evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
Slezská univerzita v Opavě
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Stochastické algoritmy pro globální optimalizaci

Josef Tvrdík

ČÍSLO OPERAČNÍHO PROGRAMU: CZ.1.07
NÁZEV OPERAČNÍHO PROGRAMU:
OP VZDĚLÁVÁNÍ PRO KONKURENCESCHOPNOST
PRIORITNÍ OSA: 2
ČÍSLO OBLASTI PODPORY: 2.3

**POSÍLENÍ KONKURENCESCHOPNOSTI
VÝZKUMU A VÝVOJE INFORMAČNÍCH TECHNOLOGIÍ
V MORAVSKOSLEZSKÉM KRAJI**

REGISTRAČNÍ ČÍSLO PROJEKTU: CZ.1.07/2.3.00/09.0197

OSTRAVA 2010

Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky

Název: Stochastické algoritmy pro globální optimalizaci
Autor: Josef Tvrdík

Vydání: první, 2010
Počet stran: 80

Studijní materiály pro distanční kurz: Stochastické algoritmy pro globální optimalizaci

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

© Josef Tvrdík
© Ostravská univerzita v Ostravě

Obsah

1 Úvod **2**

2 Problém globální optimalizace **3**

2.1 Formulace problému globální optimalizace 3

2.2 Stochastické algoritmy pro globální optimalizaci 6

3 Experimentální porovnávání algoritmů **7**

3.1 Experimentální ověřování algoritmů 7

3.2 Testovací funkce 10

4 Slepé náhodné prohledávání **20**

5 Řízené náhodné prohledávání **23**

6 Evoluční algoritmy **30**

6.1 Genetické algoritmy 32

6.2 Evoluční strategie 36

6.3 Diferenciální evoluce 44

6.4 Experimentální porovnání jednoduchých evolučních algoritmů 52

7 Algoritmy modelