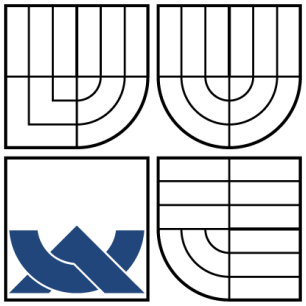
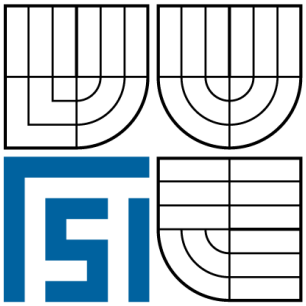


VYSOKÉ UČENÍ TECHNICKÉ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV MATEMATIKY
FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF MATHEMATICS

HYBRIDNÍ MODEL METAHEURISTICKÝCH ALGORITMŮ

HYBRID MODEL OF METAHEURISTIC ALGORITHMS

DIZERTAČNÍ PRÁCE
DOCTORAL THESIS

AUTOR PRÁCE
AUTHOR
Ing. Čeněk Šandera

VEDOUCÍ PRÁCE
SUPERVISOR
prof. RNDr. Ing. Miloš Šeda, Ph.D.

BRNO 2013

Abstrakt

Hlavním tématem této disertační práce jsou metaheuristické algoritmy v obecnějším pojetí. Úvodní kapitoly se věnují popisu širšího kontextu metaheuristik, tedy různým optimalizačním problémům, určování jejich složitosti a samozřejmě přístupům k jejich řešení. Navazující obsáhlá diskuze o metaheuristikách a jejich typických vlastnostech je následována ukázkami několika vybraných metaheuristických konceptů. Na odpozorovaných vlastnostech je vybudován obecný metaheuristický model vhodný pro vývoj nových i hybridních algoritmů. Celá práce je zakončena ukázkami autorových publikací s diskuzí o jejich užití ve vybudovaném modelu. Na přiloženém CD je k dispozici i programová implementace obecného modelu, která tvoří nedílnou součást této disertace.

Klíčová slova

metaheuristiky, optimalizace, obecný model, hybridní algoritmy

Abstract

The main topic of this PhD thesis is metaheuristic algorithms in wider scope. The first chapters are dedicated to a description of broader context of metaheuristics, i.e. various optimization classes, determination of their complexity and different approaches to their solutions. The consequent discussion about metaheuristics and their typical characteristics is followed by several selected examples of metaheuristics concepts. The observed characteristics serve as a base for building general metaheuristics model which is suitable for developing brand new or hybrid algorithms. The thesis is concluded by illustration of author's publications with discussion about their adaptation to the proposed model. On the attached CD, there is also available a program implementation of the created model.

Keywords

metaheuristic, optimization, general model, hybrid algorithms

ŠANDERA, Č. *Hybridní model metaheuristických algoritmů*. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2013. 154s. Vedoucí diplomové práce prof. RNDr. Ing. Miloš Šeda, Ph.D.

Prohlašuji, že jsem disertační práci *Hybridní model metaheuristických algoritmů* vypracoval samostatně pod vedením školitele prof. RNDr. Ing. Miloše Šedy, Ph.D. s použitím materiálů uvedených v seznamu literatury.

10.října 2013 Čeněk Šandera

Poděkování

Na tomto místě bych rád poděkoval všem, kteří mi umožnili absolvovat doktorské studium a zdárně dokončit tuto disertační práci. Děkuji především rodině a přátelům, bez jejichž podpory by se mé studium jistě stalo brzy prakticky nemyslitelným. Z profesního hlediska jsem velmi zavázán každému, kdo mi trpělivě předával své odborné zkušenosti, a to ať už formou konzultací, výuky a nebo spoluprací na publikovaných tématech. V neposlední řadě bych rád poděkoval i svému školiteli profesoru Miloši Šedovi, a to zejména za projevenou důvěru a poskytování nezbytné tvůrčí svobody.

Obsah

1 Úvod

V teoretických i aplikačních oborech se nezřídka vyskytují úlohy, jejichž řešení je s dnešními výpočetními možnostmi nedosažitelné. Takovéto úlohy bývají motivovány problémy vyskytujícími se napříč mnoha vědními oblastmi, a to od biologie přes ekonomii až po abstraktní matematiku. Nemožnost nalézat správná řešení není dána pouze nějakými chybějícími metodami, ale spíše samotnou podstatou složitosti úloh. Analýza jejich výpočetní náročnosti naznačuje, že existuje mnoho případů, kdy žádnou dostatečně efektivní metodu řešení vytvořit prostě nelze. V současné době se sice jedná pouze o domněnku, ale s přibývajícím časem stále ubývají důvody o ní příliš pochybovat.

Z akutní potřeby řešit tyto *neřešitelné* úlohy vznikají různé postupy k jejich zjednodušení. Jedním z nich je snížení nároků na přesnost výsledků a spolehnutí se převážně na experimentální přístup. Mimo jiné se mezi takové metody řadí i *metaheuristiky*, které tvoří nosné téma celé této disertace. Hlavní myšlenkou metaheuristik je systematicky prohledávat stavový prostor úlohy a ze získaných informací určovat svůj následující postup. Metaheuristiky bývají popsány velmi obecným schématem a jsou tak aplikovatelné na mnoho různých druhů problémů. Stejným způsobem se tak mohou řešit například spojité, diskrétní nebo i kombinatorické úlohy.

Kvůli své obecnosti, nemají metaheuristiky žádnou jednotnou definici a každý autor má velkou volnost v jejich návrhu. Pravidelně tak vznikají nové metody s nekompatibilní terminologií, kde nemusí být často ani jasné, v čem se vzájemně odlišují. Ze stejného důvodu se stává velmi komplikovaným i jejich vzájemné propojování a kombinování dílčích vlastností. Tato disertační práce popisuje obecný model, který takový jednotný popis umožňuje, a přesto autorům nových algoritmů zachovává velkou tvůrčí svobodu.

Disertační práce představuje metaheuristiky v širším kontextu, a proto jsou úvodní kapitoly věnovány popisu různých tříd úloh a hodnocení jejich složitosti. Uveden je zde i stručný přehled jejich možných způsobů řešení, a to včetně méně známých a alternativních cest. Po následném úvodu do problematiky metaheuristik je vypracováno obsáhlé shrnutí charakteristik jednotlivých algoritmů a na jeho základě je navržen obecný model. Celá práce je zakončena ukázkami témat autorem publikovaných metaheuristik, společně s diskuzí vztaženou k vytvořenému obecnému modelu.

2 Optimalizace

Optimalizace je činnost, která velice úzce souvisí s rozvojem celé lidské civilizace. V každém oboru se lidé vždy snažili řešit problémy tak, aby získávali nejlepší možné výsledky, a přitom spotřebovávali co nejméně svých zdrojů. Optimalizace je natolik přirozená věc, že ji nevědomky vykonává každý živý tvor, a to ať už hledá potravu, buduje obydlí, nebo si vybírá partnera pro plození potomků. Lidé už v dávnověku dokázali metodou mnoha pokusů a omylů optimalizovat závlahové systémy pro svá zemědělství nebo stavět pyramidy, které díky svým optimalizovaným vlastnostem přetrvaly až dodnes. Takovéto výsledky ale vznikaly pouze na základě zkušeností předchozích generací a nijak nezaručovaly, že žádné lepší řešení už nalezeno být nemohlo. Slovo *optimalizace* pochází z latinského *optimus*, což znamená *nejlepší*. Striktně vzato, pojem *optimalizace* by se měl tedy vázat pouze k procesu hledání zaručeně nejlepšího řešení, ale v běžném kontextu se používá i pro prosté zlepšení vlastností uvažovaného systému. K takovému zaručení je totiž potřeba, aby byl problém precizně definován, a to včetně všech jeho možných řešení. Jednoznačná optimalita výsledku se pak prokazuje buď ohodnocením každého jednotlivého řešení, anebo logickým zdůvodněním na základě jejich společných vlastností.

První netriviální optimalizační úlohy, které byly prokazatelně řešeny v duchu jednoznačné optimalizace, vznikaly již v antickém Řecku. Euklidés 300 let před naším letopočtem formuloval a vyřešil úlohu minimální vzdálenosti bodu od přímky a dokázal i, že čtverec má největší plochu mezi všemi pravoúhelníky se stejným obvodem. Za dalších 200 let ukázal Hérón Alexandrijský, že světlo vycházející z bodu A, odrážející se od zrcadla a procházející bodem B, urazí minimální možnou vzdálenost. Tyto a podobné úlohy byly většinou geometrické povahy a geometricky se dalo konstruovat i jejich řešení. Významnější obměny nastaly až s příchodem *nové matematiky*. Na počátku 17. století určil Johannes Kepler tvar vinného sudu, který má minimální plochu pláště a zároveň maximální objem. K řešení použil striktně analytické metody[1] a naznačil tak směr optimalizace pro dalších několik století. Isaac Newton a Gottfried Leibniz následně položili základy matematické analýzy, čímž optimalizace získala zbrusu nové nástroje, a tím i obrovské možnosti svého rozvoje. Velmi významný byl ideologický posun k vytváření obecných algoritmů pro optimalizaci celých tříd úloh místo hledání řešení pouze konkrétního problému. Už Newton vytvořil metodu pro numerické nalezení kořenů nelineárních funkcí[2], čímž následně dokázal snadno hledat i extrémy diferencovatelných funkcí. Navazující generace matematiků dále rozvíjely tyto nové nástroje a budovaly tak základy moderní optimalizace. Živelně začalo vznikat mnoho nových výsledků, a to napříč všemi přírodovědnými obory - Johan Bernoulli popsal Brachistochronu, Joseph Lagrange přišel se svými multiplikátory, Leonhard Euler vybudoval variační počet, Augustin Cauchy vymyslel gradientní metodu a mnoho dalších, více či méně, významných příspěvků pomohlo k položení pevných teoretických základů. První ucelená

[1] Použil metodu rozdělení tělesa na nekonečně malé části - jedna z prvních aplikací infinitezimálního počtu.

[2] Metoda je po Isaacu Newtonovi i pojmenována jako *Newtonova metoda*.

publikace věnovaná pouze optimalizaci byla ale vydána až v roce 1917 Harrisem Hancockem[3]. Ve 20. století se už buduje teorie optimalizace systematicky a hlavně díky příchodu počítačů zažívá nebývalý rozvoj. Dvě světové války popohnaly výzkum svou bezodkladnou potřebou rychle a efektivně zásobovat armády potřebnými surovinami, a světlo světa tak mohla spatřit i tzv. *simplexová metoda* dovolující řešit obrovské lineární úlohy ve velice krátkém čase.

Počítače dokáží rychle a bezchybně zpracovat enormní množství jednoduchý operací a vytváří tak zcela nová paradigmata přístupu k optimalizaci. Dnes lze běžně využívat takovou *hrubou výpočetní sílu*, která byla ještě před léty nepřekonatelnou bariérou. Optimalizační algoritmy tím mohou levně získávat množství experimentálních informací o řešeném problému a vhodně je kombinovat s teoretickými modely a předpoklady. Časté je dnes i využívání principů učících se systémů a umělé inteligence, které se i přes svou relativní nedospělost[4] stávají velice významnými nástroji přinášející pozoruhodné výsledky. Tato disertační práce je jedné z takových metod věnována, a to konkrétně *metaheuristikám*.

2.1 Formální úvod do optimalizace

Matematická optimalizace (často též *matematické programování*[5]) je soubor teoretických konceptů, které dovolují vybrat z dané množiny X *optimální* prvek x^* vzhledem k určené funkci f.

Pro formálně korektní definici musí být řádně definovány i pojmy, které jsou v ní obsaženy. Jmenovitě je zde potřeba osvětlit vágní pojem *optimální*. Primárním účelem optimalizace je vybrat *nejlepší řešení* ze všech možných, což v různých případech může znamenat různé věci. Všechna možná (akceptovatelná) řešení jsou obsažena v množině X a funkce $f : X \to Y$ umí ohodnotit kvalitu každého z nich. Základním stavebním kamenem optimalizace je tedy schopnost porovnat získané hodnoty $y = f(x)$ mezi sebou a rozhodnout tak, které řešení je nejlepší. Pro tento účel musí být definována relace uspořádání R na množině Y a *nejlepší řešení* potom je takový prvek $x^* \in X$, pro který platí:

$$f(x^*)Rf(x) \text{ pro } \forall x \in X.$$

Ne vždy je ale opravdu nutné, aby se pro každou dvojici řešení dalo jednoznačně určit, které z nich je *horší* a které *lepší*. V některých úlohách se mohou vyskytovat i páry řešení, které jsou vzájemně nesouměřitelné a pokud je úloha takto definována, tak výsledkem její optimalizace může být i množina všech těchto *nejlepších a zároveň nesouměřitelných* řešení. Příkladem takového systému může být vicekriteriální optimalizace (více viz bod 2.2.4 v kapitole 2.2).

---

[3] Harris Hancock - Theory of Maxima and Minima, 1917, Boston, USA

[4] relativní vůči analytické optimalizaci

[5] Pojem *programování* se začal používat ve 40. letech 20. století, což bylo dlouho před tím, než se stal prakticky synonymem k *počítačovému programování*.

Obecnou matematickou abstrakci optimalizační úlohy lze formulovat různými způsoby. Jedním z možných je následující formulace říkající, že nejlepší řešení jsou taková, pro která v množině X už není žádné lepší (vzhledem k R).

$$x^* = \{x \in X | \nexists x_0 \in X : f(x_0) R f(x)\},$$

kde X je množina všech možných řešení (nazývaná také *stavový prostor*), $f : X \to Y$ je funkce ohodnocení kvality řešení, R je slabě antisymetrická[6], tranzitivní[7] binární relace na Y.

Funkce f se nazývá *účelová funkce*[8] a zastupuje teoretický model studovaného jevu. Do účelové funkce mohou vstupovat i libovolné další prvky, jako např. penalizační funkce, která uměle snižuje kvalitu vybraných řešení nebo omezující podmínky (Lagrangeovy multiplikátory) ovlivňující přípustnost řešení.

Nejběžnější varianta účelové funkce zobrazuje body množiny X do nějaké číselné množiny (např. $f : X \to \mathbb{R}$) a relace R je tak obvykle volena jako standardní *menší nebo rovno* $\leq$ pro minimalizaci, popř. *větší nebo rovno* $\geq$ pro maximalizaci[9]. Tato úloha se pak zapisuje jako

$$\begin{aligned} x^* &= \operatorname{argmin}\{f(x)\} \\ x &\in X \end{aligned}$$

Na množinu X bývají často kladeny dodatečné podmínky, které jsou buď v její definici implicitně obsaženy, anebo se připojují k zadání a stávají se tak jeho nedílnou součástí. Z hlediska pracnosti a univerzálnosti formulování matematického modelu je výhodnější považovat množinu X za nějakou základní množinu v celé své šíři (např. $\mathbb{R}, \mathbb{Z}, \ldots$), a dodatečná omezení připojit ve formě podmnožin základní množiny $x \in X_0 \subset X$. Speciálně u číselných množin je zvykem podmínky zadávat ve formě rovností a nerovností

$$\begin{aligned} x^* &= \operatorname{argmin}\{f(x)\} \\ x &\in X \\ g(x) &\leq 0 \\ h(x) &= 0 \\ &\vdots \end{aligned}$$

kde $f, g, h, \ldots : X \to \mathbb{R}$.

---

[6] $\forall a, b \in X$ platí $(aRb \wedge bRa) \Rightarrow a = b$

[7] $\forall a, b, c \in X$ platí $(aRb \wedge bRc) \Rightarrow aRc$

[8] V angličtině se nejčastěji nazývá *objective function*, ale často i *cost function* (při minimalizaci), *utility function* (při maximalizaci), nebo v některých oblastech jako *energy function*.

[9] Tyto dvě úlohy jsou navzájem duální ve smyslu, že je lze libovolně převádět pouhým obrácením znaménka $f(x) \to -f(x)$.

Z interpretačního hlediska je potřeba ještě rozlišovat mezi různými typy optim, která mohou být (ne)očekávaným výstupem celé optimalizace. Hlavní rozdělení mezi optimy je na *lokální* a *globální*. U globálních je situace přesně taková, jak byla popsána výše, tedy že výsledné optimum je vztaženo k celé zadané oblasti X. Lokální optimum je naopak vztaženo pouze k určité podmnožině množiny X, a to takovým způsobem, že řešení x^* je optimální pouze ve svém nejbližším okolí. Úplně obrácená situace nastává, když například na zadané množině žádné optimum neexistuje[10], potom je na tvůrci modelu jak hledané optimum definuje (zdali může být optimum v nevlastním bodě, popř. nevlastní hodnotu apod.).

2.2 Speciální případy matematické optimalizace

Matematický model popsaný výše je natolik univerzální, že je pro něj extrémně obtížné nalézt nějaký rozumný způsob řešení. Pozornost výzkumníků se proto soustředí na dílčí speciální případy obecného modelu, pro které je mnohem snazší takové řešení nalézt[11]. Následující seznam shrnuje nejběžnější, nejzajímavější[12] a nebo dále v textu zmíněné třídy optimalizačních modelů. Modely se dělí podle typu účelové funkce, podle druhu množiny řešení, podle omezujících podmínek i podle typu uspořádání množiny hodnot účelové funkce.

Lineární optimalizace

Zdaleka nejvíce prozkoumaným a nejvíce využívaným optimalizačním modelem je lineární optimalizace. Účelová funkce i omezující podmínky jsou ve formě lineárních rovnic a nerovnic a množina možných řešení jsou nezáporná reálná čísla. Prvopočátky řešení podobných systémů se dají vystopovat až do konce 18. století k Josephu Fourierovi, ale první moderní formulace byla přinesena až v roce 1939 Leonidem Kantorovičem, který za ni obdržel Nobelovu cenu za ekonomii[13]. Zásadní roli v aplikovatelnosti modelu hrál zrod simplexového algoritmu[14], který efektivně prohledává stavový prostor a dovoluje tak řešit velice rozsáhlé úlohy. Základní formulace lineárního programu je obvykle vyjadřována v maticové formě:

$$\min_{\mathbf{x}\in X} \{\mathbf{c}^T\mathbf{x}\}$$

$$\text{vzhledem k } A\mathbf{x} \leq \mathbf{b}$$

$$\mathbf{x} \geq 0$$

---

[10] Pokud například maximalizovaná funkce f roste nade všechny meze, nebo má množina X otevřené hranice s optimální hodnotou na hranici apod.

[11] Ne však jednoduché!

[12] Nezajímavější z pohledu autora - tzn. bez snahy o objektivnost.

[13] V roce 1975 obdržel společně s Tjallingem Koopmansem *Cenu Švédské národní banky za rozvoj ekonomické vědy na památku Alfreda Nobela* - nepřesně označovanou jako Nobelova cena za ekonomii.

[14] Simplexový algoritmus byl vybrán jako jeden z deseti nejvýznamnějších algoritmů 20. století [1]

kde X je n-rozměrný stavový prostor reálných čísel, $\mathbf{c}$ je n-rozměrný konstantní vektor vah jednotlivých složek v účelové funkci (skalárním součinu), A je $n \times m$ rozměrná matice koeficientů omezujících podmínek a vektor $\mathbf{b}$ určuje jejich pravé strany.

Lineární optimalizace má naprosto výsadní postavení mezi všemi ostatními druhy optimalizace. Snahou každého výzkumníka[15] je formulovat svůj optimalizační problém právě pomocí lineárního modelu, čímž se získá záruka vysoké výpočetní efektivity a snadné škálovatelnosti.

Konvexní optimalizace

V počátcích výzkumu optimalizačních modelů a jejich řešících algoritmů se dala rychle rozpoznat výjimečnost lineárních modelů. Věřilo se, že hranice mezi *jednoduchou* a *složitou* optimalizací je totožná s hranicí mezi lineárním a nelineárním modelem. Postupem času se ale ukázalo, že tato klíčová hranice leží mezi úlohami konvexními a nekonvexními [2]. Konvexní modely mají totiž několik užitečných vlastností, které hledání optima značně zjednodušují a např. lineární optimalizace je jenom jeden z mnoha jejich speciálních případů. Nelinearita modelu tedy obecně neimplikuje vysokou složitost úlohy, ale pokud je model nekonvexní, je téměř zaručené, že jeho řešení nebude vůbec snadné.

Konvexní optimalizaci lze definovat jako optimalizaci konvexní funkce na konvexní množině. *Konvexní množina* je taková množina, ve které leží celá úsečka mezi každou dvojicí jejich bodů[16] a *konvexní funkce* je taková funkce, jejíž hodnoty v každém podintervalu leží pod úsečkou, která spojuje hodnoty v jeho krajních bodech[17]. Úloha má tedy tvar

$$\min_{\mathbf{x} \in X} \{f(\mathbf{x})\}$$

$$\text{vzhledem k } g_i(\mathbf{x}) \leq 0 \text{ pro } i = 1, \ldots, m$$

kde $X \subset \mathbb{R}^n$ je konvexní podmnožina reálného n-rozměrného prostoru a funkce $f(\mathbf{x})$ a $g_i(\mathbf{x})$ jsou na ní konvexní.

Hlavní výhoda konvexní optimalizace je, že pokud algoritmus nalezne jakékoliv lokální optimum, tak automaticky nalezl i globální optimum. Pokud je účelová funkce *striktně konvexní*[18], tak je její optimum dokonce jedinečné a když striktně konvexní není, tak je alespoň zaručené, že množina všech optim tvoří zase konvexní množinu.

Konvexní optimalizace je široká obecná třída, a tudíž existují i její důležité speciální případy pro něž byly vytvořeny samostatné řešící algoritmy. Kromě lineární optimalizace patří mezi

---

[15] obrazně řečeno

[16] $\forall x, y \in X, \forall t \in [0,1] : tx + (1-t)y \in X$

[17] pro jednorozměrný případ: $\forall x, y \in X, \forall t \in [0,1] : f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$

[18] Oproti konvexní funkci se její definice liší pouze vyžadováním striktní nerovnosti $<$ místo $\leq$.

významnější z nich např. *kvadratická optimalizace*, jejíž účelová funkce je definována pomocí symetrické matice Q a vektoru c jako $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}Q\mathbf{x}^T + \mathbf{c}^T\mathbf{x}$. Omezující podmínky kvadratické optimalizace jsou stejné jako v lineární optimalizaci, tedy soustava lineárních nerovnic zadaná pomocí matice A jako $A\mathbf{x} \leq \mathbf{b}$.

Stochastická optimalizace

Při modelování reálných úloh se často naráží na neznalost přesných hodnot konstantních parametrů vstupujících do vytvářeného modelu. Takové parametry mohou být náhodnými proměnnými, které mají při každém pozorování jinou hodnotu. Odhadnutím a použitím jejich střední hodnoty (nebo jiného skalárního parametru) lze u některý úloh narazit na neadekvátní výsledky. Jedním ze způsobů, jak se tomu vyhnout, je vytvořit stochastický model, který v sobě explicitně náhodné veličiny obsahuje. Na takové úlohy ale nelze přímočaře použít klasické algoritmy z deterministické optimalizace[19], ale musí být vytvořen kvalitativně nový přístup k jejich řešení.

Hodnota účelové funkce stochastického modelu je náhodnou veličinou a musí se tedy nejdříve rozhodnout, jakým způsobem se bude posuzovat její optimum. Intuitivním a nejběžnějším způsobem je optimalizace střední hodnoty účelové funkce a taková stochastická optimalizace má obecný tvar

$$\begin{gathered}\min_{\mathbf{x}\in X(\xi)} \{\mathrm{E}f(\mathbf{x},\xi)\} \\ \text{vzhledem k } \; g_i(\mathbf{x},\xi) \leq 0 \,\text{pro}\; i = 1,\ldots,m \\ \mathbf{x} \in X(\xi) \text{ skoro jistě},\end{gathered}$$

kde x je deterministický vektor z X a ξ je náhodný vektor ovlivňující chování modelu. Pojem *skoro jistě* znamená, že $x \in X$ s pravděpodobností $p = 1$.

Častým typem stochastické optimalizace je vícestupňová optimalizace, která v sobě zahrnuje několik na sebe navazujících optimalizačních podproblémů. V lineární podobě má dvoustupňová stochastická optimalizace většinou tvar

$$\begin{gathered}\min_{\mathbf{x}\in X} \{\mathbf{c}^T\mathbf{x} + \mathrm{E}Q(\mathbf{x},\xi)\} \\ \text{vzhledem k } \; A\mathbf{x} \leq \mathbf{b} \\ \mathbf{x} \geq 0,\end{gathered}$$

kde $Q(\mathbf{x},\xi)$ je optimální hodnota problému na druhém stupni

[19] Optimalizace, kde všechny hodnoty jsou běžné skalární veličiny.

$$\min_{\mathbf{y}\in Y} \{\mathbf{q}(\xi)^T \mathbf{y}\}$$

$$\text{vzhledem k } T(\xi)\mathbf{x} + W(\xi)\mathbf{y} \leq \mathbf{h}(\xi)$$

$$\mathbf{y} \geq 0.$$

Výraznou konceptuální odlišností vícestupňových modelů od klasických modelů je nutnost hledat řešení za základě neznámé budoucí realizace náhodných hodnot. Výsledným řešením popsaného modelu je hodnota vektoru $\mathbf{x}$, který v modelu ovlivní počáteční stav a až se realizuje hodnota proměnné ξ, ovlivní i výsledek druhostupňové optimalizace. Názorným příkladem takového systému je model produkce slitin s předepsanou koncentrací prvků v tavicích pecích. Na začátku se do roztavené železné rudy přimíchá definované množství základních prvků ($\mathbf{x}$) a po několika procedurách se změřením zjistí jejich reálné koncentrace a úbytky ξ. Následně se v druhém stupni musí do slitiny přidat chybějící prvky v potřebném množství y. Vícestupňové modely jsou většinou stavěny tak, že cena práce ve vyšším stupni je výrazně vyšší než v nižším, a proto se musí co nejlépe optimalizovat už první stupeň $\mathbf{x}$.

Exaktní výpočet stochastických optimalizačních úloh vyžaduje velice časově náročné výpočty vícerozměrných integrálů (pro střední hodnoty v účelové funkci), proto je častým přístupem k řešení přeformulování na deterministickou verzi, tzv. *scénářový přístup*. Náhodný vektor ξ se vhodně navzorkuje na k nezávislých realizací a účelová funkce se přetransformuje na vážený součet prvků k stejných optimalizačních problémů, kde se místo náhodné veličiny ξ dosadí jeho zvolené realizace $\hat{\xi}_i$ pro $i = 1, \ldots, k$. Váha jednotlivých optimalizačních podproblémů je rovna pravděpodobnosti použité realizace (vzhledem k množině všech zvolených realizací). Pro dvoustupňovou lineární stochastickou optimalizaci lze tímto způsobem získat odpovídající deterministický model, avšak k-krát větší než původní stochastický:

$$\min_{\mathbf{x}\in X} \left\{ \mathbf{c}^T\mathbf{x} + \frac{1}{k}\sum_{i=1}^{k} Q(\mathbf{x}, \hat{\xi}_i) \right\}$$

$$\text{vzhledem k } A\mathbf{x} \leq \mathbf{b}$$

$$\mathbf{x} \geq 0,$$

kde $Q(\mathbf{x}, \hat{\xi}_i)$ je optimální hodnota problému na druhém stupni pro konkrétní realizaci $\hat{\xi}_i$

$$\min_{\mathbf{y}\in Y} \{\mathbf{q}(\hat{\xi}_\mathbf{i})^T \mathbf{y}\}$$

$$\text{vzhledem k } T(\hat{\xi}_i)\mathbf{x} + W(\hat{\xi}_i)\mathbf{y} \leq \mathbf{h}(\hat{\xi}_i)$$

$$\mathbf{y} \geq 0.$$

Vícekriteriální optimalizace

Základní druhy optimalizace jsou většinou spojeny s účelovou funkcí typu $f : X^n \to \mathbb{R}$, tedy zobrazením z n-rozměrné množiny do množiny reálných čísel. Často je ale nezbytné optimalizo-

vat více různých účelových funkcí zároveň, tzn. že optimalizační funkce má tvar vícerozměrného zobrazení $f : X^n \to \mathbb{R}^m$.

Toto rozšíření klasického konceptu s sebou ale přináší značné problémy s interpretací hledaného optima. Body obecného vícerozměrného prostoru totiž běžným způsobem nelze jednoznačně seřadit, tím pádem nelze ani jednoznačně určit optimum na definované podmnožině. Obecný tvar vícekriteriální optimalizace lze zapsat jako

$$\min_{\mathbf{x} \in X} \{(f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x}))\}$$

$$\text{vzhledem k } \mathbf{x} \in X,$$

kde $f_i(\mathbf{x})$ jsou funkce $X \to \mathbb{R}$ a dohromady tvoří m-rozměrnou vektorovou účelovou funkci. Existuje několik rozdílných způsobů, jak chápat minimalizaci vektorové funkce. Jednoznačně optimální řešení je takové, které má všechny komponenty nejmenší možné, a neexistuje tedy žádné jiné řešení s některou komponentou menší. Tímto přístupem se řídí tzv. *Paretova dominance*, která zavádí *relaci dominance* $\preceq$ (popř. $\prec$) ve významu *"je preferováno před"*. Pokud má jedno řešení $\mathbf{x_0} = (x_0^1, x_0^2, \ldots, x_0^m)$ menší všechny komponenty než řešení $\mathbf{x_1} = (x_1^1, x_1^2, \ldots, x_1^m)$, tak řekneme, že řešení $\mathbf{x_1}$ je dominováno řešením $\mathbf{x_0}$ (tedy $\mathbf{x_0} \prec \mathbf{x_1}$). Pokud jsou naopak některé komponenty větší a některé menší, říkáme, že jsou řešení $\mathbf{x_0}$ a $\mathbf{x_1}$ *vzájemně nedominantní*. Výsledkem optimalizace založené na Paretově dominanci je tak množina všech vzájemně nedominantních řešení[20], ze kterých si uživatel může vybrat podle svého uvážení.

Pokud je ale požadováno, aby výsledkem optimalizace byl jediný bod, musí být explicitně definován způsob uspořádání, anebo očekávané kvality řešení. Mezi nejběžnější způsoby předdefinování *nejlepšího* řešení je např. podmínka maximální velikosti konkrétních komponent nebo přímo jejich očekávaných hodnot[21]. V takovém případě nemusí být cílem řešení modelu optimalizace ve smyslu nalezení nejlepšího řešení, ale pouze takového, které splňuje předepsané vlastnosti.

Velice oblíbený je i způsob převodu vícerozměrné účelové funkce na funkci jednorozměrnou. Příkladem může být lineární kombinace

$$f(\mathbf{x}) = \sum_{i=1}^{m} w_i f_i(\mathbf{x}),$$

kde w_i jsou konstantní váhy pro jednotlivé komponenty a celá víceúčelová optimalizace je tak převedena na klasickou jednoúčelovou úlohu. K tomuto účelu se nemusí používat pouze lineární kombinace, ale jakákoli funkce $U : \mathbb{R}^m \to \mathbb{R}$ nazývaná *funkce užitečnosti*[22].

---

[20] V literatuře označováno obvykle jako *paretovsky optimální* množina řešení.

[21] Tzv. cílové programování (goal programming).

[22] Anglicky *utility function*.

Další významné typy optimalizace

Optimalizace v obecném smyslu je potřebná pro většinu vědeckých i praktických disciplín. Není tedy překvapením, že existuje nepřeberné množství typů optimalizačních úloh. Předchozí zmíněné typy budou využívány v následujících kapitolách této disertace, a proto jsou popsány ve větší šíři. Z dalších existujících, avšak nikoli méně významných, tříd optimalizačních úloh lze ještě stručně uvést například

- *Celočíselná optimalizace* je taková, kde je stavový prostor tvořen pouze vektory se složkami patřící do celých čísel. Tvar účelové funkce a podmínky mohou být v podstatě libovolné. Mohou tak vznikat stochastické, víceúčelové nebo jakékoliv jiné celočíselné optimalizační úlohy. Za jeden ze základních typů jsou ale považovány celočíselné lineární úlohy. Ty mají stejný tvar jako spojité lineární úlohy pouze s tím rozdílem, že proměnné mohou být jen z množiny $\mathbb{Z}$. Častým případem bývá i spojení celočíselné a spojité optimalizace dohromady, kdy část proměnných může nabývat pouze celá čísla a část libovolných spojitých hodnot. Takové úlohy se označují jako *smíšené celočíselné programování*.
- *Kombinatorická optimalizace* řeší problematiku spojenou s výběrem uspořádaných i neuspořádaných podmnožin z definovaných konečných množin. Velmi častým příkladem je určování nejlepšího pořadí vybraných prvků. Nejznámějším zástupcem je bezesporu problém obchodního cestujícího, který hledá nejkratší uzavřenou cestu mezi všemi zadanými městy. Jiným velmi populárním příkladem je tzv. problém batohu, vybírající z určených předmětů nejhmotnější podmnožinu, kterou ještě batoh unese. Kombinatorické úlohy jsou velmi často reprezentovány jako grafové struktury.
- *Robustní optimalizace* je velice užitečnou třídou, která se snaží vycházet z pozorování na reálných modelech. Stejně jako ve stochastické optimalizaci se totiž nepředpokládá, že parametry modelu jsou přesně stanovitelné a neměnné. Robustní přístup k optimalizaci tak nehledá jen nejlepší možné řešení, ale takové, které má nejlepší vlastnosti vůči malým chybám v parametrech. Pokud existuje řešení úlohy, jehož blízké okolí má významně horší hodnoty než nalezené optimum, není považováno za robustní. Naopak, pokud celé okolí je dostatečně kvalitní, pak řešení prohlásíme za robustní. Interpretace a ospravedlnění tohoto přístupu spočívá v tom, že v realitě nic nejde nastavit naprosto přesně, a proto si musíme být jisti, že i nepřesně nastavený systém bude pracovat (téměř) optimálně a bude tedy tzv. robustní vůči malým chybám měření.
- Ve výčtu tříd optimalizačních úloh by bylo možné pokračovat ještě velice dlouho, ale to není cílem práce. Smyslem je pouze ukázat šíři a variabilitu optimalizačních úloh, které, ač mají mnoho společného, se mohou lišit ve velice významných detailech. Optimalizační úlohy se nemusí lišit jen v typem účelové funkce a stavového prostoru, ale mohou být založeny i na odlišných matematických přístupech. Časté například začínají být modely s prvky fuzzy matematiky, tedy tzv. *fuzzy optimalizace*, nebo optimalizace úloh s parametry proměnnými

v čase. K odlišným přístupům patří i tzv. *metaoptimalizace*, která optimalizuje parametry jiných optimalizačních metod.

2.3 Složitost úloh

V první polovině 20.století si vědci při konstrukci nových algoritmů začali uvědomovat, že ne každá úloha má stejně časově náročné řešení, ale naopak, že mohou mezi nimi existovat značné rozdíly. Výpočetní zařízení, která navržené algoritmy zpracovávají, mají omezenou jak velikost paměti, tak i maximální rychlost vykonávaných instrukcí. Navrhované algoritmy proto musí být výpočetním zařízením přizpůsobovány, aby byl jejich běh co nejvíce *efektivní*. I přes značnou snahu mnoha odborníků se ale pro některé typy úloh nepodařilo žádný takový *efektivní* algoritmus nalézt. Předním tématem teoretické informatiky se proto stala otázka, co tyto úlohy dělá takto složitými a zda nějaký *efektivní* algoritmus vůbec může existovat. Takovéto a podobné otázky pak daly základ nové vědní oblasti - *teorii složitosti*[23] [3].

Základním nástrojem teorie složitosti je matematický model obecného výpočetního zařízení nazvaný Turingův stroj.[24] Princip jeho činnosti spočívá v systematickém měnění jeho vnitřních stavů pomocí přiřazené přechodové funkce

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\},$$

kde Q je konečná množina všech možných vnitřních stavů Turingova stroje, Γ je symbol abecedy, které Turingův stroj rozumí a která ho ovládá. Symboly L, R jsou posuny čtecí hlavy (vlevo, vpravo) zajišťující přístup k paměti stroje. Právě druhem své paměti se Turingův stroj výrazně odlišuje od jednodušších modelů, jako je např. konečný automat. Paměť si lze představit jako pásku, ze které lze zapisovat a číst symboly abecedy Γ. Jakýkoli úkon s páskou se provádí pomocí čtecí hlavy, která se po ní může libovolně pohybovat. Celkově je Turingův stroj definován jako šestice.

$$M = (Q, \Gamma, s, b, F, \delta),$$

kde zbývající výše nezmíněné symboly jsou $s \in Q$ - počáteční stav, $b \in \Gamma$ reprezentuje prázdný symbol[25] a množina $F \subseteq Q$ obsahující koncové stavy.[26].

Principem činnosti Turingova stroje je tedy přečíst symbol na pozici pásky, kde se právě nachází čtecí hlava, a následně pomocí přečteného symbolu a přechodové funkce změnit vnitřní

[23] Anglicky *complexity theory*.

[24] Jako základní nástroj teorie složitosti lze uvažovat i jiné objekty (např. konečný automat), ale pro obsah této práce je významný primárně Turingův stroj.

[25] Prázdný symbol se ale nevyužívá ve vstupní abecedě.

[26] V případě, že vstupní abeceda Σ není identická s $\Gamma \setminus \{b\}$, přidává se do definice, která se tak stává uspořádanou sedmicí.

stav stroje. Tyto kroky se opakují tak dlouho, dokud se nenarazí na ukončovací symbol. Definovaný Turingův stroj tak reprezentuje konkrétní algoritmus i s jeho průběhem. (Detailnější vysvětlení problematiky Turingova stroje by bylo na tomto místě nadbytečné a lze jej najít v každé učebnici teoretické informatiky.)

Klíčový prvek pro hodnocení *efektivity* algoritmů (Turingových strojů) se skrývá ve vztahu délky vstupu ku počtu kroků, které stroj musí vykonat, než je úspěšně ukončen. Pokud je algoritmus vytvořen způsobem, že dokáže řešit daný typ úlohy pro její obecný rozměr, potom lze studiem této závislosti rozhodnout o jeho relativní efektivitě vůči jiným algoritmům.

Asymptotická složitost

Ustálená notace klasifikace algoritmů podle jejich časové nebo prostorové náročnosti, prováděná podle výše naznačeného schématu, je formalizována tzv. *asymptotickou složitostí*[27]. U zkoumaného algoritmu se zvolí parametr n, který popisuje velikost jeho vstupu a pomocí Turingova stroje se následně algoritmus analyzuje. Počet vykonaných instrukcí Turingova stroje lze takto vyjádřit jako funkci délky vstupu, tzn. $f(n)$. Ne každý algoritmus vykoná přesně stejný počet kroků pro stejnou délku vstupu[28], proto je vhodné funkci $f(n)$ chápat jako vhodně definovanou aproximaci konkrétních počtů. Nejtypičtější a nejdůležitější aproximací je *horní závora* označována symbolem $\mathcal{O}(\cdot)$. Říkáme, že funkce $f(n)$ má asymptotickou složitost $g(n)$, pokud existuje pevné n_0 takové, že pro všechna n větší než n_0 je funkce $f(n)$ menší než $g(n)$. Dokonce stačí, aby tato vlastnost platila pouze pro libovolný skalární násobek funkce $g(n)$. Exaktně je tedy $\mathcal{O}(g(n))$ množina všech funkcí, které jsou *asymptoticky ohraničeny shora* pomocí funkce $g(n)$ a lze ji definovat jako

$$f(n) \in \mathcal{O}(g(n)) \Leftrightarrow \exists(k > 0)\exists n_0 \forall n > n_0 : |f(n)| \leq |k \cdot g(n)|.$$

Obdobným způsobem se definuje i *dolní asymptotické ohraničení* $\Omega(g(n))$, kde je nerovnost zaměněna za $\geq$ nebo *oboustranné ohraničení* $\Theta(g(n))$, kde je funkce $f(n)$ ohraničena zespoda i shora stejnou funkcí $g(n)$, ale s různými násobícími konstantami k_1 a k_2. Je vhodné podotknout, že asymptotické ohraničení funkcí se týká pouze části funkce od určitého fixního n_0 až do nekonečna, tzn. že na *začátku* funkce (pro $n < n_0$) mohou být funkce f a g i v obráceném poměru.

Koncept asymptotické ohraničenosti není výhradně pojmem z teoretické informatiky, ale běžně je používán i v mnoha matematických oborech, zejména v matematické analýze nebo numerické matematice. Ve spojení s již zmíněnou závislostí vstup a výstupu Turingova stroje

[27] V anglicky psané literatuře je většinou nazývána *Big O notation* nebo *Landau notation*.

[28] Např. vyhledávací algoritmy svůj běh ukončí v případě nalezení prvku, který se může nacházet na libovolné pozici prohledávané struktury.

je tento koncept velmi názorným nástrojem k vzájemnému porovnání *efektivity* zkoumaných algoritmů.

2.4 Třídy složitosti

Na základě znalosti konceptu asymptotické složitosti můžeme přehledně roztřídit úlohy do skupin podle náročnosti jejich řešení. Zjednodušeně řečeno, úloha je tak složitá, jak náročný je nejefektivnější algoritmus, který ji dokáže vyřešit. Pokud tedy pro vybranou úlohu nalezneme algoritmus, který ji dokáže vyřešit v čase $\mathcal{O}\left(n^2\right)$, tak můžeme říct, že její složitost je stejná jako každé jiné úlohy řešitelné stejně náročným algoritmem. Můžeme dokonce říct, že jelikož funkce n^3 roste rychleji než n^2, tak úlohy, pro které je znám jen algoritmus $\mathcal{O}\left(n^3\right)$, jsou složitější než úlohy s algoritmem $\mathcal{O}\left(n^2\right)$. Z toho konceptu však vyplývá, že roztřídění úloh do příslušných tříd by záleželo pouze na schopnosti výzkumníků nalézt vhodný algoritmus a s každým vymyšlením nového a efektivnějšího algoritmu by i úloha přešla do lepší třídy. Částečně tomu tak opravdu je, ale právě z důvodu obecnějšího porovnání složitostí úloh byl vytvořen navazující koncept *vzájemné redukovatelnosti*.

Nejefektivnější algoritmy mají složitost $\mathcal{O}(1)$, což znamená, že pro libovolně velký vstup je čas potřebný k výpočtu pořád stejný. Takové algoritmy ale z výpočetního hlediska nebývají zrovna moc zajímavé a mnohem běžnější jsou ty, jejichž čas je na vstupu nějak závislý. Pokud má naopak algoritmus složitost $\mathcal{O}(e^n)$, je při dnešních možnostech prakticky nepoužitelný. Čas potřebný pro běh takového algoritmu roste s velikostí úlohy tak rychle, že by v mnoha případech trval stovky let.[29] Za ještě rozumně zvládnutelné úlohy jsou považovány takové, jejich výpočetní čas roste s velikostí vstupu maximálně polynomiálně, tzn. takové úlohy, pro něž existuje algoritmus se složitostí $\mathcal{O}(n)$, $\mathcal{O}\left(n^2\right)$,...,$\mathcal{O}\left(n^k\right)$ apod.

Koncept *vzájemné redukovatelnosti* stanovuje vztah mezi dvěma různými úlohami. Pokud lze převést úlohu A na úlohu B pomocí *rychlého* algoritmu, tak úloha A je určitě lehčí nebo stejně těžká jako úloha B. Pokud lze tuto redukci provést obousměrně, je jejich složitost v podstatě ekvivalentní. Stačí tedy znát složitost jedné úlohy a za pomoci konceptu redukovatelnosti ověřit, jestli je jejich složitost ekvivalentní.

Oblíbené třídy složitosti

Nejběžnější třídy složitosti jsou třída P a třída NP. Obě jsou definovány jako třídy sdružující rozhodovací problémy, které jsou řešitelné v polynomiálním čase. Podstatný rozdíl je ale v tom, že pro algoritmy z třídy P existuje řešící algoritmus pro *deterministický Turingův stroj*, kdežto pro třídu NP existuje pouze pro *nedeterministický Turingův stroj*. Deterministický Turingův

[29] Není problém konstruovat i relativně malé úlohy, jejichž výpočetní čas by byl delší, než je stáří našeho vesmíru, a to dokonce i při použití všech dnešních počítačů na Zemi zároveň.

stroj je přitom takový stroj, který nabývá v jeden okamžik právě jeden stav. U nedeterministického Turingova stroje toto omezení neplatí a stroj může v jeden okamžik nabýt libovolný počet svých vnitřních stavů. Je zřejmé, že deterministický stroj je pouze speciálním případem nedeterministického stroje, a proto všechny problémy z třídy P patří zároveň i do třídy NP.

Není snadné si představit stroj, který může nabývat více stavů zároveň a současně je měnit pomocí asociované přechodové funkce. Ve skutečnosti žádný takový stroj v reálném světě neexistuje a všechny dnes běžně používané počítače jsou *pouze* deterministického typu. Třída P tedy obsahuje problémy, které jsou efektivně řešitelné na počítačích dnešního typu, kdežto třída NP obsahuje, kromě úloh třídy P, i problémy, které dnes není nikdo schopen efektivně vyřešit.

Jak již bylo uvedeno, obě třídy obsahují výhradně rozhodovací problémy, tedy problémy, na něž existuje odpověď *ano* nebo *ne*. Důležitým společným znakem úloh z obou tříd je, že pokud je odpověď na zadaný vstup *ano*, musí existovat polynomiální algoritmus pro deterministický Turingův stroj, který ji dokáže ověřit.[30]

Celá třída NP obsahuje obrovské množství velice užitečných úloh, které jsou dále seskupovány na další podtřídy. Příkladem může být rychlá třída L obsahující úlohy řešitelné v logaritmickém čase[31] a jelikož jakýkoliv polynom roste rychleji než funkce logaritmus, je třída L dozajista i podmnožinou třídy P.

Na druhém konci rychlostního spektra (vztaženo zase k deterministickému stroji) jsou úlohy z třídy NP-úplná, která sdružuje nejtěžší úlohy z třídy NP. Každou úlohu z NP totiž lze polynomiálně redukovat na úlohu z NP-úplná, což zjednodušeně znamená, že tato třída obsahuje úlohy, které jsou stejně těžké, nebo dokonce těžší, než jakékoliv jiné z NP. Z podmínky, že každá úloha z třídy NP je redukovatelná na libovolnou úlohu z NP-úplná plyne, že všechny úlohy v třídě NP-úplná jsou na sebe redukovatelné navzájem. Toto je velice významný poznatek, který vede k důsledku, že by stačilo nalézt jediný polynomiální algoritmus pro deterministický Turingův stroj řešící libovolnou úlohu z NP-úplná, a ihned by jím bylo možné vyřešit i všechny zbývající úlohy této třídy (a následně i všechny úlohy z třídy NP). V této nejtěžší třídě z NP se nachází mnoho zajímavých a užitečných úloh, ale dodnes nebyl nikdo schopný přijít s polynomiálním algoritmem, který by nějaký z nich vyřešil. Není ani známo, jestli takový algoritmus vůbec může existovat, a tato nejasnost je dokonce tak významná, že byla zařazena mezi sedm nejdůležitějších matematických problémů tisíciletí[32]. Formálně se jedná o dokázání nebo vyvrácení rovnosti P=NP, tedy že každý algoritmus vykonaný na nedeterministickém Turingově stroji by bylo možné efektivně vykonat i na stroji deterministickém. Většina současných odborníků na teoretickou informatiku se ale k dokázaní této rovnosti staví spíše skepticky a své práce

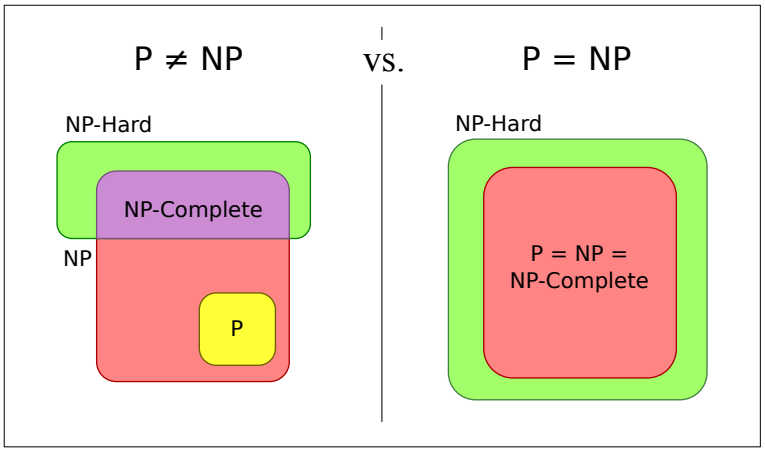
[30] Tato vlastnost může být dokonce použita i jako alternativní definice třídy NP.

[31] Řešitelná v logaritmickém čase na deterministickém Turingově stroji.

[32] Ze sedmi stanovených nevyřešených problémů je dnes už jen šest. Poincarého domněnka byla vyřešena v roce 2006.

staví na hypotéze, P≠NP. Několik profesionálních i amatérských výzkumníků již publikovalo *svůj důkaz* zmíněné nerovnosti, ale všechny byly nakonec shledány chybnými. Největší naděje a výjimečnou pozornost vzbudil v létě 2010 stostránkový důkaz Vinaye Deolalikara, jednoho z nejvýznamnějších soudobých teoretiků, ale i jeho důkaz byl nakonec po několika týdnech prohlášen za nesprávný.

I přesto, že úlohy třídy NP-úplná jsou pro dnešní počítače neřešitelné, není to zdaleka nejtěžší třída úloh obecně. Další v pořadí složitosti[33] jsou problémy třídy NP-těžká. Tato třída, i přes svůj lehce zavádějící název, neobsahuje problémy pouze z třídy NP, ale přesto k nim má blízký vztah. Každý problém z třídy NP-těžká totiž musí být alespoň tak těžký, jako nejtěžší problémy z třídy NP. Jinak řečeno, na každý problém z třídy NP-těžká musí být polynomiálně redukovatelný nějaký problém z NP-úplná. Vyřešením problému z NP-těžká by se tedy vyřešily i všechny problémy z třídy NP-úplná, a následně i všechny problémy z celé třídy NP. Úlohy v třídě NP můžou být pouze rozhodovacího typu (tedy odpověď *ano* nebo *ne*), pro třídu NP-těžká už ale žádné takové omezení neplatí. Typickými zástupci této třídy jsou úlohy prohledávání stavových prostorů a vybírání prvků s definovanou vlastností - tedy *optimalizace*.



Obrázek 2.1 Grafická reprezentace problému P=NP

Mezi další *základní* a *zajímavé* třídy složitosti patří například třída coNP obsahující rozhodovací problémy, která je v jistém smyslu doplňková k třídě NP. Její definice vyžaduje, aby pro každou obsaženou úlohu existoval polynomiální algoritmus, který ověří správnost odpovědi *ne* (narozdíl od NP, kde se vyžaduje polynomiální ověření správnosti odpovědi *ano*). Přesný vztah mezi třídami NP a coNP není v současné době znám, ale předpokládá se, že jsou to dvě různé třídy s neprázdným průnikem. Některé problémy tedy mohou ležet zároveň v NP i coNP, ale přesná podoba tohoto průniku není zatím úplně jasná. Stejně jako jsou z třídy NP odvozeny třídy NP-úplná a NP-těžká, tak i třída coNP má své odvození třídy coNP-úplná a coNP-těžká, které jsou definovány ekvivalentním způsobem jako pro NP.

[33] Jedná se jen o řečnický obrat, žádné jednoznačné *pořadí složitosti* není definováno.

Koncept odvozování tříd úplná a těžká nemá smysl pouze pro třídy NP a coNP, ale je snadno zobecnitelný i pro mnoho jiných tříd složitosti. Pro vhodnou třídu X je třída X-úplná její podtřída nejtěžších úloh. A nejtěžší úlohy jsou takové, na které lze *snadno* převést libovolnou úlohou z X. Pojem *snadno* je relativní vůči třídě X, tedy vhodná redukce musí patřit do snadnější třídy složitosti. Třída X-těžká je následně taková, která je alespoň tak těžká jako X-úplná, tedy na každou úlohu z X-těžká lze redukovat nějakou úlohu z X-úplná. Vhodným příkladem může být třída P-úplná, kde jsou k redukování použity algoritmy s maximálně polylogaritmickou složitostí $\mathcal{O}\left(\log^k n\right)$, kde $k \in \mathbb{N}$. Významnou interpretací třídy P-úplná je, že sdružuje problémy, které je obtížné paralelizovat na deterministickém Turingově stroji (a v případě, že P≠NP, tak paralelizace nebude možná vůbec).

Teoretický popis složitostí algoritmů a rozřazení problémů do vhodných tříd je stále aktuální a živé vědecké téma. Na základě nových výzkumů se stále definují nové třídy složitosti seskupující problémy podle potřebných vlastností. Aktuální seznam tříd lze nalézt např. na stránkách komunitního projektu *Complexity Zoo*[34], kde je v současné době uvedeno na 495 různých tříd složitosti. Třídy se od sebe liší typem výsledků, časovou i prostorovou náročností, druhem Turingova stroje a dalšími vlastnostmi.

2.5 Metody řešení

Předchozí kapitola ukázala, že pro různé úlohy existují různě náročné způsoby řešení. Existují dokonce úlohy, které žádný efektivní způsob řešení při dnešních možnostech nemají. Mnoho reálných i teoretických problémů ale vyžaduje nalézt řešení i pro takto složité úlohy, proto je není možné nechat bez povšimnutí. Pokud se zaměříme pouze na problémy spojené s optimalizací, lze rozpoznat několik vzájemně odlišných přístupů k jejich řešení.[35]

Exaktní algoritmy

Snem každého výzkumníka jsou algoritmy, jejichž výstupem je přesně požadovaný výsledek a které jej jsou schopny při každém opakování znovu bezpečně nalézt. Takový algoritmus ale nemusí být možné vůbec nalézt a nebo je jeho náročnost tak obtížná (časově nebo prostorově), že je prakticky nepoužitelný.

Exaktní algoritmy vychází vždy z precizního pochopení tvorby stavového prostoru. Musí být vždy známo, jaké vlastnosti mají jednotlivá řešení a jaké jsou mezi nimi vztahy. Pokud lze účelovou funkci popsat pomocí analytické funkce s existující derivací, pak můžeme snadno

[34] http://complexityzoo.uwaterloo.ca/

[35] Rozčlenění algoritmů do skupin podle přístupů a myšlenek není nijak jednotný. Přístupy se mohou různě kombinovat a jejich definice nemusí být také jednotná. Každé takové rozdělení je *nutně* nekompletní a rozporuplné. Uvedený přístup je poplatný pouze pro zaměření publikace, nikoliv optimalizaci obecně.

nalézt všechny její stacionární body[36], a tím následně odhalit i ve kterém z nich se nachází optimum. Ne vždy je tento úkol zrovna snadný, ale díky znalosti funkčního předpisu pro celý definiční obor dokážeme nahlédnou do jejich zákonitostí, z nichž lze odvodit vše potřebné.

Občas ale derivace není z nějakých důvodů užitečná. Při úloze lineárního programování je účelová funkce sice popsána analyticky a derivace existuje na celém definičním oboru, ale nikde není nulová. Neexistují v ní žádné stacionární body a optimum se nalézá v nějakém z rohů oblasti určené omezujícími podmínkami. Díky znalosti zákonitostí lineárního programu ale existují algoritmy, které přesné řešení najdou vždy. Příkladem může být slavný simplexový algoritmus procházející postupně zmíněné rohové body, dokud nenarazí na hledané optimum.[37]

Pokud řešení lineárního programu existuje, tak jej simplexový algoritmus nalezne. Nemusí to být ale zrovna nejefektivnější cesta. Z principu algoritmu plynou nevýhody při zpracování úloh s mnoha omezeními, ale málo proměnnými. Matematicky čistá cesta vede přes transformaci úlohy na úlohu duální. Při transformaci se zamění minimalizace účelové funkce za maximalizaci (popř. obráceně) a proměnné z účelové funkce se prohodí s konstantami v omezeních. Úloha se tak zredukuje sice na více rozměrnou úlohu, ale zato s menším počtem omezujících podmínek. Z nalezeného optima duální úlohy se snadno zrekonstruuje řešení původní úlohy.

Dualita lineárních úloh zefektivňuje průběh výpočetního algoritmu. I bez transformace by algoritmus byl schopen řešení nalézt, ale bylo by to náročnější. Transformací lze ale exaktně řešit i úlohy, které by jinak byly opravdu výrazně obtížnější. Příkladem může být *geometrické programování*, jehož účelová funkce má tvar

$$f(x) = \sum_{k=1}^{K} c_k x_1^{a_1 k} x_2^{a_2 k} \cdots x_n^{a_n k},$$

kde $x_i, c_i \in \mathbb{R}^+$, $a_i \in \mathbb{R}$ a omezující podmínky ve formě rovností a nerovností mohou mít obdobné tvary. Tato optimalizační úloha je obecně nekonvexní a tedy složitě řešitelná. Použitím logaritmické transformace ale přejde úloha na součet exponenciálních funkcí a stane se konvexní. Přesnost nalezeného řešení pak už závisí pouze na zvoleném algoritmu pro jeho nalezení.

Existuje mnoho dalších exaktních přístupů k optimalizaci zaručující nalezení optimálních hodnot, všechny ale vyžadují hluboké studium a náročnou matematickou práci, která ale v praktických aplikacích nemusí být vždy rentabilní.

Jako exaktní metodu lze označit i prosté prozkoumání všech možných bodů stavového prostoru. Tzv. *prohledávání hrubou silou* je účinný nástroj pro rychlé řešení malých úloh. Jak ale bylo ukázáno v předchozích kapitolách, počet možností složitějších úloh může růst tak rychle, že by takové prohledávání trvalo celá staletí.

---

[36] Body, ve kterých má funkce nulovou derivaci.

[37] Jakmile algoritmus nalezne lokální optimum, může výpočet ukončit, protože v lineární optimalizaci je lokální optimum zároveň i globální.

Téměř exaktní algoritmy

Pod vágní nadpis *téměř exaktní algoritmy* lze zařadit velmi širokou paletu algoritmů. Jejich společným rysem je, že nalezená řešení nejsou exaktními optimy, ale pohybují se někde blízko. Konkrétní vzdálenost od optima nemusí být přesně definována, ale měla by být určitým způsobem řiditelná. Před spuštěním algoritmu si tedy uživatel zvolí, jakou chybu je schopný tolerovat a algoritmus mu ji zaručí.

Jeden z nejstarších algoritmů takovéhoto typu je metoda *bisekce*. Klasická numerická metoda pro hledání nulových bodů spojité funkce funguje na principu iterativního omezování intervalu, kde se může nulový bod nalézat. Za předpokladu, že spojitá funkce má právě jeden nulový bod, se dokáže algoritmus postupným prověřováním znamének funkčních hodnot v polovinách uvažovaných intervalů dostat k velice malé chybě řešení. Chyba je přímo řiditelná ukončovací podmínkou, tedy jakmile je délka děleného intervalu kratší než předdefinované δ, tak algoritmus výpočet ukončí. Tato chyba se ale týká pouze maximální vzdálenosti od hledaného bodu v definičním oboru funkce, ale ne funkční hodnoty samotné. Přímočará aplikace metody bisekce na optimalizační problémy může být založená na hledání nulových bodů derivace účelové funkce. Řízení maximální odchylky od optima ve smyslu funkčních hodnot, ale již tak přímočaře nelze dosáhnout a musí být použity jiné metody.

Pro mnoho úloh byly vyvinuty metody, které dokáží nalézt i řešení se zaručenou kvalitou funkční hodnoty. Metody se souhrnně nazývají *aproximační algoritmy* [4] a buď si v nich lze minimální kvalitu předem stanovit, anebo vyplyne z jejich samotného principu. Problém minimálního vrcholového pokrytí hledá v grafu minimální podmnožinu jeho vrcholů, která zaručí každé hraně alespoň jeden přilehlý vrchol. Triviálním přístupem je vybrat vždy jednu nepokrytou hranu a do řešení přidat oba její vrcholy. Takové řešení určitě nemá více než dvojnásobek vrcholů oproti optimálnímu počtu. Říkáme, že algoritmus má konstantní aproximační faktor 2, tedy pro minimalizace platí

$$x^* < f(x) < 2x^*,$$

kde x^* je optimální hodnota a $f(x)$ je výsledek získaný aproximačním algoritmem. Obdobně lze definovat například i meze

$$x^* - c < f(x) < x^* + c,$$

pro $c > 0$. I přes zjevnou užitečnost aproximačních algoritmů jich zatím neexistuje příliš velké množství. Jejich vývoj je totiž časově náročný, obtížný a velmi specifický pro daný typ úloh.

U některých typů úloh lze pozorovat velmi striktní typy omezení, jejichž porušením by mohl vzniknout model s jednodušší strukturou. Jedná se o tzv. *relaxační metody*, které vedou k modelům se snazší řešitelností. Typickým příkladem je *relaxace* celočíselné lineární úlohy. Převedením podmínky celočíselnosti řešení na spojitost lze změnit model na spojitou lineární optimalizaci a k řešení tak využít nějakou běžnou lineární řešící metodu (např. výše zmíněnou

simplexovou metoda). Obdržený výsledek ale velmi pravděpodobně celočíselný nebude a nebude tedy exaktně vyřešen ani původní celočíselný model. Obdržené neceločíselné řešení slouží pouze jako optimistický odhad pro celočíselný problém. Pouze v případě, že by náhodou spojitá metoda vrátila celočíselný výsledek, lze jej považovat i za řešení původní úlohy. Obecně tedy platí, že při relaxaci minimalizační funkce je optimální řešení relaxované úlohy vždy menší nebo rovno než řešení původní úlohy.

Mnoho úloh lze řešit i pomocí pravděpodobnosti za použití (pseudo)náhodných čísel, tedy tzv. pravděpodobnostními algoritmy. Hlavním a nejznámějším příkladem je metoda *Monte Carlo*, která generuje náhodná řešení se stejnou pravděpodobností přes celý stavový prostor a vybírá z nich to nejlepší. Při konečném čase běhu algoritmu nemůže být ale zaručeno, že algoritmus naleze optimum ve stavovém prostoru s nekonečnem bodů. Zaručit to dokonce nelze ani pro prostory s konečným počtem bodů, protože vždy existuje šance, že použitý zdroj náhodných čísel se bude hledanému optimu stále vyhýbat. S dostatečně dlouhým časem ale lze dosáhnout hodnoty pravděpodobnosti nalezení $p = 1$ nebo jí velice blízké. Metoda Monte Carlo má předem definovaný rozsah využitelných zdrojů (čas, počet vyzkoušených bodů apod.), a proto nemůže být zaručeno nalezení optima. Obdobou této metody, ale se zaručeným nalezením řešení, je metoda *Las Vegas*, která ale naopak nemá předem známu dobu svého běhu a může tedy sklouznout až k úplnému projití celého stavového prostoru.

Neexaktní metody

Jako neexaktní metody lze označit takové metody, které nejsou založeny na žádném exaktně ověřitelném postupu a nemohou tak garantovat žádné výsledné vlastnosti nalezených řešení. Takové metody vznikají ze dvou hlavních důvodů. Buď nic přesnějšího pro daný problém neexistuje, anebo by exaktnější způsob řešení vyžadoval příliš mnoho úsilí (peněz), které se zkrátka nevyplatí.

Typickým zástupcem těchto metod jsou *jednoúčelové heuristiky*. V průběhu historie lidé velice často řešili problémy metodami, které vznikaly ze zkušeností předchozích generací a prostě fungovaly. U heuristik není důležité proč a jak fungují, hlavní je, že dokáží dostatečně uspokojivě vyřešit daný problém. Nelze tedy nijak z metody zjistit, jak kvalitní řešení poskytuje, tedy např. jak daleko je řešení od optima. Kvalitu takového řešení posoudí pouze jeho koncový uživatel. Pokud je uživatel spokojen a řešení mu vyhovuje, použije heuristiku nejspíš i příště. Pokud je řešení nedostatečné a nenaplňuje jeho očekávání, tak prostě příště použije jinou.

Příkladem takové heuristiky může být *hladový algoritmus*, který například problém obchodního cestujícího řeší procházením měst v pořadí nejbližších sousedů. Začne tedy v jednom městě, pak se přesune do jemu nejbližšího a tak dále, dokud se nevrátí zpět na počátek. Existují i úlohy, pro které hladový algoritmus zaručuje optimální řešení (např. určení minimální kostry), ale u složitějších úloh (NP-těžká,...) nelze nijak dopředu určit, jak bude výsledek nakonec

kvalitní. Navzdory tomu se ale tyto algoritmy stávají často dostatečným způsobem řešení, a to i díky jednoduché implementaci a vysoké výpočetní rychlosti.

Odlišnou cestou se rozhodly jít metody nazývané souhrnně *softcomputing* metody. Jedná se o několik samostatných algoritmických tříd, které se vyznačují velkou obecností, robustností a hlavně tolerancí k nepřesnostem ve výsledku i zadání. Používají se především pro úlohy tříd NP-úplná a složitějších a jejich snahou je poskytovat dostatečně dobrá řešení v polynomiálním čase.

Velmi populárním zástupcem softcomputingových metod jsou tzv. *neuronové sítě*. Jejich modelem jsou biologické principy objevené při zkoumání fyziologie mozku a procesů myšlení. Lidský mozek je tvořen miliardami jednoduchých neuronových buněk, které jsou vzájemně propojeny a schopny mezi sebou přenášet elektrické impulzy. Každá buňka je napojena na několik sousedních, a pokud jí jsou přes neuronová spojení předávány nějaké impulzy, tak všechny příchozí sečte a pošle dál. Díky nepravidelné topologii buněk vznikají velice složitá zapojení poskytující mozku schopnost vnímání a zpracovávání informací. Umělé neuronové sítě, které zjednodušují biologický model a zprostředkovávají implementační logiku, proto byly primárně vyvíjeny pro rozpoznávací a klasifikační aplikace. Neuronové sítě jsou obvykle v počítačích reprezentovány pomocí několika jednotek až desítek základních buněk uspořádaných do propojeného orientovaného grafu. Obecná buňka (vrchol grafu) má několik vstupních a několik výstupních spojení (hran) s jinými buňkami. Výpočetní postup neuronových sítí spočívá v tom, že na vstupní buňky se přivede definovaný signál (vstupní parametry úlohy), který je následně šířen do celé sítě. Každá buňka převezme hodnoty signálů na svých vstupních spojeních, spočítá jejich lineární kombinaci a předá výsledek svým výstupním spojením. Jakmile signál přejde až do buněk bez přiřazených výstupních spojení, výpočet končí. Výsledkem algoritmu je tedy numerický vektor odpovídající hodnotám na výstupních buňkách. Průběh výpočtu je ovlivňován topologií zapojení buněk a váhami přiřazenými na jednotlivá spojení mezi buňkami. Pro každou úlohu je tak potřeba nelézt nejvhodnější topologii, váhy i systém reprezentace zadání a řešení. I když původní aplikační oblastí neuronových sítí byly klasifikační a aproximační úlohy, tak dnes jsou používaný k celé řadě naprosto odlišných úloh, včetně optimalizace [5].

Mezi další neexaktní metody lze zařadit *fuzzy matematiku*, která se stala oblíbeným nástrojem pro rozšiřování již vytvořených výpočetních postupů. Do kategorie softcomputing se ale většinou zařazuje pouze jedna z jejích speciálních aplikací - *fuzzy logika*. Přístup fuzzy matematiky je založen na práci s obecnými pojmy jako jsou např. "blízko", "daleko", "rychlý", "pomalý" atd. Fuzzy logika na těchto pojmech definuje odvozovací pravidla typu "Když objekt je blízko a pohyb je rychlý, tak čas je krátký". Fuzzy logika je sama o sobě exaktně definovanou matematickou disciplínou, ale co ji dělá *neexaktní*, jsou právě fuzzy pojmy, díky kterým se mohou definovat neexaktní odvozovací pravidla. Složité systémy se tak dají jednoduše popisovat pomocí seznamu intuitivních tvrzení, s nimiž lze systematicky pracovat a na jejich základě odvozovat příslušné výsledky. Fuzzy logiku lze úspěšně použít i pro optimalizační účely, kde fuzzy pravidla popisují cestu ke zlepšování hodnot účelové funkce (viz např. [6]).

Třetím klíčovým tématem softcomputingu jsou *metaheuristické algoritmy*. Zjednodušeně řečeno, metaheuristiky poskytují *rozumný* způsob prohledávání velkých stavových prostorů, a to za významné pomoci hrubé počítačové síly. Metaheuristické metody tvoří hlavní nosné téma této disertace a následující kapitoly se téměř výhradně věnují pouze jim.

Nekonvenční výpočetní zařízení

Popsané metody a algoritmy jsou obvykle navrhovány pro běžné stolní nebo přenosné počítače. Existují ale i jiné cesty, jak získat potřebnou výpočetní kapacitu. Pomineme-li paralelní superpočítače a podobná rozšíření konvenčních strojů[38], tak zajímavých výsledků se dá dosáhnout i za pomocí naprosto odlišných výpočetních paradigmat.

Světelné počítače [7] například dokáží vyřešit grafové úlohy, jako je nalezení Hamiltonovy kružnice nebo problému obchodního cestujícího pomocí vzájemně propojených optických kabelů. Problém obchodního cestujícího je řešen tak, že města jsou reprezentována fyzickými uzlovými body a vzdálenosti mezi nimi jsou reprezentovány poměrně dlouhými optickými kabely. V počátečním městě je vybuzen světelný impuls a v každém uzlovém bodě je světlo unikátně označeno a rozesláno do všech ostatních připojených měst. V počátečním uzlu se pak měří, kdy dorazí první světelný impuls označený všemi uzly.

Dalším nekonvenčním přístupem vytvořeným speciálně k řešení úloh s rozsáhlým stavovým prostorem jsou *DNA počítače* [8] fungující na principu míchání velkých objemů deoxyribonukleové kyseliny (DNA). DNA je biologická makromolekula kódující dědičné informace ve všech živých organizmech a není ničím jiným něž dlouhým lineárním řetězcem základních stavebních prvků (nukletidů). Tyto prvky lze na sebe uměle navazovat podle jednoduchých pravidel a vytvářet tak složitější struktury. Problém obchodního cestujícího tak lze řešit jednou velkou nádobou naplněnou DNA různých typů. Každému městu odpovídá jeden typ DNA a zamícháním tekutiny se na sebe mohou jednotlivé typy navazovat. Při dostatečném promíchání dostatečně velkého objemu DNA je vysoká pravděpodobnost, že vznikne mnoho různých posloupností měst. Pak už stačí jen vyfiltrovat ty DNA, které obsahují každé město pouze jednou a vybrat z nich to nejkratší. DNA počítače fungují na principu ohromné paralelizace, ovšem pro velké úlohy se i tento postup setkává s problémy (hmotnostní bariéra, chybovost výroby DNA, nedostatečné promíchání apod.)

Kvantové počítače jsou částečně hudbou budoucnosti a částečně soudobou skutečností [9]. Kvantová fyzika, na jejímž základě je postaven tento zcela nový přístup k výpočtům, dovoluje ohromující optimalizační možnosti. Na rozdíl od konvenčních počítačů nemají kvantové počítače uložena data ve formě jedniček a nul (bitů), ale místo toho využívají jejich pravděpodobnostní superpozici nazývanou *qbit*. Operace jsou tedy prováděny nad těmito qbity, čímž

[38] počítače s tzv. Von Neumannou architekturou

se dosahuje výpočtu nad všemi jejich možnými realizacemi zároveň. V polovině devadesátých let byl představen Shorův algoritmus [10], který na kvantovém počítači dokáže rozložit zadané číslo na prvočísla v polynomiálním čase (konkrétně potřebuje $\mathcal{O}\left(\log^3 N\right)$). Bylo tak ukázáno, že nejsložitější úlohy pro Turingův stroj (třídy NP apod.) mohou být řešitelné kvantovými výpočty velmi efektivně. V dnešní době je ale technická realizace kvantového počítače ještě vzdálená praktickému využití. Existují sice již komerční firmy dodávající na trh kvantové počítače až s 512 qbity[39], ale odborníci se nemohou stále shodnout, zdali jde opravdu o pravý kvantový počítač a nebo jen určitou náhražku.

2.6 Co z toho plyne?

Jak bylo v této kapitole ukázáno, optimalizace zahrnuje velice širokou paletu různých úloh. Úlohy se mohou navzájem velmi významně lišit, a to jak druhem možných řešení, tak i třeba typem účelové funkce nebo významem optima. Efektivní způsob řešení se některým úlohám podařilo najít velice snadno, ale existuje i mnoho tříd, u kterých dodnes neexistuje ani náznak rychlého způsobu řešení. Mezi takové problematické třídy se řadí například třída NP-úplná nebo třída NP-těžká, u kterých je většina odborníků dokonce přesvědčena o jejich neřešitelnosti (ve smyslu nalezení efektivního řešení).

Potřeba řešit i úlohy z těchto složitějších tříd vedla ke snížení nároků na výsledné řešení. Byly vyvinuty aproximační metody, které pro vybrané typy úloh dokáží zaručit, že nalezené řešení není od přesného optima vzdálenější, než je předem definováno. Jiné způsoby řešení *slevily* například na striktnosti dodržování podmínek úlohy (relaxační metody) a některé pouze vsadily na dostatečně vysoké pravděpodobnosti (např. Monte Carlo).

Samostatnou kapitolou jsou pak metody založené na exaktně nepodložených myšlenkách, u kterých bylo praxí ověřeno, že často fungují, ale nelze to jednoznačně prokázat. Tyto metody jsou buď vytvořeny se záměrem optimalizovat pouze jeden konkrétní problém (heuristiky) a nebo naopak jsou navrženy tak obecně, že přesný druh úlohy u nich nehraje významnější roli (metody umělé inteligence, metaheuristiky). Pro mnoho NP-těžkých (NP-úplných) úloh jsou tyto metody jedinou schůdnou možností, a to i za cenu, že negarantují žádnou kvalitu nalezeného řešení, nebo dokonce negarantují ani to, že nějaké řešení vůbec naleznou.

[39] V červnu 2013 si takový kvantový počítač koupila NASA společně se Google za cca 300 milionů korun.

3 Metaheuristiky

Metaheuristiky jsou třídou výpočetních metod používaných na optimalizační problémy, jejichž řešitelé nevyžadují nutně nalezení optimálního řešení. Pomocí metaheuristických algoritmů lze prohledávat stavový prostor úlohy, avšak bez záruky nalezení dostatečně kvalitního řešení v rozumném čase.

Slovo *metaheuristika* vzniklo spojením dvou původem řeckých slov *meta* a *heuristika*. Předpona *meta* se používá k označení přesahu kořenového pojmu a znamená *něco za* nebo přeneseně i *nad*. Termín *heuristika* (z řeckého *heuriskein* znamenající *objevovat* nebo *vynalézat*) označuje metody, které vznikly pozorováním a jsou ověřeny pouze zkušeností bez exaktní prokazatelnosti.

První použití výrazu *metaheuristika* se přisuzuje Fredovi Gloverovi v článku [11] z roku 1986, kde jím označuje nadvrstvu nad běžnou heuristikou[40]. Jako alternativní název pro stejné metody se občas používal i pojem *moderní heuristiky* (viz [12]), od konce 90.let se ale již téměř výhradně používá široce rozšířeného pojmu *metaheuristika*.

3.1 Definice

Odborná komunita se zatím jednoznačně neshodla na definitorickém vymezení pojmu *metaheuristika*, a to převážně z důvodu, že k vývoji nových metod nebyla zatím v podstatě potřeba. V literatuře lze nalézt několik existujících definic (nebo alespoň zobecněných popisů), které si v mnoha ohledech odpovídají a pokrývají podobné třídy algoritmů, v dílčích detailech se ale liší. Příkladem mohou být následující definice.

- [13]: *Metaheuristika je množina algoritmických konceptů, které mohou být využity k definování heuristických metod aplikovatelných na široké spektrum různých problémů. Jinak řečeno, metaheuristiky mohou být chápány jako zobecněné heuristické metody navržené k určování postupu problémově závislých heuristik (lokální prohledávání, konstrukční heuristiky) směrem k oblastem ve stavovém prostoru obsahující vysoce kvalitní řešení. Metaheuristika je tedy obecný algoritmický rámec aplikovatelný na mnoho různých problémů s relativně malým počtem úprav nutných pro přizpůsobení se konkrétnímu problému.*
- [14]: *Metaheuristika je proces iterativního generování, který řídí podřízenou heuristiku kombinováním rozdílných konceptů pro prohledávání a využívání stavového prostoru pomocí učících se strategií k strukturování informací pro efektivní nalezení téměř optimálního řešení.*
- [15]: *Metaheuristika je iterativní proces, který řídí a mění operace podřízených heuristik, aby efektivně vytvářely kvalitní řešení. Může manipulovat s úplným (nebo neúplným) jednotlivým řešením nebo s celou množinou řešení během každé iterace. Podřízené heuristiky*

[40] "Tabu search may be viewed as a "meta-heuristic" superimposed on another heuristic."

mohou být vysoko (nebo nízko) úrovňové procedury, jednoduchá lokální vyhledávání a nebo jen konstrukční metody.

- [16]: *Metaheuristika je inteligentní iterativní proces, který vykonává prohledávání a může být aplikovaný na optimalizační problémy, jako např. problém obchodního cestujícího.*
- [17]: *Metaheuristický algoritmus může být definován jako vysokoúrovňová obecná metodologie (šablona), která je používána jako návod k odvození heuristik pro řešení specifických optimalizačních problémů.*
- Yang ve své knize [18] podotýká, že žádná obecně uznávaná definice zatím neexistuje a trendem poslední doby je pojmenovávat metaheuristikou jakýkoliv stochastický algoritmus s randomizací a lokálním prohledáváním, čehož se drží ve svých publikacích i on.

Tyto popisy ukazují, že autoři intuitivně chápou pojem *metaheuristika* obdobným způsobem a vyhýbají se příliš svazujícím definicím. Striktní definice je vhodná především pro matematickou práci, ale nemusí být vhodná pro experimentální tvorbu nových algoritmů[41].

Hlavními problémy uvedených popisů je buď přílišná obecnost, nebo naopak vyžadování příliš konkrétní vlastnosti. Metaheuristika ve skutečnosti nemusí být inteligentní[42], samoučící se[43] a nemusí být ani nutně stochastická[44]. Tyto silné předpoklady mohou být z definice vypuštěny, protože zbytečně zužují třídu využitelných metod.

Vyjasnění pojmu

Napříč publikovanou literaturou je pozorovatelná nekonzistence nejen v přesné definici metaheuristiky, ale i v jejím obecnějším pojetí. Tímto pojmem se označují jak vysokoúrovňové koncepty, které dávají jenom velmi zevrubný návod na řešení konkrétních úloh, tak i kompletní algoritmy čekající jen na povel ke spuštění. Mnoho metaheuristik je inspirováno nějakým specifickým přírodním (biologickým) jevem a jsou popsány pouze pomocí terminologie a jevů pozorovaných ve studovaném systému. Jako metaheuristika jsou označovány i metody přesně definované matematickým popisem a fungující na úlohách s definovanými vlastnostmi. Stejně tak lze často číst věty typu "*Vytvořená metaheuristika nalezla řešení našeho problému za 28 sekund ...*", kde je zjevně za metaheuristiku označován spuštěný algoritmus na konkrétní instanci úlohy.

Pokud vyjdeme z významu pojmu *heuristika*, tak její spojení s předponou *meta* by mělo znamenat, že *metaheuristika* je heuristika vytvářející heuristiky. Ani pojem *heuristika* nemá ale

[41] Za striktně definovaný pojem lze označit definici v kapitole 3.3 věnující se No Free Lunch teorému.

[42] *Inteligence* je stejně problematický nedefinovatelný pojem.

[43] *Samoučící* je pojem vzbuzující přílišná očekávání a kladoucí vysoké nároky na práci s informacemi.

[44] Z principu není nutné, aby se algoritmus choval při každém běhu různě.

svou přesnou definici, a proto nemůže mít přesnou definici ani jeho odvozený pojem. Obecně lze ale chápat heuristiku jako smysluplný návod na řešení konkrétní úlohy. Metaheuristika je tedy smysluplný návod na vytváření smysluplných návodů pro řešení konkrétních typů úloh. Nevyjasněnou otázkou ovšem zůstává význam pojmu *konkrétní typ úlohy*. Míra konkretizace není jednoznačná, protože heuristika může dávat slušné výsledky pouze pro jednu konkrétní instanci problému, nebo zvládne řešit instance úlohy s podobnými vlastnostmi (podobné hodnoty parametrů), popřípadě může být i schopna řešit všechny možné instance konkrétní úlohy. Vztáhneme-li tuto úvahu na *metaheuristiky*, není jasné, jestli má metaheuristika poskytovat návody na vytváření heuristik pro úlohy typu *kvadratická optimalizace*, *konvexní optimalizace*, *spojitá optimalizace* nebo *optimalizace obecného zobrazení* $f : X \to Y$. Uvedené příklady jsou postupně svými podtřídami a pro definici metaheuristiky by musela být dána jasná hranice, která by určila, co je příliš obecná úloha pro heuristiku a co naopak příliš konkrétní pro metaheuristiku. Je jasné, že na takové hranici by se odborná komunita těžko jednoznačně shodovala a navíc by ve své podstatě ani nebyla nijak významně užitečná (vzhledem k základní myšlence heuristik, tedy že se exaktně nemusí dokazovat).

Z této rozvahy ale může plynout i rozumnější následující závěr. **Metody, které dokáží přímo řešit libovolnou třídu optimalizačních úloh, jsou označovány jako heuristiky, kdežto pokud je potřeba metody nějak funkčně upravovat, jedná se o metaheuristiky**. *Přímým řešením* je myšlen jasně definovaný funkční koncept, který se dá měnit pouze numerickými (nebo jinak přesně definovanými) parametry. Metaheuristiky ale dávají obecnější popis, takže k přímému řešení musí být ještě dodefinovány jednotlivé funkční bloky (specifické operátory pro daný typ úlohy - např. spojitá vs. kombinatorická optimalizace). Tím je ve výsledku zajištěno, že metaheuristika generuje příslušné heuristiky.

Metaheuristiky tedy mohou být vytvářeny pro široké spektrum úloh, stejně jako pro úzce specializované problémy. Z praktického pohledu je ale vhodnější vytvářet metaheuristiky pro co nejobecnější třídy a klást na ně co nejméně dodatečných podmínek. Metaheuristiky mají často stochastické rysy, ale nejsou tím nijak omezeny. Kvalitní metaheuristiky mohou být jak stochastické tak i deterministické povahy. (Meta)heuristiky obecně nezaručují žádnou kvalitu řešení, ale nic nebrání jejich exaktnímu prozkoumání a stanovení potřebných vlastností. Metaheuristika zůstane metaheuristikou, i když bude dokázána její případná konvergence. Speciálně u NP-těžkých a NP-úplných problémů lze konstruovat metaheuristiky, které při dostatečně dlouhém čase projdou celý stavový prostor, a řešení tedy zaručeně naleznou. Heuristický přístup je ale v tomto případě zakotven v pořadí prohledávaných bodů, protože není v silách žádného dnešního výpočetního zařízení projít celý stavový prostor v rozumném čase. Po uplynutí stanovené doby jsou i konvergentní heuristiky přerušeny a jako výsledek reportují pouze nejlepší nalezené řešení.

Uvedený výklad je založen na porozumění pojmu *metaheuristika* v obecné a jazykovědné rovině. Zvolené pojmenování ale v reálném světě nemusí nutně přesně odpovídat typu metod, které jsou pod něj zařazovány. Jak již bylo řečeno, rozebírané metody se nějakou dobu nazývaly

i *moderní heuristiky* a pojem *metaheuristika* byl vytvořen v brzkých počátcích výzkumu, kdy ještě nebyl příliš jasně vytyčen směr rozvoje celého oboru. Mohlo by se tak zdát vhodnější změnit název na nějaký více jednoznačný, ale v dnešní době je již originální termín tak zaužívaný, že by jakákoliv jeho změna neměla žádnou šanci na prosazení.

V literatuře jsou ale všechny zmíněné koncepty různě zaměňovány a explicitně mezi nimi není rozlišováno. Většinou to nesnižuje srozumitelnost textu, protože je z kontextu zjevné, jestli se jedná o metaheuristiku (podle výše uvedeného popisu) a nebo heuristiku (algoritmus vygenerovaný na základě nějaké metaheuristiky). Tohoto přístupu bude použito i v této disertaci, pokud nebude vyloženě potřeba rozlišovat mezi těmito pojmy, tak budou oba souhrnně a zjednodušeně označovány za metaheuristiky.

3.2 Specializace metaheuristik pro optimalizační úlohy

V kontextu softcomputingu a optimalizačních metod je *metaheuristika* často chápána jako návod jak zkonstruovat algoritmus generující body stavového prostoru pouze na základě předchozích získaných znalostí. Při obecném pohledu na optimalizovanou funkci $f : X \to Y$ to znamená, že pomocí metaheuristiky lze vytvářet algoritmy (heuristiky), které pro danou úlohu dokáží generovat posloupnost bodů $(x, f(x))$ tak, aby se hodnoty $f(x)$ s časem zlepšovaly (ve smyslu optimalizačního kritéria).

Jako základní předpoklad pro úspěšný běh metaheuristiky tedy musí být zajištěno, že nalezené dvojice hodnot $(x, f(x))$ spolu navzájem *nějak korelují*. Taková korelace nemusí existovat globálně na celém stavovém prostoru, ale stačí, když určitý vztah mají mezi sebou body pouze na nějakém lokálním okolí. V případě, že by neexistovala žádná závislost mezi polohou bodu x ve stavovém prostoru a hodnotou jeho účelové funkce $f(x)$, neměla by se metaheuristika podle čeho rozhodovat a generování nových bodů by tedy probíhalo zcela náhodně.

Metaheuristiky tedy obecně nepotřebují žádné apriorní předpoklady o vlastnostech účelové funkce, protože jejich základní schopností je pohyb v neznámém prostoru a vyhledávání tamních nejlepších hodnot. Zjednodušeně lze chování metaheuristik přirovnat k hledání nejvyšší hory v neznámé krajině za naprosté tmy. Algoritmus musí o dalších krocích rozhodovat pouze ze znalosti nadmořských výšek v předchozích navštívených místech. Je zřejmé, že pokud je ráz krajiny předem znám, je větší šance na nalezení efektivnějšího způsobu prohledávání. Některé metaheuristiky tedy mohou pracovat pro určitý typ úloh výrazně lépe než pro jiné, ale dopředu nemusí být nutně úplně jasné, které typy jsou pro danou metaheuristiku vhodné a které nikoliv. A jelikož je vysoce nad schopnosti každého *průzkumníka* pamatovat si všechna navštívená místa, musí být získané informace nějakým způsobem navíc postupně zpracovávány a nepotřebné odstraňovány.

3.3 Nic není zadarmo

Historicky první metaheuristiky se začaly vyvíjet z potřeby reálných aplikací, kde bylo nutné řešit NP-těžké a podobně složité úlohy. V těchto pionýrských dobách byl ale velice mladý[45] i koncept rozdělení úloh do tříd složitosti a nebylo ještě známo mnoho o jejich vlastnostech. Heuristický přístup se tehdy používal převážně z důvodu akutní potřeby nalézt nějaké řešení, kdežto dnes má své opodstatnění založené právě ztrátou důvěry v existenci efektivních algoritmů (pokud P≠NP).

Z počátku se používala na každou úlohu jiná heuristická metoda, ale s postupem času se ukazovalo, že některé z nich lze zobecnit a některé naopak použít i na jiné typy problémů, než bylo původně zamýšleno. Zobecnění heuristik tak započalo vznik metaheuristik. Mezi prvními metaheuristikami (obecnými heuristikami) byl velice přímočarý koncept, známý již minimálně od dob Isaaca Newtona, a to *metoda největšího spádu*. Pro daný bod se prozkoumaly všechny body v jeho okolí (získané malými změnami původního bodu) a následně se vybral nejlepší z nich. Tento princip se opakoval tak dlouho, dokud se nenalezlo k nějakému optimu. Pro zamezení zacyklení byl koncept rozšířen pomocí tzv. *Tabu listu* (viz kapitola 4.4), ale základní princip zůstal stejný. Významná byla ovšem abstrakce řešícího algoritmu, tedy oddělení způsobu řešení od samotného významu problému. Ve stejné době vznikají i genetické algoritmy (viz kapitola 4.1), jejichž abstrakce šla ještě dále. Princip prohledávání stavového prostoru se změnil na rekombinaci vektorů řešení bez ohledu na jejich geometrickou blízkost. V genetických algoritmech bylo řešení reprezentováno pouze v binární soustavě, a tedy zkombinováním dvou takto zakódovaných řešení lze získat nové, které geometricky vůbec s původními nesouvisí. Vysoká abstrakce ve způsobu procházení stavového prostoru dala vzniknout novým metodám, jako např. *Ant colony optimization* a dalším (viz kapitola 4), které díky evolucí *prověřeným* přístupům slibovaly snadné vyřešení libovolné úlohy.

V polovině 90.let 20.století ale přišlo rychlé vystřízlivění v podobě *No Free Lunch teorému*. Tato exaktně prokázaná věta říká, že žádný algoritmus nemůže efektivně vyřešit všechny problémy. Nelze tedy zkonstruovat heuristiku, která by dokázala optimalizovat všechny účelové funkce lépe než kterákoliv jiná. Matematicky je toto tvrzení specifikováno jako

$$\sum_f P(d_m^y | f, m, a_1) = \sum_f P(d_m^y | f, m, a_2),$$

tedy pravděpodobnost získání konkrétní posloupnosti hodnot pro všechny účelové funkce je stejná při použití libovolného algoritmu. Účelové funkce f jsou reprezentovány jako zobrazení $f : X \to Y$, kde X i Y jsou konečné diskrétní množiny[46]. Uvažované algoritmy a_i pracují tak, že ze stavového prostoru postupně vybírají body x_i a zjišťují pro ně hodnoty $f(x_i)$. Vzniká

[45] První úlohy patřící do třídy NP-úplná byly objeveny až v roce 1972 [19].

[46] Každá hodnota reprezentovaná v počítači musí být nutně konečná, a proto není potřeba uvažovat nekonečné množiny.

tak posloupnost $d_m^y = \langle f(x_1), f(x_2), \ldots, f(x_m) \rangle$, jejíž pravděpodobnost P při uvažování všech účelových funkcí nezávisí na použitém algoritmu.

Interpretace No Free Lunch teorému vede v důsledku k tomu, že každý sebesofistikovanější algoritmus je stejně efektivní jako slepé náhodné prohledávání. Pro metaheuristiky by to fakticky znamenalo ukončení vývoje, ale po důkladných analýzách vyplynulo, že existuje mnoho tříd úloh, kde No Free Lunch neplatí. Klíč k porozumění těmto třídám leží v pojmu *uzavření vůči permutaci*. Vzhledem k tomu, že každá počítačově implementovaná účelová funkce je zobrazením mezi diskrétními a konečnými množinami, tak se v podstatě jedná a konečný vektor dvojic (x_i, y_i). O třídě funkcí F řekneme, že je uzavřená vůči permutaci, jestliže lze každé její funkci $f = \langle (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \rangle$ libovolně proházet (permutovat) hodnoty x_i, a i přesto bude stále náležet do F. Jinými slovy, pokud algoritmus určí hodnoty funkce f v několika bodech, tedy sestaví posloupnost $\langle (x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k) \rangle$, pak na jejím základě nelze pro bod x_{k+1} odhadnout, jestli její hodnota $f(x_{k+1})$ bude lepší nebo horší. Všechny možnosti jsou totiž stejně pravděpodobné a tím pádem všechny algoritmy budou mít na F stejnou průměrnou efektivitu. Pokud ale třída F není vůči permutaci uzavřená, pak existuje možnost sestavit algoritmus, který využívá znalosti o *chybějících* permutacích a podle toho generuje body x_{k+1} s vyšší pravděpodobností na lepší hodnotu účelové funkce. Bylo tedy dokázáno [20], že No Free Lunch teorém platí tehdy a jen tehdy, když je uvažovaná množina účelových funkcí uzavřená vzhledem k permutaci.

Vývoj nových metaheuristických algoritmů se tedy nemusel zastavit, protože brzy bylo nalezeno několik konkrétních tříd, kde No Free lunch neplatí [21]. U tříd praktických úloh spojených s reálným světem je málo pravděpodobné, že budou uzavřené vůči permutaci, a proto nebyl problém začít konstruovat metaheuristiky s dobrou efektivitou na vybraných typech problémů. V teoretické rovině pak bylo ukázáno, že počet podmnožin množiny všech účelových funkcí, které jsou uzavřeny vůči permutaci, je zanedbatelně malý [22].

Pokud je zkoumaná třída funkcí uzavřená vůči permutaci, potom je určitě nevhodná pro řešení pomocí metaheuristických algoritmů. Opak nemusí automaticky platit, protože ještě záleží, jestli třída neobsahuje takovou podmnožinu v sobě. Přirozeně by vznikal obdobný problém, jako v uzavřenosti celé třídy. Popřípadě si lze představit uzavřenou třídu účelových funkcí, z nichž jednu funkci odebereme. Taková třída už nadále uzavřená vůči permutaci není, ale efektivita algoritmů se prakticky nezmění. Obecně lze říci, že metaheuristiky mohou získávat tím vyšší efektivitu, čím více je potřeba doplnit (odebrat) funkcí, aby třída byla uzavřená vůči permutaci.

Dalšími nevhodnými funkcemi pro metaheuristiky jsou takové, které mají velké oblasti konstantních hodnot. Například funkce, kde všechny hodnoty jsou rovny nule a pouze jedna je rovna odlišné hodnotě, prakticky nejde optimalizovat. Neexistuje totiž závislost mezi hodnotou v optimu a hodnotami v ostatních bodech. Ze stejného důvodu by byla nevhodná i funkce s náhodnými (ale pevnými) hodnotami.

Jelikož metaheuristiky zakládají veškeré své rozhodnutí na informacích získaných z vyhodnocených bodů stavového prostoru, je přirozeně nejvhodnější mít těchto bodů k dispozici co nejvíce. Každá úloha z třídy NP musí mít k dispozici ověřovací algoritmus třídy P. Tyto úlohy tedy mají potenciál efektivně[47] poskytovat informace právě díky svým ověřovacím algoritmům (přeneseně mohou mít takový potenciál i úlohy třídy NP-těžká.) Naopak úlohy, jejichž vyhodnocení je velmi časově náročné, nejsou pro metaheuristiky příliš vhodné. Jak bude ukázáno v kapitole 9.4, i takové úlohy jsou ale s vhodným přístupem řešitelné.

3.4 Druhy metaheuristických algoritmů

Jelikož metaheuristiky ani heuristiky nejsou exaktně odvozovány ze základních principů, bývá jejich činnost založena na inspiraci známými procesy. Zdaleka nejčastějším zdrojem inspirace při vývoji nových metaheuristik je živá příroda. Autoři těchto algoritmů se odvolávají na evoluční principy, na genetické zákonitosti, na chování mravenců, ptáků, včel i netopýrů. Metaheuristiky lze tedy rozlišovat na základě druhu inspirace, ale toto rozdělení nepřináší kvalitativní rozdělení z pohledu chování algoritmů. Podstatnějším znakem z hlediska výkonnosti a chování algoritmu je, jestli se jedná o *populační* nebo *jednoduché* prohledávání. Při jednoduchém (*single based*) je v paměti udrženo pouze jedno hlavní řešení, kdežto u populačních algoritmů je jich udržována celá skupina a nová řešení jsou generována na základě její kombinace. Velmi významnými jsou i velikosti oblastí, které jsou prohledávány. Při *lokálních* typech algoritmů jsou nová řešení generována v blízkosti původních, naopak při *globálních* metodách je možné nové řešení generovat od původní skupiny velmi daleko. Další důležité rozdíly mezi metaheuristikami jsou shrnuty v kapitole 5.

Pro zlepšení výkonu metaheuristiky na konkrétních typech úloh se zkouší mnoho různých postupů. Kromě základního nastavování řídicích parametrů a jednoduchých úprav dílčích částí metaheuristiky lze použít i metody pro kombinování různých metaheuristických konceptů dohromady. Příkladem jsou tzv. *paralelní metaheuristiky*, kde se spustí více metaheuristik pro stejný problém současně a v určitých okamžicích si vyměňují informace. Tyto spolupracující metaheuristiky mohou být stejných typů, ale lze spolu kombinovat i jejich různé varianty, nebo dokonce zcela odlišné typy. Pokud jsou spuštěny dvě instance, existuje pouze jeden způsob jejich vzájemného propojení (z topologického pohledu). Pokud jich je ale propojeno více dohromady, potom mohou vznikat různé topologie s odlišnými vlivy na celkové chování algoritmu. Šíře variability topologií počíná výměnou informací každého dílčího celku s každým přes hierarchické pospojování až po zcela obecné nebo náhodné spojení. Jednotlivé samostatné metaheuristiky se často nazývají *ostrovy*, čímž se zdůrazňuje nezávislost jejich prohledávání na ostatních dílčích celcích. Ostrovy si mezi sebou mohou vyměňovat jak nejlepší nalezená řešení, tak i např. náhodné skupinky řešení. Vhodným propojením několika metaheuristik se tady nedocílí pouze

[47] Efektivně v tomto případě znamená, že výpočet patří do třídy P.

urychlení výpočtu a větší šíře prohledávané oblasti, ale vznikají tak zcela nové vzorce chování, které mohou napomoci k efektivnějšímu řešení zadaných úloh (více viz [23]).

Druhým významným způsobem kombinování různých metaheuristik je tzv. *hybridizace*. Jedná se o spojení několika různých konceptů (nebo jen jejich dílčích částí) do jednoho funkčního celku.[48] Hybridizace metaheuristik nemusí probíhat pouze spojením s odlišnou metaheuristikou, ale může být založena i na spojení s naprosto odlišným konceptem. Často jsou metaheuristiky spojovány s exaktními algoritmy, jako je například lineární nebo dynamické programování, ale takový způsob hybridizace vyžaduje apriorní znalost typu řešené úlohy a není tedy problémově nezávislý. Algoritmy tohoto typu jsou často velice efektivní, díky čemuž získávají značnou popularitu. Jejich oblibu dokazuje nejen fakt, že jim jsou věnovány samostatné konference, ale získaly dokonce i vlastní pojmenování *matheuristiky* [24]. Hybridizace je často prováděna i s dalšími nástroji soft computing, jako jsou neuronové sítě nebo fuzzy matematika. Tyto a další nástroje z nejrůznějších oblastí výzkumu (umělá inteligence, strojové učení, operační výzkum, statistika, ...) nahrazují nebo rozšiřují některé ze základních částí běžných metaheuristik, popř. mohou sloužit i pro řízení algoritmu nebo komunikaci dílčích podčástí.

Hybridizace se často rozlišuje podle úrovně spolupráce na tzv. *slabou* a *silnou* v závislosti na zachované integritě. Pokud kombinací několika konceptů vznikne úplně nový algoritmus, ve kterém lze spatřit už jen náznaky původních konceptů, je hybridizace považována za silnou. Naopak slabou hybridizaci lze dosáhnout spíše vnějším propojením základních konceptů, kde celistvost původních algoritmů zůstane zachována v nezměněné podobě. Protože hybridizace je z principu velmi variabilní koncept, bylo vyvinuto několik způsobů klasifikace takto vzniklých algoritmů. Jednotný pohled na hybridizaci a její rozdělení do tříd lze nalézt v přehledném shrnutí v [25]. Hybridizace je jedno z nosných témat této disertační práce a v následujících kapitolách jí je věnován ještě další prostor.

Kromě značné variability v koncepčních přístupech jednotlivých metaheuristik, lze mnoho odlišností spatřit i v implementačních detailech. Nejvýznamnějším důvodem rozdílností při implementaci je odlišná architektura výpočetních zařízení poskytující metaheuristikám běhové prostředí. Zdaleka nejběžnější je spouštění metaheursitky na klasickém *procesoru stolního počítače*. Procesor zde sériově vykonává definované instrukce a podle potřeby zapisuje nebo čte z přístupné paměti. Výrobci pravidelně zvyšovali výkon procesorů a nebyl tedy důvod k významným obměnám základních metaheuristických principů. Posledních pár let ale zvyšování výkonu procesorů významně zpomalilo a místo toho se začaly prosazovat vícejádrové procesory. Každý takový procesor zvládá několik paralelních výpočtů zároveň a znatelně tak může urychlit průběh výpočtu. Metaheuristiky jsou ze své povahy velice dobře hardwarově paralelizovatelné na úrovni vyhodnocování účelové funkce. Jádra vícejádrových procesorů ale většinou nemají možnost nezávisle přistupovat k paměti, a proto musí být metaheuristiky pečlivě navrhovány tak, aby se jádra vzájemně neomezovala a aby se předešlo možným bezpečnostním problémům

[48] Výše zmíněnou paralelizaci lze podle uvedeného popisu považovat za speciální případ hybridizace.

[26]. Jiným směrem se rozhodli jít výrobci grafických karet, kde je mutliprocesorový přístup základním rysem jejich činnosti. Grafické karty vykonávají desítky až stovky identických úkonů současně a jejich celkový výkon se rychle začal vyrovnávat běžným procesorům. Největší výrobci začali ke svým kartám poskytovat vývojové nástroje[49], které dovolovaly jejich potenciál naplno využít. Architektura grafických karet má ale mnoho omezení a vyvíjené metaheuristiky se jim musely silně přizpůsobovat[50]. Při úspěšném návrhu byly ale výsledky aplikovaných metaheuristik na grafických kartách velice slibné [27].

Metaheuristiky nemusí být spouštěny pouze na jediném zařízení, ale jsou velmi vhodné i pro současný běh na více počítačích zároveň. Velmi efektivní je *distribuované řešení*, kde je propojeno libovolné množství nezávislých počítačů komunikujících podle stanoveného protokolu. Na každém počítači tak může být spuštěna samostatná metaheuristika, nebo může být využit pouze pro dílčí části výpočtu. Spojené počítače nemusí mít stejný výkon ani stejný operační systém, pouze musí vědět, kdy, kam a co zaslat. Příkladem je opensource systém Boinc, který umožňuje propojení počítačů přes internet a díky kterému lze spouštět metaheuristiky na stovkách ([28]) či tisících počítačích zároveň.

Velmi moderním trendem v informačních systémech je tzv. *cloud computing*. Podobně jako u distribuovaných výpočtů je i zde propojeno velké množství nezávislých počítačů, avšak s tím rozdílem, že každý pouze shromažďuje a zpřístupňuje data. Řídicí logika je postavena na základním přístupu odvozeného z funkcionálního programování a pojmenovaného *MapReduce*[51]. Pouze pomocí dvou základních funkcí `map` a `reduce`, lze velice efektivně implementovat řadu náročných algoritmů. Jedny z hlavních výhod jsou vysoká škálovatelnost algoritmu a rychlé zpracování enormních objemů dat [29]. Úspěšně implementovány byly tímto přístupem např. genetické algoritmy [30], diferenciální evoluce [31] a další.

Jak je z uvedeného souhrnu zřejmé, ke zvýšení výkonu metaheuristik se lze dostat několika odlišnými cestami. Od vývoje nových základních konceptů přes jejich různé kombinace a hybridizace až po využívání nejmodernějších hardwarových nástrojů. V některých případech se lze obejít i bez klasického počítače a vytvořit si vlastní hardware přímo na míru[52] nebo lze pro vyhodnocování účelové funkce využít přímo i lidský mozek (avšak ne pro zvýšení rychlosti výpočtu, ale spíše pro jeho zatím nezastupitelné schopnosti v rozpoznávání a klasifikaci obrazových dat)[53]. Hlavním pilířem všech uvedených tříd jsou ale základní metaheuristické koncepty a v následujících kapitolách bude několik takových typů představeno a následně i zobecněno.

---

[49] Nvidia vytvořila nový nástroj CUDA a firma AMD nástroj OpenCL.

[50] Grafické karty používaly pouze jednoduchou přesnost čísel a všechny procesory musely ve stejný okamžik vykonávat naprosto stejnou instrukci.

[51] Přístup byl vyvinut firmou Google ke zpracování obrovských objemů dat.

[52] Genetický algoritmus byl realizován na FPGA, tedy na programovatelných klopných obvodech [32]

[53] Např. projekt *picbreeder.org* provozuje metaheuristiku vyvíjející obrazy, kde je jejich průběžné ohodnocování přenecháno na lidské úvaze.

4 Metaheuristické algoritmy

Následující odstavce popisují vybrané metaheuristiky a ukazují jejich základní princip a vlastnosti. Jak již bylo uvedeno, metaheuristika nemusí být nutně definována jako algoritmus a nemusí mít nutně přesně definovaný sled operací. V následujícím textu tedy nejsou uvedeny jejich pseudokódy, i přestože některé ze zmíněných publikací je pro lepší srozumitelnost uvádějí. Smyslem této kapitoly je ukázat šíři jednotlivých konceptů a možnosti, které dnešní metaheuristiky nabízí a pro detailní popis algoritmů je tedy vhodné prostudovat originální publikace. Při řešení reálných úloh může být každý koncept libovolně obměňován, dokud nebude dávat očekávané výsledky. Není proto nutné zabývat se přílišnými detaily, ale spíše smyslem, inspirací a přístupem jednotlivých konceptů.

4.1 Genetic algorithms[54]

Jednou z nejznámějších metaheuristik je genetický algoritmus, který napodobuje procesy přírodního výběru a využívá základní principy biologického rozmnožování. Vznik se datuje do sedmdesátých let dvacátého století, kdy ji ve své publikaci zpopularizoval J. Holland [33]. Díky získané popularitě vzniklo mnoho rozličných modifikací, ale zároveň se zapříčinila i neexistence jednotné definice. Mnoho autorů si tak definici přizpůsobuje svým potřebám a názorům, ale i přesto základní charakteristiky obvykle zůstávají stejné:

Algoritmus udržuje populaci jedinců, kde každý reprezentuje jedno z možných řešení. Častou reprezentací je zakódování souřadnic řešení do dvojkové soustavy a jejich spojení v jeden delší řetězec symbolizující DNA[55]. V každé iteraci je z populace vybráno několik dvojic, aby se staly rodiči nových jedinců a rozšířily tak své geny. Zkřížením rodičů z takovéto dvojice vznikne nové řešení, které je následně ohodnoceno a pokud má dostatečnou kvalitu, může se připojit k populaci a produkovat vlastní potomky. Tento princip simuluje přirozený evoluční výběr, protože dvojice ke zkřížení jsou vybírány s ohledem na jejich kvalitu a čím vyšší kvalitu jedinec má, tím vyšší je i jeho šance na předání svých genů do následující generace. Selekční tlak nedovoluje dlouhodobě přežívat slabším jedincům. Pokud nové řešení, nebo některé ze stávající populace, má příliš nízkou kvalitu, nevratně je odstraněno.

Algoritmus tedy iterativně vylepšuje náhodně vygenerovanou populaci postupným kombinování nejslibnějších jedinců a uchováváním pouze těch nejlepších vyprodukovaných potomků. Jednotlivé implementace genetických algoritmů se od sebe liší způsobem reprezentace řešení,

[54] Jména algoritmů jsou ponechána v originálním anglickém jazyce, protože u většiny z nich není český překlad ještě příliš zaužíván a asi ani nikdy nebude. Navíc původní jména bez českých překladů lépe slouží i k vyhledávání navazujících informacích na internetu a v odborných publikacích.

[55] Deoxyribonukleová kyselina - je nukleová kyselina nesoucí genetickou informaci každého živého organismu

výběrem rodičů i jejich křížením. Původní implementace byly navrženy pro řešení kombinatorických problémů, ale jejich úspěch dal rychle vzniknout i verzím pro řešení spojitých [34] a mnohých dalších tříd optimalizace.

4.2 Particle swarm algorithm

Česky je tento algoritmus obvykle nazýván jako *hejnový algoritmus*, protože jeho základní princip je odvozen od chování pozorovaného na hejnech zvířat v přírodě. Ať už se jedná o ryby, ptáky nebo savce, jejich skupinový pohyb se dá modelovat pomocí několika jednoduchých rovnic. Algoritmus udržuje v paměti množinu samostatných řešení, která se pohybují po stavovém prostoru a vzájemně se ovlivňují. Celá množina reprezentuje jedno společné hejno a každé samostatné řešení reprezentuje jedno zvíře v něm. Iterace algoritmu způsobují přesun jedinců na nové pozice podle pozic ostatních. Základní koncept je definován pro jedince j v iteraci i pomocí jeho pozice x a rychlosti v jako

$$v_j^i = \chi(\omega v_j^{i-1} + c_1 r_1 (G^{i-1} - x_j^{i-1}) + c_2 r_2 (L_j^{i-1} - x_j^{i-1}))$$
$$x_j^i = x_j^{i-1} + v_j^i.$$

Jedincům se tedy v každé iteraci spočítá nová rychlost (vektor) a podle ní se přesunou na novou pozici. Velikost rychlosti je určována nastavitelnými parametry χ, ω, c_1, c_2 a náhodnými parametry r_1, r_2. Směr je určován pomocí polohy nejlepšího jednice v populaci a zároveň i nejlepší pozice, které jedinec j již dosáhl. Prvek G tedy znamená globálně nalezené optimum, L lokální optimum, které je pro každého jedince obecně různé. Celé hejno se tak pohybuje směrem k nejlepšímu jedinci (nejlepší ve smyslu hodnoty účelové funkce) a společně prohledává stavový prostor. Tento přístup byl pro metaheuristickou optimalizaci použit poprvé v [35] a od té doby se stal jedním z nejoblíbenějších schémat. Na jeho základě vzniká mnoho odvozených metaheuristik s nepřeberným množstvím aplikací. Např. v [36] je shrnuto přes 500 publikací definujících rozšiřující postupy pro vylepšení vlastností hejnových algoritmů.

4.3 Artificial immune systems

Umělými imunitními systémy je nazývána třída algoritmů inspirovaných procesy biologických imunitních systémů. Nejedná se jen o metaheuristky, ale značné aplikace jsou i v klasifikaci/shlukování, detekci anomálie a samoučích se algoritmů [37]. Jako metaheuristiky se nejčastěji používají dvě základní paradigmata [38].

- *CLONALG: Clonal selection algorithm*[56] [39] je název algoritmu, který je inspirován obrannou strategií imunitního systému při napadení cizími buňkami. V krevním řečišti se pohybují

[56] algoritmus klonální selekce

buňky imunitního systému nazývané B-lymfocity, které se při detekci antigenů (cizích buněk) začnou množit a postupně je likvidovat. Každý lymfocit dokáže detekovat a zahubit pouze jeden druh antigenu, a proto se množí pouze ty, které detekci zvládly nejlépe. Čím lepší detekce (rozpoznání útočné buňky), tím vícekrát se lymfocit namnoží. Rozmnožování probíhá klonováním, a aby systém lépe odolával široké paletě proměnlivých antigenů, jsou lymfocity zároveň i mutovány. Mutace je nepřímo úměrná schopnosti detekce, tedy čím hůře byla buňka schopna identifikovat antigen, tím více bude mutována. Tento jednoduchý princip zajišťuje dostatečnou kapacitu imunitního systému pro boj se známými i neznámými antigeny.

Metaheuristika založená na přístupu klonální selekce nahrazuje schopnost detekce antigenu hodnotou účelové funkce a jednotlivá prověřovaná řešení zastupují lymfocity. Čím vyšší hodnota účelové funkce, tím více klonů daného řešení vznikne a zároveň tím menší mutaci jsou vystaveni. Populace lymfocitů má konstantní velikost a zachovávají se vždy jen nejlepší nalezená řešení. Pro zabránění předčasné konvergence a udržování dostatečné variability jsou navíc nejhorší uchovávané lymfocity odstraněny a nahrazeny novými náhodně vygenerovanými řešeními.

Počet klonů n daného řešení k se řídí pravidlem

$$n_k = \left\lfloor \frac{\beta \cdot N}{i_{(k)}} + 0.5 \right\rfloor,$$

kde N je celkový počet uchovávaných řešení, β je parametr klonování (obvykle $\beta \in (0, 1]$) a $i_{(k)} \in [1, n]$ je pořadí řešení v seřazené množině aktuální populace podle kvality hodnoty účelové funkce.

- *opt-AiNET: Immune network algorithm* [40] má několik základních znaků společných s předchozím algoritmem klonální selekce, ale navíc jej rozšiřuje ještě o další operátory podporující robustnější chování. Hlavním charakteristickým rysem je snaha zabránit předčasné konvergenci populace do jediného shluku. Každých n iterací se určí průměrná hodnota velikosti účelové funkce v aktuální populaci $\bar{f}_i$ a porovná se s předchozí uloženou $\bar{f}_{i-1}$. Velikost změny se počítá následujícím vzorcem a porovnává s předem definovanou hodnotou (nejčastěji 0.001).

$$\left| 1 - \frac{\bar{f}_i}{\bar{f}_{i-1}} \right| < 0.001$$

Pokud je nerovnost splněna, je množina aktuálních řešení redukována odstraněním geometricky blízkých prvků. Pro každou dvojici je spočítána vzdálenost (eukleidovská metrika) a pokud je její velikost menší než daná hodnota σ, je řešení s horší hodnotou účelové funkce odstraněno.

Dalším důležitým rysem opt-AiNET pro zabránění shlukování je způsob výběru uchovávaných řešení. Generování nových řešení se provádí stejným principem jako v algoritmu

CLONALG, tedy proporčním klonováním a mutací. Uchováváno je však pouze jediné nejlepší řešení vzniklé z daného rodiče a zbytek je odstraněn. Celkově je tedy tato metaheuristika vhodná pro multimodální účelové funkce, kde je velmi žádoucí zamezit vytváření jednoho velkého shluku řešení.

4.4 Hill Climbing, Simulated annealing, Tabu search

Hill climbing algoritmus, neboli horolezecký algoritmus, je základním algoritmem metaheuristické optimalizace. Z názvu lze odtušit, že se algoritmus pohybuje stavovým prostorem stejně jako horolezci po skalách. Horolezec je zde zastupován aktuálním bodem hledajícím nejvyšší hodnotu účelové funkce[57]. V každý okamžik se algoritmus rozhlédne po svém nejbližším okolí a vybere si bod s nejvyšší účelovou funkcí, do kterého se přesune. Opakováním tohoto úkonu se algoritmus dostane až do místa, kde žádný bod v okolí nemá vyšší účelovou funkci a tento bod prohlásí za výsledek optimalizace a skončí. Je zřejmé, že algoritmus může zaručit nalézt pouze lokální optimum. V případě že se jedná o konvexní funkci, zaručí se tím i nalezení ke globálnímu optimu. Kanonická verze algoritmu v každé iteraci akceptuje nejlepší řešení z aktuálně prozkoumaného okolí, bez ohledu na hodnotu účelové funkce v aktuálním bodě. Může se tedy stát, že algoritmus své aktuální řešení i zhorší, a je tedy velmi vhodné, průběžně ukládat nejlepší dosažené řešení.

Jednoduchým, ale efektivním rozšířením horolezeckého algoritmu je metoda nazvaná *Simulated annealing*. Jelikož základní horolezecký algoritmus nezvládne z principu opustit okolí nalezeného lokálního extrému, byla mu přidána možnost krátkodobě akceptovat i řešení s nižší hodnotou účelové funkce, než má v aktuální pozici. Hlavním nástrojem pro řízení přesunu bodu je tzv. *teplota systému*, která nepřímo udává pravděpodobnost změny pozice k horšímu. Na počátku algoritmu je teplota T nastavena na hodnotu T_0 a s každou novou iterací je teplota snížena konstantním multiplikativním koeficientem $\alpha \in (0, 1)$. Pokud je v okolí aktuálního bodu vybrán bod x_1 s nižší hodnotou účelové funkce, je o přesunu rozhodnuto pomocí pravděpodobností funkce $P(f(x), f(x_1), T)$ závislé na hodnotách účelových funkcí v porovnávaných bodech a na aktuální teplotě T. Funkce P má většinou exponenciální klesající tvar vzhledem k teplotě a hodnoty v intervalu $(0, 1)$. Pokud je náhodně vygenerované číslo (ve stejném intervalu) menší než spočítaná hodnota funkce P, přechod se do *horšího* bodu uskuteční. Tímto způsobem je dosaženo toho, že na počátku algoritmus může stochasticky prohledávat i oblasti, kam se běžný horolezecký algoritmus nedostane a s přibývajícím časem má naopak tendenci se mu v chování přibližovat.

Algoritmus byl poprvé popsán dvěma na sobě nezávislými autory ([41] a [42]) a dnes existuje nespočet jeho variant. Jméno získal z metody žíhání kovů, kde při zahřátí atomy základních

[57] Parafráze platí pouze v případě maximalizace účelové funkce. V opačném případě by horolezec hledal cestu z vrcholu zpět dolů.

prvků získají vyšší energii a mohou tak vykonávat poměrně velké skoky v materiálu. S přibývajícím časem a snižující se teplotou mohou atomy vykovávat čím dal menší skoky až postupně skončí ve svých rovnovážných polohách. Stejný princip, tedy velké skoky na začátku a malé na konci, je také často považován za hlavní kanonickou formu simulovaného žíhání.

Odlišným způsobem rozšíření konceptu horolezeckého algoritmu je velmi oblíbená metoda nazvaná *Tabu search*. Základní princip prohledávání stavového prostoru je stejný, jen po vybrání nejlepšího řešení z aktuálního okolí se nejprve zkontroluje, jestli již nebylo dříve navštíveno. Algoritmus udržuje seznam posledních n navštívených bodů, tzv. *tabu list*, jímž zabraňuje opakovanému navštívení stejných bodů. Záleží na konkrétní povaze algoritmu, jak tabu list reprezentuje ony navštívené body. Lze využívat například kompletní souřadnice ve stavovém prostoru nebo jen posloupnost akceptovaných transformací, které postupně vytvořily aktuální řešení. V druhém případě se přístup kombinuje s tzv. *akceptačním kritériem*, které povoluje transformace nalézající se v tabu listu, pokud vedou ke zlepšení hodnoty účelové funkce. Důsledkem využívání tabu listu je, že se algoritmus nezacyklí a prohledá větší oblast stavového prostoru. Čím delší seznam (čím větší n), tím delší cykly jsou vyloučeny.

4.5 Harmony search

Tvůrci algoritmu Harmony search se při jeho návrhu nechali inspirovat procesem improvizace jazzových muzikantů a jejich způsobem skládání samostatných tónů v jeden harmonický celek. Poprvé byla tato metaheuristika publikována v roce 2001 [43] a od té doby se stala poměrně populární a často aplikovanou metodou [44].

Inspirace procesem hudební improvizace k prohledávání stavového prostoru je postavena na paralelách mezi hodnotami účelové funkce s estetickým dojmem složené hudby; globálního optima s dokonalým hudebním dílem; počtem a rozsahem proměnných s počtem a rozsahem hudebních nástrojů. Každá iterace algoritmu je pak přirovnávána k procvičování hudebníka až k dosažení dokonalé kompozice a aktuální skupina řešení je považována za množinu zajímavých harmonií.

Algoritmus na začátku generuje a ohodnotí nevelký počet (řádově jednotky) náhodných řešení problému a nová jsou pak skládána z každého z nich. Hodnota proměnných nového řešení je zkopírována z některého náhodně (rovnoměrně) vybraného řešení. Pro zajištění prohledávání i nových oblastí stavového prostoru se s nízkou pravděpodobností může mutovat každá z vytvořených proměnných. Mutace jsou dvojího druhu, buď náhodná výměna proměnné za naprosto novou hodnotu, nebo pouze její malá změna (přirovnávaná ke změně noty o jeden půltón). Pokud je nově zkonstruované řešení lepší než nějaké z rodičovských prvků, tak jej algoritmus zahrne do populace a nahradí jím nejslabšího jedince.

4.6 Firefly algorithm

Firefly algoritmus je inspirován hejnovým chováním světlušek a jejich způsobem přitahování partnerů pro spáření [45]. Algoritmus využívá zjednodušený model, kde jsou všechny světlušky unisexuální a mohou tedy být všechny přitahovány k sobě navzájem. Nejatraktivnějším jedincem je nejvíce svítící světluška a všechny světlušky se navzájem přitahují proporcionálně ke své světelné intenzitě. V metaheuristické paralele zastupuje svítivost hodnota účelové funkce a tedy čím lepší hodnota, tím více světluška svítí.

Algoritmus po náhodném vygenerování hejna světlušek vyhodnotí pro každou dvojici jejich vzájemnou atraktivitu a méně svítivá světluška se přesune směrem k více svítivé (atraktivnější). Konkrétní hodnota vzájemné atraktivity se vyhodnocuje v závislosti na vzdálenosti mezi světluškami. S rostoucí vzdáleností klesá svítivost každé světlušky a tím i zdánlivá atraktivita. Posun méně svítivé světlušky směrem k atraktivnějšímu partnerovi je realizován vztahem:

$$x_i^{t+1} = x_i^t + \beta_0 \exp -\gamma r_{ij}^2 (x_j^t - x_i^t) + \alpha(rand_{[0,1]} - 0.5),$$

kde x_i^k jsou souřadnice světlušky v k-té iteraci a x_j jsou souřadnice svítivějšího protějšku. Vzdálenost r_{ij} je měřena euklidovskou metrikou a je využita pro výpočet exponenciálního poklesu zdánlivé atraktivnosti. Výpočet je ovlivněn třemi nastavitelnými parametry (β_0,γ,α) a k celému výsledku je nakonec přidán šum náhodným číslem z intervalu $[-\alpha, \alpha]$. Algoritmus iteračně provádí přiblížení pro všechny dvojice světlušek a jakmile je dosažený předem stanovený počet iterací, algoritmus končí.

4.7 Differential evolution

Diferenciální evoluce je populární metaheuristický algoritmus primárně určený pro spojitou optimalizaci. První publikace se dočkala v [46] a i když od té doby procházela různými dílčími proměnami [47], základní schéma zůstalo stejné.

Stejně jako ostatní evoluční algoritmy, tak i diferenciální evoluce udržuje populaci samostatných řešení, ze kterých produkuje nové. Počáteční populace je generována náhodně a většinou obsahuje několik desítek jedinců. Pro vytvoření nového jedince je použita speciální metoda mutace, kde se vybraný prvek kombinuje s třemi dalšími. Pro každý prvek $\mathbf{x_0}$ z aktuální generace se vyberou náhodně tři rozdílní jedinci $\mathbf{x}_1$,$\mathbf{x}_2$, $\mathbf{x}_3$ a vytvoří nový vektor

$$\mathbf{v}_0 = \mathbf{x}_1 + r \cdot (\mathbf{x}_2 - \mathbf{x}_3),$$

kde parametr r je náhodné číslo v rozsahu [0,2], většinou ale menší než 1. Vytvořený vektor $\mathbf{v}$ se následně zkombinuje s vybraným vektorem $\mathbf{x}_0$ tak, že vytvoří nový vektor $\mathbf{u}$. Pro každý prvek u_i se vygeneruje náhodné číslo $c \in [0, 1]$ a pokud je menší než předem definovaná hodnota CR, použije se $u_i = v_i$ a v opačném případě $u_i = x_i$. Aby se předešlo náhodnému identickému

zkopírování prvku $\mathbf{x}$ (tedy že náhodná čísla c budou stále větší než CR), je navíc vygenerováno náhodné číslo $i_r \in \{1, \ldots, D\}$, kde D je dimenze úlohy, přikazující, že $u_{i_r} = v_{i_r}$.

Pro generování vektoru v se kromě zmíněné strategie používají i další možnosti lišící se použitými jednici nebo i jejich počtem [48]. Například

$$\mathbf{v} = \mathbf{x}_{\text{best}} + r \cdot (\mathbf{x}_2 - \mathbf{x}_3),$$
$$\mathbf{v} = \mathbf{x}_i + r \cdot (\mathbf{x}_{\text{best}} - \mathbf{x}_i) + r \cdot (\mathbf{x}_1 - \mathbf{x}_2),$$
$$\mathbf{v} = \mathbf{x}_{\text{best}} + r \cdot (\mathbf{x}_1 - \mathbf{x}_2) + r \cdot (\mathbf{x}_3 - \mathbf{x}_4),$$
$$\mathbf{v} = \mathbf{x}_1 + r \cdot (\mathbf{x}_2 - \mathbf{x}_3) + r \cdot (\mathbf{x}_4 - \mathbf{x}_5),$$

kde $\mathbf{x}_i$ jsou vzájemně rozdílní jedinci a $\mathbf{x}_{\text{best}}$ je nejlepší řešení aktuální populace. Po vyhodnocení nového jedince $\mathbf{u}$ se podle hodnoty jeho účelové funkce rozhodne, jestli nahradí původní prvek $\mathbf{x}_i$, nebo jestli bude nevratně odstraněn.

4.8 Estimation of Distribution

Algoritmus Estimation of Distribution je založen na odlišném konceptu než většina ostatních popsaných algoritmů. Za jeho prvopočátek je obvykle považována publikace [49], avšak velmi obdobný koncept byl popsán již o pár let dříve v [50].

Základní myšlenka algoritmu je založena na odhadu pravděpodobnosti jednotlivých proměnných tvořících řešení optimalizované úlohy. Z aktuální populace jedinců je statistickým zhodnocením odhadnuto rozdělení pravděpodobnosti a podle něj jsou následně generována a ohodnocována nová řešení. Algoritmus Estimation of Distribuion je populární koncept a existuje velké množství jeho variant. Nejjednodušší varianta je UMDA (Univariate Marginal Distribution Algorithm [51]), která odhaduje rozdělení pomocí jednoduché pravděpodobnosti bez uvažování interakcí mezi jednotlivými proměnnými. V případě diskrétní optimalizace je na počátku distribuční funkce inicializována se stejnou pravděpodobností pro všechny prvky a podle ní vygenerováno n prvků populace. Po ohodnocení účelovou funkcí je z nich vybráno m prvků, ze kterých se vytvoří nový odhad pro rozdělení pravděpodobnosti. UMDA nijak nezahrnuje vazby mezi proměnnými, a proto je odhad pravděpodobnosti založen pouze na počtu výskytů konkrétní hodnoty parametru ve vybrané podpopulaci.

$$P(x_i = X) = \frac{\sum_{j=1}^{m} \delta_j(x_i = X)}{m},$$

tedy pravděpodobnost, že proměnná x_i bude mít hodnotu X je rovna podílu počtu výskytů X mezi všemi vybranými jedinci ku jejich celkovému počtu m. Funkce $\delta(x_i = X)$ je klasická delta funkce, která je rovna jedné když $x_i = X$, jinak vrací nulu.

Po odhadnutí distribuční funkce je vygenerována celá nová populace a vše se opakuje tak dlouho, dokud není splněna nějaká předepsaná ukončovací podmínka.

Algoritmus tedy nemá standardní operátory *křížení* nebo *mutace* pro vytváření nových prvků pomocí přímé modifikace předchozích, avšak lze jej o ně i rozšířit [52]. Pro výběr prvků z aktuální populace se používají běžné operátory selekce, tedy např. k nejlepších jedinců, turnajový výběr apod. Velikost výběru nesmí být ani příliš malá ani příliš velká. Při malém výběru velmi rychle zdegeneruje odhad pravděpodobnosti do několika bodů a nemá možnost se z nich dostat, kdežto při velkém výběru je v podstatě stále kopírováno aktuální rozdělení pravděpodobnosti a algoritmus tak nezískává žádné nové informace.

Model pravděpodobnostního rozdělení bez interakcí je velmi jednoduchý a velmi často není pro popis úlohy dostatečný. Existuje naštěstí ale i mnoho dalších způsobů jak vytvářet vhodnější odhady. Kromě UMDA patří mezi modely bez interakcí i tzv. *compact Genetic Algorithm* [53], kde se odhad distribuční funkce inkrementálně mění vždy po vygenerování dvou jedinců. *Bivariate Marginal Distribution Algorithm* (BMDA) [54] je jedním z algoritmů, který zahrnuje binární interakci mezi proměnnými, tedy počítá podmíněné pravděpodobnosti pro každou dvojici proměnných. Pro interakce mezi více proměnnými zároveň se používají metody Bayesovských sítí [55], [56] a podobné.

4.9 Shuffled Frog-Leaping Algorithm

Způsob, jakým žáby sedící na kamenech v rybníce hledají potravu, se stal inspirací pro vznik metaheuristiky Shuffled Frog-Leaping. První publikovaná aplikace byla optimalizace sítě pro distribuci vody [57], kde ji autoři popisují jako kombinaci memetického genetického algoritmu a hejnového algoritmu. Počáteční populace je generována náhodně a obsahuje N samostatných řešení, která jsou ohodnocena a seřazena podle velikosti účelové funkce. Každé řešení reprezentuje žábu sedící v daném místě v rybníku. Populace je následně rozdělena do m skupin tak, že první žába patří do první skupiny, druhá žába do druhé skupiny, $\ldots, m$-tá žába do m-té skupiny. Další žáby jsou přiřazovány zase do skupin od začátku, tedy $m+1$ žába do první skupiny atd. Každá takováto skupina se nazývá memeplex[58] a v každém z nich probíhá samostatné vyhledávání. V každém memeplexu i se vždy modifikuje pouze nejhorší řešení X_w^i, a to pomocí jeho nejlepšího jedince X_b^i. Nová pozice nejslabšího řešení $X_{w(new)}^i$ se počítá pomocí následujících vztahů:

$$D = rand_{(0,1)} \times (X_b^i - X_w^i)$$
$$X_{w(new)}^i = X_w^i + D$$

Pokud není nově vzniklé řešení lepší než předchozí, procedura se opakuje, ale za použití globálně nejlepšího řešení X_b místo X_b^i. Pokud ani druhý pokus není úspěšnější než původní řešení, prvek se nahrazuje náhodně vygenerovaným. Každý memeplex se takto optimalizuje v k iteracích. Po

[58] memeplex je kombinace memů, tedy kulturních obdob genů.

optimalizaci každého memeplexu se celá populace zase seřadí a roztřídí do m skupin stejným způsobem jako na počátku. Celý algoritmus končí po dosažení předem definovaného počtu iterací. Základní princip je vytvořený primárně pro spojitou optimalizaci, ale existují i diskrétní varianty algoritmu, které definují vlastní modifikované operátory vytváření nových řešení [58].

4.10 Angels & Mortals

Hlavním rysem metaheuristiky Angels & Mortals je využití jednoduchého iterativního modelu chování ekologických systémů. Princip modelu spočívá v rozmístění několika andělů (angels) a smrtelníků (mortals) na toroidní mřížce, kde se mohou postavy volně pohybovat. V každé iteraci modelu se všechny postavy náhodně přesunují na jedno z osmi nejbližších sousedních políček. Smrtelníci mají předem danou dobu života (počet iterací) a po jejím uplynutí jsou z plochy odstraněni. V simulaci se uplatňuje jediné pravidlo, a to, že když se anděl a smrtelník potkají, tak je smrtelník zkopírován na nejbližší prázdné sousední políčko. Výsledkem je nestabilní adaptivní simulace schopná modelovat i velice složité evoluční systémy.

V metaheuristice Angels & Mortals reprezentuje množinu řešení populace smrtelníků a jejich doba života je přímo úměrná hodnotě jejich účelové funkce. Navíc každá iterace modelu způsobí nejen posun všech postav po ploše, ale zároveň i mutaci každého jednotlivého smrtelníka. Jedinci s lepší účelovou funkcí mají tedy delší dobu života, a tím i vyšší pravděpodobnost, že potkají anděla a budou naklonováni.

Algoritmus je popsaný v publikaci [59], kde jsou dvě jeho varianty otestovány na problému obarvování grafu a je i porovnán se standardním genetickým algoritmem. Druhá modifikovaná varianta spočívá ve změně chování smrtelníků při potkání anděla, kdy je jim místo klonování prodloužen život.

4.11 Spiral optimization

Spirála je geometrická křivka, která má významně vyšší hustotu u svého středu než ve více vzdálených oblastech. Této vlastnosti se využívá v metaheuristice Spiral optimization [60], kde jsou jednotlivá řešení vybírána na základě tvaru Archimedovy spirály a zmíněná hustota křivky určuje velikost rozsahu prohledávání. Nejjednodušší spirálu lze vytvořit postupnou rotací bodu kolem pevnému středu společně s jeho současným přibližováním. K diskretizaci spirály lze použít následující vztah

$$x_{i+1} = r \cdot R(\theta) \cdot x_i - (r \cdot R(\theta) - I) \cdot x^*, \tag{4.1}$$

kde $r \in (0, 1)$ je parametr přibližování, $R(\theta)$ je matice rotace s úhlem θ, I je jednotková matice a x^* je pevný střed, kolem kterého je bod o souřadnicích x_i přesunut na souřadnice x_{i+1}. Rekurzivními iteracemi bodu x_i tedy vznikne diskretizovaná spirála konvergující k bodu x^*.

Použitá matice rotace pro n rozměrný prostor se skládá ze součinu standardních matic rotace ve všech dvourozměrných podprostorech a v každém z nich je použita stejná velikost úhlu θ.

Metaheuristika *Spiral optimization* na začátku generuje n náhodných bodů v prohledávaném prostoru a nejlepších z nich se považuje za střed rotace. Všechny ostatní body jsou potom přesunuty pomocí vztahu 4.1. Tento jednoduchý princip se opakuje, dokud není splněna daná ukončovací podmínka s tím, že jako střed rotace pro každou iteraci je zvoleno vždy nejlepší řešení z aktuální populace. Rotace všech bodů je prováděna se stejným parametrem r a o stejný úhel θ. Adaptivní změna těchto parametrů na základě statistické analýzy průběhu optimalizace byla autory vyšetřována v publikaci [61].

Publikované porovnání s jinými metaheuristikami [60] naznačuje velice slušné výsledky, a to zejména díky schopnosti algoritmu jednoduše spojit globální prohledávání s lokálním zintenzivněním v závislosti na kvalitě nalezených řešení. Čím blíže se pohybující body nachází středu rotace (nejlepší řešení), tím menší posun dělají, a tím podrobnější prohledávání probíhá.

4.12 Hyper-Heuristic

V rámci co nejobecnějšího přístupu k řešení problémů byly vyvinuty algoritmy, které nemají pořadí svých operátorů předem striktně definované, ale naopak je přizpůsobováno podle aktuálních potřeb. *Hyperheuristiky*, jak jsou tyto algoritmy nazývány, vybírají z množiny nízko-úrovňových heuristik vždy takové, které mají nejvyšší šanci produkovat kvalitní řešení. Zjednodušeně se tak hyperheuristiky mohou popisovat jako *heuristiky, které vybírají heuristiky*. Poprvé byl tento koncept použit a pojmenován v [62], ale podobné myšlenky lze nalézat i o téměř půl století dříve [63].

Mezi nízko-úrovňové heuristiky mohou patřit jak problémově závislé, tak i obecnější operátory. Např. při řešení problému obchodního cestujícího lze jako nízko-úrovňové heuristiky využít operátory náhodného prohození n prvků, záměnu prvků s nejdelší spojující hranou nebo i hladový algoritmus konstruující řešení propojením vrcholů s nejkratšími hranami.

Hyperheuristika potom z množiny těchto heuristik vybírá podle různých pravidel. Jednoduchým přístupem je například náhodně vybrat jednu heuristiku a opakovaně ji aplikovat tak dlouho, dokud produkuje zlepšující se řešení. Obdobně lze nejdříve vytvořit náhodnou permutaci heuristik a vykonávat je v tomto pořadí, dokud vytváří dostatečně dobrá řešení. Algoritmus může ale každé nízko-úrovňové heuristice přiřazovat i hodnotu pravděpodobnosti přímo určenou kvalitou produkovaných řešení. V takovém případě spolu heuristiky soutěží a nejkvalitnější bude mít nejvyšší pravděpodobnost výběru pro další produkci (více viz [64]). Hyperheuristiky tedy mohou vykazovat vyšší *stupeň inteligence*, protože jejich chování je závislé na předchozích *zkušenostech* s vytvářením řešení pro zadanou instanci problému.

Hyperheuristiky jsou často odlišovány od metaheuristik, protože jejich stavovým prostorem není prostor řešení úlohy, ale místo toho prohledávají prostor nízko-úrovňových heuristik. Snahou hyperheuristik je tedy nalézt nejvhodnější heuristiku pro zadaný problém a zadané aktuální

řešení. Výběr heuristik lze řídit širokým spektrem heuristických, statistických nebo samoučících se postupů jejichž velice dobrý přehled je uveden v publikaci [63].

4.13 Ant colony optimization

Algoritmus *Ant colony optimization* napodobující chování mravenců při hledání potravy je jedním z nejoblíbenějších biologicky inspirovaných metaheuristik. Primárně je určen pro kombinatorickou optimalizaci ([65]), ale existuje i několik variant pro optimalizaci diskrétních ([66]) nebo spojitých ([67]) proměnných.

Mravenci hledají v přírodě potravu skupinově a implicitně při tom spolupracují. Každý mravenec při pohybu za sebou zanechává feromonovou stopu, která s postupem času ztrácí na své intenzitě až následně vymizí úplně. Mravenci při opuštění mraveniště náhodně prohledávají okolí a v případě nalezení potravy se ihned vrací zpět. Nejkratší cesta mezi nalezenou potravou a mraveništěm má nejsilnější feromonovou stopu, protože ji procházející mravenci udržují s nejvyšší frekvencí. Metaheuristika *Ant Colony Optimization* využívá tento koncept nejkratší cesty jako paralelu k nejlepšímu řešení zadané úlohy. Řešení jsou na rozdíl od většiny ostatních metaheuristik konstruována postupně a nikoliv považována za pevné body stavového prostoru, jako je tomu o mnoha jiných metaheuristik. Konstrukce řešení probíhá pomocí udržovaného grafu, jehož vrcholy reprezentují jednotlivé prvky vstupní množiny úlohy (prvky, z nichž se skládá kombinatorické řešení úlohy) a ohodnocené hrany značí kvalitu konkrétní návaznosti dvou vrcholů v řešení. Každé vyhodnocení nového řešení změní hodnoty přiřazené hranám obdobným způsobem, jako mravenci zvyšují svým feromonem zápach na cestě. Čím lepší je aktuálně vyhodnocené řešení, tím vyšší je feromonová stopa na hranách spojující proměnné řešení v daném pořadí.

Pokud je zadáním úlohy například nalézt řešení kombinatorické úlohy o pěti proměnných, tedy určit optimální pořadí pěti prvků (A, B, C, D, E) vůči nějaké účelové funkci f, je reprezentace feromonů tvořená grafem[59] o pěti vrcholech s dvaceti hranami[60]. Řešení $[B, E, C, A, D]$ s hodnotou účelové funkce f_0 pak zvýší hranám (B, E), (E, C), (C, A), ... váhy přímo úměrně hodnotě f_0. Konstrukce nových řešení probíhá zvolením startovacího vrcholu a pak postupným přidáváním dalších vrcholů podle pravděpodobnosti určené váhami hran. Cesty v grafu, které odpovídají nejlepším řešením, mají největší váhy a jsou tedy nejpravděpodobněji konstruovány. Navíc je zahrnut mechanizmus vypařování feromonů, který při každé iteraci exponenciálně sníží hodnoty všech hran. Nově konstruovaná řešení tak nejsou vystavena přílišnému riziku předčasné konvergence nebo uváznutí v lokálním extrému[61].

---

[59] Graf je většinou reprezentován pomocí matice, která pak přímo bývá nazývána *matice feromonů*.

[60] Spojením každého vrcholu s každým.

[61] Tato rizika, nejsou úplně odstraněna, ale vypařování feromonů sníží jejich pravděpodobnost.

4.14 Další algoritmy

Metaheuristiky čerpají inspiraci z mnoha odlišných vědeckých oblastí. Běžná nejsou jen propojení s relativně blízkými odbory, jako je matematika, statistika nebo umělá inteligence, ale velmi často jsou základní koncepty odvozeny z oblastí, které u aplikované informatiky působí velmi exoticky. Výše zmíněné algoritmy jsou více či méně typickými zástupci dnešních metaheuristik. Jejich princip je založen buď na jednoduchých intuitivních pravidlech pohybu v neznámém prostoru (Hill climbing, Tabu search, ...), nebo na inspiraci přírodními fenomény, jako je evoluce, genetika nebo pohyb zvířat. Poslední jmenovaný přístup má mezi novými metaheuristikami výsadní postavení získané především atraktivností vlastního pojmenování než reálným inovativním přínosem. Následující odstavce názorně demonstrují šíři využité inspirace v publikovaných metaheuristikách.

Bacterial Foraging Optimization Algorithm [68] je algoritmus založený na chování bakterie Escherichia-coli. Hlavní stavební kameny jejího chování jsou odvozeny z chemotaxe[62], rozmnožování a obecného hejnového chování bakterií. *Bat-Inspired Algorithm* [69] simuluje schopnosti netopýrů, kteří loví hmyz pomocí echolokace. Pohyb netopýrů je ovlivněn vzdáleností od potravy, která zde zastupuje kvalitu řešení účelové funkce. Na principu echolokací je postaven i algoritmus napodobující delfíny při lovu [70]. Obecnějším přístupem zvířat (lvů, vlků, ...) k lovu je inspirován algoritmus *Hunting search* publikovaný v [71]. Metaheuristika *Cuckoo search* [72] je inspirována kukačkami a jejich zvykem klást svá vejce do hnízd jiných ptáků. Mezi další živočišné druhy, které posloužily k inspiraci metaheuristik, patří například kočky [73], lososi [74], opice [75], orli [76], ...

Chování včelích rojů je významnou inspirací pro více různých metaheuristik zároveň. *Honey bee mating optimization* [77] napodobuje chování včelí královny při páření a následném kladení vajíček. Včelí královna si zde vybírá z velkého množství trubců, jejichž genofond kombinuje se svým. *Artificial bee colony algorithm* [78] naopak napodobuje chování včel při hledání nektaru na loukách a každá včela reprezentuje jedno řešení úlohy. Běžné včely spolu komunikují pomocí signál, předávaných v úlu, což je u metaheuristiky nahrazeno následováním včely s nejlepší hodnotou účelové funkce. Metaheuristiky založené na chování včel jsou jedny z velmi populárních, a proto vzniklo mnoho jejich dalších druhů a aplikací (viz [79]).

Bumblebees algorithm [80] je založený na podobném konceptu jako Angels & Mortals, kde se jednotlivé postavy pohybují po toroidu a vzájemně se ovlivňují. Tento algoritmus obdobně využívá čmeláky, kteří na toroidu hledají květiny. Nejedná se o klasický hejnový algoritmus, ale spíše o sofistikovanější přístup k náhodnému prohledávání a udržování různých řešení v paměti. Čmeláci byli inspirací i pro algoritmus *Bumble Bees Mating Optimization* [81], který ke konstrukci nových řešení využívá speciálně upravený hladový algoritmus. Hmyz obecně patří mezi velmi časté objekty využívané v metaheuristické inspiraci, protože se v přírodě musel naučit

[62] Pohyb organismu či buňky ve směru chemického gradientu.

kooperovat ve velkých počtech a vykazuje tak vysoký stupeň samoorganizace. Další příklady mohou být termiti (*Termite colony optimization* [82]), červi (*Glowworm swarm optimization* [83]) nebo i octomilky (*Fly optimization Algorithm* [84]).

Zajímavým zdrojem inspirace je i astrofyzika a mechanika. Na pohybu vesmírných těles způsobených gravitací jsou založeny algoritmy *Gravitational algorithm* [85] a *Central force optimization* [86], kde hlavní úlohu hrají přitažlivé síly mezi jednotlivými vesmírnými objekty reprezentující řešení ve stavovém prostoru. Podobně i algoritmy *Big Bang-Big Crunch*[87] a *Big Crunch* [88] využívají principů gravitace a iterativně přitahují a odpuzují jednotlivá tělesa. Stejně byl postaven i algoritmus *Galaxy-based Search Algorithm* [89], kde jsou hlavními objekty celé rotující galaxie a algoritmus *Black hole* využívající fyziku černých děr [90].

Z dalších fyzikálních jevů byly k vybudování nových metaheuristik využity principy elektromagnetismu (*Electro-magnetism optimization* [91]), elektrostatického náboje (*Charged system search* [92]), deterministického chaosu [93] nebo i kvantové fyziky používané k obohacení genetických algoritmů [94] či simulovaného žíhání [95]. Pro přehledný souhrn dalších fyzikálně inspirovaných metaheuristik lze doporučit stručnou publikaci [96].

Využity byly i procesy formování přírody a procesy zajišťující životní koloběh. Způsob, jakým voda stéká po hornatém terénu a tvoří řeky a potoky, se používá v metaheuristice *River Formation Algorithm* [97], tvorba mraků je základem *Atmosphere Clouds Model Optimization Algorithm* [98] a celý vodní koloběh je zahrnut v *Water cycle algorithm* [99]. *Biogeography-based optimization* [100] čerpá inspiraci z ostrovní biogeografie a za základní princip používá myšlenku, že rychlost změny počtu živočišných druhů na ostrově je výrazně závislá na rovnováze mezi počtem imigrujících a emigrujících druhů. Stejně jak živočichové, tak i rostliny daly svým chování vzniknout několika metaheuristikám, jmenovitě to byl princip opylování v *Flower Pollination Algorithm* [101] nebo růst trávy v *Weed colonization Algorithm* [102].

Mimo přírodních procesů se využívá i konceptů pozorovaných v lidském chování a sociální interakci. *Cultural algorithms* [103] rozšiřují koncept genetických algoritmů o tzv. *belief space*, který symbolizuje předávání znalostí mezi generacemi. Vývoj lidské kultury je pozvolnější a není přímo závislý na změnách v genetickém vývoji populace, vzájemně se ale oba vývoje ovlivňují. Belief space je rozdělen na pět samostatných prostorů, které reprezentují určité oblasti znalostí a jejich řízené propojení je následně využíváno k tvorbě nových řešení. Podobně i *Imperialist competitive algorithm* [104] je považován za obdobu genetických algoritmů v sociální oblasti, tedy že se jedná o sociální evoluci populace. Sociálním postavením jedince je inspirován algoritmus *Social emotional optimization* [105], kde každé řešení symbolizuje konkrétního člověka, jehož cílem je dosáhnout co nejvyššího postavení ve společnosti. Do stejné oblasti se řadí i algoritmus *League championship algorithm* [106] simulující taktiku sportovních klubů ve snaze dosáhnout co nejlepšího rozložení sil při soutěžním šampionátu.

Vývoj nových algoritmů pokračuje ale i bez přímé inspirace reálnými procesy, a to za pomoci konceptů založených spíše na informatické nebo matematické představě o fungování metaheuris-

tik. *Nested partition algorithm* [107] průběžně klasifikuje části stavového prostoru podle pravděpodobnosti výskytu kvalitních řešení a následně podle toho generuje nová řešení z nejslibnějších oblastí. *Scatter Search* [108] vybírá z aktuální populace reálných řešení vhodnou podskupinu a různé dvojice jejich prvků následně lineárně interpoluje. Prvky takové vhodné podskupiny nemusí být jen nejlepšími prvky z pohledu hodnot účelové funkce, ale i například prvky zajišťující udržení dostatečné variability. Lineární kombinace dvou vybraných prvků mohou být chápany jako body tvořící jejich spojující cestu. Touto myšlenkou se řídí i metaheuristika *Path relinking* [108], která zmíněný Scatter search zobecňuje i pro nespojité proměnné. Pro každou vybranou dvojici řešení se nalezne cesta postupně měnící jedno řešení na druhé a jako nově vyprodukované řešení se použije nějaký její bod. Jelikož takových spojovacích cest může existovat více, uvažuje se pouze cesta s nejlepšími hodnotami účelové funkce.

Metaheuristika *Variable Neighborhood Search* [109] je příkladem rozšíření klasického lokálního prohledávání rozšiřující základní koncept o vlastní přístup k útěku z lokálního optima. V případě, že v okolí není žádný bod s lepší hodnotou účelové funkce než má aktuální pozice, je způsob generování tohoto okolí změněn. Pokud se ani v nově vytvořeném okolí (např. okolí s větším průměrem) nenachází žádný lepší bod, tak se buď zvolí další způsob generování okolí, nebo se algoritmus prostě ukončí. *Iterated local search* [110] volí odlišnou strategii pro vyváznutí z lokálního optima. Jakmile v okolí už nelze nalézt žádné lepší řešení, je aktuální bod náhodně pozměněn, a to natolik významně, že by se při následném pokračování v lokálním prohledávání nemělo vrátit zpět do stejného optima. Jedná se v podstatě o obdobu úplného restartu algoritmu, ovšem s tou výjimkou, že jsou zachovány všechny získané znalosti i celá paměť (např. tabu list). Lokální prohledávání je využíváno i metaheuristikou *Memetic Algorithms* [111], kde je standardní koncept evolučních algoritmů prokládán zlepšováním vybraných řešení pomocí lokálních (meta)heuristik. Tento přístup může být také interpretován jako postupné zlepšování jedinců v průběhu jejich života (učení se lepšímu způsobu přežití) a nespoléhání se pouze na sílu evolučních principů. Mezi další samostatné rozšíření lokálního prohledávání lze zařadit i *Reactive search optimization* [112], které zajišťuje adaptivní přizpůsobování parametrů metaheuristiky za jejího běhu. Typickým příkladem je přizpůsobování délky tabu listu v závislosti na předpokládané fázi algoritmu[63].

[63] Na začátku algoritmu stačí krátká paměť zajišťující rychlý výpočet, kdežto v blízkosti lokálních optim je potřeba tabu list s větším počtem řešení, aby nedošlo k zacyklení

5 Souhrnný pohled na metaheuristické přístupy

Publikovaných metaheuristik je nepřeberné množství. Některé algoritmy se liší pouze drobnými detaily, kdežto některé přináší zásadně nové koncepty. Navíc existuje bezpočet publikovaných kombinací a modifikací základních algoritmů, včetně jejich spojování s metodami jako jsou neuronové sítě nebo buněčné automaty. Společným pilířem všech metaheuristických přístupů musí ale nutně být práce s informacemi získanými vyhodnocováním účelové funkce v různých bodech stavového prostoru. Jakým způsobem jsou jednotlivá řešení vybírána a jaké informace jsou z nich následně získávány, už ale závisí na konkrétní metaheuristice. Následující odstavce porovnávají různé metaheuristické koncepty z pohledu jejich základních stavebních kamenů. Napříč různými koncepty lze takových společných znaků vypozorovat hned několik a s jejich pomocí lze pak jednotlivé metaheuristiky i rozlišovat.

5.1 Reprezentace řešení

Aby metaheuristika mohla vyhodnocovat účelovou funkci, musí být jednoznačně definovaný způsob převodu objektů udržovaných v paměti na řešení uvažované úlohy. Velmi často je tento převod triviální, protože objekty v paměti přímo reprezentují řešení, ale obecně tomu tak nemusí být vždy. Potřeba hledat hodnoty n spojitých proměnných x_i tak bývá nejčastěji zakódována do jedince x ve tvaru

$$x = (x_1, x_2, \ldots, x_n)$$

a metaheuristiky, které takto reprezentují vektor rozhodovacích proměnných úlohy, prohledávají přímo i její stavový prostor. Typické jsou tím hejnové nebo evoluční algoritmy. Jinak je tomu u standardních genetických algoritmů, kde je řešení nejdříve transformováno (zakódováno) do dvojkové soustavy a metaheuristické prohledávání tedy neprobíhá přímo ve stavovém prostoru úlohy, ale v k rozměrném binárním prostoru ($\{0,1\}^k$). V závislosti na počtu bitů použitých k vyjádření jedné rozhodovací proměnné se mění celkový rozměr stavového prostoru k. Např. při osmi bitech a třech proměnných se místo třírozměrného stavového prostoru úlohy prohledává 24 rozměrný prostor binárních proměnných. Transformace je bijektivní, a body obou prostorů lze tedy na sebe vzájemně jednoznačně převádět[64]. Transformační funkce má na výsledné chování metaheuristiky významný vliv, protože přechodem z jednoho prostoru do druhého se nemusí zachovávat celková struktura úlohy. Například vzdálenosti mezi body v jednom prostoru nemusí odpovídat vzdálenostem v transformovaném prostoru, proto může malý pohyb v jednom z nich znamenat velmi velký pohyb v druhém. Významné je to z pohledu definice okolí bodu,

[64] Aby transformace byla opravdu bijektivní, musí být zajištěna stejná mohutnost obou množin. U převodu reálných proměnných na binární se tak většinou uvažuje pouze omezená přesnost reálných čísel (např. dvě desetinná místa).

kde okolí v transformovaném prostoru vůbec nesouvisí s okolím v původním prostoru. Jedinou záměnou nejvýznamnějšího bitu ve zmíněné 8-bitové soustavě lze docílit pohyb o 2^7 v metrice původního prostoru. K předejití popsanému problému se používá tzv. *Grayovo kódování*, které transformuje vstup do stejného prostoru $\{0,1\}^k$, avšak takovým způsobem, že změna jednoho bitu v reprezentaci rozhodovací proměnné změní její velikost pouze o jeden krok[65]. Jako transformační funkce může být použito mnoha různých funkcí, prohledávání lze provádět např. v různých číselných soustavách, ve frekvenčním spektru a nebo prostoru hlavních komponent [113].

Metaheuristiky však nemusí prohledávat pouze prostory, jejichž body přímo reprezentují (transformované) řešení úlohy. Například *Ant colony optimization* lze chápat jako metaheuristiku, která prohledává prostor matic feromonových stop. *ACO* neustále tuto matici modifikuje a jako "vedlejší produkt" vznikají jednotlivá řešení úlohy. Stejně tak lze chápat i algoritmus *Estimation of Distribution*, kde se prohledává prostor distribučních funkcí a na jejich základě se pak *pouze* generují příslušná řešení úlohy. *Hyperheuristické algoritmy* se zase pohybují v prostoru nízkoúrovňových heuristik a hledají takové, které produkují nejlepší řešení. Algoritmem popsaným v kapitole 9.3 se přímo prohledává pouze prostor s malým rozměrem a mnoharozměrné řešení úlohy vznikne až spojením všech aktuálních jedinců populace dohromady.

Kvalita prvků

Ohodnocování kvality prozkoumaných řešení je základním prvkem metaheuristického prohledávání. Jednotlivé objekty je mezi sebou potřeba porovnávat a z těch nejlepších vytvářet nová řešení. Ve většině metaheuristik reprezentují základní objekty přímo řešení úlohy a jejich kvalita tak může být určena vypočítanou hodnotu účelové funkce. Kvalitnější jedinci jsou závislí na druhu optimalizačního kritéria - při minimalizaci platí, že čím nižší hodnota, tím kvalitnější jedinec a při maximalizaci je závislost přirozeně obrácená. Např. při víceúčelové optimalizaci ale už není takto definovaná relace takového uspořádání jednoznačná (viz kapitola 2.2.4) a způsoby určování kvality se tomu musí přizpůsobit. Častým způsobem porovnávání kvality jedinců (nejen ve víceúčelové optimalizaci) je transformace jejich charakteristik do jednorozměrné skalární veličiny. Genetické algoritmy definují tzv. *fitness function*, neboli míru přizpůsobení jedince aktuální úloze a čím vyšší je, tím lepší má i jedinec kvalitu. Takové výpočty je možné provádět mnoha různými způsoby od násobení účelové funkce vhodnou konstantou přes vážené lineární kombinace až po odhady pomocí metod umělé inteligence. Například v [114] je použita metoda neuronových sítí ke zvýšení rychlosti výpočtu metaheuristiky. V případě, že výpočet účelové funkce trvá velmi dlouho dobu (numerické simulace atp.), je neuronová síť použita k odhadu kvality a reálné výpočty se provádí pouze pro vybranou podskupinu nejslibnějších jedinců. V

[65] Pokud jsou Grayovým kódováním transformována celá čísla, krok odpovídá hodnotě 1. Pokud jsou kódována reálná čísla s dvěma desetinnými místy, je velikost kroku 0.01.

této implementaci je kvalita objektů měřena dvojím druhem (odhad a vypočítaná hodnota), a to dokonce ještě může být i proměnné hodnoty v závislosti na natrénování neuronové sítě. Nekonstantní kvalita se objevuje i v algoritmu *CLONALG*, kde je vypočítávána v závislosti na celé aktuální populaci (normováním seřazené populace podle účelové funkce do rozsahu $[0, 1]$). Při každé iteraci tak může mít jedinec jinou kvalitu. I algoritmus popsaný v kapitole 9.3 počítá kvalitu jedinců odvozením z celé populace, ovšem v tomto případě má svou kvalitu i samotná populace a ne jenom její jedinci.

Inicializace algoritmu

Každý algoritmus musí nějak nastavovat prvotní hodnoty proměnných na počátku svého běhu. Nejčastějším způsobem je náhodné vygenerování definovaného počtu nezávislých řešení, ze kterých je pak následně pokračováno v dalších iteracích. V případě nulových apriorních předpokladů o tvaru účelové funkce se pro generování používá rovnoměrné rozdělení pravděpodobnosti na celém prostoru. Stavové prostory ovšem bývají většinou opravdu velké a vygenerování malého počtu náhodných řešení nemusí dostatečně pokrývat potřebné oblasti. Důležitou vlastností metaheuristik je schopnost i z nekvalitních (náhodných) řešení získat řešení vysoce kvalitní. Jelikož ale takový výsledek nelze zaručit, většinou se volí doplňkové metody zvyšující šance na úspěšné nalezení potřebného řešení. Častým způsobem je opakovaný běh metaheuristiky, tedy nezávislé opětovné spuštění algoritmu po jeho dokončení. Inicializace počátečních řešení tak hraje zásadní úlohu, a to zvláště v deterministických[66] nebo lokálních[67] metaheuristikách.

Pokud o řešené úloze nějaké apriorní předpoklady existují, mohou být do generování počáteční populace promítnuty pomocí nerovnoměrné distribuční funkce. Větší pravděpodobnost vygenerování je tak přiřazena řešením, která mají lepší předpoklady na kvalitní hodnotu účelové funkce. Z hlediska robustnosti běhu algoritmu bývá ale výhodné generovat i několik méně kvalitních řešení, aby byla zajištěna dostatečná variabilita populace a tím i odolnost vůči rychlé konvergenci do lokálního optima.

Jako startovací body metaheuristiky je možné používat i výsledky jiné metaheuristiky, popř. jakéhokoliv jiného vhodného algoritmu. Užitečné výsledky mohou poskytovat různé verze hladových algoritmů umožňující rychlou konstrukci řešení s *rozumnými* vlastnostmi. Při řešení velkých úloh v prostorech s mnoha rozměry lze také použít generování počáteční populace, která nepokrývá celý stavový prostor rovnoměrně, ale místo toho soustředí své síly na výrazně menší podprostor. Lze například zafixovat některé z proměnných (nebo jim dovolit pouze malý rozptyl) a výpočet takto opakovat pro různé zafixované proměnné.

---

[66] Metaheuristiky, které při opakovaném běhu ze stejného startovacího bodu vždy vrátí stejný výsledek.

[67] Metaheuristiky, které nevykonávají velké skoky v prostoru, ale přesouvají se pouze v rámci svého nejbližšího aktuálního okolí.

Vytváření nových řešení

Vytváření nových řešení je základní a nejdůležitější funkcí metaheuristických algoritmů. Způsoby, jakými jsou řešení postupně generována, tvoří samotnou podstatu metaheuristik a jsou velmi závislé na typu bodů ve stavovém prostoru. Principem metaheuristik je kombinovat informace získané procházením stavového prostoru a na jejich základě postupovat do dalších bodů. Metaheuristika tedy musí definovat (často pouze implicitně), jaké ze získaných informací považuje za důležité a jak je využije v generování nových řešení úlohy.

V genetických (evolučních) algoritmech je klasickým způsobem vytváření nových řešení tzv. *křížení*, kde nový jedinec vzniká kombinací hodnot proměnných ze dvou rodičovských prvků. Při *rovnoměrném křížení* jsou hodnoty proměnných vybírány s pravděpodobností 0.5 z každého z rodičů. Informace, která se tímto způsobem dědí, je přímo hodnota proměnné a zároveň i jejich vzájemný vztah (50% proměnných zůstává ve stejném vztahu jako v rodičovském prvku). Geometrická poloha řešení vygenerovaného tímto způsobem může být velmi vzdálená od rodičovských prvků, ale díky přenosu dědičné informace si i přes to zachovává *významovou blízkost*.

Naopak hejnové algoritmy vytváří nová řešení pomocí geometrické blízkosti k rodičovským prvkům. Lineární kombinací několika jedinců se získá řešení, které nemusí mít žádnou hodnotu proměnné stejnou jako některý z rodičů, ale přesto je míra přenesené informace velmi významná. Oba zmíněné typy lze zobecnit pod jeden společný postup, kde lineární kombinací k rodičovských jedinců p_i

$$\begin{aligned} p_1 &= (p_1^1, p_1^2, \ldots, p_1^n) \\ p_2 &= (p_2^1, p_2^2, \ldots, p_2^n) \\ &\ldots \\ p_k &= (p_k^1, p_k^2, \ldots, p_k^n) \end{aligned}$$

lze získat nového jedince $x = (x^1, x^2, \ldots, x^n)$ jako

$$x^i = \sum_{j=1}^{k} c_j^i p_j^i,$$

kde c_j^i jsou konstanty buď zvolené před začátkem běhu algoritmu, nebo jsou dynamicky generovány až při samotném výpočtu. Pro rovnoměrné křížení v genetických a evolučních algoritmech je počet rodičů $k = 2$ a konstanty $c_j^i \in \{0, 1\}$ takové, že $p_i^1 + p_i^2 = 1$ pro $i = 1, \ldots, n$, tedy, že je vždy použita hodnota proměnné z právě jednoho z rodičů. Standardní hejnové algoritmy při vytváření kombinují tři různá řešení ($k = 3$) - aktuálního jedince, jeho nejlepší nalezenou pozici a globálně nejlepší nalezené řešení. Hodnoty c_j^i jsou generovány podle vhodných vzorců uvedených v kapitole 4.2. Stejný obecný postup je platný pro mnoho dalších metaheuristických

algoritmů, např. *Harmony search*, *Firefly algorithm*, *Differential evolution*[68]. Dokonce i standardní operátor mutace lze chápat podle tohoto schématu, kde jeden z rodičů je nové náhodné řešení a druhý je samotný mutovaný jedinec (hodnoty $c_j^i \in \{0, 1\}$ s vhodnou distribuční funkcí).

Další typy děděné informace mohou být různé povahy. Algoritmus *Estimation of distribution* zachovává u nových jedinců stejnou pravděpodobnost výskytu určité hodnoty proměnné, jako měli vybraní kvalitní jedinci. To je dosaženo odhadem a následným využitím distribuční funkce proměnných. V závislosti na použitém typu distribuční funkce se zachovávají vzájemné vztahy mezi jednotlivými proměnnými. Podobně je tomu i u optimalizace pomocí mravenčích kolonií, kde je předávaná informace obsažena v pravděpodobnosti výskytu daných dvojic hodnot proměnných. Algoritmus *Spiral optimization* rotuje všechna řešení kolem definovaného středu a zachovává tak informaci v podobě vzdálenosti mezi rotovaným řešením a středem.[69]. Odlišný případ lze pozorovat u hyperheuristik, kde je předávaná informace dynamicky měněna na základě aktuálních použitých operátorů.

Důležitým faktorem tvorby nového řešení je počet jedinců, kteří se na něm rodičovsky podílejí. Čím více jedinců je k tomu využito, tím méně je nové řešení podobné některému z nich. Naopak při malém počtu rodičů se zděděná informace zachovává výrazně čistěji, protože není rozmělněna velkým počtem různých vlivů.[70] Jednotlivé přístupy vytváření nových řešení tak lze rozdělit podle počtu rodičů následovně

- *Žádný jedinec* - na začátku algoritmu probíhá inicializace, která vytváří řešení zcela náhodně a nedědí se tím od nikoho žádná informace. Stejný přístup se občas uplatňuje i pro udržení (zvýšení) variability populace nebo opakovaný start algoritmu (speciálně v *Iterated local search*).
- *Jeden jedinec* obecně generuje dva typy nových řešení. Buď je nová proměnná zcela nezávislá na její hodnotě v rodičovském prvku, a nebo je jí nějakým způsobem ovlivněna. První z případů je typický pro mutaci v evolučních algoritmech, kde je některá z proměnných náhodně změněna na novou hodnotu. Tento typ vytváření řešení zachovává velkou část rodičovské informace a zároveň významně obměňuje její tvar. Za zmínku stojí, že například u algoritmů *Angles&Mortals* nebo *CLONALG* jde o jediný způsob vytváření nových řešení.

  Druhý způsob produkce nových řešení z jediného rodiče je modifikací aktuálních hodnot. V *PSO* je každý prvek z hejna přitahován k nejlepšímu jedinci. Samotný nejlepší jedinec ale nemá ke komu být přitahován, a tak je jenom mírně posunut náhodným směrem. Nové řešení tak vznikne malými změnami v proměnných vstupního řešení.

---

[68] *DE* dokonce dovoluje i extrapolaci, tedy $\sum_{j=1}^{k} c_j^i > 1$ pro libovolné i.

[69] Vzdálenost není zachována konstantně, ale postupně se zmenšuje.

[70] Samozřejmě lze nadefinovat i operátory, které se chovají přesně obráceně něž je popsáno, ale jedná se spíše o výjimečné přístupy.

Počet produkovaných řešení z jediného rodiče se může lišit podle potřeb metaheuristiky. Mutace v *GA* obvykle generují pouze jediné nové řešení, kdežto v algoritmu *CLONALG* je počet závislý na kvalitě hodnoty normalizované účelové funkce. Čím lepší hodnota, tím více prvků vzniká. Speciálním typem produkce nových řešení z jediného prvku je vytváření *okolí*, kde výstupem je množina (všech) jedinců s definovanou vzdáleností od vstupního prvku.

- *Dva jedinci* jsou typičtí pro genetické algoritmy, kde nová řešení vznikají jejich křížením. Kromě výše zmíněného rovnoměrného křížení je definováno několik dalších standardních křížení, např. jednobodové (rozdělením vektoru proměnných na dvě části a jejich následné prohození mezi rodiči), dvoubodové (obdobné jako jednobodové, ale s rozdělením vektoru na tři části) atd. Dva jedince pro vznik nového řešení používá dále i algoritmus *Spiral optimization*, kde jeden hraje roli středu rotace a druhý je kolem něj rotován.
- *Tři jedinci* produkují řešení v klasickém hejnovém algoritmu. Jedná se o lineární kombinaci aktuálního, lokálně nejlepšího a globálně nejlepšího jedince (viz též výše).
- *Čtyři jedinci* produkují nové řešení v algoritmu *Differential evolution* pomocí lineárních kombinací. Stejně tak existují implementace této metaheuristiky, které využívají kombinace pěti i šesti jedinců k produkci jediného nového řešení.
- *Celá aktuální populace* se podílí na vytváření nového řešení například v algoritmu *Harmony search*. Každá nová hodnota proměnné je vybrána náhodně z jednoho z řešení a jedná se tak v podstatě o zobecnění rovnoměrného křížení dvou jedinců v *GA*. V *Harmony search* většinou populace mívá nízký počet jedinců, v opačném případě by byla hodnota zděděné informace velice nízká. Stejně tak i *Firefly algorithm* vytváří nová řešení pomocí lineární kombinace všech jedinců, ale každý má na výsledek jinou váhu, čímž je zabezpečena vyšší hodnota předané informace (čím lepší hodnota účelové funkce, tím větší váhu jedinec na výsledek má).
- *Nepřímý vliv* mají jedinci na tvorbu nových řešení v algoritmech jako je *Estimation of distribution*, *Nested partition algorithm* nebo *Ant colony optimization*. Každý prozkoumaný jedinec přispívá do způsobu generování nových řešení (pravděpodobností rozdělení, feromonová matice), ale jejich konkrétní hodnoty v proměnných nejsou nijak přímo použity při další konstrukci.

V rámci jedné metaheuristiky se většinou kombinuje více druhů produkce nových řešení. Genetické algoritmy využívají primárně křížení dvou jedinců a pro udržení variability doplňkově i jedno-jedincovou mutaci. Popsané rovnoměrné křížení pouze přesunuje hodnoty mezi odpovídajícími si proměnnými různých jedinců, ale nikdy nemůže vytvořit novou hodnotu, která se u žádného řešení nevyskytuje. Jednotlivé procedury tedy nemusí být nutně schopny produkovat libovolné řešení, ale k dosažení potřebné účinnosti je stačí pouze vhodně kombinovat.

5.2 Populace

Pojem *populace* bývá nejčastěji používán pro množinu jednotlivých nezávislých řešení úlohy, ze kterých jsou vytvářeny nové kombinace. Podle velikosti udržované populace se metaheuristiky většinou dělí na algoritmy s *jedním* a nebo s *více řešeními*. Velikost udržované populace hraje významnou roli ve variabilitě nově produkovaných řešení. Algoritmy s jedním hlavním jedincem, který je postupně modifikován, aby nabýval stále lepších hodnot účelové funkce, jsou většinou zaměřeny na lokální prohledávání stavového prostoru. Mezi typické příklady patří např. *Tabu search* nebo simulované žíhání. Míra jejich lokální povahy je odvislá od způsobu vytváření nových řešení. Algoritmus simulovaného žíhání na svém počátku prohledává stavový prostor s možností poměrně významných odskoků od aktuálního řešení, a tak by z tohoto pohledu nemusel být mezi lokální metaheuristiky zařazován, avšak při pohledu na přenos dědičné informace jej za takový považovat lze. Při dostatečně velkém stavovém prostoru jsou nově vzniklá řešení v průměru nejpodobnější právě aktuálnímu řešení a všem ostatním by měly být podobné méně.[71] Při udržování větších populací se předaná informace v nových řešeních více mísí a během několika málo iterací algoritmu mohou původní informace naprosto vymizet. Velké populace různorodých řešení v genetických algoritmech rekombinují jednotlivé geny v podstatě náhodným způsobem a zachovávání informací (např. zachování daných hodnot v různých proměnných současně) probíhá až při menší variabilitě populace.

Nově vytvářená řešení tedy v sobě většinou kombinují informace z celé populace. Do nových řešení z populací s jediným jedincem se dostává přenesená informace přímo, kdežto u větších populací se přenáší postupně, až po dostatečném promíchání hodnot mezi sebou. Velikostí populace se tedy přímo ovlivňuje rychlost, s jakou jsou vstřebávány nové informace z prohledaných bodů. Malé populace (např. algoritmus *Harmony search* s typicky jednotkami řešení) jsou výrazně citlivější na nově prohledávané body stavového prostoru a mnohem rychleji přenáší tyto informace do nových řešení než např. *umělé imunitní systémy*, kde se běžně pracuje s desítkami až stovkami různých řešení zároveň. Malé populace tedy mohou rychleji konvergovat avšak i k neoptimálním hodnotám, protože díky nižší variabilitě nejsou schopny změnit falešně vytypovaný směr. Samostatným případem jsou algoritmy jako je *Estiamtion of distribution*, kde není žádná udržovaná populace a všechny získané informace jsou přímo transformovány do parametrů distribuční funkce. Takové algoritmy lze přeneseně považovat také za populační, protože mají obdobné chování (pozvolné vstřebávání informací, ...) a generování nových řešení je nepřímo závislé na více jedincích současně.

Velikost populace nemusí být v průběhu algoritmu stále stejná, ale může se různě vyvíjet. Nejběžnějším případem je, že se populace nijak nemění a počty nově vytvářených řešení jsou stále stejné. Například standardní genetické algoritmy uchovávají stálou populaci s konstantní velikostí a stejně tak se nemění ani velikost dočasných populací tvořených nově vzniklými řešeními. Obdobně je tomu u *Harmony search*, *Tabu search*, hejnových algoritmů nebo u dife-

[71] Není potřeba nijak explicitně definovat míru *podobnosti* a lze ji bez problému chápat ve smyslu běžných metrik.

renciální evoluce. Jinak je tomu u algoritmu *CLONALG*, který uchovává v populaci stále stejný počet jedinců, ale počet dočasných se mění podle aktuálních hodnot účelových funkcí. Algoritmus *Angles & Mortals* mění počet jedinců nepředvídatelným způsobem závislým na náhodném potkávání andělů se smrtelníky. Teoreticky si lze představit libovolný vývoj velikosti populace v čase, například postupně rostoucí, klesající, oscilující nebo určené aktuálním stavem algoritmu, ale takové případy jsou mezi publikovanými metaheuristikami spíše výjimečné.

Ať už se jedná o populaci libovolné velikosti, tak významné rozdíly mezi jednotlivými metaheuristickými přístupy lze spatřovat v množství modifikovaných prvků během jedné iterace algoritmu. Stálá populace se používá pro generování nových řešení, a tak rychlost její obměny hraje v algoritmu významnou roli. Čím větší procento jedinců se obmění, tím méně aktuální informace je zachováno a algoritmus se tak může rychleji pohybovat směrem k lepším (i horším) hodnotám. U algoritmů jako jsou hejnové algoritmy se každou iterací změní hodnoty všech jedinců, ale díky nevelkým skokům se alespoň částečně zachovává informace v podobě geometrické blízkosti ve stavovém prostoru. Naproti tomu u genetických algoritmů se obměňuje pouze menší část s nejslabšími jedinci a velikost této části je závislá na kvalitách nalezených řešení. V počátku může docházet k obměnám celých populací a ke konci běhu algoritmu, kdy je pravděpodobnost nalezení kvalitnějších řešení významně nižší, se nemusí obměnit vůbec žádný prvek. Aby populace nezůstávala dlouho stejná, zajišťuje se pomocí mutace změna alespoň minimálního počtu jedinců. U diferenciální evoluce se změní v aktuální populaci vždy nejvýše jeden prvek, čímž se dosahuje rychlejších odezev algoritmu na získané informace. Speciálním případem jsou algoritmy s jediným udržovaným řešením, kde se u většiny z nich mění řešení při každé iteraci a pokud se nezmění, je algoritmus buď ukončen (*Hill climbing*), změněn prohledávací postup (*Variable Neighbourhood Search*) nebo jinak vyhodnocen nastalý stav.

Jelikož jsou v mnoha případech nově generovaná řešení závislá pouze na aktuální populaci, je nezbytné, aby byla složena z jedinců s vhodnými vlastnosti. Hlavní snahou algoritmů je udržovat v populaci co možná nejkvalitnější řešení, protože základní myšlenka, implicitně obsažená v každé metaheuristice, je založena na předpokladu, že z kvalitních jedinců lze odvozovat kvalitní potomky. Do populace jsou často ale navíc přidávána i řešení slabší (nebo bez předem známé hodnoty účelové funkce), aby jimi byly zajištěny některé další požadované charakteristiky. Předně jde o přísun nových informací, zabránění předčasné konvergenci a opouštění lokálních extrémů. Různé metaheuristiky volí různé způsoby nakládání se špatnými řešeními a podle toho vykazují specifické druhy chování. U algoritmu *Harmony search* se do nové populace zahrnou pouze ta řešení, která jsou lepší než některé z aktuální populace. Pokud je vygenerované řešení horší, je nevratně odstraněno a žádná nová řešení z něj odvozována nikdy nebudou. Stejně tak špatní jedinci neprodukují nová řešení ani v algoritmech *CLONALG*, *Differential evolution*, *Estimation of Distribuiton* a dalších. V genetických algoritmech je situace lehce odlišná, protože při každé iteraci se zároveň provede i mutace, která může významně snížit kvalitu jedince. Zmutovaný jedinec ale teoreticky může v další generaci produkovat nová řešení a vyřazen může být až nahrazením za nějaké lepší řešení.

V algoritmu *Angels & Mortals* se může stát, i když jen s malou pravděpodobností, že špatné řešení bude přežívat velmi dlouho a bude i produkovat stále nová řešení, což vede k následnému snížení kvality celé populace. Předcházet takovým situacím se algoritmus snaží definováním délky života všem jedincům a nucené mutaci v každé iteraci. Jelikož ale doba života může být náhodně prodlužována, je jistá šance, že se budou zbytečně prohledávat i neužitečné oblasti stavového prostoru. Umělé imunitní systémy k podobnému účelu používají strategii, kdy je odstraněno nejhorších n aktuálních řešení a následně jsou pak nahrazena za nová. Je tomu tedy trochu obráceně než u genetických algoritmů, kde je počet odstraněných řešení závislý na jejich kvalitě.

V některých metaheuristikách (např. u *Harmony search*, *Firefly algorithm*) je k produkci nových řešení použita celá aktuální populace, ale ve velkém množství ostatních algoritmů je každé nové řešení produkováno pouze určitým malým podvýběrem aktuálních jedinců. Jednotlivých přístupů k výběru těchto rodičovských jedinců je nespočet, ale v drtivé většině jsou pravděpodobnosti jejich výběru závislé na relativní kvalitě hodnot účelové funkce vůči ostatním jedincům v populaci. Naopak počet nově vyprodukovaných řešení je většinou na hodnotě účelové funkce nezávislý. Genetické algoritmy, jakmile vyberou dvojici rodičovských jedinců, je počet jejich potomků vždy stejný (nejčastěji jeden nebo dva potomci). Podobně je tomu i při generování okolí v *Hill climbing algoritmu*, což lze také považovat za produkci nových řešení s konstantním počtem potomků. Jednou z mála výjimek je algoritmus *CLONALG*, který produkuje počet řešení v závislosti na normované hodnotě účelové funkce vůči celé populaci.

5.3 Další významné vlastnosti metaheuristik

Kromě populace mají mnohé metaheuristiky i další struktury, které slouží k uchovávání informací získaných během prohledávání stavového prostoru. Některým algoritmům stačí pouze přístup k hodnotám jednorozměrných numerických parametrů, které vyžadují pro svůj běh. Příklady jsou *teplota systému* v simulovaném žíhání nebo čítač generací v genetických algoritmech. Významnou uchovávanou hodnotou je ukazatel na nejlepšího jedince v populaci nebo obecněji na libovolného jedince. Taková potřeba se často vyskytuje v hejnových algoritmech všeho druhu. *Tabu search* vyžaduje strukturu *tabu list*, což lze implementovat stejně jako kteroukoliv jinou populaci, pouze s výjimkou zajištění potřeby jejího řazení podle pořadí vkládání jednotlivých řešení. Speciální požadavky mají algoritmy bez stálé populace, například *Ant colony optimization*, kde je potřeba udržovat matici feromonových stop mezi jednotlivými hodnotami proměnných. Stejně tak i algoritmus *Estimation of Distribution* potřebuje reprezentovat odhadnuté rozdělení pomocí vhodné struktury. Pokud je rozdělení popsáno pouze parametrickým modelem, je pro algoritmus dostatečné mít přístup pouze k jednorozměrným numerickým proměnným, pokud je ale zvolená reprezentace složitější (např. Bayesovské sítě), jsou potřeba obecné grafové nebo obdobné struktury.

Lokální optima

Kvůli omezené velikosti paměti metaheuristiky se může stávat, že jsou neustále generována řešení z malé oblasti obsahující lokální extrém. Metaheuristiky se v takovém případě často nedokáží efektivně vzdálit od aktuální pozice a konvergují stále do stejného místa. V takovém případě je potřeba vnést do generovacího algoritmu novou informaci a soustředit výpočetní síly i na další slibné oblasti. Metaheuristiky obecně ale nemají o stavovém prostoru žádné globální informace a nemohou tedy nijak zjistit, jestli nalezená oblast obsahuje globální optimum, anebo bylo nalezeno pouze jedno z mnoha lokálních. Každý metaheuristický koncept tedy může volit vlastní přístup k rozhodování o šířce prohledávané oblasti. Běžné algoritmy většinou volí přístup průběžného přísunu nových informací do aktuálních populací. Genetické algoritmy stále mutují několik proměnných v malém procentu jedinců, umělé imunitní systémy průběžně nahrazují neužitečné jedince pomocí kompletně nových řešení. Algoritmus *Very large scale neighbourhood* ([115]) s každou iterací generuje pro aktuální bod velké množství okolních řešení, čímž si snaží zvýšit šanci útěku z lokálních extrémů pomocí prozkoumávání rozsáhlejších oblastí. Podobný myšlenkový základ volí i diferenciální evoluce, která vytváří řešení interpolací i extrapolací více různých řešení, čímž vzniká velká variabilita a šance na opuštění lokálního extrému tak roste s ní. Za velký problém je považováno uváznutí v lokálním optimu algoritmem *Estimation of Distribution*, protože odhadované rozdělení většinou popisuje pouze jednu specifickou oblast stavového prostoru a nedokáže generovat body s jiným rozdělením. Vhodnou cestou ke změně prohledávaných oblastí mohou být podmíněné odskoky, tedy akce, které se uskutečňují pouze pokud algoritmus stagnuje a delší dobu nedosahuje žádná zlepšení v hodnotách účelové funkce. Metaheuristika *Variable neighbourhood* v takovém případě obměňuje způsob generování okolí aktuálního bodu postupným zvětšováním jeho poloměru. Při stagnaci lze také zvyšovat například počet mutovaných jedinců, nebo prostě restartovat celý algoritmus z nových počátečních populací. Stejné metody, které jsou určeny k opuštění lokálních extrémů, jde většinou chápat i jako metody pro zvyšování (udržování) dostatečné variability. Obě problematiky mají nemálo společného a jejich řešení tak jdou spolu ruku v ruce.

Ukončovací kritéria

Absence globálních informací o řešeném problému vede k problému určení vhodného okamžiku pro zastavení běhu metaheuristiky. V obecném případě nelze nijak určit, jestli je ještě užitečné dále prohledávat stavový prostor a jestli tedy ještě mohou existovat řešení s lepší hodnotu účelové funkce. Velmi často se volí přístup, kdy se jednotlivé iterace algoritmu opakují tak dlouho, dokud uživateli nedojde trpělivost. Přičemž za míru trpělivosti lze považovat například počet iterací algoritmu, počet prohledaných řešení, maximální akceptovatelný čas strávený výpočtem apod. U některých typů úloh lze odhadnout optimum (např. pomocí aproximačních metod nastíněných v kapitole 2.5) a tedy ukončovací podmínky je mohou patřičně zohledňovat. Protože u metaheuristik nelze zaručit výsledné vlastnosti, mohou být používány nejen pro optimalizaci,

ale i pouze pro zlepšování známých řešení. Takový přístup lze transformovat do ukončovacích podmínek pomocí sledování nejlepší nalezené hodnoty účelové funkce tak, že se algoritmus ukončí, jakmile je poprvé dosaženo požadovaného zlepšení. Pokud je algoritmus navržen způsobem, že může uváznout v lokálním extrému, je vhodné sledovat maximální zlepšení účelové funkce během jedné iterace a pokud je menší než nějaké stanovené ε, pak je buď restartován nebo ukončen. Uváznutí v lokálním optimu tedy nemusí být nutně považováno za negativní jev, ale spíše jako vhodný indikátor pro změnu prohledávané oblasti. Místo sledování změny účelové funkce se ke stejnému účelu používají i různé míry homogenity v populaci nebo měření maximálních vzdáleností mezi jednotlivými řešeními. V publikacích je ale zdaleka nejčastěji specifikace ukončovací podmínky vynechána úplně a autoři se spokojí s pouhým obecným konstatováním `"while not (stop condition)"`.

Řídicí proměnné metaheuristik

Běh metaheuristiky, stejně jako většina ostatních algoritmů, je řízen logikou rozhodující na základě hodnot proměnných uložených v paměti počítače. Metaheuristiky využívají velké množství řídicích proměnných, jejichž hodnoty mohou pocházet z několika rozličných zdrojů a stejně tak mohou i velmi různorodě ovlivňovat běh algoritmu. Konkrétní hodnoty proměnných mohou měnit chování algoritmu a vhodnými změnami jej lépe přizpůsobovat pro řešení zadaného problému. Rozhodovací proměnné lze řadit do následujících tří skupin.

- Velmi charakteristickou skupinou jsou *náhodné generované hodnoty*. Mnoho výzkumníků považuje stochasticitu za klíčovou vlastnost metaheuristik bez které si je nelze ani představit (viz definice v kapitole 3.1). Náhodná čísla slouží k adaptivitě algoritmů a omezení opakování jednoho neúčinného vzorce chování. Mezi nejčastější místa, kde se náhodná čísla v metaheuristikách využívají, patří generování nových řešení nebo úpravy stávajících, tedy typicky při vytváření počátečních populací. Drobnější úpravy řešení jsou také často řešeny pomocí náhodných čísel, např. mutace v genetických algoritmech nebo geometrický posun nejlepšího jedince v hejnových algoritmech. Přímý vliv mají u genetických algoritmů naopak náhodná čísla ve výběrech rodičovských jedinců nebo v jejich následném křížení. Každý jedinec v populaci má definovanou pouze jistou pravděpodobnost, se kterou bude ke křížení vybrán, a proto není předem dané, jaká data a kam budou v průběhu výpočtu předávána. Konkrétní běh metaheuristiky je tak významně podmíněn vygenerovanými náhodnými čísly, díky čemuž získává velmi robustní vlastnosti vzhledem k nepředpokládaným tvarům účelové funkce. Náhodná čísla mohou být použita v mnoha dalších částech algoritmů (počet vyprodukovaných jedinců - *CLONALG*, zahrnování vygenerovaných řešení do aktuální populace - simulované žíhání, náhodné prohazování prvků mezi populacemi - *Shuffled frog leaping*, ...), nebo mohou tvořit samotnou základní myšlenku prohledávání, jako je tomu například u algoritmu *Estimation of distribution* a jeho variant.

- Častá možnost ovlivňování běhu algoritmů je za pomoci *nastavitelných parametrů*, které algoritmus za svého běhu využívá a řídí pomocí nich své chování. Mezi nejtypičtější příklady patří nastavitelná velikost populace nebo pravděpodobnost výběrů prvků pro mutování, křížení apod. Stejně tak se nastavují i počty produkovaných řešení a hlavně maximální počty iterací a podobné ukončovací podmínky. Nastavení těchto charakteristik má relativně přímý dopad na chod algoritmu a lze tak poměrně dobře odhadovat jeho výsledné chování. V metaheuristikách se ale často objevují i parametry, jejichž vliv není úplně jasný a korektní nastavení vyžaduje vhodný experimentální přístup. Mezi takové parametry patří různé neurčité konstanty vstupující do výpočtu nových řešení v hejnových algoritmech (např. koeficient svítivosti u *Firefly algorithm* nebo koeficienty echolokace u *Bat algorithm* apod.). Jelikož chování algoritmu je silně závislé na konkrétní optimalizované účelové funkci, je pro ni vhodné experimentálně ověřovat všechny parametry a nastavovat jim tak nejvhodnější hodnoty. Velká variabilita účelových funkcí zapříčiňuje i velkou variabilitu vlivů jednotlivých parametrů a získané chování tak nemusí vždy odpovídat očekávání. Autoři většinou zároveň s algoritmem publikují i doporučené rozsahy hodnot pro všechny parametry, ale k jejich získání bývají použity především testy na klasických úlohách a pro reálné úlohy tak nemusí být jejich hodnoty nejvhodnější.

- Třetí významnou skupinou řídicích proměnných v metaheuristikách jsou *vypočítané hodnoty*. Jako nejdůležitější vypočítaná (vytvořená) hodnota proměnné je beze sporu samotná reprezentace řešení, tedy např. hodnota genů v genetickém algoritmu apod. Hodnoty v těchto proměnných většinou neovlivňují běh algoritmu ve smyslu toku dat, ale pouze vstupují do tvorby nových hodnot. Mezi výjimky patří algoritmy, které vybírají řešení pro produkci nových na základě jejich geometrické polohy, například pomocí shlukování dat nebo korelace souřadnic (např. [116]). Druhým výrazně nejdůležitějším typem vypočítaných hodnot proměnných jsou hodnoty účelové funkce pro jednotlivá řešení. Protože tyto hodnoty jsou většinou jediným zdrojem informací o řešené úloze, musí nutně ovlivňovat chod algoritmu v celé své šíři. Pokud by tomu bylo naopak a algoritmus by neměnil tok dat podle zjištěných hodnot účelových funkcí, znamenalo by to, že algoritmus buď funguje zcela náhodně (Monte Carlo), nebo naopak pro všechny úlohy naprosto identicky. Obě možnosti jsou v přímém rozporu se smyslem metaheuristik, tedy se snahou *inteligentně* prohledávat stavový prostor. Hodnoty účelových funkcí vstupují do rozhodovacích procesů buď přímo, nebo v transformovaných podobách (např.*fitness function* v genetických algoritmech nebo *normovaná účelová funkce* v algoritmu CLONALG). Mezi nejobvyklejší části algoritmů, které přímo závisí na těchto hodnotách, patří výběr jedinců pro produkci nových řešení. V genetických algoritmech, stejně jako v mnoha dalších metaheuristikách, je pravděpodobnost výběru jedince přímo úměrná vypočítané hodnotě účelové funkce a její vliv je tedy snadno rozpoznatelný. Na základě hodnot účelových funkcí se mohou řadit populace, určovat délky života, stanovovat ukončovací podmínky a nebo podmíněné akce, jako např. v algoritmu *opt-Ainet*, kde se za podmínky nerostoucí průměrné hodnoty účelové funkce v populaci pro-

vede její významná obměna. V některých algoritmech je role účelové funkce velmi omezená, což se týká i algoritmu *Differential evolution*, který využívá vypočítané hodnoty pouze k rozhodování o zahrnutí nalezeného řešení do aktuální populace.

Dynamický výpočet hodnot řídicích proměnných je hlavním rysem i tzv. *adaptivních metaheuristik*. V těchto metaheuristikách mohou být všechny výše zmíněné konstantní řídicí proměnné upravovány na základě aktuálního chování algoritmu. Dynamickými změnami parametrů lze docílit vhodného ovlivnění strategie prohledávání, a tím efektivnějšího využití výpočetních zdrojů.

Konvergence

Konvergence je prokazatelná schopnost algoritmu nalézt optimální řešení. Jinými slovy, konvergentní metaheuristika zaručuje, že se nezacyklí ani neuvázne v žádném bodě před nalezením optima. V kontextu s *No Free Lunch teorémem* to znamená, že konvergentní algoritmus musí postupně projít celý stavový prostor úlohy, protože neexistuje žádné jiné pravidlo, které by obecně zajišťovalo ověření dosažení optima. Konvergence se tedy dá dokázat schopností kompletně projít stavovým prostorem, nebo se musí omezit na nějakou konkrétní uchopitelnou třídu účelových funkcí (spojité, kladné, ...). Autoři metaheuristických algoritmů se většinou této vlastnosti příliš nevěnují a spokojí se s experimentálním ověřením jejich funkčnosti. I přesto, že většina algoritmů nemá konvergenci řádně prokázanou, jsou hojně používány a jejich výsledky dostatečně naplňují očekávání.

Některé základní verze jednodušších algoritmů mají rozumně vyjádřitelnou alespoň pravděpodobnost nalezení optimálního řešení. Tato pravděpodobnost se ale velmi rychle snižuje s rostoucí velikostí úlohy (viz např. algoritmus *Harmony search* [43]) a pro praktické účely tak není příliš užitečná.

Metaheuristiky nejsou vytvářeny ani navrhovány za účelem zajištění nalezení skutečného optima, takže neexistence zajištěné konvergence nemusí být na obtíž. Velmi vítanou naopak může být v některých aplikacích pouze lokální konvergence, která zaručuje dostatečné prohledávací schopnosti v blízkém okolí. Za lokálně konvergující metaheuristiky lze považovat všechny, kterým hrozí uváznutí v lokálních extrémech, tedy např. *Hill climbing*, *CLONALG* apod.

Z pohledu návrhu metaheuristik je důležité sledovat dobu přežívání nejlepších řešení. Jednotlivé algoritmy se liší v přístupu k nejlepším nalezeným řešením. Některé jej nechávají v aktuální populaci během celého výpočtu a u jiných naopak hrozí, že bude za běhu změněno, nebo dokonce odstraněno. Na konci výpočtu se zachovává nejlepší řešení u metod jako *Tabu search*, ale u simulovaného žíhání to už pravda být nemusí. Stejně tak poslední generace v genetických algoritmech by teoreticky mohla obsahovat nejlepší nalezené řešení, avšak hrozí, že jej mutace náhodně změní a zhorší tak i jeho kvalitu. Tím, že algoritmus *Estimation of Distribution* neuchovává žádná řešení v populaci, nemůže uchovávat ani to nejlepší a všechna

prozkoumávaná řešení ihned maže. Pro zapamatování nejlepšího řešení se tedy používá (často bez explicitního zmínění) automatické uchovávání nejlepšího jedince, který se hledá ihned po výpočtu příslušných účelových funkcí.

5.4 Co z toho plyne?

Přístupy jednotlivých metaheuristik se od sebe mohou velmi lišit. Neexistuje jediné obecné schéma, které obsáhne všechny popsané možnosti a přitom popisuje pouze a jen právě metaheuristiky. Rozdíly jsou patrné již v samotných základech algoritmů, tedy ve způsobu reprezentace řešení nebo v určování jejich kvality. Následně se liší i způsoby odvozování nových řešení a předávání informací získaných předchozím prohledáváním. Ne všechny metaheuristiky jsou určeny ke stejnému účelu (lokální vs. globální apod.), což obecné uchopení metaheuristik samozřejmě ztěžuje. Nejzákladnější princip metaheuristik je ale stále stejný, a to vyzkoušet velké množství různých řešení a vybrat z nich to nejlepší. Hlavním rozdílem oproti zcela náhodnému prohledávání je, že metaheuristiky implicitně předpokládají jistou souvislost mezi reprezentací řešení a hodnotou jeho účelové funkce. Většina odlišností tedy pramení z rozdílných předpokladů o struktuře účelové funkce a o významu jednotlivých získaných informací v průběhu prohledávání.

Z implementačního pohledu jsou většinou metaheuristiky chápány jako modifikující se množina řešení. Mezi klíčové charakteristiky jednotlivých přístupů, které definují rozdíly mezi jednotlivými koncepty, patří

- velikosti množiny řešení,
- doba, po kterou může jedno řešení přímo odvozovat nová řešení,
- počet prvků (rodičů) přímo podílejících se na produkci nových řešení,
- způsob uchovávání získaných informací,
- způsob předávání uchovaných informací,
- druh stochasticity,
- počet generovaných řešení

a několik dalších (zmíněných v předchozích podkapitolách) v závislosti na obecnosti pojmu metaheuristika.

Na základě svého konceptu trpí každá metaheuristika i svými nedostatky. Některé algoritmy vznikají pouze jako vylepšení stávajících metod k překonání jejich problémů, avšak z *No Free Lunch teorému* plyne, že nikdy nemůže překonat všechny své nedostatky. Uživatelé metaheuristik si proto musí být vědomi jejich omezení a neočekávat nesplnitelné. Jako nejčastější problémy jsou uváděny předčasná konvergence[72], nedostatečná šíře prohledávané oblasti, možnost generování stále stejných řešení nebo přílišná variabilita a neschopnost se ustálit. Specifické

[72] Příliš rychlá konvergence do lokálního extrému bez možnosti dále prohledávat stavový prostor.

problémy mohou mít i metaheuristiky pouze s určitými typy účelových funkcí. Například algoritmus *Differential evolution* má takové problémy se zašumělými funkcemi, kdežto konvenčním evolučním algoritmům tolik obtíží nezpůsobují [117]. Vhodnou modifikací ale lze diferenciální evoluci obměnit a schéma prohledávání přizpůsobit i přidanému šumu (viz [118]). Metaheuristika musí být vždy poplatná svému účelu, který může být pro různé uživatele různý, a proto bude v další kapitole navržen obecný model dovolující kombinovat různé metaheuristické přístupy a ověřovat jejich podstatné vlastnosti.

6 Zobecněný model metaheuristik

Hlavním účelem metaheuristik je zprostředkovávat prohledávání stavového prostoru, a to pouze na základě hodnot účelových funkcí v prozkoumaných bodech. V předchozí kapitole bylo popsáno několik speciálních případů metaheuristik, jejichž principy mohou být vzájemně velmi významně odlišné. I přes velkou různorodost ale lze vypozorovat několik společných vzorců chování a využít je tak pro formulování obecného modelu.

Základním stavebním kamenem jakéhokoliv metaheuristického modelu musí být práce s dvojicemi typu (x, y), kde x je navštívený bod a $y = f(x)$ je zjištěná hodnota optimalizačního kritéria (účelové funkce). Z takto uspořádaných informací musí být metaheuristika schopna generovat nové body x, které mají *jistou šanci*[73] na lepší hodnotu y. Reálná výpočetní zařízení jsou navíc omezena limitovanou pamětí, a proto musí metaheuristika neustále rozhodovat, které páry (x, y) z paměti odstraní a které si ponechá pro další použití. Základními schopnostmi metaheuristických algoritmů tedy musí být

- schopnost vyhodnotit kvalitu konkrétního bodu x (tzv. vyhodnotit hodnotu účelové funkce),
- schopnost uchovat nebo zpracovat získané informace (dvojice (x, y)),
- schopnost rozhodnout o odstranění získaných informací z paměti,
- schopnost vygenerovat nová řešení na základě uchovaných informací.

Každá metaheuristika definuje vlastní způsoby zajišťování uvedených schopností. Při jejich jednotném popisu pomocí obecného modelu je lze snadno vzájemně kombinovat a využívat pro vytváření nových odvozených metaheuristik. Takové spojení několika principů různých metaheuristik dohromady bude dále nazýváno **hybridizací**.

Obecné modely

Myšlenka zobecnění existujících metaheuristik pod jeden koncept není nová a v odborných publikacích se tak již takové modely nepravidelně objevují řadu let. Například v [119] je vyvinut obecný popis určený primárně k lokálnímu prohledávání. Mimo základních verzí *Hill climbing* algoritmu, *Tabu search*, *Variable neighbourhood* algoritmu apod. jím lze popsat i genetické algoritmy a některé další populační metaheuristiky. Naopak model publikovaný v [120] se zaměřuje především na evoluční metaheuristiky, pod něž lze řadit jak genetické algoritmy, tak i *Ant colony optimization* nebo *Scatter search*. Mezi obecné modely, které si ale získaly větší pozornost patří výsledky publikované v [121] a popisující metaheuristiky následujícím pseudokódem

[73] Právě tento pojem je hlavním rysem metaheuristik - zlepšení hodnoty účelové funkce je negarantované a stojí pouze na heuristickém ověření.

```
inicializuj P pomocí externí procedury
while termination = FALSE do
    S <- OF(P)
    if |S|>1 then
        S' <- SCM(S)
    else
        S' <- S
    S'' <- IM(S')
    P <- IF(S'')
použij post-optimalizační proceduru na P.
```

Množina `P` je zde označována jako *Pool*, tedy množina jedinců reprezentující samostatná řešení, `IF/OF` jsou *input/output function* a zajišťují vstupy/výstupy z/do poolu. `SCM` je *Solution Combination Method*, která je použita pouze v případě, že je pro produkci nového řešení vybráno z poolu více než jedno řešení. `IM` je *Improvement method* vylepšující jedno předané řešení a již zmíněná funkce `IF` rozhodne o jeho následném začlenění do aktuální populace (poolu). Tímto popisným schématem lze modelovat velkou část metaheuristik, příkladem může být *Tabu search* nebo simulované žíhání, které mají $|P| = 2$, $|S| = 1$ a rozhodovací logika (adaptivní paměť a akceptační kritérium) je implementována v `IM`. Genetické algoritmy jsou definovány pro $|P| > 1$ a $|S| > 1$ a operátory křížení a mutace jsou zahrnuty v SCM, krok IM vůbec není využit. Další příklady i s obsáhlou diskuzí lze nalézt v [121].

Jiná základní myšlenka popisu metaheuristik byla publikována v [122] a nazvána *Adaptive memory programming* (AMP). Hlavní roli zde sehrává *paměť*, která slouží ke konstrukci nových řešení. Například u metody *Tabu search* je pamětí *tabu list*, u genetických algoritmů je to celá populace a u optimalizace mravenčí kolonií je pamětí matice feromonových stop. Pseudokódem je Adaptive Memory Programming v publikaci [122] popsaný jako

```
inicializuj paměť M
while termination = FALSE do
    pomocí paměti generuj nové řešení S
    vylepši S pomocí metody lokálního prohledávání
    vylepšené řešení S' použij k aktualizaci paměti M.
```

V uvedeném popisu lze vypozorovat mnoho shodných myšlenek s předchozím obecným modelem. Nejvýznamnější rozdíl mezi nimi je v tom, že *paměť* nemusí být tvořena jen populací řešení, ale může být v podstatě libovolnou datovou strukturou. Vyjadřovací silou se oba modely vzájemně vyrovnají, protože přidaná obecnost, která je u AMP docílena zavedením pojmu *paměti*, může být u předchozího modelu realizována přímo ve funkcích `IF, IM,` ...

Ze zcela jiného pohledu jsou metaheuristiky popisovány pomocí přístupu nazvaného *Agent Metaheuristics Framework* (AMF) [123]. Základem jsou zde *multiagentní systémy*, které udržují

skupinu vzájemně spolupracujících a interagujících objektů (*agentů*). Využit je zde tzv. RIO přístup, který každému agentovi definuje jeho *Roli*, mezi rolemi definuje vzájemné *Interakce* a vše dohromady sjednocuje *Organizace*. V popisu metaheuristiky jsou použity agenti čtyř rolí - *intensifier, diversifier, guide a strategist*[74]. Role intensifier a diversifier zajišťují prohledávání prostoru, a na rozdíl od AMP modelu jsou chápány separovaně. Intensifier se soustředí na slibné oblasti, kde je potřeba prohledávat prostor více důkladně, kdežto diversifier prohledává globálně celý stavový prostor. Role guide zprostředkovává interakci mezi intensifierem a diversifierem a určuje jim, které oblasti budou prohledávat. V AMP modelu této roli částečně odpovídá funkce *paměti*. Poslední rolí je strategist, která interaguje s agenty typu guide a zastupuje adaptivitu algoritmů. Vyhodnocuje tedy aktuální stav prohledávání a podle toho modifikuje parametry celé metaheuristiky. Tímto způsobem lze popisovat většinu běžných metaheuristik, například v *Tabu search* jsou role intensifier a diversifier spojené do jedné, která prohledává aktuální okolí a roli guide hrají funkce pracující s tabu listem a výběrem nového aktuálního řešení. V běžném Tabu search není role strategist zastoupena, jejím přidáním může vzniknout algoritmus *Reacitve search*, který adaptivně mění délku tabu listu. Genetické algoritmy mají roli intensifier reprezentovánu křížením jedinců, roli diversifier mutací a roli guide výběrem rodičů pro křížení.

Definování stávajících metaheuristik pomocí popsaných rolí lze jednotlivé agenty kombinovat a vytvářet nové metaheuristiky. Například *Coalition-based metaheuristika* [124] vznikla použitím AMF modelu a spojením s konceptem hyperheuristik.

V následujících kapitolách bude představen nový obecný model, který se od předchozích odlišuje nejen mírou obecnosti, ale i celkovým přístupem k pojmu metaheuristika. Metaheuristika zde není chápána jako rigidní osnova, která v každém kroku definuje smysl prováděné operace, ale naopak je v návrhu nechávána maximální volnost. Za existenci metaheuristik vděčíme především výpočetní síle dnešních počítačů, a tak i obecný model je vybudován s ohledem na rozumný způsob jejich počítačové implementace.

6.1 Koncepční model

Významným společným znakem většiny kanonických tvarů metaheuristik je, že jedno konkrétní řešení reprezentují jako jednu samostatnou entitu umístěnou ve vhodném prostoru. Většina výzkumníků si tak může představit metaheuristiku jako body rozmístěné ve stavovém prostoru, které se různě pohybují, vytváří a zanikají a vzájemně se ovlivňují. Tato představa se ukázala jako vhodná a intuitivní, a tedy i budovaný obecný model musí umět tuto představu jednoduše zprostředkovat. Za základní stavební kámen modelu musí být zvolena dostatečně obecná struktura, která je schopná přímočaře popisovat běžné metaheuristiky, ale zároveň nesmí nijak omezovat vývoj metaheuristik s odlišným přístupem.

[74] Pojmenování je ponecháno v původním anglickém jazyce

Klíč

Základní a nejobecnější datová struktura je entita nazvaná `klíč`. `Klíč` je abstraktní pojem, pouhé jméno, pod který se může schovat libovolná povolená hodnota. Termín `klíč` je zvolen kvůli významové podobnosti s databázovým klíčem. `Klíč` si lze představit jako jméno proměnné, do které se ukládají vhodné hodnoty. Ve vytvářené metaheuristice může `klíč` zastupovat jedno řešení, nebo i celou jejich skupinu, stejně tak zastupuje i jednotlivé proměnné a nebo pomocné hodnoty. `Klíč` jenom pojmenovává hodnoty, se kterými metaheuristika pracuje a na které se lze průběhu dotazovat.

Každý `klíč` je unikátní a mohou mezi nimi existovat různé vztahy. `Klíče` jsou primárně hierarchicky uspořádány, ale nejsou tím nijak omezeny. Jejich topologie (vzájemné vztahy) mohou být v podstatě libovolné.

Ke každému `klíči` může být připojena určitá hodnota, ale není to nezbytně nutné. `Klíč` může sloužit i jenom jako abstraktní pojmenování pro určitou skupinu hodnot. Pokud má `klíč` nějakou hodnotu přiřazenou, budeme jej psát jako uspořádanou dvojici

$$q_i = (k_i, v_i)\,, \text{ pro } i \in \mathbb{N},$$

kde k_i je jedinečný klíč a v_i je jeho přiřazená hodnota. Vyhledáním daného klíče k_i tedy může být vždy získána i jeho přiřazená hodnota v_i. Soubor všech vytvořených párů q_i reprezentuje *databázi* všech hodnot používaných a sdílených za běhu metaheuristiky.

Základní typy hodnot v, které lze ke `klíči` k přiřazovat jsou

- `klíč` - mohou tak být vytvářeny páry (k_1, k_2),
- `operátor` - základní funkční mechanismus metaheuristického modelu (viz dále),
- datová hodnota - jde o jakýkoliv běžný datový typ, který metaheuristika vyžaduje. Může se jednat o přiřazení čísel, znaků, textových řetězců apod.
- speciální hodnoty - jsou hodnoty využívané k řízení běhu metaheuristiky. Jedná se pouze o hodnoty `LOOP_STOP` a `NONE` (viz dále).

Struktura všech `klíčů` je tvořena stejnými pravidly. `Klíče` jsou hierarchicky uspořádány postupným navazováním základních datových typů. K již vytvořeným `klíčům` lze snadno přidávat nové podklíče, nebo je spolu kombinovat. Pro existující `klíč` k_1 lze vytvořit například následující klíče

$$k_2 = [k_1, x]$$
$$k_3 = [k_1, y]$$
$$k_4 = [k_2, k_3] = [k_1, x, k_2, y],$$

kde `klíč` k_2 je hierarchicky pod klíčem k_1 a jeho jméno je rozšířeno o hodnotu x. Obdobně `klíč` k_3 je hierarchicky pod klíčem k_1 a je rozšířen o hodnotu y. Pokud by `klíč` k_1 reprezentoval jedno řešení (bod ve stavovém prostoru), tak `klíče` k_2 a k_3 mohou reprezentovat jeho souřadnice na dvou osách x a y.

`Klíč` k_4 je vytvořený spojením `klíčů` k_2 a k_3 a reprezentuje jejich vzájemný vztah. Přiřazením hodnoty k takovému `klíči` lze vytvářet v podstatě libovolné relace, a to dokonce i fuzzy relace nebo grafové struktury s definovanými váhami. Uspořádané dvojice

$$q_1 = ([k_1, k_2], 0.4)$$
$$q_2 = ([k_1, k_3], 0.8)$$

tak mohou reprezentovat například fuzzy relaci prvků k_1 a k_2 s hodnotou 0.4 a nebo hranu v grafu mezi vrcholy k_1 a k_3 s váhou 0.8. Důležité je ale si uvědomit, že prvek $[k_1, k_2]$ je pouze novým `klíčem`, ke kterému lze přistupovat jako ke kterémukoliv jinému `klíči` (tzn. vytvářet mu nové podklíče, řetězit apod.)

K intuitivnímu uchopení navrhovaných metaheuristik je velice důležité snadno pracovat s celými skupinami `klíčů` zároveň. Uspořádaná posloupnost `klíčů` by tak mohla být jedním ze základních datových typů modelu, ale jelikož ji lze snadno implementovat za pomoci ostatních typů, tak je definována pouze jako doplňková struktura. Klíč k_s reprezentující skupinu je tedy definován jako

$$q_0 = ([k_s, n], 4)$$
$$q_1 = ([k_s, 1], v_1)$$
$$q_2 = ([k_s, 2], v_2)$$
$$q_3 = ([k_s, 3], v_3)$$
$$q_4 = ([k_s, 4], v_4),$$

kde dvojice q_0 udává celkovou délku skupiny a každá další dvojice přiřazuje danému indexu i hodnotu v_i. V tomto případě není nutné, aby klíč k_s měl přiřazenou nějakou konkrétní hodnotu a *fyzicky* existovat v databázi mohou pouze jeho podklíče.

V dalším textu bude tato struktura používána ve zkrácené formě

$$q_s = (k_s, \langle v_1, v_2, v_3, v_4 \rangle).$$

Operátor

Základním funkčním prvkem v hybridním metaheuristickém modelu je entita nazvaná `operátor`. Myšlenkově jde o koncept blízký funkcím ve funkcionálním programování[75] popř. i v

procedurálním programování[76]. `Operátorem` je tedy rozuměn ucelený blok instrukcí, které jsou vykonávány nad předanými `klíči`. Každý `operátor` má přiřazený `klíč`, který obsahuje vstupní hodnoty a `klíč`, ke kterému se přiřadí zpracované výstupní hodnoty. Výstupem je vždy pouze jeden `klíč`, kdežto vstupních `klíčů` může být více. Ve shodě s běžnou syntaxí budeme pro zjednodušení psát

$$k_{\text{out}} = \text{op}(k_{\text{in}}^1, k_{\text{in}}^2, \ldots, k_{\text{in}}^m),$$

což znamená, že vstupem do operátoru je skupina `klíčů` $k_{\text{in}}^1, k_{\text{in}}^2, \ldots, k_{\text{in}}^m$ a výstup je přiřazen ke `klíči` k_{out}.

Jednotlivé instrukce v `operátoru` mohou být buď další `operátory` a nebo přímé volání nízkoúrovňových funkcí poskytované programovým prostředím implementovaného modelu (matematické funkce, uživatelsky definované funkce, ...). Každý `operátor` má neomezený přístup k databázi uložených hodnot, tedy ze znalosti klíče k může kdykoliv získat jeho přiřazenou hodnotu v. Stejně tak mohou `operátory` data i ukládat, tedy přiřazovat `klíčům` k jejich hodnoty v.

Běh metaheuristiky je popsán pouze jako posloupnost po sobě jdoucích operátorů s definovanými vstupními a výstupními `klíči`.

$$\begin{aligned} k_1 &= \text{op}_1(k_{\text{in}}) \\ k_2 &= \text{op}_2(k_1) \\ k_3 &= \text{op}_3(k_1, k_2) \\ k_4 &= \text{op}_4(k_3) \\ &\ldots \end{aligned}$$

Z Churchovy-Turingovy teze plyne, že pro každý algoritmus lze vytvořit ekvivalentní Turingův stroj (tzn. algoritmus je *naprogramovatelný*) pokud je použit programovací jazyk s alespoň jednou z následujících vlastností

- *neomezená rekurze* - možnost (teoreticky) neomezeného volání funkce sebe sama,
- *cyklus while* - opakování bloku kódu tak dlouho, dokud nepřestane platit definovaná podmínka,
- *podmíněný skok* - výsledek vyhodnocení definované podmínky rozhodne o místě v programu, kterým se bude pokračovat.

---

[75] přeneseně také v lambda kalkulu - výpočetním modelu funkcionálního programování.

[76] V procedurálním programování jsou funkce chápány jako prostředek pro zjednodušení psaní programu a teoreticky se bez nich lze obejít. Ve funkcionálním programování tvoří funkce nedílnou součást jeho paradigmatu, a proto je toto pojetí bližší popisovanému pojmu `operátor`.

Aby byl i metaheuristický model schopný popsat jakýkoliv algoritmus, musí mít také alespoň jednu z uvedených vlastností. Pouhým definováním sledu jednotlivých operátorů by nebylo možné realizovat komplexní řídicí logiku.

K získání potřebných vyjadřovacích schopností je koncept `operátorů` rozšířen o možnost opakování. Každý `operátor` tedy definuje posloupnost instrukcí a při jejich dokončení jsou všechny znovu opakovány. Opakování probíhá tak dlouho, dokud nějaká z instrukcí nepřiřadí ke svému výstupnímu klíči k_{out} speciální hodnotu `LOOP_STOP`, která zapříčiní okamžité ukončení operátoru a pokračování jeho následníkem. Pokud je tedy `operátor` op_A definován např. jako posloupnost následujících `operátorů`

$$\begin{aligned}\text{op}_A{:=}&\\ &k_1 = \text{op}_1(k_{\text{in}})\\ &k_2 = \text{op}_2(k_1)\\ &k_3 = \text{op}_3(k_1, k_2)\\ &k_4 = \text{op}_4(k_3),\end{aligned}$$

přiřazují se výstupním `klíčům` při postupném vyhodnocování určené hodnoty a po dokončení `operátoru` op_4 se pokračuje znovu `operátorem` op_1. Pokud v dalším běhu přiřadí `operátor` op_2 ke svému výstupnímu `klíči` k_2 hodnotu `LOOP_STOP`, operátory op_3 a op_4 se přeskočí a pokračuje se následujícími operátory (pokud žádné v posloupnosti již nejsou, běh metaheuristiky se ukončí). Jako výstup `operátoru` op_A je použita poslední platná přiřazená hodnota před přiřazením hodnoty `LOOP_STOP`. V uvedeném příkladě by k výstupnímu `klíči` k_A `operátoru` op_A byla přiřazena hodnota z `klíče` k_1.[77] Operátor tedy může vracet buď `klíč`, nebo hodnotu `LOOP_STOP` a nebo speciální hodnotu `NONE`, která značí prázdný výstup. Tímto jednoduchým principem lze snadno realizovat *cyklus while*, a tím splnit jednu z výše uvedených podmínek pro úplný programovací jazyk. V každém `operátoru` lze například definovat `operátor` op_T, který bude sloužit pouze k vyhodnocování ukončovací podmínky a v případě jejího splnění vrátí hodnotu `LOOP_STOP` a v případě nesplnění naopak vrátí hodnotu `NONE`.

$$\begin{aligned}\text{op}_B{:=}&\\ &k_1 = \text{op}_1(k_1)\\ &k_2 = \text{op}_\text{T}(k_1)\end{aligned}$$

Zde se bude `operátor` op_1 vykonávat tak dlouho, dokud jej `operátor` op_T neukončí. Za povšimnutí stojí, že výstupní `klíč` `operátoru` op_1 je stejný jako jeho vstupní `klíč` a může tedy opakovaně zpracovávat vstup, dokud není podle předdefinovaných kritérií v op_T.

---

[77] Tento přístup je sice dostatečně obecný, ale z uživatelského hlediska není nejpřívětivější. V následující kapitole 7 bude vhodně přizpůsoben.

Cyklus *while* je přirozeným vyjadřovacím prvkem v metaheuristických algoritmech a je i hlavním koncepčním rozdílem oproti funkcionálnímu programování, kde tuto úlohu sehrává *rekurze*. Protože cyklus *while* je dostačující logickou strukturou pro naprogramování libovolného algoritmu, lze jím vyjádřit i ostatní *standardní* procedurální techniky jako například cyklus *for* nebo následující větvení běhu pomocí *if - then - else*:

$$\begin{aligned} \mathrm{op}_{\text{if-then}} := \\ k_1 &= \mathrm{op}_{\text{if}}(k_{\text{in}}) \\ k_2 &= \mathrm{op}_{\text{then}}(k_1) \\ k_3 &= \mathrm{op}_{\text{exit}}(), \end{aligned}$$

kde op_{if} nejdříve vyhodnotí zadanou podmínku a pokud je splněna, tak vrátí `NONE` a vykoná se `operátor` $\mathrm{op}_{\text{then}}$ a následně $\mathrm{op}_{\text{exit}}$, který vrací `LOOP_STOP` vždy. V případě nesplnění podmínky se vrátí `LOOP_STOP` a `operátor` se ukončí. V prvním případě je návratová hodnota rovna k_2 v druhém případě je `NONE`, protože se do ní ještě nestihlo nic uložit. Rozšíření na `operátor` $\mathrm{op}_{\text{if-then-else}}$ lze docílit

$$\begin{aligned} \mathrm{op}_{\text{if-then-else}} := \\ k_1 &= \mathrm{op}_{\text{if-then}}(k_{\text{in}}) \\ k_2 &= \mathrm{op}_{\text{is-not-none}}(k_1) \\ k_3 &= \mathrm{op}_{\text{else}}(k_{\text{in}}) \\ k_4 &= \mathrm{op}_{\text{exit}}(), \end{aligned}$$

kde `operátor` $\mathrm{op}_{\text{is-not-none}}$ otestuje jestli byla vrácena nějaká hodnota. Když je `klíči` k_1 přiřazena hodnota `NONE`, tzn. podmínka v *if* je neplatná, vrátí se `NONE` a provede se operátor $\mathrm{op}_{\text{else}}$. V opačném případě se `operátor` ukončí.[78]

Interpretace modelu

Důvodem, proč je metaheuristický model vybudován popsaným způsobem, je jeho hlavní smysl, tedy snadná hybridizace. Koncept `operátorů` vytváří jednotné abstraktní pojetí řídicí logiky a jejích jednotlivých procedur. V kontextu typických metaheuristik, např. genetických algoritmů, zastupuje `operátor` např. způsoby křížení chromozomů, výběr populace nebo ukončovací podmínku. Standardizací těchto operátorů lze docílit přímočaré hybridizace, tedy kombinování operátorů napříč různými metaheuristickými koncepty.

[78] Větvení běhu metaheuristiky lze provádět různými způsoby a popsaný způsob by nefungoval kdyby $\mathrm{op}_{\text{then}}$ mohl vracet i hodnotu `NONE`.

Pojem `klíč` naopak vytváří abstrakci pro datové hodnoty a zároveň i pro myšlenkové celky použití v metaheuristikách. V případě genetických algoritmů mohou `klíče` reprezentovat jednotlivé chromozomy, jejich geny i celé populace. Pro úspěšnou hybridizaci tedy stačí, aby byla tato data reprezentována stejnými strukturami, a tím i využívána různými `operátory`.

6.2 Matematický model hybridního metaheuristického modelu

Pro snadnější implementaci a srozumitelnost konceptu je popsaný hybridní metaheuristický model převeden do rigorózní terminologie. Tento druh popisu usnadňuje a zpřesňuje použité myšlenky. Koncepční ani následující matematický model nejsou přímo vhodny k implementaci pomocí programovacích jazyků. Jejich smysl je podávat základní, ale zároveň úplný, popis hybridního metaheuristického modelu. Pro implementaci se pak může zvolit nadstavba vybudovaná na popsaném základním modelu, která usnadní a zpříjemní vytváření nových algoritmů.

Klíč

`Klíč` tvoří základní stavební kámen modelu a slouží k pojmenování zpracovávaných hodnot i jejich celých skupin. Klíče mohou být složeny z libovolných prvků, ale nejsnáze se představují ve formě textových řetězců. Není však nutno se omezovat pouze na množinu písmen, ale lze využít libovolnou obecnou množinu, na které je možné definovat relaci uspořádání.

Pro definitorické účely tedy lze uvažovat množinu A jako základní *abecedu*, ze které jsou složena *slova* využitelná pro vytváření `klíčů`. Jelikož *slovo* je posloupnost prvků *abecedy*, lze relací uspořádání abecedy A uspořádat i množinu všech *slov*, tzv. *lexikografickým uspořádáním*. Možnost řadit vytvořená slova není pro matematický model podstatná, ale je velmi užitečná z následných implementačních důvodů. Množinu všech neprázdných *slov* označme jako A^+ a její vhodnou podmnožinu jako $\Sigma \subset A^+$. Množina Σ je tedy *jazyk* nad *abecedou* A.

Jazyk Σ lze považovat za *abecedu*, ze které jsou tvořeny `klíče`. Σ tedy obsahuje vhodná pojmenování proměnných použitých v modelované metaheuristice. Množinu všech neprázdných klíčů označme jako Σ^+, tedy jako *jazyk nad abecedou* Σ.

K vytváření `klíčů` ze slov jazyka Σ^+ je dobře využitelná standardní operace *zřetězení* označovaná "." (tečka). `Klíč` $\sigma \in \Sigma^+$ tak může být získán jako $\sigma = \sigma_1.\sigma_2$. (Platným `klíčem` jsou jak prvky Σ, tak i prvky Σ^+, protože $\Sigma \subset \Sigma^+$). Operace zřetězení je asociativní a platí tedy $(\sigma_1.\sigma_2).\sigma_3 = \sigma_1.(\sigma_2.\sigma_3)$.

Pro implementační důvody je také důležitý pojem *prefix* (předpona) `klíče` $\sigma \in \Sigma^+$, což je takové slovo σ_p, že platí $\sigma = \sigma_p.\sigma_x$ pro nějaké $\sigma_x \in \Sigma^+$.

Pro popis běhu metaheuristického modelu jsou důležité dva speciální symboly

- τ odpovídající hodnotě `LOOP_STOP` a

- ε odpovídající hodnotě `NONE` a reprezentující prázdné slovo.

Ani jeden z těchto symbolů nesmí být využit pro vytváření klíčů. Jako množinu Σ^* označme rozšíření množiny všech `klíčů` o prázdné slovo ε, tedy $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

Paměť metaheuristiky je definována jako množina uspořádaných dvojic $M = \{(\sigma_i, v_i), \ldots\}$, kde $\sigma_i \in \Sigma^+$ jsou vytvořené klíče a $v_i \in \mathcal{V}$ jsou hodnoty, které jsou ke klíčům přiřazeny. Hodnoty v_i mohou být buď `klíče`, `operátory`, prázdný klíč ε a nebo běžné datové hodnoty (čísla, řetězce atd.). Množina těchto přípustných hodnot je označena jako $\mathcal{V}$.

Následující definice popisují druhy *skupin klíčů*, které jsou v modelu využity. Jako *skupina klíčů* délky k bude označována uspořádaná posloupnost $p^k = \{\sigma_i\}_{i=1}^k$, kde $\sigma_i \in \Sigma^+$ a $i = 1, \ldots, k$. Jako p^0 je označována prázdná posloupnost nulové délky. V některých případech ale není přesný počet klíčů v posloupnosti důležitý, a proto je definována pouze nějaká jeho hranice. Definujme následující symboly pro libovolné $k \in \mathbb{N}$

- p^{k+} je posloupnost klíčů délky alespoň k.
- $p^+ \equiv p^{1+}$ je posloupnost klíčů libovolné nenulové délky.
- $p^* \equiv p^{0+}$ je posloupnost klíčů libovolné délky, včetně prázdné posloupnosti (tedy posloupnosti délky 0).
- $\mathcal{P}^k$ je množina všech posloupností p^k, tedy posloupností délky k.
- $\mathcal{P}^{k+}$ je množina všech posloupností p^{k+}, tedy posloupností délky alespoň k.
- $\mathcal{P}^+$ je množina všech nenulových posloupností p^+.
- $\mathcal{P}^*$ je množina všech posloupností p^*.

Všechny uvedené posloupnosti jsou složeny z prvků množiny Σ^+, tedy neobsahují prázdný klíč ε. Pokud jej budou moci obsahovat, tak bude k označení posloupnosti připsán dolní index ε (například p_ε^{k+} nebo $\mathcal{P}_\varepsilon^*$).

`Operátory` zpracovávají hodnoty získané z předaných klíčů a poskytují jednu výstupní hodnotu. V matematickém modelu budou operátory z důvodu kategorizace rozděleny na dva základní typy. Obecnější typ operátoru je definován za použití `klíčů`, tedy jako zobrazení n-tice klíčů do množiny přípustných hodnot:

$$X : \Sigma_1 \times \ldots \times \Sigma_n \to \mathcal{V} \cup \{\tau\}.$$

Tyto operátory zobecňují všechny procedury potřebné pro běh metaheuristiky. Hodnota $n \in \mathbb{Z}_0^+$ a může být tedy i nulová, což značí žádný vstupní klíč. Jako množinu všech operátorů obecného typu označme množinu $\mathcal{X}$.

Druhý typ operátorů se váže k základním principům metaheuristiky a je vytvořen kvůli klasifikaci jejích jednotlivých operátorů. Vstupem do tohoto operátoru je uspořádaná n-tice

posloupností, kde $n = 0, 1, \ldots$ a výstupem `operátoru` může být buď posloupnost `klíčů` a nebo nějaký ze speciálních symbolů. Obecně lze tedy tento `operátor` definovat jako zobrazení

$$X_{\mathcal{P}} : \mathcal{P}_1 \times \ldots \times \mathcal{P}_n \to \mathcal{P}_{n+1} \cup \{\tau, \varepsilon\},$$

kde $\mathcal{P}_i$ jsou vhodné podmnožiny $\mathcal{P}^*$, přičemž pro $i \in \{1, \ldots, n\}$ se jedná o množiny vstupních posloupností a pro $i = n + 1$ je množina výstupní.

Operátory typu $X_{\mathcal{P}}$ lze převést na operátory obecného typu pomocí speciálního operátoru $X_S \in \mathcal{X}$, který je obaluje. Operátor X_S pouze převezme hodnoty klíčů k_i, najde pro ně v paměti odpovídající hodnoty v_i a použije jako vstup pro operátor $X_{\mathcal{P}}$. Výstupní posloupnost je uložena do paměti a odpovídající klíč je použit jako výstup X_S.

Metaheuristika $\mathcal{M}$ je posloupnost instrukcí $(X, \sigma_{\text{in}}, \sigma_{\text{out}})$, kde X je `operátor` se vstupy z `klíče` σ_{in} a výstupem přiřazeným do σ_{out}. Ke vstupnímu `klíči` může být přiřazena celá n-tice klíčů vstupující do operátoru.

Stroj, který definovanou metaheuristiku $\mathcal{M}$ umí provést, je stroj s jednoduchou řídicí logikou a schopností číst a zapisovat data do paměti M. Ke čtení paměti slouží funkce $R_M(k)$ vracející pro `klíč` k hodnotu v takovou, že $(k, v) \in M$. Zápis naopak provádí funkce $W_M(k, v)$ tak, že odstraní předchozí záznam (k, v_0) z M, tedy $M_{\text{temp}} = \{(k_1, v_1) : (k_1, v_1) \in M \wedge k_1 \neq k\}$ a následně množinu M_1 rozšíří o nový klíč $M = M_{\text{temp}} \cup \{(k, v)\}$.

Logika provádění metaheuristiky je rekurzivním voláním jednoduchého vykonání operátoru

$i = 1$

dokud $i \leq |\mathcal{M}|$

  přečti instrukci $q_i = (X_i, \sigma^i_{\text{in}}, \sigma^i_{\text{out}})$

  přečti vstupní parametry $p = R_M(\sigma^i_{\text{in}})$

  vykonej operátor $v = X_i(p)$

  když $v = \tau$, tak ukonči provádění operátoru a vrať výsledek

  ulož výstup $W_M(\sigma^i_{\text{in}}, v)$

  zvyš hodnotu i na hodnotu i+1 a opakuj vykonání instrukce

opakuj vykonání operátoru

Dokud nějaký z operátorů nevrátí symbol τ, stále se opakuje vykonávání celého operátoru. Jakmile se symbol τ objeví, je běh operátoru ukončen a reportuje se poslední platná hodnota ($v \neq \tau$). Operátory, které jsou obsaženy v přečtených instrukcích, se vykonávají stejným postupem. Rekurzivně tak lze tedy vykonat celou metaheuristiku.

6.3 Typické operátory

Principem metaheuristiky je popisovat způsob prohledávání stavového prostoru na základě zjištěných hodnot v předchozích bodech. Typické operátory tedy přijímají posloupnosti klíčů, ze kterých vybírají nové podposloupnosti a nebo z nich vytváří nové klíče. Operátory také často spojují více posloupností do jedné nebo z nich extrahují informace. Napříč publikovanými metaheuristikami lze pozorovat mnoho operátorů s podobnou funkcí, a lze je tak kategorizovat do samostatných tříd. Některé operátory mohou patřit do více tříd zároveň, a může tak záležet na úhlu pohledu a způsobu, jak je operátor implementovaný. V následujících odstavcích se jedná především o operátory, jejichž vstupní a výstupní `klíče` reprezentují přímo konkrétní řešení úlohy. Může se tak jednat o *chromozomy*, *světlušky*, *atomy* a nebo jakékoliv jiné metafory použité v konceptech různých metaheuristik. Společně budou označovány pouze jako *objekty*.

Výběr

Operátory výběru neprodukují žádná nová řešení, pouze z předané posloupnosti vyberou nějakou její podposloupnost a vrátí ji na výstup. Typickým modelem je zobrazení

$$X_{v1} : \mathcal{P}^n \to \mathcal{P}^m,$$

kde z předané posloupnosti libovolné nenulové délky n je vybráno m jejích prvků. Obvykle je $m < n$. V genetických algoritmech se jedná o výběr rodičů z celé populace (nejčastěji $m = 2$) nebo výběr populace nejslabších jedinců k následnému odstranění.

Speciálním typem tohoto operátoru je výběr jednoho prvku z předané posloupnosti:

$$X_{v2} : \mathcal{P}^n \to \mathcal{P}^1.$$

Nejčastěji jde o výběr nejlepšího prvku (tedy minimálního nebo maximálního), ale výjimkou nejsou ani výběry prvku náhodně. Například v metodě *Tabu search* je vybíráno z vygenerovaného okolí nejlepší řešení, nebo v genetických algoritmech může být náhodně vybírán jedinec pro následnou mutaci.

Generování

Generováním je myšlena třída operátorů, které k vytvoření posloupnosti nových objektů nevyžadují žádné předané vstupní klíče

$$X_g : \mathcal{P}^0 \to \mathcal{P}^k.$$

Nejčastějším příkladem tohoto operátoru je generování počátečních populací pomocí náhodného generátoru. Operátor tak pouze na základě znalosti rozdělení pravděpodobnosti dokáže vygenerovat zadaný počet jedinců (např. chromozomů v genetických algoritmech). Dalším příkladem může být načítání jedinců z externích souborů.

Produkce

Produkce je jedna z nejdůležitějších tříd, protože výstupy jejích operátorů jsou vytvářeny na základě kombinování dat z předaných objektů. V těchto operátorech dochází velkou měrou k předávání informací o stavovém prostoru a na základě logiky příslušné metaheuristiky jsou zpracovány a použity pro vytvoření nových informací. Typický příklad je křížení rodičů v genetických algoritmech:

$$X_{p1} : \mathcal{P}^2 \to \mathcal{P}^1,$$

kde vstupem jsou dva chromozomy a kombinací jejich genů je získán nový potomek. V různých algoritmech vstupují do operátoru různé počty rodičovských objektů. Ve zmíněných genetických algoritmech jsou to dva rodiče, v diferenciální evoluci jsou to čtyři prvky a v algoritmech založených na hejnovém chování to může být i celá populace. Obecně lze uvažovat, že `operátory` produkce nemají pevně daný počet vstupních objektů a počet nově vytvořených objektů může být také různý:

$$X_{p2} : \mathcal{P}^n \to \mathcal{P}^m,$$

kde $n, m \in \mathbb{N}$ jsou libovolná přirozená čísla, která nemusí být pro konkrétní typ metaheuristiky pevně stanovena a mohou být jedním z nastavitelných parametrů. Obecné operátory mohou na vstupu akceptovat i posloupnosti s délkou, která se v průběhu algoritmu různě mění.

Délka vstupní posloupnosti n může být i výrazně kratší než délka výstupní posloupnosti m. Například metoda *Tabu search* generuje z jediného objektu (aktuální řešení) celou množinu okolních bodů, kterých může být i několik desítek.

Speciálním typem produkce je *klonování*, kde výstupem je posloupnost objektů identicky zkopírovaných ze vstupní posloupnosti. Nově vytvořené *klony* jsou následně ihned modifikovány podle potřeb algoritmu. V metaheuristice *CLONALG* je počet takto vytvořených klonů závislý na hodnotě účelové funkce daného objektu.

Modifikace

Modifikace objektů posloupnosti bývá často označována jako tzv. *mutace*. Jejím hlavním smyslem je náhodně změnit některou z vnitřních proměnných vstupních objektů a vrátit stejnou posloupnost, jako byla na vstupu. Modelem je tedy operátor:

$$X_m : \mathcal{P}^n \to \mathcal{P}^n,$$

kde $n \in \mathbb{N}$ a $\mathcal{P}^n$ označuje posloupnost objektů, která je stejná na vstupu i výstupu. V genetických algoritmech je mutace většinou založena na výměně některé souřadnice řešení za náhodně

vygenerovanou. Tento operátor lze chápat i jako speciální typ produkčního operátoru, který *kříží* předané řešení s náhodně vygenerovaným. Významným rozdílem ale je, že *fyzicky* nevzniká žádné nové řešení, ale je vráceno původní pouze s pozměněnými vlastnostmi.

Spojování

Velmi často metaheuristiky udržují jednu množinu nejlepších řešení a z ní stále vytváří nové objekty. Pokud mají nové objekty dostatečnou kvalitu, bývají zpravidla zahrnuty do původní množiny. Operátory spojování takto zajišťují vytvoření jedné souhrnné posloupnosti objektů z několika samostatných. Ze vstupu je získána n-tice nezávislých posloupností, které po vhodném zkombinování dávají novou posloupnost tvořenou vybranými prvky původních.

$$X_s : \mathcal{P}_1^{n_1} \times \mathcal{P}_2^{n_2} \times \cdots \times \mathcal{P}_k^{n_k} \to \mathcal{P}^n,$$

kde různě rozměrné vstupní posloupnosti jsou přetvořeny na výstup, jehož délka n nemusí být na vstupních délkách n_i nijak přesně závislá.

Výpočet parametrů

Operátory musí často zajišťovat i výpočet dodatečných hodnot pro předané objekty. Každý objekt může mít, kromě svých souřadnic určujících konkrétní řešení i libovolný počet dalších vlastností realizovaných systémem podklíčů. Například objekty hejnových algoritmů vyžadují přiřazenou skalární veličinu charakterizující rychlost, v gravitačním algoritmu je obdobně vyžadována hybnost objektu. Tyto parametry nejsou v čase konstantní a jejich aktuální hodnota musí být určována za běhu. Její velikosti mohou záviset pouze na vlastnostech objektu, nebo mohou být spojené s vlastnostmi celé populace. Obecný zápis operátoru pro výpočet doplňkových parametrů lze psát

$$X_c : \mathcal{P}_1^{n_1} \times \mathcal{P}_2^{n_2} \times \cdots \times \mathcal{P}_k^{n_k} \ \to \mathcal{P}_1^{n_1},$$

kde $\mathcal{P}_1^{n_1}$ je posloupnost objektů, pro které se výpočet provádí a další předané posloupnosti $\mathcal{P}_2^{n_2}, \ldots, \mathcal{P}_k^{n_k}$, které jsou pro výpočet potřeba jsou doplňkové. Výstupem je pak identicky stejná posloupnost objektů $\mathcal{P}_1^{n_1}$ s doplněnými (aktualizovanými) parametry.

Zdaleka nejvyužívanějším typem výpočtového operátoru je učení hodnoty účelové funkce, kde je pro předaný objekt vypočítána jeho funkční hodnota a uložena k příslušnému podklíči. Tedy v nejčastější variantě má operátor tvar

$$X_{e_1} : \mathcal{P}^1 \to \mathcal{P}^1.$$

V umělých imunitních systémech je potřeba určovat i tzv. *normalizovanou účelovou funkci*, která je vztažena k hodnotám účelové funkce objektů v aktuální populaci a normalizována na interval $[0, 1]$. Pro konkrétní objekt může být takový operátor konstruován jako

$$X_{e_2} : \mathcal{P}^1 \times \mathcal{P}^n \to \mathcal{P}^1,$$

kde vstupem je uvažovaný objekt a celá aktuální populace.

Operátory výpočtu parametrů a operátory modifikace jsou koncepčně velmi podobné a lze je v obecném smyslu považovat za prvky stejné třídy. Jejich hlavním rozdílem je typ podklíčů, které modifikují (vytvářejí). Operátory modifikace mění hodnoty objektů přímo reprezentující řešení, kdežto výpočtové operátory modifikují klíče, které zastupují doplňkové informace předaných objektů. Z tohoto pohledu jsou modifikační operátory více obecné, protože je lze použít napříč různými metaheuristikami. Výpočtové operátory jsou více svázány s konkrétním metaheuristickým algoritmem a jejich využití je tedy více limitované. Neplatí to ovšem pro operátor vyhodnocení účelové funkce, ten ve stejné podobě tvoří základní kámen většiny používaných metaheuristik.

Ukončovací kritéria

Ukončovací operátory jsou velice speciální třídou operátorů. Jejich hlavním smyslem je testovat, jestli jsou splněna kritéria pro ukončení vykonávané smyčky operátoru. Lze je využít i pro udržování informací o běhu algoritmu. Například typická ukončovací podmínka, která je splněna při překročení předdefinovaného počtu iterací, může být realizována pomocí operátoru, který při každém zavolání načte z paměti aktuální počet iterací a pokud je menší než stanovený počet, zvýší je, uloží zpět a vrátí hodnotu `NONE` (ε). Pokud naopak operátor zjistí, že aktuální hodnota už stanovenou mez přesáhla, je vrácena hodnota `LOOP_STOP`, čímž se vykonávaný operátor ihned ukončí. Obecný tvar ukončovacího operátoru závisí na zvolené strategii ukončování. V nejširším smyslu může být tento tvar shodný s nejobecnějším typem operátoru, ale pro přehlednost jej zavedeme pouze jako:

$$X_t : \mathcal{P}_1^{n_1} \times \mathcal{P}_2^{n_2} \times \cdots \mathcal{P}_k^{n_k} \to \{\tau, \varepsilon\},$$

tedy na základě k-tice vstupních posloupností se rozhodne, jestli bude vrácena hodnota `NONE` nebo `LOOP_STOP`. Předchozí popsané operátory byly definovány pomocí zobrazení několika vstupních posloupností na jednu výstupní. Tento typ operátoru je odlišný tím, že musí někdy vracet i hodnotu `LOOP_STOP`.

Variabilita výstupů operátoru je klíčovým znakem celého modelu. Lze například spojit ukončovací operátor s operátorem výběru, kde budou z předané posloupnosti selektovány objekty a pokud by se žádné vhodné nenašly, ukončí se vrácením `LOOP_STOP`. Tím se může významně zpřehlednit implementace metaheuristiky a zjednodušit i následný návrh odvozených typů.

Možnosti aplikace

Popsaný algoritmický modelu je natolik univerzální, že v něm lze popsat jakýkoliv algoritmus[79]. Pro mnoho běžných úloh by byl ale naprosto nevhodný a byl vytvořen pouze se záměrem usnadnit práci s metaheuristikami. Pro metaheuristiky je přirozené, že jsou definovány jako operátory předávající si posloupnosti obsahující různá řešení úlohy. Hlavní síla modelu je v tom, že reflektuje tento přístup a dovoluje tak vytvářet množství nových konceptů. Jednotný způsob ukládání dat a operátorů, spolu s přímočarým seskupování proměnných do logických struktur, vede ke snadnějšímu zobecňování a znovupoužitelnosti metaheuristických konceptů.

Mnoho metaheuristik se zdá být na první pohled významně rozdílných, ale popsaným modelem je lze sjednotit. Následující příklady demonstrují šíři konceptů, které lze modelem jednoduše realizovat

- Základním přístupem je udržování *populace* jedinců, kde každý jedinec reprezentuje samostatné řešení. Jedinec je tedy určen klíčem k_i, který je zřetězený s klíči odpovídajícími souřadnicím řešení (tedy $k_i.x$, $k_i.y$, ...). Populace je posloupnost klíčů k_i.
- Posloupnosti jsou vhodně využitelné i v lokálním prohledávání, kde se přímo nepracuje s populacemi, ale i přesto vznikají větší soubory řešení. *Hill climbing* algoritmus nebo *Tabu search* generují pro aktuální řešení jeho *okolí*, které je snadno pomocí posloupnosti reprezentovatelné. V *Tabu search* se posloupnost klíčů přímo hodí i *Tabu list*, tedy seznam posledních navštívených bodů.
- Z pohledu vývoje nových prohledávacích strategií je velice výhodná *bezschémovost* databáze. Například rozšířit standardní genetický algoritmus o pohlavní křížení lze docílit pouze jednoduchým přiřazením pohlaví každému vygenerovanému řešení k_i, tedy např. $(k_i.\text{sex}, \text{female}$ a přeformulovat operátory, aby jej braly v potaz.
- Mravenci v *Ant colony optmization* zanechávají feromonové stopy mezi jednotlivými uzlovými body prohledávaného prostoru a podle toho se řídí jejich následující cesty. Každý uzlový bod lze reprezentovat vlastním klíčem b_i a aktuální feromonovou stopu mezi uzly b_u a b_v přiřadit ke klíči vzniklému jejich zřetězením, tedy např. $(b_u.b_v, 0.2456)$.
- Pokud je potřeba pracovat s *více instancemi jedné metaheuristiky* zároveň (více nezávislých feromonových stop zároveň), docházelo by při předchozím způsobu ke kolizi dat v databázi. Stačí ale přiřadit pro každou instanci unikátní klíč u_i a problém je ihned vyřešen: $(u_i.b_u.b_v, 0.2456)$.
- Stejně, jako může fungovat více instancí metaheuristik dohromady, tak může být i *jedna instance použita na různé populace*. Jedná se o přístup typický pro paralelní metaheuristiky, kde existuje několik nezávislých populací, které jsou zpracovávány stejným přístupem jen s

[79] Pro deterministický Turingův stroj.

odlišnými parametry. Populace jsou uspořádány do nejrůznějších topologií a podle nich si mezi sebou vyměňují určité informace (jedince). Každá populace má svůj klíč a topologie je popsána jejich zřetězením. Například $(p_1.p_2, 1)$ znamená, že mezi populací p_1 a p_2 je nějaký vztah, který lze specifikovat např. počtem vyměňovaných jedinců označených jako n: $(p_1.p_2.n, 8)$.

- Paralelizace je často řešena tak, že jeden prvek patří do právě jedné populace. Není ale problém prvek zařadit do více populací zároveň. Zařazení prvku do libovolného počtu populací je pouze otázkou jeho vložení do příslušné posloupnosti. Správné zacházení s takovými prvky musí být ale realizované pečlivě navrženými operátory.

- Zařazení prvku do populace nemusí být pouze jednoznačné (tedy *patří* nebo *nepatří*), ale může tam patřit s různými stupni příslušnosti. Lze tak formulovat fuzzy metaheuristiku, kde pro populaci p_i a prvek k_j se snadno zapíše stupeň příslušnosti jako např. $(p_i.k_j, 0.8)$. Stejný zápis může být použit i pro pravděpodobnostní nebo jakýkoliv jiný vztah mezi populací a nějakým prvkem.

- I samotné prvky mezi sebou mohou mít různé relace, které lze definovat libovolně podle potřeb. Například v názvosloví evolučních algoritmů má každý prvek k_i svého rodiče r_j. Takový vztah lze zapsat buď jako $(k_i.r_j, \text{parent})$, nebo i jako $(k_i.\text{parent}, r_j)$. Rozdíl mezi těmito záznamy je ve způsobu vyhledávání této informace, tedy v otázce, kterou chceme danou informací zodpovědět. V prvním případě by otázka zněla *"Jaký je vztah mezi k_i a r_j?"* a odpověď by byla *parent*. Druhým záznamem odpovídáme na otázku *"Kdo je rodičem klíče k_i?"*. Zvolení správného zápisu informace do paměti tedy musí být vždy přizpůsobeno povaze operátorů, které s nimi pracují.

- Relace mezi klíči (prvky, populacemi, ...) nemusí být pouze binární[80]. Stejným způsobem lze postupovat i při definování obecných n-arních relací. Zařazení tří klíčů k_1,k_2 a k_3 do relace lze docílit jejich zřetězením $k_1.k_2.k_3$ a přiřazením odpovídající hodnoty, např. $(k_1.k_2.k_3, 1)$. Pokud je relace R symetrická vůči všem kombinacím klíčů, tzn. když $k_1.k_2.k_3 \in R \Rightarrow k_3.k_1.k_2 \in R$ apod., nemusí se do databáze zapisovat všechny tyto kombinace, ale stačí je uložit pouze v lexikografickém uspořádání a považovat ji za kanonickou formu.

- Různé metaheuristiky vyžadují různé druhy informací přiřazených ke konkrétním řešením. Model dovoluje snadnou hybridizaci v tom smyslu, že jedno řešení může mít zároveň přiřazenou normalizovanou hodnotu účelové funkce (potřebnou pro operátory umělých imunitních systémů), tak i např. rychlost (potřebnou pro operátory hejnových algoritmů). Vše je realizováno pouze přidáním odpovídajícího klíče do databáze. Pokud by hrozila kolize používání stejných jmen pro různé parametry, lze tomu předejít zřetězením s novým klíčem definujícím konkrétní operátor.

---

[80] Binární relace určuje vztah mezi dvěma prvky. Obecné relace mohou určovat vztah mezi libovolným počtem prvků.

- Díky definici databáze takovým způsobem, že klíčům mohou být přiřazována jak data tak i operátory, lze mnoho operátorů navržených pro práci s běžnými řešeními úlohy použít i pro práci s operátory. Koncept hyperheuristik je založený na zkoušení a ohodnocování různých nízkoúrovňových heuristik (operátorů) a podle jejich výsledků je používat na další hledání. Místo populace řešení tak lze používat populaci operátorů, což pro velkou část tradičních operátorů není žádný rozdíl, protože obojí je reprezentováno pomocí klíče. Např. operátory výběru tak lze používat naprosto beze změny, stačí jen přiřadit operátorům v populaci odpovídající hodnoty účelových funkcí.

7 Programová implementace hybridního metaheuristického modelu

Jedním z primárních účelů popisu obecného modelu je jeho implementace dovolující následné snadné vytváření a kombinování metaheuristických algoritmů. Obecný model by neměl být závislý na zvoleném programovacím jazyce, avšak některé jazyky se k tomuto účelu hodí lépe než jiné. I přestože metaheuristiky sdílí velké množství společných rysů, vykazují značnou různorodost v typech využívaných proměnných i ve způsobech jejich zpracování a předávání. Implementovat takový systém ve staticky typovaných jazycích[81] může být těžkopádné a nepřehledné, a proto byl zvolen dynamicky typový jazyk[82] *Python*. Jazyk Python se vyznačuje velkou flexibilitou a plnou objektovostí, takže do proměnných se dají ukládat, kromě běžných datových typů, i celé funkce, třídy nebo objekty[83]. Mezi další důležité vlastnosti Pythonu patří hlavně multiplatformnost[84], multiparadigmatické možnosti programování[85], snadná spolupráce s jinými programovacími jazyky[86] a hlavně velice užitečné základní datové typy[87]. Jazyk Python má velice snadno čitelnou syntaxi a bude tedy použit i pro ukázky implementací (místo běžného pseudokódu).

7.1 Základní technologické prvky

Z popsaného metaheuristického modelu plyne, že základními technologickými prvky jsou `klíče` a `operátory` a na ně navázané algoritmické struktury paměti a řídicí logiky. Aby byly vytvořené metaheuristiky opravdu hybridní, musí umožňovat snadnou výměnu operátorů při zajištění funkčnosti celého algoritmu.

Operátory tedy musí být samostatnými celky, které nesmí být nijak vázány na konkrétní metaheuristiku. Jelikož jeden operátor může v sobě zahrnovat libovolný počet dalších operátorů, měly by být všechny specializovanější části sdruženy do jednoho výsledného operátoru a pouze ten by měl být používán pro další hybridizaci. Komunikace mezi operátory je zajištěna primárně výměnou posloupností klíčů, jejichž přiřazené hodnoty jsou operátorem vhodně modifikovány.

---

[81] Staticky typovaný jazyk znamená, že každá proměnná má předem známý datový typ (*integer, float, string, objekt*, ...) a za běhu programu jej nelze změnit.

[82] Dynamické typování jazyka znamená, že o typu proměnné (jeho *fyzické* reprezentaci v paměti počítače) se rozhoduje až za běhu programu, a tedy do jedné proměnné lze postupně ukládat jakýkoliv datový typ.

[83] V Pythonu je každý datový typ (čísla, řetězce, funkce, třídy, ...) zároveň i objektem, a tedy každé přiřazení hodnoty do proměnné je implementováno pouze pomocí ukazatelů.

[84] Schopnost spustit vytvořený program na různých platformách (Windows, Linux, OS X, .Net apod.)

[85] Možnost psát programy s využitím procedurálních, objektových i funkcionálních přístupů.

[86] Nejjednodušší možností je spolupráce s jazyky C/C++.

[87] Speciálně typ *dictionary*, na kterém je postavená základní databáze programu.

Doplňkovým způsobem komunikace operátorů je využití sdílené paměti klíčů. Každý operátor tedy musí mít plný přístup k paměti a musí být schopen získat libovolnou hodnotu přiřazenou ke konkrétnímu klíči. To ovšem neznamená, že operátor může číst celou paměť i bez znalosti jmen klíčů, ale je tomu právě naopak. Základní podmínka pro přístup do paměti je znalost jména klíče jehož hodnota má být přečtena. Paměť metaheuristiky nedovoluje (nemusí dovolovat) iterativní procházení všech uložených hodnot, ale garantuje pouze přístup k hodnotám libovolného zadaného klíče. Toto omezení usnadní a zpřístupní hardwarovou paralelizaci běhu metaheuristiky.

Paměť je realizována pomocí tzv. NonSQL databáze[88] a celý metaheuristický algoritmus je v ní uložen jako sled po sobě jdoucích operátorů. Vykonávání jednotlivých instrukcí řídí jedna k tomu určená funkce[89], která čte instrukce z databáze, spouští operátory a řídí správné předávání vstupů a výstupů.

Stručně řečeno, implementace metaheuristického hybridního algoritmu je založena na jedné databázi sloužící jako paměť, jedné řídicí struktuře a operátorech nezávislých na druhu vykonávané metaheuristiky.

7.2 Paměť

Jak již bylo v úvodu naznačeno, paměť je realizována pomocí databáze, ke které může mít každý operátor neomezený přístup. V předešlých několika letech lze zaznamenat na poli databázových systémů významnou změnu oproti hlavnímu schématu posledních 40 let. Naprostá většina databází byla založena na relačním typu a na dotazovacím jazyce SQL vyvíjeným již od 70.let 20.století. Toto schéma se časem osvědčilo jako vynikající nástroj pro rychlý přístup k obrovským množstvím dat stejného tvaru. Ale právě podmínka, že data musí mít stejný tvar (a tedy je lze zapisovat do přehledných tabulek), se postupem času stávala překážkou. Mnoho dnešních aplikací musí pracovat s velice nestrukturovanými daty. Data jsou různých typů, mají různá uspořádání a jsou vůči sobě naprosto nekonzistentní. Z potřeby ukládat takovýto typ dat a nějak rozumně s nimi pracovat se začaly prosazovat tzv. *NoSQL databáze*.

Objekty metaheuristik představují přesně takový typ nestrukturovaných dat, se kterými si standardní přístupy poradí jen s velkými obtížemi. Nabídka NoSQL databázových systémů se rychle zvětšuje, a to hlavně díky stále rostoucím potřebám internetových aplikací, pro něž je NoSQL schéma často více než vhodné. Existuje několik hlavních skupin NoSQL databází lišících se především základním typem uchovávaných dat. Existují tak například databáze specializované na data ve formátu grafu[90], dokumentové databáze[91] pro uchovávání kolekcí sestávajících z

[88] Více viz následující kapitola 7.2.

[89] Ve skutečnosti jde o třídu a z ní odvozený objekt, ale to v tomto stručném popisu není podstatné. Více viz kapitola 7.3.

[90] Tzv. *graph database*, kde významným zástupcem je např. Neo4j (neo4j.org).

dat bez jednotného schématu a nebo tzv. *Key-Value* databáze, která uchovává data ve formátu dvojic *klíč-hodnota*[92]. Poslední zmíněný druh je přesně ve shodě se způsobem ukládání dat ve vytvořeném modelu, a proto je jeho použití nasnadě.

Ve vytvořené implementaci není použit žádný externí databázový systém a zmíněná databáze je vytvořena pomocí základních typů programovacího jazyka *Python*. Důvodem pro vytvoření vlastního databázového systému je potřeba *ad-hoc* modifikací a naopak nevyužití mnoha obsažených funkcí externí databáze.

Ve dvojici *(klíč, hodnota)* je klíč reprezentován pomocí textového řetězce. Vytváření odvozených klíčů je tedy zredukováno na jednoduché zřetězení jednotlivých klíčů společně se symbolem pro jejich odlišení. Každý klíč v implementaci má tvar

```
k_1 = some_key_name#
k_2 = another_key_name#
```

tedy sestávající ze svého názvu a zakončený znakem '#' (dvojitý křížek). Zřetězením klíčů `k_1` a `k_2` lze získat nový klíč

```
k_3 = some_key_name#another_key_name#
```

jehož struktura je stejná (tedy *jméno#*) avšak oddělovač '#' se vyskytuje i uvnitř jména, aby odlišil jednotlivé podčásti klíče. Symbol oddělovače '#' se nesmí vyskytovat v žádném základním jménu klíče, jinak by nešlo rozeznat mezi jednotlivými klíči a mohlo by tak docházet k záměnám a nesprávným ukládáním hodnot. Ke stejným důsledkům by vedlo i vynechání tohoto symbolu na konci jednotlivých jmen.

Často je potřeba vytvářet klíče, které reprezentují obdobné entity, a je tedy výhodné, aby jejich jména byla na první pohled jednotná. K tomu je použit symbol ":" (dvojtečka) následovaný pořadovým číslem entity. Například v metaheuristice genetické algoritmy jsou jednotlivá řešení pojmenována jako *chromozomy* a jejich reprezentace ve formě unikátních klíčů může mít například tvar

```
c_1 = chromosome:1#
c_2 = chromosome:2#
...
```

Chromozomy reprezentující řešení jsou definovány pomocí souřadnic v prohledávaném stavovém prostoru. Ve dvourozměrném případě tak mohu být tyto komponenty reprezentovány pomocí klíčů

---

[91] *Document-oriented database* s nejdůležitějšími zástupci MongoDB (www.mongodb.org) a CouchDB (couchdb.apache.org).

[92] Existují i databáze se schopností udržovat data ve formátu trojic i čtveřic (tzv. *triple/quad store database*).

```
c1_x = chromosome:1#x#
c1_y = chromosome:2#y#
```

popřípadě v n-rozměrném prostoru pomocí klíčů

```
c1_x1 = chromosome:1#x:1#
c1_x2 = chromosome:1#x:2#
c1_x3 = chromosome:1#x:3#
     ...
c1_xn = chromosome:1#x:n#
```

Data se ke klíčům ukládají pomocí vytvořené *key-value* databáze. Její implementace je v jazyce *Python* přímočará, protože jednou z jeho základních datových struktur je *slovník*, který v podstatě takovou databázi zastupuje. Tato struktura funguje na principu *hashování* vstupního klíče a následném přiřazení předané hodnoty. Hashování je metoda, jak z předaného řetězce (nebo i jiného objektu) vytvořit krátké číslo unikátní pro každý zvolený řetězec[93]. Hashe jsou uchovávány v setříděném seznamu, čímž se v případě vyhledávání zadaného klíče zajistí dostatečná rychlost odpovědi. Vyhledávací algoritmus dokáže vrátit hodnotu přiřazenou ke zvolenému klíči v čase $log(n)$ i kratším.

V jazyce *Python* tedy můžeme pro vyhledání klíče psát pouze

```
value = db[key],
```

kde `db` zastupuje databázi (paměť), `key` je textová reprezentace klíče a `value` je hodnota získaná dotazem do databáze.

Typy hodnot, které musí být databáze schopná ukládat, jsou podle modelu minimálně

- *klíč* - tedy textové řetězce,
- *operátor* - objekt typu mthOperator (viz dále),
- *datové hodnoty* - čísla, řetězce, ...,
- *speciální hodnoty* - tedy ukončovací symbol `LOOP_STOP` a prázdný symbol `NONE` (viz dále).

Python je plně objektový jazyk a všechny základní datové typy jsou potomky jedné hlavní třídy. Interně tedy Python může operovat pouze s ukazateli na instance základní třídy, čímž je ukládání do databáze zobecněno na všechny podporované typy a žádný z nich tedy nijak nebrání jejich ukládání.

---

[93] V konečném důsledku hashování nemusí tvořit unikátní klíče, protože vstupní množina řetězců má mnohem větší mohutnost než výstupní hashe. Snahou je tedy je zajistit hashům co největší variabilitu, aby se pravděpodobnost kolize snížila na minimum. V případě potřeby zaručit bezkoliznost hashů existují i další doplňkové metody.

Konečná posloupnost klíčů je v modelu definována pouze jako doplňková struktura. Z pohledu používání vytvářené implementace a vysoké využitelnosti této struktury je výhodnější ji zahrnout již do návrhu mezi základní stavební kameny programu. V Python k její implementaci existuje výhodná struktura `list` značená hranatými závorkami jako

```
sequence = [key_1, key_2, ..., key_n]
```

Do databáze lze tedy ke klíčům přímo přiřazovat i celé posloupnosti a nemusí být implementovány pomocí ostatních základních typů (viz 6.1.1kapitola). Tato odlišnost nijak nemění smysl modelu, pouze urychluje a zjednodušuje vykonávání algoritmu.

Databáze je implementována jako objekt třídy `Database` a jeho základní funkce plní metody, tedy

- čtení dat z databáze je realizováno pomocí funkce `db.get(key)`
- a zápis pomocí `db.set(key, value)`.

Kromě těchto nejpodstatnějších funkcí jsou navíc implementovány i další výhodné funkce, které zefektivňují celkový návrh i běh metaheuristiky. Příkladem je automatické mazání klíčů a jejich přiřazených hodnot z paměti. Každému klíči se automaticky počítá počet referencí (v kolika proměnných a posloupnostech je uložen) a jakmile tento počet klesne na nulu, je odstraněn z databáze. Postup je založen na myšlence, že pokud mohou operátory přistupovat do paměti pouze přes známé klíče, musí být někde jejich názvy uloženy. V případě, že tedy název klíče nikde uložen není, může být jeho obsah smazán. Vedle databáze je paralelně udržován i seřazený seznam všech klíčů a při mazání nějaké z nich jsou smazány i všechny klíče z něho odvozené. Seřazený seznam umožňuje i kopírování klíče včetně jeho podklíčů, což je velice přínosná funkce při ukládání nejlepších jedinců z populací apod.

Pomocí popsaného modelu lze snadno implementovat i koncept objektového programování. Typický zástupce řešení úlohy metaheuristického algoritmu je objekt s n-ticí souřadnic a m-ticí hodnot účelové funkce. Pro snadnou hybridizaci vytvořených operátorů je vytvořena třída klíčů, se kterou umí operátory bezpečně zacházet. V databázi je uložena následující struktura dvojic *klíč-hodnota*

```
(class_name_1#num_of_var#, 3)
(class_name_1#num_of_out#, 1)
```

Třída jménem `class_name_1` definuje, že reprezentuje řešení v tří rozměrném prostoru a odpovídající hodnota účelové funkce je jednorozměrná. Přiřazením jména třídy ke klíči reprezentujícímu řešení lze dosáhnout obecné srozumitelnosti struktury klíčů pro všechny odpovídající operátory:

```
(solution_1#class#, class_name_1#)
(solution_1#var#0#,  0.15566)
(solution_1#var#1#, -1.15566)
(solution_1#var#2#,  3.33333)
(solution_1#out#0#, -19.2368)
```

Řešení *solution_1* má třídu *class_name_1*, z níž lze zjistit, že řešení má tři rozměry a jejich hodnoty jsou [0.15566, -1.15566, 3.3333]. Účelová funkce takového řešení má hodnotu -19.2368. Typický způsob vyhodnocení účelové funkce řešení reprezentované klíčem v proměnné `key` popisuje následující kód

```
1:  class_name = db.get(key,'class')
2:  number_of_variables = db.get(class_name, 'num_of_var')
3:  variables = []
4:  for i in range(number_of_variables):
5:      var = db.get(key,'var',i)
6:      variables.append(var)
7:  f = variables[0]**2+5*variables[1]*variables[2]
8:  db.set(key,'out',0, f)
```

Na řádku 1 a 2 se získá z databáze jméno třídy a počet jejich proměnných, další řádky uloží do proměnné `variables` všechny proměnné, tedy její hodnota bude [0.15566,-1.15566,3.33333]. Řádek 7 vypočítá hodnotu účelové funkce (v tomto případě $f(x, y, z) = x^2 + 5yz$) a poslední řádek výsledek uloží do databáze.

Za zmínku stojí způsob používání funkcí `db.get() a db.set()`. Obě funkce akceptují proměnný počet vstupních parametrů, ze kterých poskládají výsledný klíč. Tedy například `db.get(key,'var',1)` nejdříve vytvoří klíč z hodnoty v proměnné `key` (řekněme, že má hodnotu `solution_1#`), potom k němu přidá textový řetězec `'var'` a nakonec na text převede i číslici 1. Vznikne tak klíč `solution_1#var#1#` a ten se vyhledá v databázi. Ukládání funguje naprosto obdobně, pouze s výjimkou, že poslední parametr je vždy předáván jako ukládaná hodnota. Celá implementace je navržena tak, aby se uživatel vůbec nemusel explicitně zabývat řetězením klíčů. S výjimkou velmi speciálních případů by se tedy s oddělovacím symbolem '#' nemělo vůbec pracovat.

Pro generování proměnných v zadaných mezích, nebo pro potřeby mutace apod., může být definice třídy ještě rozšířena o maximální a minimální povolené hodnoty:

```
(class_name_1#min#0#, -100)
(class_name_1#max#0#, 100)
(class_name_1#min#1#, -2)
(class_name_1#max#1#, 2)
```

```
(class_name_1#min#2#, 0)
(class_name_1#max#2#, 100)
```

Tímto způsobem lze řešit abstrakci metaheuristik pro jejich snadné používání. Vzhledem k tomu, že do databáze lze ukládat i operátory, je takto v podstatě možné realizovat kompletní objektové programování.

7.3 Řídicí logika

Zjednodušeně řečeno, operátory jsou funkce, které manipulují s předanými daty. Na základě existujících metaheuristik lze rozeznat dva hlavní přístupy k aplikování operátorů na předaná data. Některé operátory pracují s více předanými klíči dohromady a potřebují mít v jeden čas přístup ke všem klíčům posloupnosti současně. Naproti tomu jiným operátorům stačí mít v jeden čas přístup pouze k jednomu z klíčů předané posloupnosti, protože jeho operace nejsou nijak ovlivněny ostatními objekty v posloupnosti. Volání operátoru je tedy možné dvěma způsoby:

- metoda `apply` způsobí zavolání operátoru s parametry ve stejném tvaru, jako jsou nadefinovány, kdežto
- metoda `map` volá operátor opakovaně pro každý prvek posloupnosti zvlášť a jednotlivé dílčí výsledky automaticky spojí do nové posloupnosti.

V matematickém ani koncepčním modelu nebylo toto rozdělení provedeno, protože nebylo potřeba. Z implementačního a uživatelského hlediska je ale rozdělení volání do dvou kategorií velmi významné a ušetří množství programátorské práce a eliminuje zbytečné chyby. Například většina operátorů výběru musí být volána metodou `apply`, protože potřebuje zpracovat celou populaci zároveň (seřadit, vybrat nejlepší, apod.). Naopak operátory, které pouze počítají parametry pro konkrétní objekty (nová pozice v hejnových algoritmech, vyhodnocení účelové funkce, ...) mohou být pro zjednodušení volány metodou `map`, tedy pro každý objekt zvlášť.

Celá řídicí logika metaheuristiky je tedy posloupností definovaných operátorů doplněná o způsob jejich volání. Při definování této posloupnosti algoritmus automaticky vybírá, o jaký konkrétní typ operátoru se jedná, a podle toho s ním zachází. Jsou definovány tři typy, ze kterých lze poskládat celou metaheuristiku:

- *Externí funkce* je běžná funkce popsaná v programovacím jazyce. Jedná se o nejnižší typ operátoru, který reálně manipuluje s daty, provádí výpočty nebo přímo čte a zapisuje do databáze. Příkladem jsou operátory křížení v genetických algoritmech, vytváření okolí v *Tabu search* nebo vyhodnocení účelové funkce.
- *Blok* je definovaná posloupnost operátorů libovolného typu. Blok shlukuje operátory do větších celků tak, aby dávaly náležitý smysl. Celá metaheuristika i její větší logické celky tvoří operátory tohoto typu.
- *Odkaz na operátor* obsahuje pouze pojmenování kteréhokoliv z již vytvořených operátorů.

Typ operátoru *blok* je nejvýznamnějším typem z pohledu budování metaheuristiky. Z definice modelu plyne, že každý z těchto operátorů se opakuje tak dlouho, dokud nějaký z jeho *podoperátorů* nevrátí hodnotu `LOOP_STOP`. V popisované implementaci je tento způsob specifikován, aby byl jednodušší na programování. *Blok* může být vytvářený buď jako anonymní nebo pojmenovaný (určený k pozdějšímu volání). Oběma typům je možné nastavit, zdali se mají opakovat a nebo provést jenom jednou. Operátor bez opakování je pouze speciálním typem základního operátoru s opakováním a nejedná se tedy o rozšíření modelu. Operátoru s opakováním stačí přiřadit na konec pomocný operátor vracící hodnotu `LOOP_STOP` při každém volání, čímž nikdy k jeho opakování nedojde.

Operátory v modelu standardně vrací poslední klíč s jinou hodnotou než je `LOOP_STOP`, ale v implementaci je přístup rozšířen o možnost zvolení výstupního klíče bez ohledu na pořadí přiřazení jeho hodnoty. Tento koncept je opět dosažitelný za pomoci základních operací a nemusí být tedy definován v obecném modelu.

Všechny popsané typy operátorů lze libovolně kombinovat pro získání očekávaného chování. Z vnějšího pohledu jsou všechny typy operátorů rovnocenné a uživatel mezi nimi nemůže spatřit rozdíl.

Základním objektem zodpovědným za spouštění a vykonávání celé metaheuristiky je objekt třídy `Algorithm`. Jeho metody registrují všechny potřebné operátory a klíče pro poskytování vstupů a ukládání výstupů. Základní schéma vytváření nové metaheuristiky je

```
alg = Algorithm()
main = alg.operator('main')

alg.block_start(main)
...
alg.block_stop()

result = alg(main)
```

Nejdříve je zde vytvořen objekt `alg` třídy `Algorithm`, kterým se nadefinují všechny operátory. Hlavní operátor, tedy samotná metaheuristika, je uložen do proměnné `main` a jeho definice je vložena mezi příkazy `alg.block_start(main)` a `alg.block_stop()`. Zavoláním objektu `alg(main)` s názvem operátoru je zahájen jeho běh.

Před zahájením běhu operátoru mu lze nastavit potřebné parametry. Prakticky každá metaheuristika má možnost ovlivnit své chování pomocí množiny vstupních parametrů jako je velikost populace, počet iterací před ukončením, pravděpodobnosti výběru prvků atd. Všechny tyto parametry jsou ukládány přímo do databáze, odkud je může kterýkoliv operátor snadno načíst. V případě, že by dva nezávislé operátory pojmenovaly svůj řídicí parametr stejně, byl by v databázi pouze jeden zápis s pouze jednou hodnotou. Tato nesnáz je vyřešena svázáním

parametru přímo s konkrétním operátorem pomocí metody `alg.bind_parametr(operator)` a je algoritmu dostupný přes proměnnou `this`. Ve své podstatě se jedná pouze o vytvoření klíče obsahujícího název operátoru spojený s názvem nastavovaného parametru (tedy základní typ relace mezi klíčí).

Pro snadné zapouzdření, používání a nastavování bloků operátorů je vytvořena doplňková třída `MthOperator`, která zajišťuje správné přiřazení parametrů, vnitřních operátorů a konzistenci vstupů s výstupy.

Příklad implementace metaheuristického algoritmu (konkrétně genetických algoritmů) je uveden i s popisem v příloze A.1.

Možnosti hybridizace

Smyslem budování obecného modelu je usnadnit hybridizaci různých metaheuristik dohromady a vtisknout jim jednotnou podobu. Rozložením chování metaheuristik na samostatné operátory a sjednocením komunikačního rozhraní lze dosáhnout snadné záměny a spolupráce jednotlivých konceptů. Každý operátor je z vnějšku popsán počtem a délkami svých vstupních posloupností a stejně tak i délkou své výstupní posloupnosti. Pokud tedy existují dva různé operátory, které mají stejné velikosti vstupů a výstupu, jsou vzájemně zaměnitelné.

V případě metody *Tabu search* se z jednoho bodu generuje jeho okolí a z tohoto okolí se zase vybere jediné řešení. Jsou to tedy dva operátory

$$X_{\text{tabu}_1} : \mathcal{P}^1 \to \mathcal{P}^n$$

a

$$X_{\text{tabu}_2} : \mathcal{P}^n \to \mathcal{P}^1,$$

kde n je nějaké přirozené číslo závislé na dimenzi úlohy. Operátor X_{tabu_2} má stejný typ jako například celý genetický algoritmus

$$X_{\text{GA}} : \mathcal{P}^n \to \mathcal{P}^1,$$

který z počáteční populace o velikosti n vrátí po několika iteracích nejlepší nalezené řešení. Jedním ze způsobu hybridizace těchto dvou zmíněných algoritmů tedy může být nahrazení jednoduchého výběru nejlepšího prvku z vygenerovaného okolí za celý genetický algoritmus. Doba výpočtu se tím určitě prodlouží, ale získá se tím kvalitativně nové chování algoritmu a tím i možnost lepšího výkonu pro nějaké typy účelových funkcí.

Hlavní rys algoritmu *Tabu search*, jeho hlavní odlišnost od ostatních metaheuristik, je využívání seznamu několika posledních navštívených bodů společně s metodou lokálního prohledávání. Při globálním prohledávání nemá *Tabu list* velký smysl, protože je velice malá pravděpodobnost navštívení stejných bodů víckrát po sobě během krátké doby. Zakomponovaný

genetický algoritmus do nastíněné hybridní metaheuristiky by neměl vykonávat stovky iterací, jako je tomu běžné, ale může skončit pouze po několika málo iteracích, které tak přinesou lokálnější prohledávání stavového prostoru.

Ne každé nahrazení operátoru tedy nemusí být rozumné a vnější kompatibilita operátorů by neměla být jediným indikátorem jejich možné záměny. Mnoho funkčních konceptů ale může vzniknout i z na první pohled nerozumných záměn, a proto je obecný model nijak neomezuje.

Ani samotná vnější definice celé metaheuristiky nemusí být jednoznačná. Zmíněný genetický algoritmus typicky vrací pouze nejlepší prvek, který nalezl během svého běhu. Algoritmus ale lze nadefinovat i např. jako

$$X_{\mathrm{GA}_n} : \mathcal{P}^n \to \mathcal{P}^m,$$

tedy že vrací celou závěrečnou populaci (tedy $m = n$) nebo například pouze m nejlepších prvků. Tímto přístupem lze stejný operátor využít pro hybridizaci dalších typů metaheuristik.

7.4 Shrnutí obecného modelu

Vytvořený model byl popsán třemi navzájem doplňujícími se způsoby. Koncepční model v kapitole 6.1 popisuje základní struktury určující celou povahu modelu. Jedná se o nejobecnější řídicí principy a způsob ukládání informací potřebných pro řádný běh metaheurisitky. Jako hlavní typ pro uchování dat je popsán `klíč` zastupující libovolný objekt metaheuristiky. Klíče lze spolu řetězit, čímž se mohou vytvářet vazby mezi jednotlivými entitami modelu. Hlavními funkčními objekty vykonávající práci s daty (klíči) jsou `operátory`. Operátory jsou v metaheuristikách popsány pouze pomocí vstupních a výstupních parametrů. Každá metaheuristika tedy může být definována naprosto abstraktně, tzn. bez znalosti konkrétních operátorů.

Kapitola 6.2 ukazuje matematicky formalizovaný způsob popisu vhodný pro klasifikaci operátorů podle jejich vnějšího rozhraní. Vytvořené popisy jsou vhodné pro základní popis chování, ale nejsou příliš vhodné pro reálnou práci. Implementace modelu v kapitole 7 významně zjednodušuje práci s modelem a přináší několik užitečných rozšíření (metody `map`,`apply`,anonymní bloky, automatické ukončování operátorů, ...). Všechna rozšíření jsou plně kompatibilní se základním modelem a nijak ho nemění.

Metaheuristický model je natolik silný a obecný, že dokáže přirozeně popsat velké množství metaheuristik. Lze vytvářet metaheuristiky základních typů nebo i velmi specifické, metaheuristiky pro lokální i globální prohledávání, nebo dokonce metaheuristiky paralelní s libovolnou topologií.

Hlavním smyslem modelu je ale umožnit snadnou hybridizaci různých metaheuristických konceptů. Toho lze dosáhnout rozložením známých metaheuristik na samostatné operátory (nebo jejich logické celky) a následně vytvořit abstraktní popis jejich spolupráce. Metaheuristika,

která je takto popsána, může být aplikována na řešení nějaké úlohy až po úplném dodefinování všech abstraktních parametrů. A právě toto je prostor pro hybridizaci, protože není důležité, jak přesně daný operátor uvnitř pracuje, ale je důležité, jaké dokáže zpracovat vstupy a kolik výstupů z nich vrátí. Prohazováním operátorů z jednoho konceptu do druhého lze dosáhnout odlišného chování, a tím i možnosti zvýšit šanci na nalezení optima.

Metaheuristika je tedy abstraktně popsána jako aplikování *nějakých* operátorů na *nějaké* klíče. Konkrétní operátor se přiřazuje abstraktnímu klíči až v době kompilace celého algoritmu. Některé operátory ale mají v metaheuristice své pevné místo a nemají možnost být zaměněny za jiné. Jsou to buď pomocné technické operátory a nebo takové, které tvoří samotný základ metaheuristiky a bez kterých by daná metaheuristika nebyla tím, čím je. Například pro metodu *Tabu search* to jsou operátory pracující s tabu listem a všechny ostatní jsou zaměnitelné (generování okolí, výběr nejlepšího prvku, apod.). Rozhodnutí, které operátory budou fixní a které zaměnitelné, je plně na úvaze tvůrce metaheuristiky. Obecně lze říci, že pokud operátor není závislý na typu stavového prostoru úlohy, je velmi pravděpodobné, že tvoří neoddělitelnou část metaheuristiky.

8 Práce s modelem

8.1 Měření metaheuristik

Smyslem definování metaheuristického hybridního modelu je snadná tvorba nových algoritmů pomocí kombinování klasických operátorů. K vhodnému návrhu je ovšem potřeba mít přehled o vlivu chování prováděných změn na výsledné vlastnosti prohledávání. Obecný model dovoluje, díky své přímočaré implementaci, velice snadno takové informace získávat. Stačí, když se během vykonávání metaheuristiky uloží do externího souboru všechny provedené dotazy v databázi (čtení nebo zápis), a vznikne tak sekvenční soubor, který věrně popisuje uskutečněný běh. Pro přehlednost lze do externího souboru zapisovat i spouštění jednotlivých operátorů nebo vytváření nových klíčů, ale pro analýzu to není nezbytně nutné. Struktura vytvořených *log*-souborů[94] může být např. následující

```
...
100: >|op:1#
101: c|key:1#key:2#
102: s|key:1#key:2#|5
103: g|key:5#
104: <|op:1#
105: d|key:1#
...
```

Každý řádek má stejnou strukturu `X|Y` nebo `X|Y|Z`, kde `X` znamená typ vykonaného příkazu, `Y` je použitý klíč a `Z` je předaná hodnota. Řádek 100 ukazuje vstup do operátoru uloženého pod klíčem `op:1#`, na řádku 101 je vytvořen (*create*) nový klíč `key:1#key:2#`, řádek 102 vytvořenému klíči přiřadí (*set*) hodnotu 5 a řádek 103 přečte (*get*) z databáze hodnotu přiřazenou klíči `key:5#`. Ukončení provádění operátoru `op:1#` je zapsáno na řádku 104. Poslední zobrazený řádek ukazuje vymazání klíče `key:1#` z databáze včetně jeho podklíčů (`key:1#key:2#`).

Tímto způsobem vzniknají soubory řádově s desítkami i stovkami tisíc řádků, ale díky jejich jednoduché a úplné struktuře z nich lze analyzovat celý běh metaheuristky a *normalizovanými* postupy vyvozovat závěry z většiny implementovaných agloritmů.

Rodičovské vztahy

Velmi názorným pohledem na chování metaheuristiky je vztah mezi nově vytvořenými jedinci (nová řešení problému) a mezi prvky, ze kterých byly vytvořeny (rodičovské prvky). V kontextu

[94] Soubory tohoto typu se většinou označují příponou `*.log` a zjednodušeně se nazývají jako *logovací* soubory.

log-souboru se jedná o nalezení prvků vytváření nového řešení (řádky začínající symbolem `c`, jejichŽ klíče reprezentují jednotlivá řešení) a správného přiřazení rodičů. Za rodiče jsou považováni ti jedinci, kteří poskytli svá data ve stejném operátoru, kde byl vytvořen nový prvek. Například následující ukázka demonstruje princip hledání rodičovských prvků

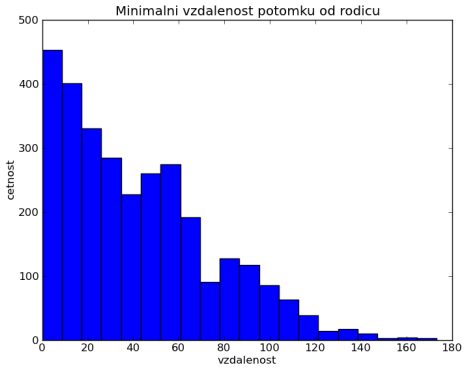
```
...
101: >|op:12#
102: c|object:128#
103: g|object:101#x#
104: g|object:101#y#
105: g|object:105#x#
106: g|object:105#y#
107: s|object:128#x#|52.8
108: s|object:128#y#|-38.1
109: <|op:12#
...
```

Nový jedinec byl vytvořen na řádku 102 uvnitř operátoru `op:12#`. Ve stejném operátoru byly získány z databáze prvky `object:101#x#`, `object:101#y#`, `object:105#x#` a `object:105#y#` (řádky 103-106). Prvky `object:101#` a `object:105#` jsou tedy považovány za rodiče prvku `object:128#`. Stejným způsobem se přisuzuje rodičovství i bez vytvoření nového klíče, ale pouze přiřazením hodnoty do podklíče objektu (např. řádky 107 a 108).

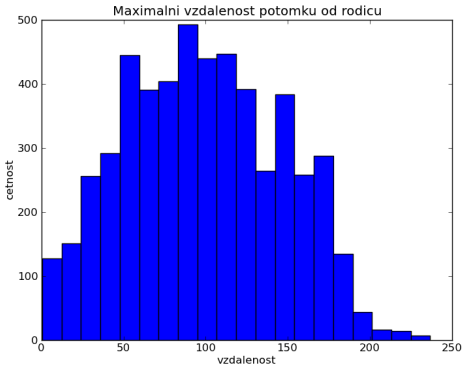
Získáním tohoto rodokmenu lze experimentálně prověřovat prohledávací schopnosti algoritmů a porovnávat je mezi sebou. Zajímavá je například geometrická vzdálenost nového řešení od rodičovských prvků, kterou lze popisovat globální a lokální povahu vytvořeného algoritmu.

Jelikož jeden prvek může mít libovolný počet rodičů, lze buď spočítat vzdálenost pro každého z nich zvlášť, nebo vytvořit jednu souhrnnou. Za rozumné souhrnné míry mohou být považovány např. vzdálenost nového řešení od nejbližšího rodiče, od nejvzdálenějšího rodiče, od těžiště rodičů nebo průměrná vzdálenost všech rodičů apod.

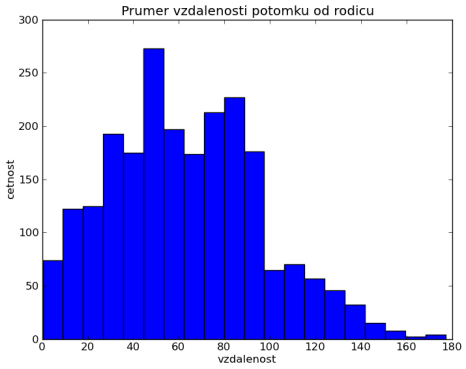
Obrázek 8.1 ukazuje výsledky genetického algoritmu použitého pro řešení Rosenbrockovy funkce. Obrázek 8.1 a) ukazuje rozdělení minimálních vzdáleností mezi rodičem a jeho potomkem v průběhu celého běhu algoritmu. Obdobně obrázek 8.1 b) ukazuje maximální vzdálenost mezi rodičem a potomkem. Jelikož v genetických algoritmech jsou pouze dva rodiče na jednoho potomka, tak jeden realizuje maximální a druhý minimální vzdálenost. Obrázek 8.1 c) ukazuje průměrnou hodnotu vzdálenosti mezi rodičem a potomkem, ale pouze pro prvních 1000 jedinců (tzn. prvních 10 generací). Obrázek 8.1 d) ukazuje také průměrné vzdálenosti, ale pro posledních 1000 jedinců. Pokud je potřeba testovat, jestli se vzdálenosti mezi rodiči a potomky v průběhu výpočtu nějak mění, stačí na data z obrázku c) a d) použít vhodný statistický test (např. Kolmogorovův-Smirnovův test apod.).



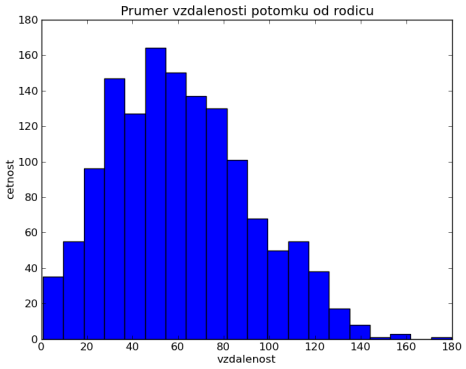
a) minimální vzdálenost



b) maximální vzdálenost



c) průměrná vzdálenost
(prvních 1000 jedinců)



d) průměrná vzdálenost
(posledních 1000 jedinců)

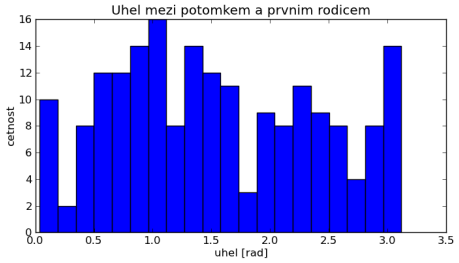
Obrázek 8.1 Histogramy vzdáleností nových řešení od rodičovských

Informace, které noví jedinci dědí od svých rodičů mohou být obecně velice různé. Některé operátory vytváří nová řešení prohazováním jednotlivých proměnných, některé vzájemným přibližování a některé lehce změní hodnoty v proměnných. Zajímavým údajem tedy mohou být i zachované souvislosti v nových řešeních. Například lze zjišťovat, kolik procent proměnných zůstává stejných nebo obdobných.

Neméně zajímavý může být i geometrický úhel mezi rodiči a potomky. Stejně jako u vzdálenosti lze definovat různé úhly (od nejbližšího prvku, ...) a vybrat z nich ty nejužitečnější pro danou aplikaci. Určité operátory produkují řešení jenom v několika málo úhlech od rodičovských prvků. Například křížení v genetických algoritmech, které pouze prohazuje hodnoty proměnných, vytváří řešení lokalizována na mřížce a úhly jsou tedy omezeny na nevelký počet možností. Operátor mutace ovšem dovoluje z mřížky vyskočit a tedy měnit i generované úhly. Navržené metaheuristiky by tedy měly být schopny, vzhledem k rovnoměrnosti prohledávání stavového prostoru, vytvářet jedince ve všech možných úhlech a nepreferovat žádný konkrétní směr.[95] Na obrázku 8.2 je zobrazeno empirické rozdělení úhlů mezi rodičovským prvkem

[95] Lze si samozřejmě představit metaheuristiku, která generuje nové jedince pouze v několika málo úhlech od

a vytvořeným jedincem (měřeno vůči kladnému směru horizontální osy). Data jsou sesbírána z dvourozměrné optimalizační úlohy řešené pomocí *Shuffled Frog-Leaping Algorithm*, kde je každé nové řešení generováno ze dvou rodičů.



Obrázek 8.2 Histogram úhlů vektoru mezi potomkem a rodičovským prvkem

Histogram 8.2 ukazuje, že výrazně preferovány nejsou žádné oblasti úhlů, i když zcela rovnoměrné prohledávání to není. Pozorované nerovnoměrnosti se mohou vytratit při vyšším počtu vzorků. Při interpretaci tří jedinců (potomek a dva rodiče) jako vrcholů trojúhelníku v rovině lze měřit i úhel při vrcholu potomka a získat tím pohled na jejich vzájemné postavení. Jelikož algoritmus ale vytváří potomky jako lineární kombinaci rodičů, tak všechny tři prvky náleží do jedné přímky a vrcholový úhel je tedy vždy π. Obdobně úhel mezi potomkem a druhým rodičem je pouze doplněk předchozího úhlu α do π (tyto informace se dají využít například pro testování správnosti naprogramovaných procedur).

Aktuální populace

Základní činností metaheuristiky je vytvářet, udržovat a mazat získaná řešení. V každý okamžik se tak *aktuální populace* může libovolně zmenšovat, rozrůstat nebo libovolně obměňovat. Analýzou *log*-souboru lze zjišťovat, jaká řešení jsou aktuálně v paměti uchovávána a podle toho usuzovat na vlastnosti metaheuristiky.

Jedna z velmi často zmiňovaných schopností metaheuristik je schopnost *explorace* versus schopnost *exploitace*. Jinými slovy, jestli dokáže algoritmus prohledávat velké oblasti stavových prostorů, nebo se spíše soustředí na malé oblasti se slibnými charakteristikami. V ideálním případě by se metaheuristika měla umět dynamicky přizpůsobit aktuální potřebě.

Z *log*-souboru lze získat údaje o každém použití objektu klíče a tedy i libovolného objektu. Nejjednodušší konstrukcí aktuální populace celé metaheuristiky je přiřazení doby života každému vytvořenému objektu (tedy doby mezi první a poslední zmínkou o objektu v *log*-souboru) a následně vytvořit chronologický soubor všech změn v aktuální populaci (tedy jak se populace mění vzhledem k iteracím algoritmu).

---

rodičovského prvku, ale její užití musí být opodstatněné a otestované.

V aktuální populaci tedy lze například měřit největší nebo průměrnou vzdálenost mezi dvěma jedinci (průměr populace), rozptyl proměnných jednotlivých řešení nebo např. počty a velikosti shluků. Všechny statistiky lze i snadno filtrovat pouze pro určité operátory.

Obrázek 8.3 Vzdálenosti mezi prvky v aktuální populaci

Graf 8.3 ukazuje vzájemné vzdálenosti prvků v aktuální populaci algoritmu *Angels & Mortals*. Zobrazeny jsou maximální průměrná a mediánová vzdálenost, přičemž aktuální populace se mění s každým nově vytvořeným řešením. Prohledávaná oblast je čtvercová se stranou $(-100, 100)$ a tedy maximální možný průměr je $200\sqrt{2}$, což zhruba odpovídá maximálním naměřeným hodnotám. Maximální vzdálenost nemá tendenci se zmenšovat a tedy algoritmus nekonverguje do jediného bodu. Křivka průměrných a mediánových vzdáleností má také víceméně konstantní charakter, což může naznačovat, že rozdělení vzdáleností v populaci se s časem příliš nemění a body rovnoměrně pokrývají celou prohledávanou oblast.

Složitost metaheuristiky

Jako lze obecné algoritmy realizované na Turingově stroji analyzovat z hlediska konceptu asymptotické složitosti, mohou být i metaheuristiky studovány z podobného pohledu. *Log*-soubor obsahuje posloupnost spouštění operátorů a přesný počet přístupů do databáze pro konkrétní instanci problému. Při zvýšení dimenze úlohy se přirozeně zvýší i počet přístupů do databáze. Ke změnám přístupů může docházet ale i změnou parametrů metaheuristiky (typicky při změně maximálního počtu iterací apod.).

Počet přístupů dává velmi dobrý náhled na složitost samotné metaheuristiky - tedy algoritmického konceptu bez implementačních detailů. Při vyšetřování klasické asymptotické složitosti se musí brát v úvahu kompletní algoritmus, což obsahuje jak řízení běhu metaheuristiky, tak i algoritmy obsažené uvnitř operátorů. Klasická složitost tedy popisuje závislost rozměru úlohy na počtu vykonaných kroků celého algoritmu. Pokud bychom uvažovali pouze počet přístupů do databáze, získáme obrázek o počtu načítání a ukládání dat bez počtu kroků, které operátory provádí uvnitř. Například v případě metaheuristiky *Estimation of distribution* (viz kapitola 4.8) se uvnitř operátoru provádí odhad rozdělení hodnot jednotlivých proměnných, který je určitě náročný na počet kroků. Z pohledu přístupu do databáze to ale pouze znamená načtení všech potřebných hodnot objektů a následné uložení parametrů odhadnutého rozdělení. Závislost počtu přístupů do databáze vzhledem k parametrům úlohy je tedy měřítkem složitosti meta-

heuristiky jako takové a lze pomocí ní experimentálně hodnotit citlivost náročnosti algoritmu na konkrétní parametry.

Algoritmy se většinou klasifikují z pohledu časové a prostorové náročnosti. Časová náročnost určuje počet vykonaných kroků, kdežto prostorová určuje paměťové nároky algoritmu. Paměťové nároky metaheuristiky lze snadno zjistit pouhým sečtením počtu záznamů v databázi, což může být hodnota přímo zjištěná z vygenerovaných *log*-souborů. Díky dynamické povaze metaheuristik je obecná paměťová náročnost metaheuristiky v jejím průběhu značně proměnná, za odpovídající a jednoduché metriky tak lze považovat například maximální nebo průměrnou dosaženou hodnotu (počet záznamů v databázi).

Další analýzy

Zmíněné experimentálně získané popisy schopností metaheuristik jsou uvedeny sice jako důležité, avšak pouze exemplární případy. Pro vývoj výkonných metaheuristik musí (může) být sledováno mnoho rozdílných ukazatelů, jejichž hodnoty mohou značně ovlivňovat celkový chod.

Významná výhoda navrženého modelu spočívá v zapouzdření celé paměti do jedné databáze. Algoritmus tak může být kdykoliv za běhu přerušen a obsah databáze uložen do jednoduchého externího souboru. Stejně tak jej lze i načíst zpět a spouštět tak algoritmus ze stejného stavu, kde předtím skončil.

Pro názornost jsou na následujících řádcích navrženy další sledovatelné vlastnosti, ale jelikož je jejich implementace díky *log*-souborům a zmíněnému ukládání vcelku snadná, tak jsou pouze stručně okomentovány.

- *Pravděpodobnost, že daný prvek bude vybrán pro produkování nových řešení.* Snadným porovnáním seznamu aktuálních populací a seznamu rodičovských jedinců lze zjistit, jestli metaheuristika systematicky nevynechává nějaká řešení. Pokud se zjistí, že aktuální populace obsahuje jedince, kteří nejsou nikdy vybíráni pro produkci nových řešení, pak je rozumné buď navrhnout operátory, které je využijí, a nebo upravit stávající, aby nevyužité jedince odstraňovaly z paměti.
- *Pravděpodobnost vyhodnocování stejného řešení vícekrát v jednom běhu.* Metaheuristiky na diskrétních stavových prostorech mohou být analyzovány na pravděpodobnost opakovaného navštívení stejného bodu. Pokud je účelová funkce pomalejší, může opakované vyhodnocování znamenat značné snížení výkonnosti metaheuristiky. V *log*-souboru lze snadno zjistit, jestli nějaký prvek nebyl vyhodnocován vícekrát a pokud ano, je vhodné tomu zabránit. V případě spojitých stavových prostorů lze analýzu provést na základě vzájemné blízkosti řešení, ale je to velmi časově náročný proces, protože musí být spočítána vzdálenost mezi všemi dvojicemi použitých řešení.[96]

- *Pravděpodobnost přežití prvku do následující generace* je primárně deskriptivní hodnota. Analýzou po sobě jdoucích aktuálních populací lze zjistit poměr mezi novými a starými jedinci v každé generaci. Jinak řečeno, kolik procent z aktuální populace se obměňuje a kolik zůstává stejných. Poměry by se ve většině metaheuristik měly v průběhu výpočtu měnit. Na začátku by mělo převažovat přidávání nových jedinců, protože nebývá obtížné nalézat lepší řešení. V koncových fázích metaheuristiky už nalezení lepších jedinců tak snadné není, a proto budou převažovat starší jedinci. Pokud jsou i ke konci běhu stále výrazně přidáváni noví jedinci, může to naznačovat, že algoritmus je stále schopný nalézat kvalitní řešení a byl tedy ukončen předčasně.

- *Uváznutí v lokálním extrému* je velmi významná charakteristika metaheuristik. V některých případech je vhodné, aby výpočet konvergoval do jednoho bodu, v jehož okolí se žádné lepší řešení nenachází, ale někdy je to naopak považováno za velmi nevhodnou vlastnost. Opakovaným spouštěním metaheuristiky na multimodální účelové funkci lze usuzovat na schopnost lokální extrém opustit. Zautomatizování analýzy může být provedeno například výpočtem těžiště a průměru aktuální populace (a jejich vzájemné srovnání s předchozími běhy).

- *Citlivost na počáteční generaci.* Většina metaheuristik začíná svůj běh z náhodně vygenerované populace, její konkrétní podoba ale může významně ovlivňovat následný průběh prohledávání. Počáteční generaci lze snadno modifikovat a testovat její různé varianty. Může být například definován operátor, který stále stejnou vygenerovanou populaci různě rotuje kolem pevného středu nebo zmenšuje/zvětšuje její rozptyl. Ze zjištěných vlastností pak mohou být odvozovány optimální tvary a pozice počátečních populací.

- *Stochastičnost algoritmu* lze posuzovat například na základě opakovaných krátkých běhů algoritmu. Ze stejné počáteční populace se algoritmus nechá vykonávat pouze několik málo iterací (např. 2 generace v genetických algoritmech). Pokud je míra stochastičnosti malá, dosažené populace si budou navzájem velice podobné, naopak pokud se v každém běhu dosáhne značně odlišné populace, je stochastičnost algoritmu velice významná a je rozumné zvážit její důsledky. Při vysoké stochastičnosti může být v algoritmu obsaženo tolik generování náhodných hodnot, že celkově převáží vytvořený metaheuristický koncept a prohledávání se pak stane čistě náhodné.

- *Citlivost k parametrům metaheuristiky.* Všechny výše zmíněné závislosti lze vyšetřovat v závislosti na nastavitelných parametrech metaheuristiky. Díky schopnosti modelu spouštět metaheuristiku z libovolného stavu lze získávat dostatek opakovaných měření a měnit podle potřeby jejich parametry.

---

[96] Časovou náročnost lze snížit návrhem odpovídajícího hashovacího algoritmu nebo vhodným řazením porovnávaných prvků.

Navržené způsoby popisů chování metaheuristických algoritmů demonstrují šíři a snadnost jejich experimentálního získávání pomocí hybridního modelu. Všechny informace jsou extrahovány z vytvářeného *log*-souboru a z jeho dvou derivátů - soupisu prvků v aktuálních populacích a přiřazení rodičovských prvků ke každému jedinci.

8.2 Možnosti hybridizace a další možnosti

Metaheuristický hybridní model slouží primárně k vytváření metaheuristik pomocí jednotného schématu a dovoluje tak jejich snadné vzájemné kombinování. Každá metaheuristika je rozdělena na posloupnost samostatných operátorů, které si spolu předávají data pomocí systému klíčů ve sdílené databázi. Tyto myšlenky tvoří základ pro obecnou a úspěšnou hybridizaci.

Do každého operátoru vstupují pouze abstraktní klíče, jejichž přiřazené hodnoty mohou být libovolného typu[97]. Stejné procedury tedy mohou být využívány pro jednotlivá řešení, celé skupiny řešení, a nebo dokonce pro numerické řídicí parametry metaheuristiky. Například řešení dvourozměrné spojité úlohy může být reprezentováno pomocí klíče `solution:1#` a hodnoty jeho jednotlivých proměnných pomocí klíčů `solution:1#x:1#` a `solution:1#x:2#`. Vypočítaná hodnota účelové funkce $f(x, y)$ se přiřadí ke klíči `solution:1#f#` a populace vybraných řešení pod klíč `population:1#`. Operátory, které metaheuristika využívá, jsou většinou dvojího druhu, buď problémově závislé a nebo problémově nezávislé. Mezi první z nich patří operátory, které jsou vytvořeny speciálně pro daný typ účelové funkce, daný typ stavového prostoru a podobně. Operátory nezávislé na typu problému tvoří samotnou podstatu metaheuristik a využívají pouze klíče, které jsou obecně stejné bez ohledu na typy přiřazených hodnot.

Část popisu metaheuristiky může vypadat například jako

```
...
10: vytvoř okolí nejlepšího prvku z populace 1
11: z vytvořeného okolí vyber 20% nejlepších prvků
12: z populace 2 vyber nejlepší řešení
13: přesuň vybraných 20% prvků směrem k nejlepšímu řešení z populace 2
...
```

Operátory na řádku 10 a 13 jsou problémově závislé, protože přímo vytváří řešení daného typu. Operátory na řádku 11 a 12 už ale problémově závislé nejsou (nemusí být), protože jim stačí pouze znát kritéria, podle kterých se vybírají nejlepší prvky. Hybridizaci, tedy nahrazení jednoho operátoru jiným, lze provádět pro oba typy zmíněných operátorů, avšak u problémově závislých musí být zaručeno, že jsou nahrazeny vzájemně si odpovídajícími operátory.

---

[97] Libovolný typ, který zvládne databáze uložit. Minimálním požadavkem jsou typy - klíč, operátor, posloupnosti klíčů a číselné hodnoty (textové řetězce).

Ruční hybridizace

V popisu metaheuristiky pomocí vytvořeného modelu není potřeba dopředu určovat žádné operátory a všechny mohou být dodefinovány až při vytváření konkrétní instance. Nejjednodušší hybridizace je tedy realizovatelná *ruční definicí* tzn., že v zápisu metaheuristiky si autor zvolí, jakým operátorem se potřebná operace bude provádět. Ve výše zmíněném příkladu si tak může autor lehce zaměnit řádek 11 za jakýkoliv jiný operátor výběru. Tento operátor nemusí rozlišovat mezi konkrétními typy řešení, tedy není důležité, jestli předané klíče reprezentují řešení v třírozměrném spojitém prostoru nebo pořadí měst v problému obchodního cestujícího. Operátor pouze vyžaduje, aby předaná posloupnost klíčů měla své podklíče `solution:1#f#`, podle kterých se výběr provádí. Operátor výběru může být nahrazen i celou jinou metaheuristikou, například předaná populace může sloužit jako počáteční populace v genetických algoritmech vracející definovaný počet nejlepších jedinců po určitém počtu provedených iterací.

Kromě výměny operátorů lze hybridizaci chápat i jako přenos metody z jednoho prostoru do jiného. Snadno si lze představit, že uvedený kousek pseudokódu bude fungovat v prostoru spojitých i diskrétních numerických vektorů, ale díky zobecnění metaheuristického konceptu pomocí systému klíčů může být kód využíván i pro celé skupiny řešení nebo i operátory samotné. Pokud je možné definovat operátory vytvářející okolí daného prvku a operátory přesouvající prvek směrem k jinému, lze využít model beze změn a to včetně zachování celého myšlenkového konceptu. Pokud je ale vytvoření takových operátorů nemožné, lze operátory zaměnit pouze za odpovídající si ve smyslu vstupů a výstupů. Myšlenka metaheuristiky se sice změní, ale obecný model zůstane zachován. Vstupní populace ve zmíněném pseudokódu tedy mohou být tvaru `[solution:1#,...,solution:n#]`, `[operator:1#,...,operator:n#]`, `[parameter:1#,...,parameter:n#]`, `[population:1#,...,population:n#]` atd.

Automatická hybridizace

Kromě ručně utvářené hybridizace je možné tvorbu nových metaheuristik i automatizovat. Je potřeba ale rozlišovat různé úrovně automatizace. Metaheuristika je zapsána pouze jako posloupnost operátorů, které jsou obecně nedefinované. Automatizace tvorby metaheuristiky tak může probíhat doplněním vhodných operátorů. Takový přístup ale vyžaduje klasifikaci jednotlivých operátorů pomocí jednotného schématu, aby bylo možné automaticky rozhodovat o jejich vzájemné zastupitelnosti. Například operátor očekávající vstup 10 klíčů může být nahrazen operátorem očekávajícím vstup obecného počtu n klíčů, ale obrácené nahrazení možné není. Tímto způsobem lze z databáze předvytvořených operátorů skládat stále nové metaheuristiky kombinující koncepty mnoha různých algoritmů dohromady.

Pokud je navíc nadefinována i vhodná metrika, která vytvořené metaheuristiky ohodnotí, může být celý proces zautomatizovaný i na úrovni hledání nejvhodnějšího algoritmu pro danou úlohu. Stejně jako se prohledává stavový prostor běžné úlohy, tak lze prohledávat i stavový prostor všech poskládatelných hybridních metaheuristik. Na obdobném přístupu je založeno i

genetické programování, kde se vyvíjí nové programy pomocí operátorů evolučních algoritmů. Kritická je ovšem otázka zvolení vhodné metriky (účelové funkce) a vhodných problémově závislých operátorů. Tyto otázky ovšem nejsou nijak výjimečné a každá metaheuristická aplikace je musí řešit stejně intenzivně. Díky snadné zaměnitelnosti operátorů v navrženém hybridním modelu je ale stejně technicky obtížné vytvořit metaheuristiku pro hledání nových metaheuristik jako pro hledání číselných vektorů v běžných optimalizačních úlohách.

Paralelizace

Kromě kombinování dílčích částí různých metaheuristik dohromady lze modelem dosáhnout velmi snadné paralelizovatelnosti. Pojem *paralelní metaheuristiky* je často chápán ve dvou různých souvislostech. Stejně jako u mnoha ostatních druhů algoritmů, tak i u metaheuristik, lze za jeden druh paralelizace pokládat rozdělení úlohy na samostatné celky, které mohou být odděleně prováděny na různých výpočetních zařízeních. Metaheuristiky jsou k takové fyzické paralelizaci velmi dobře uzpůsobeny, protože obvykle vyžadují mnoho nezávislých ohodnocení účelové funkce, což lze na samostatných počítačích přímočaře realizovat. Obtíže nastávají pokud je logika rozdělení úlohy komplikovanější a jednotlivé počítače mezi sebou musí sdílet různá data. K takovémuto účelu je ale navržený model velmi vhodně uzpůsobený. Ukládaná data mají velmi striktně danou strukturu , čímž lze snadno zjistit, které klíče (a přiřazená data) jsou potřeba na kterém počítači. Každý počítač tak může mít vlastní databázi s daty a klíče, které jsou vyžadovány na více místech zároveň , mohou být automaticky sdíleny. Vhodnou implementací tak lze snadno dosáhnout jak paralelního modelu (v jeden čas) tak i distribuovaného modelu na počítačích vzájemně pospojovaných komunikační sítí (internetem).

Druhým významem pojmu *paralelizace metaheuristik* je koncepční paralelizace. Tento způsob nemá žádný přímý vztah k typu výpočetního zařízení a může být prováděn i pouze na jediném počítači. Hlavní myšlenkou je vzájemná spolupráce několika samostatných metaheuristik, a to jak stejných druhů, tak i zcela odlišných. I k tomuto úkonu je navržený model dobře přizpůsoben, protože celá metaheuristika je v něm chápána jako ucelený operátor stejný jako každá jeho dílčí část. Spuštění více metaheuristik v rámci jednoho výpočtu se tak stává ekvivalentní spuštění pouze jedné. Výměna informací mezi jednotlivými metaheuristikami probíhá stejně jako mezi běžnými operátory (tedy pomocí předávání svých vstupů a výstupů nebo prostřednictvím sdílené databáze). Mohou tak vznikat nejrůznější inovativní přístupy jako například

```
100: alg.block_start()
101: p1 = alg.apply(copy,population)
102: p2 = alg.apply(copy,population)
103:
104: p1 = alg.apply(genetic_algorithm, p1)
```

```
105: p2 = alg.apply(swarm_optimization, p2)
106:
107: population = alg.apply(select_best, p1,p2)
108: alg.block_stop()
```

kde se na prvních řádcích (101,102) zkopíruje do proměnných `p1` a `p2` stejná populace `population` a každá se nechá nezávisle vylepšit genetickým a hejnovým algoritmem (například jenom pouze pomocí pár jednotek iterací). Z obou výsledků se následně vybere to nejlepší (řádek 107) a uloží zpět do proměnné *population*. Tento postup se může opakovat tak dlouho, dokud operátor `select_best` nevrátí hodnotu `LOOP_STOP`. Obdobně lze jednotlivé metaheuristiky i řadit za sebe nebo jim předávat a rozdělovat jednotlivé populace. Díky jednoduchému schématu skládání operátorů lze takových kombinací vytvářet v podstatě neomezený počet.

9 Příklady aplikace metaheuristických konceptů

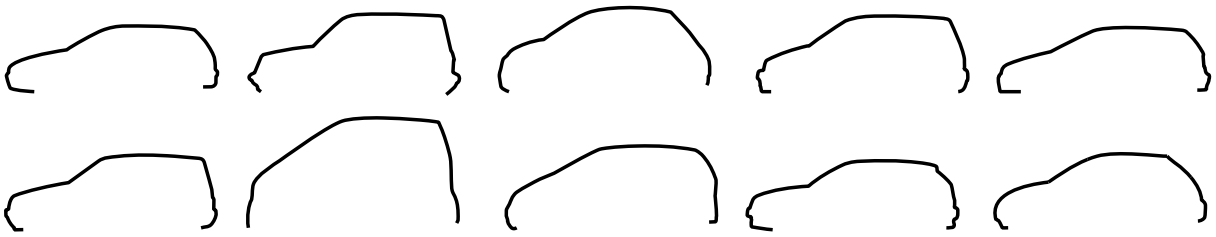
Jednou z nejdůležitějších výhod metaheuristik je široká paleta jejich možných aplikací. Vysoký počet publikovaných aplikací ukazuje na snadnou přístupnost a uchopitelnost těchto konceptů a hlavně na schopnost metaheuristik požadované řešení opravdu nalézt. Následující podkapitoly ukazují užití několika metaheuristik na řešení inženýrských i teoretických problémů. Všechny tyto příklady byly (spolu)vyvinuty a publikovány autorem této disertace a jsou seřazeny podle typizace použitých metaheuristických operátorů. První příklady využívají standardní operátory bez podstatnějších změn, kdežto poslední aplikace mají některé operátory významně přizpůsobené nebo je jejich užití něčím netypické. Uvedený popis algoritmů se nezabývá všemi implementačními detaily, ale jen hlavními myšlenkami a metaheuristickými koncepty, potenciální zájemci o chybějící informace jsou tedy laskavě odkázání na originální publikace.

Popisované aplikace vznikaly ještě před vytvořením obecného modelu a nejsou v něm tedy implementovány. Ke každé z nich je ovšem připojen i zevrubný návrh takové implementace, a to včetně popisu jednotlivých částí.

9.1 Generování siluety automobilů metodou hlavních komponent

V oblasti grafického designu nových výrobků se obvykle rozeznávají dva základní tvůrčí vývojové směry. Pozvolné drobné změny designu se označují jako *evoluční*, kdežto významné velké změny, které v předchozích verzích nemají obdoby, jsou nazývány *revoluční*. Evoluční změny jsou častější a běžný designér je schopný přicházet s takovýmito vylepšeními docela často a pravidelně. Naopak revoluční změny vyžadují značnou fantazii a schopnost oprostit se od zažitých postupů. Metaheuristiky nemají většinou žádnou doménovou znalost řešené problematiky, a proto ani nemají problém přicházet s novými a revolučními řešeními. Záleží pouze na návrhu programátora, jestli bude preferovat výsledky blízké nějakému již známému řešení, a nebo naopak nějaké vzdálené a tedy i teoreticky revoluční.

Algoritmus navržený v [113] (a dále rozšířený v [125]) dokáže zpracovat množinu vstupních křivek a na jejich základě vygenerovat novou křivku s podobnými vlastnostmi. Exemplární příklad je proveden na návrhu nové siluety automobilu. Vstupem do algoritmu je deset siluet běžných aut a výstupem je nový design, který přebírá vlastnosti vstupní množiny, ale přesto popisuje nový tvar. Výsledek je určen designérům, kterým poslouží jako inspirace, a kteří na základě zkušeností rozhodnou o jeho další použitelnosti, popř. revolučnosti. Algoritmus nejdříve analyzuje zadané křivky a potom, za pomoci získaných informací, vytváří na základě metaheuristického konceptu nový tvar.

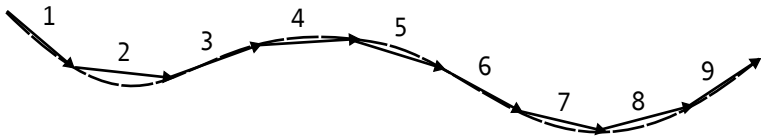


Obrázek 9.1 Křivky vstupující do algoritmu

Zpracování křivek

Každá vstupní křivka (viz obrázek 9.1) je rozdělena na stejný počet aproximujících vektorů (viz obrázek 9.2). Vstupní množinu lze tedy převést na matici M typu $m \times 2n$, kde m je počet vstupů a n je fixní počet aproximujících vektorů. Řádek $M_{(i,\cdot)}$ matice M potom reprezentuje jeden vstupní vzorek, kdežto jeden sloupec $M_{(\cdot,j)}$ reprezentuje souřadnici konkrétního vektoru napříč všemi zadanými křivkami.

$$M_{(i,\cdot)} = \left[\Delta x_1^i, \Delta y_1^i, \Delta x_2^i, \Delta y_2^i, \ldots, \Delta x_n^i, \Delta y_n^i\right] \tag{9.1}$$



Obrázek 9.2 Aproximace křivky pomocí vektorů

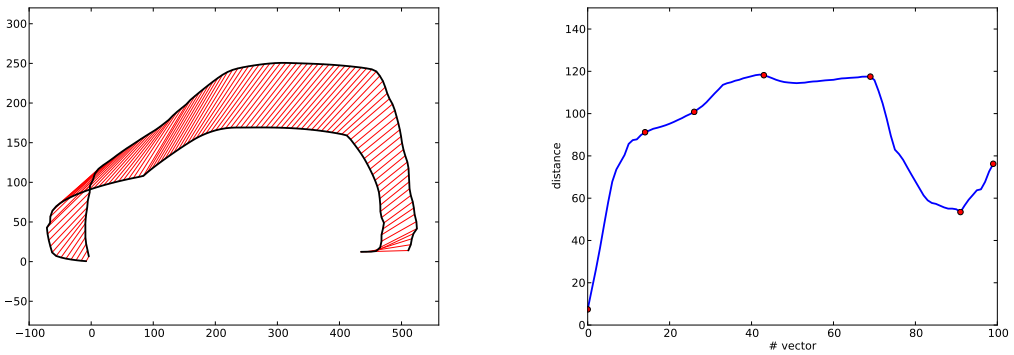
K získání *matematického modelu* automobilové siluety je použita metoda hlavních komponent ([126]). Celá silueta je nejprve rozdělena na několik dílčích částí a ty jsou pak pomocí vhodného algoritmu samostatně zpracovány. Metoda hlavních komponent je statistická metoda, jejímž primárním účelem je odhalení skrytých lineárních vazeb mezi jednotlivými sledovanými proměnnými. V kontextu křivek aproximovaných lineárními vektory jde o odhalení vazeb mezi souřadnicemi jednotlivých vektorů. Výstupem metody hlavních komponent je kovarianční matice, která umožňuje transformaci vstupních dat do nového souřadného systému, kde lze určité rozměry zanedbat[98] a tím redukovat dimenzi úlohy. Každý vstupní vektor má rozměr $1 \times 2n$ a pomocí metody hlavních komponent jej lze přetransformovat na vektor $1 \times k$, kde $k \leq 2n$ a křivku (její aproximaci) lze vyjádřit jako lineární kombinaci nevelkého počtu k vstupních souřadnic. Jednotlivé komponenty (nové souřadnice) jsou obvykle seřazeny podle jim odpovídajících rozptylů od největšího po nejmenší, a tedy první souřadnice v novém (transformovaném) systému má největší vliv na celkový tvar křivky.

Každá silueta automobilu je tedy zakódována do několika málo čísel (hlavních komponent), jejichž modifikací lze získávat nové designy. Přímočará cesta k evolučním i revolučním novým

[98] rozměry s minimálním rozptylem

designům tedy může vést skrze definování omezení a požadavků na nový design a nechat vhodný algoritmus prohledávat prostor hlavních komponent dokud nenajde odpovídající řešení. Tento přístup byl použit v článku [113], kde standardní genetický algoritmus vyhledával nové siluety na základě předem definovaného hrubého obrysu.

Modifikovaným přístupem z článku [125] lze ale získat větší variabilitu a přirozenější výsledné vlastnosti nalezeného řešení. Nejprve je potřeba rozdělit siluetu na části, které se budou analyzovat samostatně a které mají podobné charakteristiky (přední kapota, přední sklo, ...). Tyto části jsou identifikovány algoritmicky pomocí výše zmíněné metody hlavních komponent. Náhodně vybraná silueta se transformuje do prostoru hlavních komponent, kde se jí změní znaménko u první komponenty, čímž vznikne nová silueta. Vzdálenosti mezi koncovými body vektorů v nové a původní siluetě (viz obrázek 9.3 a)) jsou vyneseny jako funkce do grafu (viz obrázek 9.3 b)) a analyzovány na významné body změny charakteru funkce (lokální extrémy, inflexní body apod.). Takto identifikované body jsou pak považovány za hranice částí siluet s obdobným charakterem.



a) b)

Obrázek 9.3 Vzdálenosti mezi vektory staré a nové siluety.

Algoritmus tvorby nové siluety

Po analýze vstupních křivek a jejich rozdělení na části se náhodně vygeneruje základní *kostra* nové siluety, což jsou hraniční body rozdělených částí. Po zbytek algoritmu se hledají křivky, které nejlépe navazují na vygenerovanou kostru. Pro každou část siluety se v prostoru jejich hlavních komponent hledá takový vektor, který po zpětné transformaci dosáhne co nejblíže k aktuálnímu koncovému bodu.

Prohledávání prostoru hlavních komponent se provádí pomocí konceptu metaheuristiky *Tabu search*. Jedná se tak o optimalizaci několika reálných proměnných s minimalizací jedné účelové funkce $f(x, y)$.

$$f(x, y) = \sqrt{(x_{n,n+1} - x_C)^2 + (y_{n,n+1} - y_C)^2},$$

kde $[x_i, y_i]$ jsou souřadnice koncového bodu aktuální navrhované části siluety a $[x, y]$ jsou souřadnice koncového bodu zkoumané křivky. Prohledávání prostoru křivek probíhá v prostoru hlavních komponent, pro získání bodů $[x, y]$ je tedy nejprve potřeba aktuální pozici p v prohledávaném prostoru transformovat na data křivky d[99], ze kterých lze určit hledané koncové souřadnice křivky.

$$d = p \cdot W^T + \mu,$$

kde W je transformační (kovarianční) matice a μ je vektor středních hodnot.[100]

Pro každou část siluety se použije jeden samostatný běh metaheuristiky *Tabu search* v prostoru hlavních komponent. Počáteční pozice p se generuje náhodně a prohledáváním jeho okolí se postupuje stále blíže hledanému bodu. Pro zamezení opakovanému prohledávání je použit seznam posledních 20 navštívených bodů. V případě, že prostor hlavních komponent má více než 4 rozměry, není prohledáváno celé okolí, ale pouze jeho náhodný podvýběr. Prohledávání končí buď po nalezení minima (tzn. $f(x, y) = 0$), anebo po provedení 500 kroků.

Operátory metaheuristiky

Úloha, kterou zpracovává metaheuristika *Tabu search*, je typická jak druhem bodů prohledávaného prostoru, tak i svými využitými operátory. Prohledávaný prostor je k-rozměrný vektorový prostor $X \subseteq \mathbb{R}^k$ s klasickou eukleidovskou metrikou a operátory jsou standardní, které jsou popsané ve většině publikací o *Tabu search*.

- *Operátor generování náhodného řešení* - v prostoru hlavních komponent se vygeneruje reálný vektor délky n (tedy s počtem uvažovaných hlavních komponent), čímž se inicializuje počáteční řešení. Operátor má tvar $X_1 : \mathcal{P}^0 \to \mathcal{P}^1$.
- *Operátor generování okolí* $X_2 : \mathcal{P}^1 \to \mathcal{P}^m$. Pro každý rozměr stavového prostou je uvažované posunutí o $\pm\delta$ a operátor vrací body složené z těchto posunutí. Pro malé dimenze prohledávaných prostorů ($d < 4$) jsou vráceny všechny body okolí, pro větší dimenze je vrácena pouze náhodná podmnožina. Velikost podmnožiny a hodnota δ jsou nastavitelné parametry algoritmu.
- *Operátor výběru prvku z okolí* $X_3 : \mathcal{P}^n \to \mathcal{P}^1 \cup \{\tau\}$ vybere nejlepší prvek z předané posloupnosti (okolí). Pokud je okolí prázdné, vrátí se hodnota `LOOP_STOP` a operátor skončí.
- *Operátor Tabu search* $X_4 : \mathcal{P}^n \times \mathcal{P}^k \to \mathcal{P}^1 \cup \{\varepsilon\}$ zkontroluje, jestli se prvek nachází v tabu listu (posloupnost posledních k akceptovaných řešení). Pokud v tabu listu není, vrátí se samotný prvek a následující operátor jej přidá do posloupnosti tabu list. Pokud se ale

[99] Data d mají obdobný tvar jako ve vzorci 9.1.

[100] Jedná se o běžný vztah v metodě hlavních komponent.

naopak v tabu listu již řešení vyskytuje (je nějakému prvku blíže než zadané ω), vrátí se prázdná hodnota `NONE` a pokračuje se výběrem nového prvku z okolí.

- *Výpočet účelové funkce* $X_f : \mathcal{P}^1 \to \mathcal{P}^1$ se provádí zpětnou transformací hlavních komponent na křivku a určení vzdálenosti do aktuálního koncového bodu. K předanému klíči se přiřadí vypočítaná hodnota a prvek se vrátí zpět.
- *Ukončovací operátor* $X_{\tau_1} : \mathcal{P}^1 \to \{\tau, \varepsilon\}$ pro překročení maximálního povoleného počtu iterací. Operátor při každém zavolání zvýší svůj interní čítač o hodnotu jedna a pokud přesáhl definovaný počet, vrátí se hodnota `LOOP_STOP`.
- *Ukončovací operátor* pro dosažení optima $X_{\tau_2} : \mathcal{P}^n \to \{\tau, \varepsilon\}$. Jakmile má nějaký prvek hodnotu $f(x) = 0$ (nebo je menší než nějaké δ_τ), je vrácena hodnota `LOOP_STOP`.

Klíče

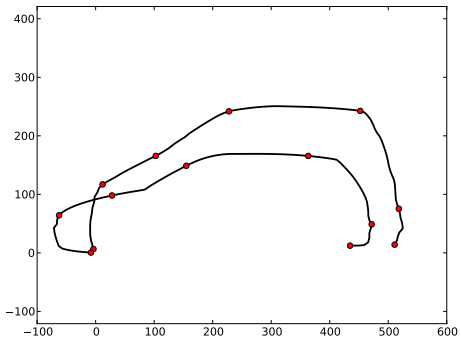
Klíče reprezentují paměť metaheuristiky a v každé z nich je jich tedy potřeba celá řada. Následující seznam z nich ukazuje pouze několik ilustračních příkladů.

- `actual_point#` je klíč pro aktuální prohledávané řešení. Tento klíč nemá žádnou hodnotu přímo přiřazenou v paměti a slouží jenom jako pojmenování asociovaných podklíčů.
- `actual_point#x:1#`,...,`actual_point#x:n#` jsou podklíče aktuálního řešení uchovávající hodnoty jednotlivých hlavních komponent.
- `actual_point#f#` je podklíč udržující hodnotu účelové funkce (vzdálenost k aktuálnímu koncovému bodu).
- `tabu_list#` má přiřazenou posloupnost klíčů předchozích řešení, která jsou zakázána pro další posun.
- `tabu_list#length#` označuje délku tabu listu.
- `previous_solution:n#` je jedno z předchozích řešení v aktuálním běhu algoritmu Tabu search, kde `n` je inkrementálně zvyšující se pořadové číslo.
- `neighbourhood#` je posloupnost okolních řešení aktuálního bodu.
- `skeleton:n#x#`, `skeleton:n#y#` jsou souřadnice n-tého bodu kostry siluety
- `skeleton:n#solution#` je řešení první části siluety.
- Potřeba je také udržovat celou řadu dalších klíčů např. `actual_number_of_iteration#`, `max_number_of_iteration#`, ...

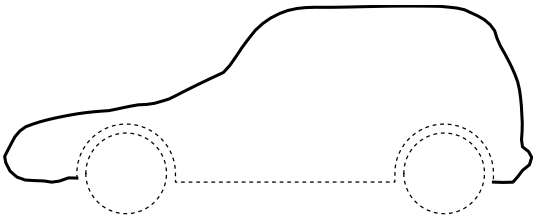
Proč je zde použita metaheuristika?

Při vhodných úpravách modelu by bylo možné některé části algoritmu řešit i lineární nebo kvadratickou optimalizací (transformace mezi prostorem hlavních komponent a prostorem křivek je realizována násobením a sčítáním; minimalizační účelová funkce může být v kvadratickém tvaru). Metaheuristický přístup je zde ale zvolen převážně z důvodu větší flexibility návrhu algoritmu a hlavně možnosti získávat nové *(r)evoluční* křivky. Každý běh algoritmu produkuje jiný výsledek a je pak až na úvaze koncového uživatele programu (designérovi), jestli nějaký ze získaných výsledků využije pro svou další práci. *Revolučnost* designu vychází ze dvou náhodných principů - nejdříve se náhodně generuje kostra a následně se náhodně generují počáteční křivky pro jednotlivé části siluet. Metaheuristika zajišťuje jen nalezení odpovídající navazující části.

Hybridizace algoritmu je možná pro většinu zmíněných operátorů. Důležité je ale zachovat lokální prohledávání, které je pro tuto úlohu charakteristickým rysem. Např. operátory, které generují okolí aktuálního bodu mohou pocházet z libovolné metaheuristiky a kombinovat tak rozličné přístupy a koncepty.



a) Rozdělení siluety na 6 částí. Zvýrazněné body tvoří kostru siluety.



b) Jeden ze získaných výstupů algoritmu (kola a podvozek jsou přikresleny pouze z ilustračních důvodů).

Obrázek 9.4

9.2 Identifikace systému pomocí zlomkového kalkulu

V publikaci [127] je navržen nový způsob k identifikaci systému, který je popsán pouze signály na svém vstupu a odpovídajícím signálem na výstupu. Klasickým způsob řešení takovéto úlohy je aproximace vnitřního mechanismu pomocí vhodné kombinace jednoduchých modelů. Každý model má vlastní přenosovou funkci a jejich vhodným počtem, parametry nebo vzájemným propojením lze dosáhnout obdobné odezvy jako v zadání problému. Nejběžnější používané modely jsou tlumiče a pružiny. Jejich závislosti mezi vstupním a výstupním signálem mají vzájemně podobný tvar $y(t) = \lambda x^{(n)}(t)$, kde $y(t)$ je výstupní signál závislý na čase t, $\lambda \in \mathbb{R}^+$ je konstanta systému (tuhost) a $x^{(n)}(t)$ je n-tá derivace vstupního signálu. Konkrétně, pružina je modelována tzv. Hookovým zákonem

$$y(t) = \lambda x(t),$$

kde výstup $y(t)$ je lineárně závislý na vstupu $x(t)$. A naopak tlumič je modelován tzv. Newtonovým zákonem

$$y(t) = \lambda x'(t),$$

kde je výstup $y(t)$ lineárně závislý na první derivaci vstupu $x(t)$.

Při identifikaci zadaného systému se tedy aproximuje transformace vstupu na výstup pomocí vhodně zvolené kombinace zapojených pružin a tlumičů dohromady. Ze známých vztahů pro paralelní a sériovou kombinaci těchto základních součástek je spočítána celková transformace, která by se měla blížit zadané transformaci co možná nejvíce.

Běžně se postupuje buď zvolením obecného modelu (konkrétního typu zapojení) a následným dopočítáním vhodných konstant λ, a nebo mohou algoritmy rovnou hledat i nejvhodnější topologii celého zapojení. Moderním přístupem k řešení problému druhým zmíněným přístupem je využití speciální metaheuristiky - evolučního programování, kde je několik různých zapojení reprezentováno stromovou strukturou a v průběhu výpočtu si vyměňují své podstromy, čímž se prohledává stavový prostor všech možných řešení úlohy.

Specialitou publikovaného článku [127] je využití nového typu základní součástky, ze které se skládá výsledné zapojení. Její matematický model zobecňuje výše zmíněný tlumič a pružinu a dovoluje tak vytvářet *hybridní součásti* kombinující jejich základní rysy chování. Matematický model je založený na *zlomkovém kalkulu*, tedy netradičním přístupu k matematické analýze dovolující pracovat s neceločíselnými řády derivací a integrálů. Součástka je tak popsána rovnicí

$$y(t) = \lambda \mathrm{D}_0^\alpha x(t),$$

kde $\lambda \in \mathbb{R}^+$ a D_0^α je derivace řádu $\alpha \in [0, 1]$. Speciálně pro $\alpha = 0$ získáváme Hookův model (pružina) a pro $\alpha = 1$ rovnice přechází do Newtonova modelu (tlumič). Ostatní hodnoty α (tedy neceločíselné derivace) tvoří plynulý přechod mezi těmito dvěma modely.

Zadáním úlohy je tedy aproximovat známou frekvenční odezvu systému $H(\omega) = \frac{Y(\omega)}{X(\omega)}$, kde $H(\omega)$ je přenosová funkce ve frekvenční oblasti a $Y(\omega)$ a $X(\omega)$ jsou Fourierovy transformace výstupu $y(t)$ a vstupu $x(t)$. Důvodem pro zvolení aproximace ve frekvenčním spektru je snadnější skládání základních součástí do propojených celků.

Fourierova transformace základního modelu součástky je

$$Y(\omega) = \lambda(\mathrm{i}\omega)^{\alpha} X(\omega),$$

a tedy její přenosová funkce může být potom zjednodušena na

$$H(\omega) = \lambda(\mathrm{i}\omega)^{\alpha}.$$

Pro paralelní zapojení dvou samostatných součástek s přenosovými funkcemi $H_1(\omega)$ a $H_2(\omega)$ lze snadno odvodit jednoduchý vztah pro společnou přenosovou funkci $H_p(\omega)$

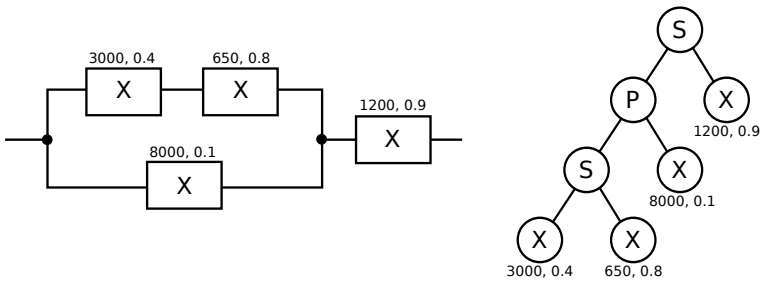
$$H_p(\omega) = H_1(\omega) + H_2(\omega)$$

a pro sériové zapojení je společná přenosová funkce $H_s(\omega)$

$$H_s(\omega) = \left(\frac{1}{H_1(\omega)} + \frac{1}{H_2(\omega)} \right)^{-1}.$$

Oba vztahy mohou být dále zobecněny na paralelní (sériové) zapojení N součástek, kde se součty ve vzorcích přímočaře zamění za sumy všech zahrnutých samostatných součástek ve stejné formě.

Algoritmus pro hledání řešení zadané úlohy je inspirován metaheuristikou evolučního programování, kde jsou jednotlivá řešení porovnávána navzájem mezi sebou a ta nejlepší spolu kombinována. Každé řešení je reprezentováno stromovou strukturou, jejíž každý samostatný podstrom tvoří určitou kombinaci základních součástí (viz obrázek 9.5). Součástky mají dva konstantní parametry - $\lambda \in \mathbb{R}^+$ a $\alpha \in [0, 1]$, kde význam odpovídá jejich protějškům v matematických modelech (tedy multiplikativní konstanta a neceločíselný řád derivace).



Obrázek 9.5

Operátory metaheuristiky

Vzhledem k běžně publikovaným operátorům evolučního programování nemusí být základní operátory pro tuto úlohu nijak zvlášť upravovány. Na začátku algoritmu se náhodně vygeneruje

100 zapojení se dvěma až šesti součástkami a každé součástce se přiřadí náhodné hodnoty α a λ. Generování a práce s těmito konstantami je jedinou významnější změnou oproti běžnému schématu. Pro generování parametru λ je použito rovnoměrné rozdělení na intervalu $[0, 10^6]$, kdežto pro parametr α je použito rozdělení tvaru U na intervalu $[0, 1]$. Při použití rovnoměrného rozdělení pro oba parametry měl algoritmus tendenci příliš rychle konvergovat k hodnotě $\alpha = 0.5$, ze které se pak už nemohl přesunout jinam.

Pro lepší přizpůsobení nalezených řešení je navíc náhodných 15% řešení vylepšováno pomocí metaheuristiky Tabu search. Při běžném tvoření nového řešení z dvou starých se totiž parametry λ a α nemění, a je proto přenecháno na *pod-metaheuristice* Tabu search, aby vhodné změny provedla.[101]

Vyhodnocení kvality konkrétního řešení, tedy výpočet účelové funkce, je prováděn na principu geometrické *blízkosti* k předepsané přenosové funkci. Zadaná přenosová funkce $H(\omega)$ je rozdělena na funkci velikosti amplitudy $G(\omega) = |H(\omega)|$ a na fázi $\phi(\omega) = \arg(\omega)$. Účelová funkce f má tvar relativní chyby a cílem prohledávání je nalézt řešení H s minimální hodnotou $f(H)$.

$$f(H) = \sum_{\omega} \left| \frac{\arg(H(\omega)) - \arg(H_0(\omega))}{\arg(H_0(\omega))} \right| + \sum_{\omega} \left| \frac{|H(\omega)| - |H_0(\omega)|}{|H_0(\omega)|} \right|$$

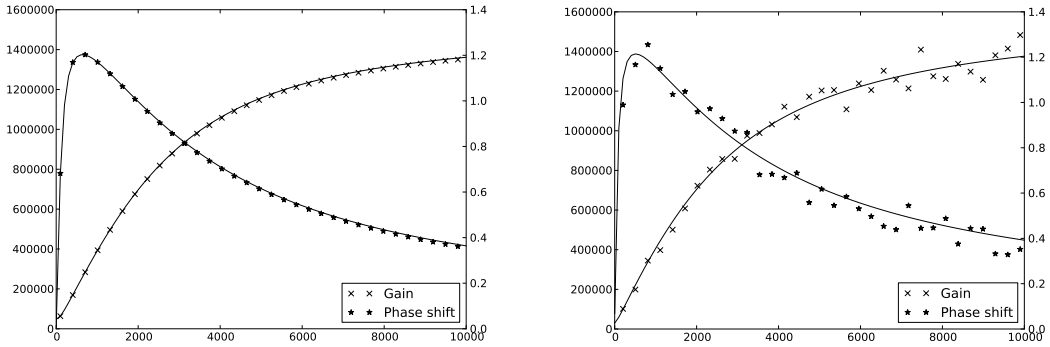
Protože v průběh algoritmu mohou nekontrolovaně růst počty základních součástek v zapojení, je každé řešení navíc penalizováno zvětšením hodnoty své účelové funkce vynásobením počtem svých součástek. Tím jsou preferována řešení s nejmenším počtem součástek v zapojení a dlouhá a komplikovaná řešení jsou průběžně odstraňována.

- *operátor generování náhodného prvku* $X_1 : \mathcal{P}^1 \to \mathcal{P}^n$ vygeneruje n náhodných zapojení a vrátí jejich seznam.
- *operátory genetických algoritmů* vyberou m párů řešení z aktuální populace, zkříží je a zahrnou zpět do populace. Další operátory vyberou malé procento prvků, kterým náhodně změní některou z jejich částí, nebo je předají operátoru Tabu search pro vylepšení numerických parametrů.
- *ukončovací operátor* vrací hodnotu `LOOP_STOP` po dosažení předdefinovaného počtu iterací.
- *operátor vyhodnocení účelové funkce* spočítá relativní chyby frekvenční odezvy řešení a penalizuje jeho délku.

Klíče

- `connection:n#` je klíč reprezentující jedno konkrétní zapojení ($n \in \mathbb{N}$). Každé zapojení je tvořeno stromem s rozličnou topologií a nekonstantním počtem vrcholů. Pro vhodnou reprezentaci lze použít zápis, kde jeden klíč odpovídá jednomu vrcholu grafu a je mu přiřazena

[101] Obdobný koncept je typický pro oblast metaheuristik nazvanou *memetické algoritmy* (viz kapitola 4.14).



a) Aproximovaná úloha b) Zadání s přidaným umělým šumem

Obrázek 9.6 Výsledky algoritmu. Diskrétní body značí zadání úlohy, spojité křivky značí nalezenou aproximaci.

posloupnost klíčů zastupující navazující vrcholy. Tedy např. klíči `connection:1#` je přiřazen vrchol `node:1#`, kterému je přiřazena posloupnost `[node:2#, node:3#]`. Každý vrchol také může mít přiřazenou libovolnou vlastnost. Speciálně je důležitý klíč `node:n#type#`, která nabývá hodnoty `S` nebo `P` označující sériové, popř. paralelní zapojení navazujících uzlů. Pokud vrchol zastupuje součástku, hodnota `node:n#type#` bude `X`. V takovém případě má vrchol ještě přiřazeny podklíče `node:n#alpha#` a `node:n#lambda#` udržující hodnoty jeho konstant. Křížení (výměna podstromů) tak může být snadno realizováno pouze jako výměna (kopie) jednotlivých podklíčů. Pro ušetření manipulace s klíči lze výhodně využít postfixovou notaci zapojení, tedy přepsání celé grafové struktury (binárního stromu) do textového řetezce (viz originální článek [127]).

- `connection:n#count#` udává počet uzlů v konkrétním zapojení a `connection:n#f#` hodnotu vypočítané účelové funkce.
- `population#` je posloupností jednotlivých aktuálních zapojení.
- `offspring#` je posloupností nově vygenerovaných řešení čekajících na zaintegrování do aktuální populace.

Proč je zde použita metaheuristika?

Úloha identifikace systému má obrovskou variabilitu možných řešení. Topologie aproximujících zapojení i parametry základních součástí mají velmi silný a těžko předvídatelný vliv na výslednou odezvu. Algoritmus musí optimalizovat celočíselné a reálné proměnné, a to zároveň i s celou dynamickou topologií zapojení. Tradiční metody k takovým úlohám nejsou dobře stavěny, a proto je zde metaheuristika velmi užitečnou volbou. Výhodou metaheuristik je i jejich snadné rozšiřování a zobecňování úlohy - lze například přidat jiné typy základních součástí nebo snadno obměnit způsob vyhodnocování kvality řešení. Silnou stránkou metaheuristického řešení je i značná robustnost vůči přidanému šumu v zadané úloze (viz obrázek 9.6).

Hybridizace algoritmu je zde velmi přímočará, kterákoliv z posloupností (population, offspring) může být přímo použita jako vstup a nebo výstup pro operátory libovolných jiných metaheuristik.

9.3 Odhad nejhoršího scénáře

Publikace [128] je věnována algoritmu pro odhad nejhorších možných rozdělení v úloze stochastické optimalizace[102]. Jedná se o úlohu dvoustupňové stochastické optimalizace, kde se v prvním stupni musí určit optimální hodnoty vektoru x tak, aby po realizaci náhodného vektoru ξ byla druhostupňová korekce co nejmenší. Využitý obecný model dvoustupňové stochastické optimalizace má tvar

$$\min\{q_1(\mathbf{x}) + \mathrm{E}Q(\mathbf{x}, \xi)\} \tag{9.2}$$

$$\text{vzhledem k } \mathbf{x} \in X \tag{9.3}$$

kde $Q(\mathbf{x}, \xi)$ je náhodná veličina závislá na náhodném vektoru ξ reprezentující optimální hodnotu problému na druhé stupni (korekce provedená po realizaci ξ)

$$\min\{q_2(\mathbf{y}, \xi)\} \tag{9.4}$$

$$\text{vzhledem k } \mathbf{y} \in Y(\mathbf{x}, \xi). \tag{9.5}$$

Funkce $q_1(\cdot)$ a $q_2(\cdot)$ jsou obecné funkce, které určují složitost modelu. V publikované práci jsou využity její lineární a kvadratické případy, ale navržený metaheuristický algoritmus je na druhu funkce nezávislý. Hlavním smyslem popsaného modelu je minimalizovat velikost akcí vyžádaných na druhém stupni vlivem realizace náhodné proměnné ξ. Akce na druhém stupni bývají mnohem nákladnější než na prvním stupni, a proto je zapotřebí, aby vstupní vektor $\mathbf{x}$ zajistil co nejlépe, že druhý stupeň nebude vyžadovat příliš velkou změnu y.

Střední hodnota $\mathrm{E}Q(\mathbf{x}, \xi)$ je velmi závislá na druhu rozdělení Ξ náhodného vektoru ξ ($\xi \sim \Xi$). V praxi není jednoduché odhadnout jaký má rozdělení Ξ tvar, a proto je vhodné studovat chování modelu v závislosti na změnách tohoto rozdělení. Pokud má ξ dostatečně malý rozptyl a funkce Q jej nijak významně nezvětší, lze celou stochastickou optimalizaci nahradit deterministickou, a tím zjednodušit řešení modelu. Pokud je ale rozptyl ξ (respektive Q) nezanedbatelný, potom významně ovlivňuje hodnotu celé účelové funkce a musí být brán v úvahu. Cílem algoritmu v popisované publikaci je nalézt takové rozdělení Ξ, které zapříčiní nejhorší výkonnost systému. Jinak řečeno, smyslem algoritmu je zjistit, co nejhoršího se může stát a jaký to bude mít dopad na studovaný systém. Úlohu lze zapsat pomocí krátkého zápisu

$$\max_{\Xi \in \Sigma_\Xi} M(\Xi), \tag{9.6}$$

kde Σ_Ξ je množina všech možných rozdělení, Ξa $M(\Xi)$ je samotná minimalizační úloha dvoustupňové stochastické optimalizace s $\xi \sim \Xi$.

[102] Stručný popis stochastické optimalizace lze nalézt v kapitole 2.2.3.

Popis algoritmu

Algoritmus je navržen primárně na základě metaheuristiky genetický algoritmus, ale využívá i klasické nástroje deterministické optimalizace. Není ovšem závislý na konkrétním typu optimalizačního algoritmu a může být tedy nahrazen libovolným jiným nástrojem schopným řešit dané úlohy.

Uchopení množiny Σ_Ξ a popis všech jejich obsažených rozdělení Ξ je provedeno aproximací pevně daným počtem k realizací náhodného vektoru $\xi \sim \Xi$. Všechna rozdělení Ξ mají stejný rozměr n a jsou aproximována ve stejně ohraničeném n-rozměrném prostoru. Blízkost a hustota realizací v tomto prostoru definuje odpovídající rozložení pravděpodobnosti.

Principem navrženého algoritmu je rozmístit k bodů na ohraničeném n-rozměrném prostoru tak, aby rozdělení, které vznikne touto aproximací, způsobilo maximální hodnotu přidružené dvoustupňové minimalizační úlohy.

Algoritmus je významně inspirován genetickým algoritmem, avšak má několik konceptuálních změn. Jednotlivé základní metaheuristické objekty (chromozomy v terminologii GA) nereprezentují řešení úlohy, jak je běžné, ale reprezentují aproximující realizace hledaného rozdělení pravděpodobnosti. Celé aproximované rozdělení (tedy řešení úlohy) je reprezentováno celou populací všech jednotlivých chromozomů dohromady. Běžný genetický algoritmus udržuje v populaci několik desítek různých řešení současně, avšak tato jeho modifikace udržuje vždy pouze jediné řešení, a to reprezentované celou populací.

Další operátory jsou už velice obdobné jako v originálních genetických algoritmech. Vybírají se jedinci s nejnižší hodnotou účelové funkce, navzájem se kříží. Křížením vznikají noví jedinci, kteří postupně nahrazují své slabší předchůdce, a to za stále stejného celkového počtu jedinců v populaci. Důležitou odlišností od standardního genetického algoritmu je ohodnocení kvality každého jedince zvlášť. Jak již bylo řečeno, hodnota účelové funkce, která má být maximalizovaná, je vztažena k celé populaci a nemůže být tedy přímo použita jako kvalita jedince (jak je v GA běžné). Proto hodnocení probíhá dvoustupňově. Nejdříve je vypočítána hodnota

$$\mathbf{x}^* = \arg M(\hat{\Xi}),$$

tedy optimální hodnota vektoru x v modelu 9.2 za předpokladu, že náhodný vektor ξ má rozdělení $\hat{\Xi}$ aproximované body v populaci algoritmu. Vektor x^* je vstupní nastavení pro první stupeň v modelu 9.2 a cena (kvalita) jedinců v populaci je interpretována jako reálná výstupní hodnota modelu, která nastane pro $x = x^*$ a $\xi = \hat{\xi}_i$. Tedy kvalita jedince je

$$f(\hat{\xi}_i) = q_1(\mathbf{x}^*) + Q(\mathbf{x}^*, \hat{\xi}_i),$$

kde funkce Q vychází z dílčí minimalizační úlohy 9.4 (druhý stupeň). Důsledkem tohoto přístupu je, že hodnota kvality jedince se mění v závislosti na aktuální populaci, a proto musí být při každé iteraci algoritmu všichni jedinci znovu ohodnoceni.

Samotný výpočet hodnoty $\mathbf{x}^*$ a kvality $f(\hat{\xi}_i)$ je přenechán na externích řešících softwarech, protože se jedná o standardizované optimalizační úlohy. Pro výpočet obou druhů hodnot jsou použity řešiče deterministických úloh, kde model 9.2 pro výpočet $\mathbf{x}^*$ je přepsán do deterministického tvaru pomocí scénářového přístupu

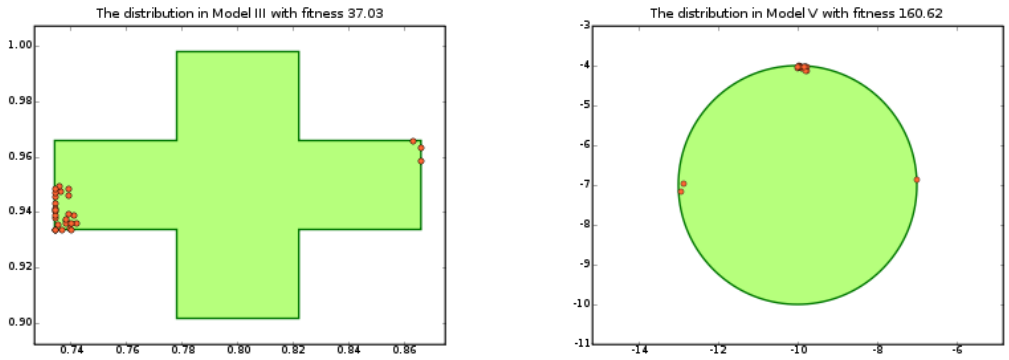
$$\mathbf{x}^* = \text{argmin} q_1(\mathbf{x}) + \frac{1}{n}\sum_{i=1}^{n} Q(\mathbf{x}, \hat{\xi}_i) \tag{9.7}$$

$$\text{vzhledem k } \mathbf{x} \in X, \tag{9.8}$$

kdežto model pro výpočet $f(\hat{\xi}_i)$ není třeba nijak modifikovat a je ponechán v původním tvaru modelu 9.4, protože se již jedná o klasickou deterministickou optimalizaci. Úloha výpočtu hodnoty $\mathbf{x}^*$ je poměrně rozsáhlá a časově náročná, velikost modelu je n-krát větší než velikost modelu pro výpočet hodnoty $f(\hat{\xi}_i)$. Stochastická optimalizace je obecně velice výpočetně náročná a celkový čas významně závisí na typu účelové funkce.

Výsledky

Pro otestování algoritmu byly použity lineární a kvadratické modely s nelineárními omezeními na definiční obory náhodné veličiny ξ. Výsledné aproximace rozdělení jsou zobrazeny na obrázku 9.7. Zelená oblast vyznačuje možné hodnoty náhodných proměnných (nejsou na sobě nezávislé) a zobrazené diskrétní body reprezentují koncovou populaci, která aproximuje hledané rozdělení pravděpodobnosti. Výsledek tedy říká, že pokud budou mít náhodné veličiny v zadaném modelu rozdělení odpovídající nalezené aproximaci, bude cena celého systému nejvyšší možná.



a) Výsledek pro lineární model b) Výsledek pro kvadratický model

Obrázek 9.7 Aproximované rozdělení způsobující největší hodnotu účelové funkce

Operátory metaheurisiky

Velká část použitých operátorů jsou klasické učebnicové případy. Nejdůležitější změna v implementaci tady nastává ve vyhodnocení účelové funkce a reprezentaci řešení.

- operátory generování počáteční populace, výběr jedinců pro křížení a zaintegrování zpět do populace jsou standardní operátory.
- *Operátor křížení* musí dbát na to, že při běžném způsobu křížení by mohla vznikat neplatná řešení (mimo definovanou oblast). Je proto přidán mechanizmus, který ověří, jestli je řešení platné a v negativním případě provede jeho korekci. Korekce se provádí přesunutím jedince na nejbližší místo do platné oblasti a navíc malý náhodný pohyb směrem dovnitř.
- Jelikož prohledávaná oblast není příliš velká a každá změna v jedinci může mít významný dopad na celou populaci, je místo klasické *mutace* prováděno pouze malé náhodné posunutí ve stavovém prostoru. Stejně jako u křížení i v tomto operátoru musí být dodrženo, že výsledný pohyb spadá dovnitř platné oblasti.
- *Operátor ohodnocení pro celou populaci* převezme všechny prvky populace a vypočítá hodnotu $\mathbf{x}^*$, kterou uloží k populaci do databáze. Výpočet se provádí externím řešičem lineárních (kvadratických) úloh.
- *Operátor ohodnocení pro každý objekt* provede externí výpočet hodnoty $f(\hat{\xi}_i)$ a uloží ji k zpracovávanému jedinci.
- *Ukončovací operátor* vrací hodnotu `LOOP_STOP` jakmile se dosáhne počet povolených iterací.

Klíče metaheuristiky

Protože je v této úloze využíváno především přístupu genetických algoritmů, odpovídá tomu i použitá struktura klíčů.

- `population#` je posloupnost klíčů reprezentující jednotlivé body aproximující hledané rozdělení,
- `point:m#` je jeden z bodů a jeho souřadnice ve v n-rozměrném prostoru odpovídají klíčům `point:m#x:1#`,...,`point:m#x:n#`,
- účelová funkce je počítána pro celou populaci zároveň a její hodnota se jí tak přímo může přiřadit pomocí klíče `population#f#`. Odpovídající hodnoty účelové funkce jednotlivých bodů jsou potom přiřazeny klíči `point:m#f#`.
- další klíče `offsprings#` atd. netřeba dále zdůrazňovat, protože odpovídají běžně užívaným proměnným v genetických algoritmech.

Proč je zde použita metaheuristika?

Stochastická optimalizace je velice náročný druh optimalizace, který pro větší úlohy může zabírat neúnosné množství výpočetních zdrojů. Speciálně uvedená *max-min úloha* vyžaduje značné

množství času a metaheuristický přístup je tedy vhodnou cestou. Absence exaktních řešicích algoritmů, způsobená náležením do třídy složitostí NP-těžká, vedla k vytvoření několika aproximačních postupů, ale jejich výsledky jsou více vzdálené od optima než ve zde uvedeném metaheuristickém přístupu.

Navržený algoritmus má velice významnou konceptuální odlišnost od klasických genetických algoritmů, a to tu, že celá populace reprezentuje pouze jediné řešení úlohy. V případě, že by genetický algoritmus pracoval klasickým způsobem, tedy že by uchovával několik desítek řešení zároveň, vzrostl by čas jeho běhu velmi rychle nad použitelnou mez. V každé iteraci se řeší dílčí optimalizační úlohy pomocí klasických deterministických algoritmů, a tedy metaheuristický přístup je tak obecný, jako jsou možnosti externích řešičů optimalizovat zadané deterministické modely.

Hybridizace je možná ve všech standardních operátorech. Křížení a mutace musí dbát na platnost řešení, protože z pohledu správnosti algoritmu musí být všechny body uvnitř platné oblasti. Operátory vyhodnocování účelové funkce tvoří hlavní myšlenku a přístup algoritmu, a proto není možná jejich koncepční záměna. Aktuální populace přiřazená jako posloupnost klíči `population#` může být ale předána jakékoliv vhodné metaheuristice (např. hejnovým algoritmům), protože princip hledání není na genetických algoritmech nijak závislý.

9.4 Optimalizace spojitého odlévání

Spojité odlévání je dnes nejvýznamnějším druhem produkce oceli a obdobných slitin. Železárny po celém světě investují obrovské částky do budování nových a vylepšování stávajících odlévacích strojů. Kvůli vysoké energetické náročnosti a neustálému tlaku zákazníků na zvyšování kvality je výroba velmi drahá a každé ušetřené procento z nákladů představuje významný obnos. Jedním ze způsobů snížení ceny tavby a zároveň zvýšení kvality výsledných odlitků je optimalizace procesu chlazení materiálu. Proces spojitého odlévání oceli lze nastavit několika numerickými parametry, které mají klíčový vliv na výslednou strukturu i na efektivitu výrobního procesu. Pro optimalizaci těchto parametrů byly na základě různých metaheuristik postupně vyvinuty vhodné algoritmy a publikované v [129], [130] a [6].

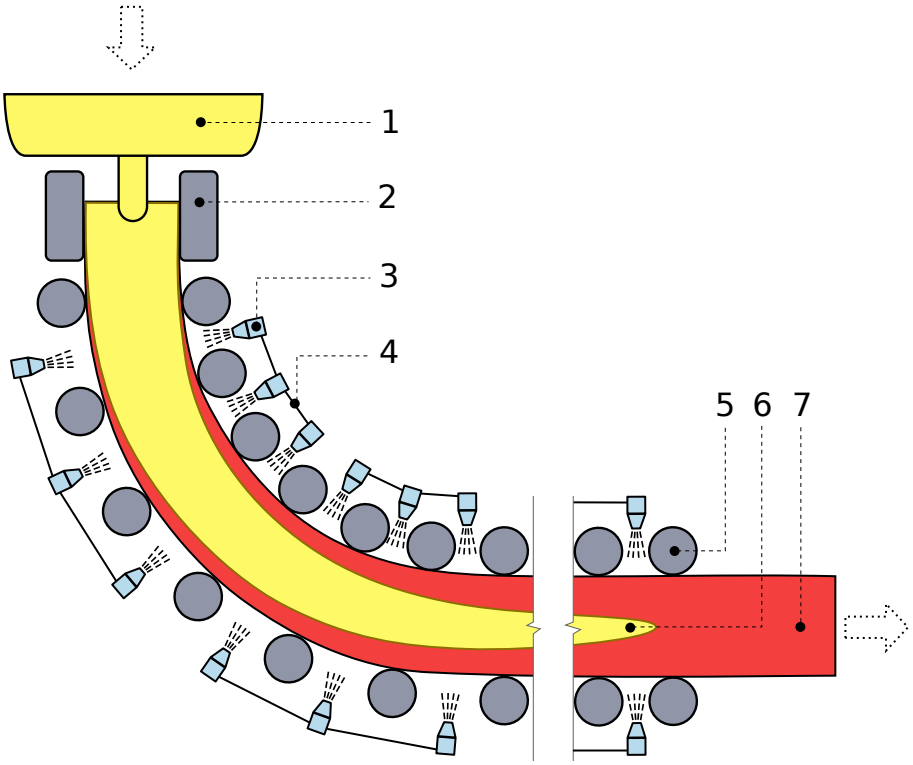
Spojité odlévání

Spojité odlévání je způsob, jak efektivně vytvářet z roztavené rudy ocelové polotovary. Schematicky je tento způsob odlévání zobrazen na obrázku 9.8. Z licí pánve v horní části stroje je vlévána roztavená ruda společně s potřebnými příměsmi do základní formy, kde mění skupenství a začíná krystalizovat. Z licí pánve neustále proudí nová roztavená ruda a částečně zkrystalizovaný materiál tak pokračuje v pohybu konstantní rychlostí. Válce formují tvar výsledného polotovaru a trysky přitom materiál vhodně ochlazují. S přibývající vzdáleností od licí pánve se zvětšuje poměr ztuhlého materiálu k tekutému a na konci licího stroje musí být materiál ztuhlý už v celém průřezu. Následně je zkrystalizovaný polotovar odříznut a ponechán samovolnému vychladnutí pouze na vzduchu.

V porovnání s původním odléváním oceli do forem je spojité odlévání mnohem rychlejší a dovoluje snadnější řízení výroby. Kvalita a vlastnosti výsledné oceli jsou dány převážně chemickým složením taveniny a procesem jejího chladnutí. Při rychlém ochlazování dochází ke koncentraci napětí a vzniku nerovnoměrné struktury materiálu, naopak pomalé chlazení sice dovoluje vytvářet velmi rovnoměrné struktury, ale za cenu významně zvýšených nákladů na spotřebované energie a čas výroby. Odlévací stroj dovoluje kontrolovat proces pomocí dvou základních parametrů, a to *licí rychlosti* (tj. rychlost, kterou je přidáván nový roztavený materiál do krystalizátoru) a průtoky chladicích trysek. Řídit průtoky v tryskách nelze v každé zvlášť, ale pouze v několika oddělených *chladicích okruzích*, do kterých jsou trysky pospojovány.

Definice úlohy

K optimalizaci popsaného spojitého odlévání byl vytvořen numerický model simulující přenos tepla v celém objemu polotvaru v průběhu celého procesu odlévání. Fyzikální proces chladnutí je popsán 2D Fourierovou-Kirchhoffovou diferenciální rovnicí, jejíž diskretizace je provedena metodou konečných diferencí. Geometrie odlévacího stroje je převzata přesně podle reálného



Obrázek 9.8 Schematický nákres spojitého odlévání. 1-licí pánev, 2-krystalizátor, 3-tryska, 4-chladicí okruh, 5-válec, 6-tekuté jádro, 7-ztuhlý materiál

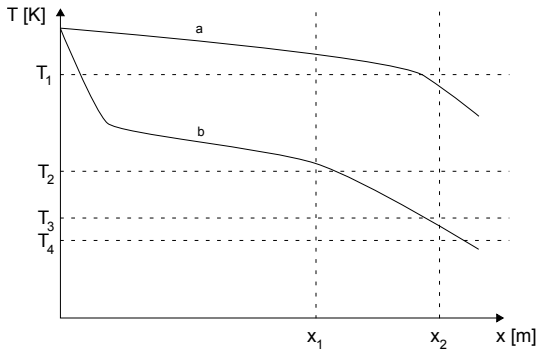
zařízení sloužícího ve Vítkovických železárnách.[103] Model dokáže na základě přesného chemického složení taveniny určit průběh teplotního rozdělení v polotovaru od počátku odlévání až do ustáleného stavu. Pro definovanou licí rychlost a průtoky tryskami (určené pomocí koeficientů přenosu tepla) se určí teplotní pole, na jehož průběh jsou následně kladeny optimalizační požadavky.

Numerický model je dvourozměrný a popisuje řez prostorem vyobrazený na obrázku 9.8. Teplotní rozložení ve třetím rozměru je považováno za stejné ve všech řezech. Optimální kvalita oceli vzniká při plynulém poklesu teploty během válcování. Velké skoky v teplotním poli mohou způsobovat koncentrace napětí, vměstky nebo nerovnoměrnost chemického složení. Před samotnou optimalizací tedy musí být odborně odhadnutý průběh teplotního pole a algoritmy pak hledají takové nastavení stroje, které zajistí co nejpodobnější výsledek.

Obrázek 9.9 zobrazuje dvě teplotní křivky. Jedná se o průběh teploty podél střední čáry odlitku (křivka a) a průběh teploty na horní ploše odlitku (křivka b). Horizontální osa určuje vzdálenost od licí pánve a vertikální osa teplotu. Bod x_1 určuje konec zahnuté části odlitku a bod x_2 konec odlévacího stroje (místo, kde je prováděno odříznutí polotovaru). Z technologického hlediska musí být dodrženy čtyři význačné teploty

[103] Celý projekt vznikal přímo ve spolupráci s Vítkovickými železárnami.

- Teplota v jádře musí ještě před bodem x_2 klesnout pod hodnotu T_1, což je teplota přeměny tekuté fáze na tuhou. V opačném případě by z licího stroje vycházely tekuté polotovary. Délka tekutého jádra od tavící pánve se nazývá *metalurgická délka* a je jedním z nejdůležitějších parametrů optimalizace.
- Teplota T_2 určuje minimální teplotu v zahnuté části odlévaného polotovaru. Pokud by pod ni teplota klesla, tak by se při rovnání mohlo vytvářet nebezpečné vnitřní pnutí.
- Teploty T_3 a T_4 jsou maximální a minimální teploty na povrchu při výstupu polotovaru ze stroje.



Obrázek 9.9 Teoreticky optimální průběh chlazení pro ocel s daným chemickým složením.

Popsané křivky jsou předány algoritmu ve formě ohraničujících intervalů pro několik vybraných bodů. Všechny body jsou zvoleny na konci chladicích okruhů na horní a dolní ploše polotovaru. Pro teplotu v jádře nejsou definovány žádné teplotní intervaly, pouze je předepsána maximální metalurgická délka, která se určí jako bod, ve kterém teplotní křivka protíná teplotu T_1.

Jedná se tedy o nelineární optimalizaci vektoru reálných proměnných za podmínky nelineárních omezení.

$$\max v \tag{9.9}$$

$$0 < v < v_{\max} \tag{9.10}$$

$$0 \leq h_i \leq h_i^{\max} \quad \text{pro } i = 1, 2, \ldots, m \tag{9.11}$$

$$T_i^{\min} \leq T_i(v, \mathbf{h}) \leq T_i^{\max} \quad \text{pro } i = 1, 2, \ldots, n \tag{9.12}$$

$$d(v, \mathbf{h}) \leq d_{\max} \tag{9.13}$$

Výsledkem modelu je licí rychlost $v \in (0, v_{\max})$ a vektor $\mathbf{h} = (h_1, h_2, \ldots, h_n)$, který reprezentuje průtoky (přesněji koeficienty přenosu tepla) v m chladicích okruzích. Maximální průtoky jsou definovány konstantami $h_i^{\max}$. Metalurgická délka je označena jako funkce $d(v, \mathbf{h})$ a je omezena maximální velikostí $d_{\max}$. Teploty v n předepsaných bodech jsou označeny jako $T_i(v, \mathbf{h})$ a omezeny teplotou $T_i^{\max}$ shora a $T_i^{\min}$ zdola.

Alternativně lze úlohu chápat i jako víceúčelovou optimalizaci, kde každá nová účelová funkce udává rozdíl mezi vypočítanou a předepsanou teplotou.

$$\min\left(-v, \Delta T_1(v,\mathbf{h}), \Delta T_2(v,\mathbf{h}), \ldots, \Delta T_k(v,\mathbf{h})\right) \tag{9.14}$$

$$0 < v < v_{\max} \tag{9.15}$$

$$d(v,\mathbf{h}) \leq d_{\max} \tag{9.16}$$

$$0 \leq h_i \leq h_i^{\max} \quad \text{pro } i = 1,2,\ldots,m, \tag{9.17}$$

kde $\Delta T_i(v,\mathbf{h}) = \left|T_i(v,\mathbf{h}) - \frac{T_i^{\min}+T_i^{\max}}{2}\right|$. Licí rychlost má v účelové funkci obrácené znaménko kvůli transformování maximalizace na minimalizaci pro dosažení kompaktnějšího zápisu.

Rozdíl mezi interpretací úlohy jako jednoúčelové nebo víceúčelové je hlavně v striktnosti dodržení požadavku na předepsané teplotní meze. Pokud je tento požadavek předepsán ve formě omezení (model 9.9), může nastat situace, že řešení vůbec nemusí existovat, čímž by běžné řešiče buď reportovaly chybu nebo by porušily některá z definovaných omezení. Model 9.14 má teplotní rozsahy zadané přímo v účelových funkcích a tedy neexistence přesného řešení nezpůsobí závažnější problém. Výsledkem řešení modelu by byl teplotní průběh, který se blíží zadanému průběhu *pouze* tak, jak je to jenom možné.

Algoritmus

Ideový princip algoritmu publikovaného v [130] je volně inspirován metaheuristikou *simulované žíhání*, kde s přibývajícími iteracemi algoritmu klesá velikost prohledávacích skoků ve stavovém prostoru. Je to ale pouze jeden z několika využitých principů, přičemž ostatní principy jsou více problémově závislé a měly by tedy být označovány jako heuristické a ne metaheuristické.

Algoritmus hledá řešení ve dvou samostatných krocích a v podstatě rozděluje i matematický model na dvě separátní úlohy. V první úloze je řešena úloha nelezení vhodných průtoků trysek pro konstantní a definovanou hodnotu licí rychlosti v_0.

$$\max_{\mathbf{h}\in H} d(v_0,\mathbf{h}) \tag{9.18}$$

$$T_i^{\min} \leq T_i(v_0,\mathbf{h}) \leq T_i^{\max} \quad \text{pro } i = 1,2,\ldots,n, \tag{9.19}$$

$$0 \leq h_i \leq h_i^{\max} \tag{9.20}$$

$$d(v,\mathbf{h}) \leq d_{\max}, \tag{9.21}$$

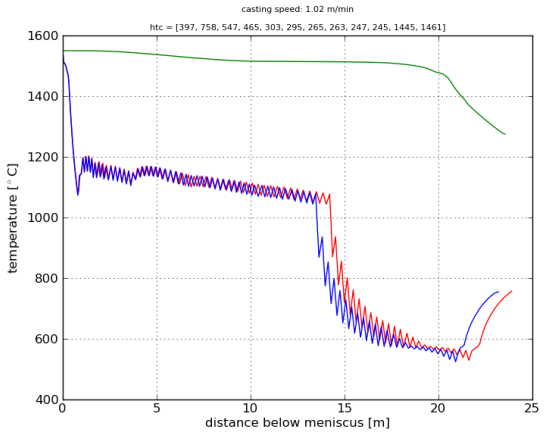
kde je smyslem modelu primárně ověřit řešitelnost zadaných podmínek pro konkrétní licí rychlost v_0. Model je formulovaný jako maximalizace metalurgické délky, protože čím delší je, tím vyšší licí rychlost může být použita. V ideálním případě je výsledná délka přesně rovna maximální povolené délce $d_{\max}$. Pro běh algoritmu ale nalezená délka není nijak důležitá, podstatné je jenom ověření řešitelnosti.

Druhým krokem algoritmu je nalezení maximální rychlosti v_0, při které ještě existuje nějaké řešení $\mathbf{h}$ splňující zadané podmínky. Pokud je pro dané v_0 model shledán řešitelným, může se rychlost zvýšit, v opačném případě se sníží. Tyto dva základní kroky algoritmu se střídají, dokud není nalezena maximální hodnota v_0.

Druhý stupeň algoritmu (maximalizace v_0) je řízen jednoduchou, ale efektivní metodou bisekce. Z fyzikální povahy procesu totiž plyne, že pokud je úloha řešitelná pro licí rychlosti v_1 a v_2, je řešitelná i pro $\forall v_0 \in [v_1, v_2]$. Ekvivalentní tvrzení platí i pro neřešitelnost modelu, a proto je takto možné postupným zkracováním intervalů hledat maximální licí rychlost.

Řešení prvního stupně však vyžaduje komplexnější přístup. Nalézt správné průtoky tryskami k dosažení optimálního poklesu teploty vyžaduje mnoho časově náročných simulací teplotního pole při různých chladicích nastaveních. Počet vyhodnocení účelové funkce (výpočet numerického modelu) je kritickou částí celého algoritmu. Při délce trvání několik minut na jedno vyhodnocení se algoritmus může lehce dostat k celkovému výpočtu na desítky hodin. Byl proto zvolen *regulační přístup*, kdy je pro definované body podél osy odlitku sledována jejich teplota a její vliv se promítne do jednoho až dvou chladicích okruhů před nimi (ve smyslu směru odlévání). Zjednodušeně řečeno, když je v kontrolovaném bodě vyšší teplota než je dovolená, začneme více chladit v chladicím okruhu těsně před ním. Naopak, pokud je teplota příliš nízká, tak chlazení snížíme. Tímto způsobem se hledá optimální průběh tak dlouho, dokud nejsou splněny všechny předepsané teplotní rozsahy a metalurgická délka není menší než maximální povolená hodnota.

Na počátku je generováno naprosto náhodné řešení a velikost prováděných změn v chladicích tryskách je největší. S každou iterací algoritmu se maximální velikost změny exponenciálně snižuje. Tento koncept je inspirován metaheuristikou simulované žíhání a zajišťuje tak postupnou konvergenci k řešení.



Obrázek 9.10 Nalezený optimální průběh chlazení. (Drobné kmity jsou způsobeny střídáním trysek s válci. Jedná se o běžný a v podstatě neovlivnitelný jev.)

Operátory metaheuristiky

Celý algoritmus lze vnímat jako dvě samostatné úlohy. Jedná se tedy o dva operátory, z nichž první, hledající optimální řešení pro konstantní rychlost, je obsažen uvnitř druhého, který rychlost mění a hledá její nejvyšší možnou hodnotu. První operátor tedy přebírá vstupem nastavenou rychlost v společně s počátečním nastavením chladicích trysek.

Na začátku algoritmu se vytvoří pomocí *operátoru generování náhodného řešení* n-tice kladných reálných náhodných čísel v dovoleném rozmezí reprezentující počáteční průtoky tryskami. Počet proměnných n je dán počtem chladicích okruhů.

Protože se často iteruje přes všechny kontrolované body s předepsanou teplotou, jsou uloženy do databáze jako samostatná posloupnost přiřazená nějakému klíči k. Každý kontrolní bod má pak svůj klíč b, stejně jako chladící okruh má svůj klíč c. Přiřazením hodnoty klíči $b.c$ definujeme přímý vliv, jaký má daný okruh c na teplotu v bodě b (čím blíže je okruh bodu, tím větší má přímý vliv).

Po vygenerování náhodného řešení se pro náhodně zvolenou rychlost v spustí operátor hledání průtoků obsahující následují pod-operátory:

- *Operátor vypočítávající aktuální teploty v kontrolovaných bodech* je založen na externí numerické simulaci, po které přiřazuje klíčům $b.t$ vypočítanou teplotu.
- *Operátor určující změnu průtoku jednotlivých trysek* je spuštěn pro každý kontrolní bod zvlášť (metodou `map`) a přiřadí každému chladicímu okruhu velikost změny (jedná se o kombinaci náhodného čísla s velikostí přímého vlivu a s definovaným znaménkem podle směru změny průtoku). Do operátoru vstupuje i aktuální hodnota maximální změny (tzv. energie v simulovaném žíhání), která úměrně zmenšuje vygenerované náhodné číslo.
- *Operátor pro řízený pokles energie* (pokles velikosti možných změn v chladicích průtocích). Každou iterací se aktuální energie vynásobí konstantou $\alpha < 0$, čímž je docílen exponenciální pokles. Jakmile hodnota klesne pod dovolenou mez, vrátí operátor hodnotu `LOOP_STOP` a algoritmus skončí.
- *Ukončovací operátor* je rovněž nastaven na kontrolování platnosti zadání. Jakmile jsou všechny podmínky splněny, není již potřeba nic měnit a pomocí hodnoty `LOOP_STOP` je algoritmus ukončen.

Výstupem operátoru, který zahrnuje všechny výše zmíněné dílčí operátory, je metalurgická délka (získávaná při numerické simulaci), která je použita při rozhodování o zvýšení nebo snížení nastavené licí rychlosti v. Rozhodnutí, jak přesně změnit licí rychlost, je vykonáno operátorem bisekce obalujícím předchozí popsaný operátor. Pokud je získaná metalurgická délka delší než maximální povolená hodnota, je rychlost snížena, a to přesně podle pravidel metody půlení intervalů. Obdobný přístup platí i pro zvýšení rychlosti za podmínky malé metalurgické délky.

Klíče metaheuristiky

- Numerický model je řešen externí rutinou a nevyžaduje tak žádné klíče pro reprezentaci svých interních proměnných. Jediné vstupy, které přebírá, jsou hodnoty chladicích průtoků a licí rychlost. Tyto hodnoty jsou přiřazeny ke klíčům `cooling_circuit:n#`. Výstupem modelu jsou hodnoty teplot ve vybraných bodech (na koncích chladících okruhů), které se přiřadí klíčům `temperature_point:n#`.
- Každý bod s hlídanou teplotou má předdefinovaný rozsah, do kterého se musí výsledná teplota vměstnat, tyto meze lze snadno přiřadit ke klíčům `temperature_point:n#min#` a `temperature_point:n#max#`.
- Všechny teplotní body jsou uchovávány v posloupnosti přiřazené ke klíči `temperature_points#`.
- Vztahy mezi jednotlivými chladicími okruhy a teplotními body jsou snadno udržovatelné pomocí klíče `temperature_point:n#cooling_circuit:m#`, kde přiřazená hodnota označuje odhadnutou velikost přímého vlivu daného okruhu na teplotu v daném bodě (čím jsou od sebe vzájemně vzdálenější, tím menší přímý vliv mají).
- V případě, že je v úloze mnoho měřicích bodů a jen málo okruhů, je z důvodu zvýšení efektivity vhodné vytvořit ke každému klíči `cooling_circuit#` podklíč s posloupností ovlivněných měřicích bodů (kterých musí být z principu pouze velmi málo).

Proč je zde použita metaheuristika?

Teplotní model spojitého odlévání, stejně jako většina modelů přenosů tepla, je velmi nelineární a optimální nastavení jeho parametrů se tak stává značně netriviální úlohou. Statistické přístupy vyžadují mnoho dat, které jsou při časově náročných simulacích velmi drahé. Klasické metaheuristické přístupy pracují na principu častého vyhodnocování účelové funkce, což zde přirozeně není schůdná cesta. Byl testován optimalizační model pomocí algoritmu Firefly [129], ale nalézt pomocí něj řešení vyžadovalo až 200 výpočtů numerického modelu. Popsaný algoritmus získává přesnější výsledky již kolem 50 vyhodnocení. Heuristický přístup se tak ukázal jako velice efektivní a vhodný i pro časově náročné účelové funkce.

Protože se nejedná o klasickou metaheuristiku s běžnými typy operátorů, není její hybridizace úplně přímočará. V publikaci [6] je algoritmus přepracován za použití fuzzy logiky, avšak hlavní myšlenky zůstaly stejné. Hybridizace je zde provedena s fuzzy přístupem a algoritmus již nevyžaduje řízení změny průtoků trysek pomocí klesající energie. Celková rychlost algoritmu se zvýšila a k nalezení řešení je potřeba méně vyhodnocení numerického modelu. Algoritmus je ovšem citlivý na pečlivé nastavení odvozovacích pravidel fuzzy logiky.

Z popsaného algoritmu lze extrahovat obecný přístup, ale ten je vhodný převážně pro úlohy se známými (alespoň přibližně) vlivy výsledku účelové funkce na jednotlivé hodnoty rozhodovacích proměnných.

9.5 Shrnutí příkladů

Zaměření a přínos zmíněných publikací je různý. V příkladě automatického generování siluet je zajímavá především celková aplikace a nepřímá reprezentace řešení. Typické metaheuristiky většinou prohledávají body stavového prostoru jejichž souřadnice přímo korespondují s hodnotami proměnných v řešené úloze. Zmíněná aplikace nejdříve ale řešení transformuje do nového prostoru (prostoru hlavních komponent), kde následně přímo probíhá i celé prohledávání. Z pohledu běhu algoritmu není podstatné, jaký konkrétní typ metaheuristiky je použit a místo navrženého *Tabu search* může být využito libovolné jiné lokální optimalizační metody. Tento příklad tak ukazuje metaheuristiky jako snadné a efektivní nástroje pro řešení nekonvenčních problémů.

Příklad s identifikací systému ukazuje jednoduchost a přímočarost rozšiřitelnosti metaheuristik o nové přístupy. Klasické identifikační metaheuristiky hledají pouze topologii sítě a vhodné typy základních buněk. V představeném příkladu je běžný způsob obohacen o využívání neceločíselných derivací, ale z pohledu metaheuristiky jde pouze o jeden spojitý parametr každé buňky navíc. Všechny operátory tak mohou zůstat prakticky nezměněné, i přestože se druh prohledávaného prostoru a matematický model v pozadí velmi významně změní.

Odhadování nejhoršího scénáře ve stochastickém programování je naopak založeno na významné obměně základního metaheuristického konceptu. Řešení úlohy není reprezentováno jedním prvkem populace, ale naopak přímo celou populací. Tento příklad ukazuje, že metaheuristické algoritmy mohou být modifikovány již ve svých základech, a přesto stále dávají velmi dobré výsledky. Metaheuristiky, speciálně genetické algoritmy, se zde ukazují pouze jako pomocná schémata, jejichž základní principy jsou zachovány, ale celková interpretace je velmi odlišná.

Poslední uvedený příklad optimalizace spojitého odlévání představuje algoritmus, který má s deklarovanou inspirací simulovaným žíháním již jen velmi málo společného. Vyhodnocení účelové funkce zde probíhá velmi zdlouhavě, a proto nemohou být použity standardní přístupy stojící na vysokém počtu ohodnocených bodů stavového prostoru. Algoritmus je navržen na míru studovanému problému a nejedná se tak přímo o metaheuristiku, ale spíše o heuristiku. Dílčí operátory ale i přesto mohou být extrahovány a použity k hybridizaci s libovolnou jinou klasickou metaheuristikou.

I přes značné koncepční odlišnosti se všechny tyto algoritmy dají rozdělit na samostatné operátory a vztahy mezi jednotlivými logickými celky a proměnnými lze popsat vhodnou strukturou klíčů. Algoritmy jsou tak přímočaře implementovatelné v navrženém obecném modelu, a lze tak snadno zajistit jejich další rozvoj a testování.

10 Závěr

Představená práce je věnována optimalizaci pomocí metaheuristických metod. Úvodní kapitoly demonstrují pestrost optimalizačních problémů stejně jako možných přístupů k jejich řešení. Následný popis jejich složitosti dále ukazuje, proč jsou některé z těchto problémů komplikovanější než jiné a hlavně, že existuje i velké množství problémů, které v dnešní době nejsme schopni efektivně řešit vůbec. Právě tyto *nejsložitější* problémy ospravedlňují existenci metaheuristického přístupu, protože jeho prostřednictvím jsou nacházeny alespoň nějaká, i když ne nutně optimální, řešení.

Metaheuristiky lze velmi zjednodušeně popsat jako *chytřejší náhodné prohledávání*. Jejich princip je založen na iterativním procházení bodů stavového prostoru úlohy a systematickém zpracovávání takto získaných informací. Metaheuristiky nejsou striktně definované algoritmy, ale spíše jen obecná schémata dovolující budovat efektivní a problémově specifické způsoby řešení. Snahou metaheuristik je tak poskytovat návod ke konstruování algoritmů bez ohledu na druh stavového prostoru úlohy. Stejným metaheuristickým konceptem proto mohou být vytvářeny algoritmy pro kombinatorické, spojité, grafové nebo jakékoliv jiné optimalizační úlohy.

Žádná metaheuristika však nemůže být univerzálně nejlepší pro všechny typy úloh. Místo hledání nejefektivnější metaheuristiky je tak rozumnější disponovat vzájemně rozdílnými koncepty, ze kterých je možné vybírat nejvhodnější typ pro danou úlohu. V současné době živelně vzniká nespočet nových metaheuristik inspirovaných nejrůznějšími přírodními fenomény. Pomocí připodobnění vytvořené metaheuristiky k nějakému objektivně fungujícímu jevu (evoluce, gravitace, biologické chování zvěře, ...) se jim často dodává potřebné zdání korektnosti a robustnosti. Všechny využité předlohy jsou ale značně zjednodušovány a často jsou u nich zanedbávány i velmi významné okolní vlivy. *Bat-inspired algorithm* tak ve skutečnosti nemá nic společného s netopýry a ani genetické algoritmy nesimulují reálné procesy biologického rozmnožování. Každá vytvořená metaheuristika však obsahuje víceméně podrobný popis jednotlivých kroků abstraktního procházení stavového prostoru úlohy. Terminologie i způsob podání se liší v závislosti na druhu deklarované inspirace, což může velmi znesnadňovat jejich používání. Při aplikaci na konkrétní problém není nijak zvlášť užitečné dogmaticky udržovat původní smysl metaheuristiky, ale může být naopak velmi přínosné inovativně obměňovat základní schéma tak dlouho, dokud nesplňuje požadované kvality. Koncept jedné metaheuristiky tak může být obohacován prvky z naprosto odlišných schémat, čímž mohou vznikat algoritmy se zcela novými vlastnostmi.

Kombinování různých schémat dohromady je obvykle nazýváno *hybridizací* a v kontextu metaheuristik se většinou jedná o intuitivní *ad-hoc* spojení několika nezávislých principů. Metaheuristiky nemají žádnou pevně danou strukturu ani definici, a proto není cesta k univerzálnímu hybridizačnímu postupu úplně přímočará. V této disertaci byl představen nový obecný způsob popisu metaheuristik vyvinutý za účelem zjednodušení návrhu i následné hybridizace.

Tento metaheuristický model je vytvořený natolik obecně, že dokáže zastoupit jakýkoliv algoritmus. Většina běžných algoritmů by ale pomocí něj byla zapsána velice těžkopádně, protože jeho hlavní rysy jsou voleny speciálně s ohledem na typické potřeby metaheuristik.

Základní datový stavební kámen modelu je *klíč* a základním funkčním prvkem je *operátor*. Klíče zobecňují přístup metaheuristiky k paměti a slouží pro předávání dat. Klíč nemá žádný datový typ a lze k němu přiřazovat libovolnou hodnotu (čísla, operátory, klíče, posloupnosti, ...). Operátory zapouzdřují určitou funkční logiku, a to buď nízkoúrovňové úkony programovacího jazyka, nebo posloupnost jiných operátorů. Každý operátor může získávat vstupy z předaných klíčů a své výstupy naopak může přiřazovat pod vlastní definovaný klíč. Každá metaheuristika je modelována jako posloupnost operátorů, které si navzájem předávají své výstupní klíče.

Všechna data jsou udržována ve sdílené databázi, do které má každý operátor neomezený přístup. Pro získání dat je ale potřeba znát přesný klíč, pod kterým jsou uložena. Klíče nezastupují pouze jednotlivá data, ale popisují i vztahy mezi nimi. Klíče tak slouží pro uspořádání dat do skupin (např. populací), pro přiřazování hodnot abstraktním objektům (např. souřadnice řešení, hodnoty účelové funkce), pro popsání relací mezi klíči (např. rodič-potomek) a mnoho dalších. Operátory převezmou předané klíče, zpracují je a vrátí výsledek. Jediná řídicí logika, která je v modelu obsažena, je cyklus `while` ukončovaný speciální hodnotou `LOOP_STOP`. Konkrétně operátory uvnitř vybrané skupiny jsou volány stále dokola, dokud libovolný z nich nevrátí `LOOP_STOP`.

Programová implementace je realizována v moderním NonSQL databázovém stylu a s pomocí plně objektového jazyka Python. V implementaci je navíc přidáno několik rozšíření značně zjednodušujících práci s modelem. Za nejvýznamnější lze považovat metody `apply()` a `map()`, které vykonají daný operátor buď jediným voláním (apply), anebo jej vykonají pro každý prvek předané posloupnosti zvlášť (map).

Navržený model významně akcentuje základní charakteristiku metaheuristik, a to jejich obecnost. Metaheuristiky jsou nezávislé na druhu stavového prostoru, a proto ani model nevyžaduje zadání konkrétního typu. Ze stejného důvodu model nevyžaduje ani zadání problémově (prostorově) závislých operátorů. Metaheuristika je pouhým konceptem a jako koncept je modelem i popsána. Definovány jsou pouze ty operátory, které tvoří samotný princip metaheuristiky, tedy např. operátory pracující se seznamem posledních navštívených bodů v metodě *Tabu search* apod.

Celý model je tedy vybudován pouze na dvou základních pojmech. Díky *klíčům* model přistupuje stejným způsobem k rozhodovacím proměnným konkrétního řešení, k vypočítaným hodnotám účelové funkce, k pořadí v posloupnosti, k parametrům celé metaheuristiky, ke všem typům relací, ke skupinám a vůbec ke všem ukládaným informacím. *Operátory* naopak zajišťují stejný způsob zacházení pro problémově závislé funkce, pro obecně definované operátory, pro skupiny operátorů i pro celé metaheuristiky. Sjednocujícím pohledem na zmíněné aspekty metaheuristik tak vznikl model, který dovoluje snadný vývoj nových hybridních postupů. Stejným

postupem se vytváří jak základní typy metaheuristik (genetické algoritmy, ...), tak i jejich významně modifikované varianty (odhad nejhoršího scénáře z kapitoly 9.3) a nebo i schémata s od základu odlišnou filosofií (hyperheuristiky). Kromě hybridizace metaheuristik je stejně tak usnadněna i jejich paralelizace a standardizace experimentálních testů chování. Schopnost popisovat obecným modelem různé metaheuristiky je demonstrována několika implementovanými ukázkami na přiloženém CD.

Závěrečná kapitola popisuje autorem (spolu)vyvinuté algoritmy pro řešení vybraných inženýrských problémů. Popsané algoritmy se od sebe významně odlišují mírou inovace v sobě obsažené. Jsou zde popsány algoritmy od téměř učebnicových případů až po velmi netypické přístupy. Všechny ale lze popsat pomocí vytvořeného jednotného hybridního modelu a využívat jejich dílčí části při dalším vývoji nových postupů.

Celá disertační práce je postavena na zkušenostech autora s úspěšnými aplikacemi metaheuristických algoritmů na inženýrské problémy. Je ale potřeba zdůraznit, že se nejedná o všelék, který vyřeší libovolný problém bez větší námahy. Z publikovaných odborných článků lze často získat pocit, že pomocí několika málo změn řídicích parametrů lze většinu metaheuristik přizpůsobit k nevídané efektivitě. Opak je ale pravdou a u komplikovanějších stavových prostorů je zapotřebí velké trpělivosti při hledání vhodné strategie a systematické zpracování mnoha experimentálních dat, k čemuž může být obecný model velmi dobře nápomocný. V mnoha publikací jsou k ověření kvality navržených algoritmů použity sice komplikované účelové funkce, ale pouze na malých ohraničených množinách. Při vyšších počtech vyhodnocení účelové funkce lze ale dosáhnout obdobných výsledků i zcela náhodným způsobem generování bodů. Metaheuristiky se tak vyplatí používat až opravdu v případě, kdy *jde to tuhého*, tedy například pro úlohy třídy NP-těžká nebo i náročnější problémy z třídy P (např. se složitostí $\mathcal{O}\left(n^{123}\right)$). Metaheuristiky nejsou samospasitelné, ale v mnoha případech již prokázaly svou užitečnost a představují tak dobrou volbu při řešení nesnadných úloh.

11 Autorovy publikace

2012

1. ŠKAROUPKA, D.; ŠANDERA, Č.; SEDLÁČKOVÁ, V.; KRÁL, J. *Developing of the manipulative sketch generator for the product design*. In 53. Mezinárodní konference kateder částí a mechanismů strojů. Brno: 2012. s. 297-302. ISBN: 978-80-214-4533- 8.
2. MAUDER, T.; ŠANDERA, Č.; ŠTĚTINA, J.; MASARIK, M. A *Fuzzy logic approach form optimal control of continuouscasting process*. In METAL 2012 Conference proceedings. Ostrava, Czech Republic: TANGER, spol. s r. o., 2012. s. 35-40. ISBN: 978-80-87294-29- 1.
3. MAUDER, T.; ŠANDERA, Č.; ŠTĚTINA, J. *A fuzzy based optimal control algotihm for a continuous castíng process*. Materiali in tehnologije, 2012, roč. 46, č. 4, s. 325-328. ISSN: 1580- 2949.
4. ŠANDERA, Č.; ŠKAROUPKA, D. *Heuristic statistical generation of graphical curves*. In Mendel 2012. Mendel Journal series. 2012. s. 494-499. ISBN: 978-80-214-4540- 6. ISSN: 1803-3814.

2011

5. MAUDER, T.; ŠTĚTINA, J.; ŠANDERA, Č.; KAVIČKA, F.; MASARIK, M. *An optimal relationship between casting speed and heat coefficients for contiuous casting process*. In METAL 2011 Conference proceedings. Metal. Ostrava: Tanger, 2011. s. 22-27. ISBN: 978-80-87294-22- 2.
6. ŠANDERA, Č.; MAUDER, T. *Optimization algorithm for continuous casting process*. In Mendel 2011. Mendel Journal series. Brno: BUT, 2011. s. 252-258. ISBN: 978-80-214-4302-0. ISSN: 1803- 3814.
7. ŠKAROUPKA, D.; ŠANDERA, Č. *Automated design of vehicle silhouette using genetic algorithms and statistical analysis*. MECCA - Journal of Middle European Costruction and Design of Cars, 2011, roč. 2011, č. 3, s. 1-5. ISSN: 1214- 0821.
8. MAUDER, T.; ŠANDERA, Č.; ŠTĚTINA, J.; ŠEDA, M. *Optimization of Quality of Continuously Cast Steel Slabs by Using Firefly Algorithm*. Materiali in tehnologije, 2011, roč. 45, č. 4, s. 347-350. ISSN: 1580- 2949.

2010

9. ŠANDERA, Č.; KISELA, T.; ŠEDA, M. *Heuristic identification of fractional- order systems*. In Mendel 2010. Mendel Journal series. 2010. s. 550-557. ISBN: 978-80-214-4120- 0. ISSN: 1803- 3814.

10. MAUDER, T.; ŠANDERA, Č.; ŠTĚTINA, J.; ŠEDA, M. *Optimization of Quality of Continuously Cast Steel Slabs by Using Firefly Algorithm*. In Program and books of abstracts of the 18 th Conference on Materials and Technology. Ljubljana, Slovenia: Institute of Materials and Technology, 2010. s. 45-46. ISBN: 978-961-92518-2- 9.

11. ŠEDA, M.; MATOUŠEK, R.; OŠMERA, P.; ŠANDERA, Č.; WEISSER, R. *Combined Heuristic Approach to Resource- Constrained Project Scheduling Problem*. Lecture notes in Electrical Engineering, 2010, roč. 2010, č. 68, s. 47-57. ISSN: 1876- 1100.

2009

12. ŠANDERA, Č.; POPELA, P.; ROUPEC, J. *The Worst Case Analysis by Heuristic Algorithms*. In Mendel 2009. Mendel Journal series. 2009. s. 109-114. ISBN: 978-80-214-3884- 2. ISSN: 1803- 3814.

13. ŠEDA, M.; MATOUŠEK, R.; OŠMERA, P.; PIVOŇKA, P.; ŠANDERA, Č. *A Flexible Heuristic Algorithm for Resource- Constrained Project Scheduling*. In World Congress on Engineering and Computer Science 2009. San Francisco, USA: IAENG, 2009. s. 730-734. ISBN: 978-988-18210-2- 7.

12 Použitá literatura

[1] DONGARRA J. and SULLIVAN F.. Top Ten Algorithms of the Century. , pages 22-23, 2000.

[2] BOYD S. and VANDENBERGHE L.. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[3] WEGENER I.. *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Springer, 2005.

[4] VAZIRANI V.. *Approximation Algorithms*. Springer, 2002.

[5] LAGOUDAKIS M. G.. Neural networks and optimization problems - a case study: The minimum cost spare allocation problem. 1997

[6] MAUDER T., ŠANDERA Č., ŠTĚTINA J. and ŠEDA M.. A fuzzy-based optimal control algorithm for a continuous casting process. *Materiali in tehnologije*, 46:325-328.

[7] OLTEAN M.. A light-based device for solving the hamiltonian path problem. In CristianS. Calude, MichaelJ. Dinneen, Gheorghe Paun, Grzegorz Rozenberg and Susan Stepney, editors, *Unconventional Computation*, number 4135 in Lecture Notes in Computer Science, pages 217-227. Springer Berlin Heidelberg, 2006.

[8] ROZENBERG G. and SALOMAA A.. Dna computing: New ideas and paradigms. In Jiri Wiedermann, Peter Emde Boas and Mogens Nielsen, editors, *Automata, Languages and Programming*, number 1644 in Lecture Notes in Computer Science, pages 106-118. Springer Berlin Heidelberg, 1999.

[9] EISERT J. and WOLF M.. Quantum computing. In AlbertY. Zomaya, editor, *Handbook of Nature-Inspired and Innovative Computing*, pages 253-286. Springer US, 2006.

[10] SHOR P. W.. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[11] GLOVER F.. Future paths for integer programming and links to artificial intelligence. *Computers and Operational Research*, 13(5), 1986.

[12] REEVES C.. *Modern heuristic techniques for combinatorial problems.*, 2005.

[13] DORIGO M. and STUTZLE T.. *Ant colony optimization*. MIT Press, 2004.

[14] OSMAN I. H. and KELLY J. P.. *Meta-heuristics: theory & applications*. Springer, 1996.

[15] VOSS S., MARTELLO S., OSMAN I., and ROUCAIROL C.. *Metaheuristics: Advances and trends in local search paradigms for optimization*. Kluwer Academic Publishers, 1999.

[16] WEBER B.. *Distributed Hybrid Metaheuristics for Optimization.*.

[17] TALBI E.-G.. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[18] YANG X.-S.. *Nature-Inspired Metaheuristic Algorithms*. Luniver Press, 2008.

[19] KARP R. M.. Reducibility among combinatorial problems. In Michael JÃijnger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser and William R. Pulleyblank et al., editors, *50 Years of Integer Programming 1958-2008*, pages 219-241. Springer Berlin Heidelberg, 2010.

[20] SCHUMACHER C., VOSE M. D. and WHITLEY L. D.. The no free lunch and problem description length. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 565–570, 2001. Morgan Kaufmann.

[21] STREETER M. J.. Two broad classes of functions for which a no free lunch result does not hold. In *Proc. Genetic and Evolutionary Computation Conference GECCO-2003*, pages 1418–1430, 2003. Morgan Kaufmann.

[22] IGEL C. and TOUSSAINT M.. On classes of functions for which no free lunch results hold. *Inf. Process. Lett.*, 86(6):317–321, 2003.

[23] ALBA E.. *Parallel Metaheuristics: A New Class of Algorithms*. Wiley-Interscience, 2005.

[24] BOSCHETTI M. A., MANIEZZO V., ROFFILLI M. and BOLUFé RöHLER A.. Matheuristics: Optimization, simulation and control. In *Proceedings of the 6th International Workshop on Hybrid Metaheuristics* HM '09, pages 171–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] RAIDL G. R.. A unified view on hybrid metaheuristics. In *Proceedings of the Third international conference on Hybrid Metaheuristics* HM'06, pages 1–12, Berlin, Heidelberg, 2006. Springer-Verlag.

[26] VAN LUONG T., TAILLARD E., MELAB N. and TALBI E.-G.. Parallelization strategies for hybrid metaheuristics using a single gpu and multi-core resources. In *Proceedings of the 12th international conference on Parallel Problem Solving from Nature - Volume Part II* PPSN'12, pages 368–377, Berlin, Heidelberg, 2012. Springer-Verlag.

[27] POSPICHAL P.. Gpu-based acceleration of the genetic algorithm. In *Proceedings of the 16th Conference Student EEICT 2010 Volume 5*, pages 234–238, 2010. Faculty of Information Technology BUT.

[28] VEGA-RODRíGUEZ M. A., GóMEZ-PULIDO J. A., ALBA E., VEGA-PéREZ D. and PRIEM-MENDES S. et al.. Evaluation of different metaheuristics solving the rnd problem. In *Proceedings of the 2007 EvoWorkshops 2007 on EvoCoMnet, EvoFIN, EvoIASP,EvoINTERACTION, EvoMUSART, EvoSTOC and EvoTransLog: Applications of Evolutionary Computing*, pages 101–110, Berlin, Heidelberg, 2007. Springer-Verlag.

[29] DEAN J. and GHEMAWAT S.. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6* OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[30] VERMA A., LLORA X., GOLDBERG D. and CAMPBELL R.. Scaling genetic algorithms using mapreduce. In *Intelligent Systems Design and Applications, 2009. ISDA '09. Ninth International Conference on*, pages 13-18, 2009.

[31] TAGAWA K. and ISHIMIZU T.. Concurrent differential evolution based on mapreduce. *International Journal of Computers*, 4:161-168, 2010.

[32] MOSTAFA H., KHADRAGI A. and HANAFI Y.. Hardware implementation of genetic algorithm on fpga. In *Radio Science Conference, 2004. NRSC 2004. Proceedings of the Twenty-First National*, pages C9-1-9, 2004.

[33] HOLLAND J.. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

[34] CHELOUAH R. and SIARRY P.. A continuous genetic algorithm designed for the global optimization of multimodal functions. *Journal of Heuristics*, 6(2):191-213, 2000.

[35] KENNEDY J. and EBERHART R.. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, pages 1942-1948 vol.4, 1995.

[36] SEDIGHIZADEH D. and MASEHIAN E.. Particle swarm optimizationmethods, taxonomy and applications. *International Journal of Computer Theory and Engineering*, 1:486-502, 2009.

[37] ZHENG J., CHEN Y. and ZHANG W.. A survey of artificial immune applications. *Artificial Intelligence Review*, 34(1):19-34, 2010.

[38] BERNARDINO H. S. and BARBOSA H. J.. Artificial immune systems for optimization. In Raymond Chiong, editor, *Nature-Inspired Algorithms for Optimisation*, number 193 in Studies in Computational Intelligence, pages 389-411. Springer Berlin Heidelberg, 2009.

[39] HAKTANIRLAR ULUTAS B. and KULTUREL-KONAK S.. A review of clonal selection algorithm and its applications. *Artificial Intelligence Review*, 36(2):117-138, 2011.

[40] TIMMIS J. and EDMONDS C.. A comment on opt-ainet: an immune network algorithm for optimisation. In *Genetic and Evolutionary Computation*, pages 308–317, 2004. Springer.

[41] KIRKPATRICK S., GELATT C. D. and VECCHI M. P.. Optimization by simulated annealing. *Science*, 220(4598):671-680, 1983.

[42] CERNY V.. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41-51, 1985.

[43] GEEM Z. W., KIM J. H. and LOGANATHAN G. V.. A new heuristic optimization algorithm: Harmony search. *Simulation*, pages 60-68, 2001.

[44] PATIL S. A. and PATEL D. A.. An overview: Improved harmony search algorithm and its applications in mechanical engineering. *International Journal of Engineering Science and Innovative Technology*, 2, 2013.

[45] YANG X.-S.. Firefly algorithms for multimodal optimization. *Stochastic Algorithms Foundations and Applications*, 5792(Lecture Notes in Computer Sciences):169–178, 2010.

[46] STORN R. and PRICE K.. Differential Evolution- A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces. Technical Report, 1995.

[47] ALI M. and TORN A.. Population set based Global Optimization Algorithms : Some modifications and Numerical Studies. 2003

[48] BREST J., BOSKOVIC B., GREINER S., ZUMER V. and MAUCEC M. S.. Performance comparison of self-adaptive and adaptive differential evolution algorithms. *Soft Computing*, 11(7):617-629, 2007.

[49] MUHLENBEIN H. and PAASS G.. From recombination of genes to the estimation of distributions i. binary parameters. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature âĂŤ PPSN IV*, number 1141 in Lecture Notes in Computer Science, pages 178-187. Springer Berlin Heidelberg, 1996.

[50] ZHIGLJAVSKY A. A.. *Theory of Global Random Search (Mathematics and its Applications)*. Springer, 1 edition, 1991.

[51] MüHLENBEIN H.. The equation for response to selection and its use for prediction. *Evol. Comput.*, 5(3):303–346, 1997.

[52] HANDA H.. Estimation of distribution algorithms with mutation. In Gunther R. Raidl and Jens Gottlieb, editors, *Evolutionary Computation in Combinatorial Optimization*, number 3448 in Lecture Notes in Computer Science, pages 112-121. Springer Berlin Heidelberg, 2005.

[53] HARIK G., LOBO F. G. and GOLDBERG D. E.. The compact genetic algorithm. In *IEEE transactions on evolutionary computation*, pages 523–528, 1998.

[54] PELIKAN M. and MUEHLENBEIN H.. The bivariate marginal distribution algorithm. In Rajkumar Roy, Takeshi Furuhashi and PravirK. Chawdhry, editors, *Advances in Soft Computing*, pages 521-535. Springer London, 1999.

[55] LARRANAGA P., ETXEBERRIA R., LOZANO J. A. and PENA J. M.. Combinatorial optimization by learning and simulation of bayesian networks. In *in Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 343–352, 2000. Morgan Kaufmann.

[56] PELIKAN M., SASTRY K. and GOLDBERG D. E.. iboa: the incremental bayesian optimization algorithm. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation* GECCO '08, pages 455–462, New York, NY, USA, 2008. ACM.

[57] EUSUFF M. M. and LANSEY K. E.. Optimization of water distribution network design using the shuffled frog leaping algorithm. *Journal of Water Resources Planning and Management*, 129(3):210–225, 2003.

[58] VAKIL BAGHMISHEH M., MADANI K. and NAVARBAF A.. A discrete shuffled frog optimization algorithm. *Artificial Intelligence Review*, 36(4):267-284, 2011.

[59] COMELLAS F. and GALLEGOS R.. Angels & mortals: A new combinatorial optimization algorithm. In WilliamE. Hart, J.E. Smith and N. Krasnogor, editors, *Recent Advances in Memetic Algorithms*, number 166 in Studies in Fuzziness and Soft Computing, pages 397-405. Springer Berlin Heidelberg, 2005.

[60] TAMURA K. and YASUDA K.. Spiral dynamics inspired optimization. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 15:1116-1122, 2011.

[61] TAMURA K. and YASUDA K.. Statistical analysis based adjustment method for convergence rate of spiral optimization. In BijayaKetan Panigrahi, Swagatam Das, PonnuthuraiNagaratnam Suganthan and PradiptaKumar Nanda, editors, *Swarm, Evolutionary, and Memetic Computing*, number 7677 in Lecture Notes in Computer Science, pages 653-661. Springer Berlin Heidelberg, 2012.

[62] COWLING P., KENDALL G. and SOUBEIGA E.. A hyperheuristic approach to scheduling a sales summit. In Edmund Burke and Wilhelm Erben, editors, *Practice and Theory of Automated Timetabling III*, number 2079 in Lecture Notes in Computer Science, pages 176-190. Springer Berlin Heidelberg, 2001.

[63] QU R., HYDE M. R., GENDREAU M., BURKE E. K. and KENDALL G. et al.. Hyperheuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 2012.

[64] ÖZCAN E., BILGIN B. and KORKMAZ E. E.. A comprehensive analysis of hyperheuristics. *Intell. Data Anal.*, 12(1):3–23, 2008.

[65] DORIGO M. and STüTZLE T.. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[66] DORIGO M., CARO G. D. and GAMBARDELLA L. M.. Ant algorithms for discrete optimization. *Artificial Life*, 5(2):137-172, 1999.

[67] SOCHA K. and DORIGO M.. Ant colony optimization for continuous domains. *European Journal of Operational Research*, 185(3):1155 - 1173, 2008.

[68] DAS S., BISWAS A., DASGUPTA S. and ABRAHAM A.. Bacterial foraging optimization algorithm: Theoretical foundations, analysis, and applications. In Ajith Abraham, Aboul-Ella Hassanien, Patrick Siarry and Andries Engelbrecht, editors, *Foundations of Computational Intelligence Volume 3*, number 203 in Studies in Computational Intelligence, pages 23-55. Springer Berlin Heidelberg, 2009.

[69] YANG X.-S.. A new metaheuristic bat-inspired algorithm. In Juan R. GonzÃąlez, David Alejandro Pelta, Carlos Cruz, German Terrazas and Natalio Krasnogor, editors, *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, number 284 in Studies in Computational Intelligence, pages 65-74. Springer Berlin Heidelberg, 2010.

[70] KAVEH A. and FARHOUDI N.. A new optimization method: Dolphin echolocation. *Advances in Engineering Software*, 59(0):53 - 70, 2013.

[71] OFTADEH R., MAHJOOB M. and SHARIATPANAHI M.. A novel meta-heuristic optimization algorithm inspired by group hunting of animals: Hunting search. *Computers and Mathematics with Applications*, 60(7):2087 - 2098, 2010.

[72] GANDOMI A., YANG X.-S. and ALAVI A.. Cuckoo search algorithm: a metaheuristic approach to solve structural optimization problems. *Engineering with Computers*, pages 1–19, 2011.

[73] CHU S.-C., TSAI P.-w. and PAN J.-S.. Cat swarm optimization. In Qiang Yang and Geoff Webb, editors, *PRICAI 2006: Trends in Artificial Intelligence*, number 4099 in Lecture Notes in Computer Science, pages 854-858. Springer Berlin Heidelberg, 2006.

[74] MOZAFFARI A., FATHI A. and BEHZADIPOUR S.. The great salmon run: a novel bio-inspired algorithm for artificial system design and optimisation. *Int. J. Bio-Inspired Comput.*, 4(5):286–301, 2012.

[75] KAMMERDINER A., MUCHERINO A. and PARDALOS P.. Application of monkey search meta-heuristic to solving instances of the multidimensional assignment problem. In MichaelJ. Hirsch, ClaytonW. Commander, PanosM. Pardalos and Robert Murphey, editors, *Optimization and Cooperative Control Strategies*, number 381 in Lecture Notes in Control and Information Sciences, pages 385-397. Springer Berlin Heidelberg, 2009.

[76] YANG X.-S. and DEB S.. Eagle strategy using levy walk and firefly algorithms for stochastic optimization. In Juan R. Gonzalez, David Alejandro Pelta, Carlos Cruz, German Terrazas and Natalio Krasnogor, editors, *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, number 284 in Studies in Computational Intelligence, pages 101-111. Springer Berlin Heidelberg, 2010.

[77] AFSHAR A., HADDAD O. B., MARINO M. and ADAMS B.. Honey-bee mating optimization (hbmo) algorithm for optimal reservoir operation. *Journal of the Franklin Institute*, 344(5):452 - 462, 2007.

[78] KARABOGA D.. An idea based on Honey Bee Swarm for Numerical Optimization. Technical Report, Erciyes University, 2005.

[79] KARABOGA D. and AKAY B.. A survey: algorithms simulating bee swarm intelligence. *Artificial Intelligence Review*, 31(1-4):61-85, 2009.

[80] COMELLAS F. and NAVARRO M.. Bumblebees: A multiagent combinatorial optimization algorithm inspired by social insect behaviour. , 2009.

[81] MARINAKIS Y., MARINAKI M. and MATSATSINIS N.. A hybrid bumble bees mating optimization - grasp algorithm for clustering. In Emilio Corchado, Xindong Wu, Erkki Oja, Alvaro Herrero and Bruno Baruque, editors, *Hybrid Artificial Intelligence Systems*, number 5572 in Lecture Notes in Computer Science, pages 549-556. Springer Berlin Heidelberg, 2009.

[82] HEDAYATZADEH R., SALMASSI F., KESHTGARI M., AKBARI R. and ZIARATI K.. Termite colony optimization: A novel approach for optimizing continuous problems. In *Electrical Engineering (ICEE), 2010 18th Iranian Conference on*, pages 553-558, 2010.

[83] KRISHNANAND K. and GHOSE D.. Glowworm swarm optimization for simultaneous capture of multiple local optima of multimodal functions. *Swarm Intelligence*, 3(2):87-124, 2009.

[84] ABIDIN Z., ARSHAD M. and NGAH U.. A simulation based fly optimization algorithm for swarms of mini autonomous surface vehicles application. *Indian journal of goe marine sciences*, 2011.

[85] RASHEDI E., NEZAMABADI-POUR H. and SARYAZDI S.. Gsa: A gravitational search algorithm. *Information Sciences*, 179(13):2232–2248, 2009.

[86] FORMATO R. A.. Central force optimization: A new deterministic gradient-like optimization metaheuristic. *OPSEARCH*, 46(1):25-51, 2009.

[87] EROL O. K. and EKSIN I.. A new optimization method: Big bang-big crunch. *Adv. Eng. Softw.*, 37(2):106–111, 2006.

[88] KRIPKA M. and KRIPKA R. M. L.. Big crunch optimization method. In *EngOpt - International Conference on Engineering Optimization*, 2008.

[89] SHAH T. and HOSSEINI H.. Principal components analysis by the galaxy based search algorithm a novel metaheuristic for continuous optimisation. *Int. J. Comput. Sci. Eng.*, 6(1/2):132–140, 2011.

[90] HATAMLOU A.. Black hole: A new heuristic optimization approach for data clustering. *Information Sciences*, 222(0):175 - 184, 2013. Including Special Section on New Trends in Ambient Intelligence and Bio-inspired Systems.

[91] CUEVAS E., OLIVA D., ZALDIVAR D., PéREZ-CISNEROS M. and SOSSA H.. Circle detection using electro-magnetism optimization. *Inf. Sci.*, 182(1):40–55, 2012.

[92] KAVEH A. and TALATAHARI S.. A novel heuristic optimization method: charged system search. *Acta Mechanica*, 213(3-4):267-289, 2010.

[93] LU H.-j., ZHANG H.-m. and MA L.-h.. A new optimization algorithm based on chaos. *Journal of Zhejiang University SCIENCE A*, 7(4):539-542, 2006.

[94] NARAYANAN A. and MOORE M.. Quantum-inspired genetic algorithms. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 61-66, 1996.

[95] CHEN Z. and LUO P.. Qisa: Incorporating quantum computation into simulated annealing for optimization problems. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 2480-2487, 2011.

[96] BISWAS A., MISHRA K. K., TIWARI S. and MISRA A. K.. Physics-inspired optimization algorithms: A survey. *Journal of Optimization*, 2013, 2013.

[97] RABANAL P., RODRIGUEZ I. and RUBIO F.. Using river formation dynamics to design heuristic algorithms. In SelimG. Akl, CristianS. Calude, MichaelJ. Dinneen, Grzegorz Rozenberg and H.Todd Wareham, editors, *Unconventional Computation*, number 4618 in Lecture Notes in Computer Science, pages 163-177. Springer Berlin Heidelberg, 2007.

[98] GAO-WEI Y. and ZHANJU H.. A novel atmosphere clouds model optimization algorithm. In *Computing, Measurement, Control and Sensor Network (CMCSN), 2012 International Conference on*, pages 217-220, 2012.

[99] ESKANDAR H., SADOLLAH A., BAHREININEJAD A. and HAMDI M.. Water cycle algorithm - a novel metaheuristic optimization method for solving constrained engineering optimization problems. *Computers and Structures*, 110-111(0):151-166, 2012.

[100] SIMON D.. Biogeography-based optimization. *IEEE Transactions on Evolutionary Computation*, 12, 2008.

[101] YANG X.-S.. Flower pollination algorithm for global optimization. In Jerome Durand-Lose and Natasa Jonoska, editors, *Unconventional Computation and Natural Computation*, number 7445 in Lecture Notes in Computer Science, pages 240-249. Springer Berlin Heidelberg, 2012.

[102] MEHRABIAN A. and LUCAS C.. A novel numerical optimization algorithm inspired from weed colonization. *Ecological Informatics*, 1(4):355 - 366, 2006.

[103] REYNOLDS R. G.. An introduction to cultural algorithms. In *Proceedings of the third annual conference on evolutionary programming*, pages 131–139, 1994. World Scientific.

[104] ATASHPAZ-GARGARI E. and LUCAS C.. Imperialist competitive algorithm: An algorithm for optimization inspired by imperialistic competition. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 4661-4667, 2007.

[105] XU Y., CUI Z. and ZENG J.. Social emotional optimization algorithm for nonlinear constrained optimization problems. In BijayaKetan Panigrahi, Swagatam Das, PonnuthuraiNagaratnam Suganthan and SubhransuSekhar Dash, editors, *Swarm, Evolutionary, and Memetic Computing*, number 6466 in Lecture Notes in Computer Science, pages 583-590. Springer Berlin Heidelberg, 2010.

[106] KASHAN A.. League championship algorithm: A new algorithm for numerical function optimization. In *Soft Computing and Pattern Recognition, 2009. SOCPAR '09. International Conference of*, pages 43-48, 2009.

[107] SHI L. and ÓLAFSSON S.. Nested partitions method for global optimization. *Oper. Res.*, 48(3):390–407, 2000.

[108] GLOVER F., LAGUNA M. and MARTI R.. Fundamentals of scatter search and path relinking. *Control and cybernetics*, 39:653–684, 2000.

[109] MLADENOVIC N. and HANSEN P.. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097 - 1100, 1997.

[110] LOURENCO H. R., MARTIN O. C. and STÃijTZLE T.. Iterated local search: Framework and applications. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, number 146 in International Series in Operations Research & Management Science, pages 363-397. Springer US, 2010.

[111] MOSCATO P.. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report, California Institute of Technology, 1989.

[112] BATTITI R. and BRUNATO M.. *Reactive Search Optimization: Learning while Optimizing*. Springer Verlag, 2009.

[113] ŠKAROUPKA D. and ŠANDERA Č.. Automated design of vehicle silhouette using genetic algorithms and statistical analysis. *MECCA - Journal of Middle European Costruction and Design of Cars*, 2011:1-5.

[114] PERSSON A., GRIMM H. and NG A. H. C.. Metamodel-assisted global search using a probing technique.. In Sio Iong Ao, Oscar Castillo, Craig Douglas, David Dagan Feng and Jeong-A. Lee, editors, *IMECS* Lecture Notes in Engineering and Computer Science, pages 83-88, 2007. Newswood Limited.

[115] AHUJA R. K., ERGUN O., ORLIN J. B. and PUNNEN A. P.. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3):75-102, 2002.

[116] HANSEN N.. The cma evolution strategy: A comparing review. In Jose A. Lozano, Pedro Larranaga, Inaki Inza and Endika Bengoetxea, editors, *Towards a New Evolutionary Computation*, number 192 in Studies in Fuzziness and Soft Computing, pages 75-102. Springer Berlin Heidelberg, 2006.

[117] KRINK T., FILIPIC B., FOGEL G. and THOMSEN R.. Noisy optimization problems - a particular challenge for differential evolution?. In *In Proceedings of 2004 Congress on Evolutionary Computation*, pages 332–339, 2004. IEEE Press.

[118] DAS S. and KONAR A.. An improved differential evolution scheme for noisy optimization problems. In SankarK. Pal, Sanghamitra Bandyopadhyay and Sambhunath Biswas, editors, *Pattern Recognition and Machine Intelligence*, number 3776 in Lecture Notes in Computer Science, pages 417-421. Springer Berlin Heidelberg, 2005.

[119] VAESSENS R., AARTS E. and LENSTRA J.. A local search template. *Computers and Operations Research*, 25(11):969 - 979, 1998.

[120] CALéGARI P., CORAY G., HERTZ A., KOBLER D. and KUONEN P.. A taxonomy of evolutionary algorithms in combinatorial optimization. *Journal of Heuristics*, 5(2):145–158, 1999.

[121] GREISTORFER P. and VOSS S.. Controlled pool maintenance for metaheuristics. In Ramesh Sharda, Stefan Voss, CÃľsar Rego and Bahram Alidaee, editors, *Metaheuristic Optimization via Memory and Evolution*, number 30 in Operations Research/Computer Science Interfaces Series, pages 387-424. Springer US, 2005.

[122] TAILLARD E. D., GAMBARDELLA L. M., GENDREAU M. and POTVIN J.-Y.. Adaptive memory programming: A unified view of metaheuristics. *European Journal of Operational Research*, 135(1):1 - 16, 2001.

[123] MEIGNAN D., CREPUT J.-C. and KOUKAM A.. An organizational view of metaheuristics. 2008

[124] MEIGNAN D., KOUKAM A. and CRéPUT J.-C.. Coalition-based metaheuristic: a self-adaptive metaheuristic using reinforcement learning and mimetism. *Journal of Heuristics*, 16(6):859–879, 2010.

[125] ŠANDERA Č. and ŠKAROUPKA D.. Heuristic statistical generation of graphical curves. *18th International Conference on Soft Computing, MENDEL 2012*, 2012.

[126] JOHNSON R. A. and WICHERN D. W.. *Applied Multivariate Statistical Analysis (6th Edition)*. Pearson, 6 edition, 2007.

[127] ŠANDERA Č., KISELA T. and ŠEDA M.. Heuristic identification of fractional-order systems. *MENDEL 2010 - 16th International Conference on Soft Computing*, pages 550-557.

[128] ŠANDERA Č., POPELA P. and ROUPEC J.. The worst case analysis by heuristic algorithms. *MENDEL 2009 - 15th International Conference on Soft Computing*, pages 109-114.

[129] MAUDER T., ŠANDERA Č., ŠTěTINA J. and ŠEDA M.. Optimization of quality of continuously cast steel slabs by using firefly algorithm. In *Program and books of abstracts of the 18th Conference on Materials and Technology*, 2010.

[130] MAUDER T., ŠANDERA Č., ŠTěTINA J. and ŠEDA M.. Optimization algorithm for continuous casting process. In *17th International Conference of Soft Computing*, 2011.

A Apendix

A.1 Příklad spouštění implementovaného genetického algoritmu

```
01: alg = Algorithm()
02:
03: main = alg.operator('main')
04: alg.block_start(main)
05:
06: create = mtho_creation.CreateClassObjects(alg)
07: termination = mtho_termination.Terminate_number_of_iterations(alg)
08: evaluation = standardBenchmarks.ObjectiveFunctionWithSaveMinimum(alg)
09: integrate = commonOperators.ReplaceTheWorst(alg)
10: crossover = commonOperators.ExchangeGenes(alg)
11: mutation = commonOperators.MutateNumVariables(alg)
12: selection = commonOperators.SelectPairsTresholds(alg)
13: select_mutants = commonOperators.SelectFromPopulation(alg)
14:
15: genetic_algorithm = GeneticAlgorithm(alg)
16: genetic_algorithm['termination'] = termination
17: genetic_algorithm['evaluation'] = evaluation
18: genetic_algorithm['crossover'] = crossover
19: genetic_algorithm['mutation'] = mutation
20: genetic_algorithm['integration'] = integrate
21: genetic_algorithm['selection'] = selection
22: genetic_algorithm['select_mutants'] = select_mutants
23:
24: population = alg.apply(create)
25: alg.map(evaluation, population)
26: best = alg.apply(genetic_algorithm, population)
27:
28: alg.block_stop(result = best)
29:
30:
31: class_of_objects = supportFunction.create_class_vector(alg, ('x','y'), ((-100,100),(-
100,100)), 'value')
32: evaluation['evaluation'] =  standardBenchmarks.objective_function_cls
33: create['class_name'] = class_of_objects
```

```
34: create['number_of_objects'] = 100
35: mutation['mutated_variables'] = 1
36: selection['num_of_couples'] = 60
37: selection['treshold_parent1'] = 0.2
38: selection['treshold_parent2'] = 0.5
39: select_mutants['percent'] = 0.1
40: termination['max_iteration'] = 50
41:
42: result = alg(main)
43:
44: class_name = alg.db.get(result,'class')
45: n_var = alg.db.get(class_name, 'num_of_var')
46: n_out = alg.db.get(class_name, 'num_of_out')
47: print 'result', [alg.db.get(result,'var',i) for i in range(n_var)]
48: print 'objective function', [alg.db.get(result,'out',i) for i in range(n_out)]
```

Objekt `alg` třídy `Algorithm` je hlavní objekt, který konstruuje a řídí celou metaheuristiku. Metaheuristika je implementována jako operátor s názvem `main`, jehož definice je mezi řádky 04-28. Řádky 6-13 nahrávají operátory, které budou potřeba pro genetický algoritmus. Samotný genetický algoritmus je již implementován jako externí operátor a stačí ho tedy pouze nahrát (řádek 15). Nahraná metaheuristika je pouze schránkou na předané operátory, takže řádky 16-22 postupně specifikují, které operátory se budou používat.

Popis, jak má operátor `main` fungovat, začíná na řádku 24, kde se vytvoří patřičná populace a její klíč se uloží do proměnné `population`. Na řádku 25 se každý prvek ohodnotí (metoda `map()` spouští zadaný operátor pro každý prvek populace zvlášť) a řádek 26 spustí genetický algoritmus, kterému předá ohodnocenou populaci (metoda `apply()` spustí předaný operátor pro celou populaci dohromady). Výsledkem genetického algoritmu je nejlepší jedinec, který se uloží pod klíč `best`. Výstupem operátoru `main` je právě klíč uložený v proměnné `best` (definováno na řádku 28). Řádky 31-40 nastavují parametry operátorů předaných genetickému algoritmu.

Samotné spuštění operátoru `main`, který obaluje genetický algoritmus, je provedeno na řádku 42 a výsledek (klíč) je uložen do proměnné `result`. Poslední řádky (44-48) zjistí, kolik má výsledek vstupních a výstupních proměnných a všechny je vypíše.

Hybridizace

Základní způsob hybridizace je vidět na řádcích 16-22, kde lze nadefinovat libovolné operátory, které ve vytvořeném genetickém algoritmu budou hrát roli křížení, mutace, ... Není potřeba, aby operátor *výběru rodičů* pracoval běžným způsobem a vracel z předané populace již existující prvky, ale může být využit libovolný jiný operátor, který například modifikuje celou populaci

(např. hejnovým operátorem) a teprve potom vybere vhodné dvojice pro křížení. Obdobně i operátor pro křížení může být zaměněn za jakákoliv operátor schopný zpracovat dva vstupní jedince a předat nějakého nového. Hybridizace je tímto způsobem velice zjednodušena, ale při zcela náhodném zvolení operátorů může docházet k nesmyslným operacím (například použití operátoru selekce pro křížení by nepřinášelo žádné nové informace do algoritmu).

A.2 Implementance genetického algoritmu

Operátory mohou být definovány buď přímo třídou `Algorithm`, nebo prostřednictvím třídy `MthOperator`, která práci s operátory zjednodušuje a zapouzdřuje. Každý nový operátor je třídou odvozenou z `MthOperator`, kde se samotné tělo algoritmu zapíše do metody `__init__()` spouštěné při vytváření nové instance třídy. Například genetický operátor má následující zápis:

```
01: self.block_start('genetic_algorithm')
02: local_best_key = self.apply('create_local_best')
03:
04: population = self.input()
05:
06: self.block_start(repeat=True)
07:
08: parents = self.apply('selection', population) # [p1a,p1b, p2a,p2b, ...]
09: offsprings = self.apply('create_empty_offsprings')
10:
11: self.block_start(repeat=True)
12: pair_of_parents = self.apply('select_two', parents)
13: one_offspring = self.apply('crossover', pair_of_parents)
14: self.apply('evaluation', one_offspring)
15: self.apply('append', offsprings, one_offspring, output = offsprings)
16: self.apply('remove_two', parents, output = parents) # if return empty => exit
17: self.block_stop()
18:
19: self.apply('integration', population, offsprings, output = population)
20: mutants = self.apply('select_mutants', population)
21: self.map('mutation', mutants, output = mutants)
22: self.map('evaluation', mutants)
23:
24: local_best = self.apply('save_best', population, local_best_key)
25: self.apply('termination')
26: self.block_stop()
27: self.block_stop(result = local_best)
```

Metody `self.map()` a `self.apply()` odpovídají stejně pojmenovanému volání metod `alg.apply()` a `alg.map()`. Jediný rozdíl je ten, že jako první parametr je předáváno textové jméno abstraktního operátoru. Reálné přiřazení operátorů jejich abstraktním klíčům se provádí až v době kompilace modelu. Některé operátory jsou ale napevno zafixované a to takové, které jsou buď pomocné a nemá smysl je za něco zaměňovat, a nebo takové operátory, které tvoří jádro celé metaheuristiky (tedy operátory specifické pro konkrétní typ metaheuristiky).

A.3 Obsah přiloženého CD

Popsaný obecný hybridní metaheuristický model byl implementovaný v jazyce Python a k vyzkoušení je k nalezení na přiloženém CD.

Základní adresářová struktura programu:

```
./components
./mth_operators
./testFunctions
angles_and_mortals.py
clonalg.py
frogs.py
genetic_algorithm.py
harmony_search.py
particle_swarm_optimization.py
simulated_annealing.py
spiral.py
tabu_search.py
transform_log.py
```

Soubory v tomto základním adresáři definují konkrétní podobu spustitelné metaheuristiky řešící konkrétní účelovou funkci. Všechny příklady jsou vytvořeny pro Rastriginovu účelovou funkci $f(\mathbf{x}) = \sum_i^n (10 + x_i^2 - 10\cos 2\pi x_i)$, kde n je počet dimenzí stavového prostoru, a která má minimum $f(\mathbf{x}) = 0$ v bodě $\mathbf{x} = \mathbf{0}$. Záměna za libovolnou jinou účelovou funkci se provádí v hlavním spouštěcím souboru metaheuristiky.

Jediným souborem s jiným významem je `transform_log.py`, který vygenerované *log-soubory* transformuje do formy vhodné pro následující analýzy (viz kapitola 8.1). Spouštění programu je prováděno standardním způsobem pomocí příkazového řádku, tedy např. v Linuxu:

```
$ python genetic_algorithm.py
```

nebo ve Windows:

```
> C:\python27\python.exe genetic_algorithm.py
```

(Pro správný běh je potřeba mít aktuální pracovní adresář nastavený na základním adresáři modelu. Popsaný způsob spouštění může být lehce odlišný v závislosti na zvoleném způsobu instalace Pythonu.)

Adresář `components` obsahuje základní stavební kameny celé implementace, tedy:

- třídu `Algorithm` vykonávající řídicí logiku modelu metaheuristiky,
- třídu `Database` sloužící pro práci s klíči,
- třídu `mthOperator` reprezentující operátory (externí funkce, bloky operátorů i celé metaheuristiky).

Zmíněné třídy jsou dostatečné k vytvoření libovolné metaheuristiky.

Adresář `mth_operators` obsahuje implementované ukázky různých metaheuristik. Tyto soubory, na rozdíl od souborů v základním adresáři, definují logiku a posloupnost vykonávání jednotlivých dílčích operátorů. Soubory v základním adresáři tyto vytvořené operátory volají a dodefinovávají jim potřebné chybějící údaje (parametry, operátory, ...). Zbylé soubory jsou využity pro běžné typy operátorů, které jsou společné pro více metaheuristik (generování základní populace, ukončovací kritéria, řadící operátory, ...).

K běhu programu je potřeba instalace Python verze 2.x (přednostně 2.7) a k příkladu spirální optimalizace (`spiral.py`) je navíc potřeba i knihovna NumPy. Verze pro Windows je dostupná ke stažení buď na stránkách `http://www.python.org` a nebo v adresáři ./Python na CD. (Ve většině linuxových distribucí není potřeba nic doinstalovávat, protože Python bývá standardní součástí vydání.)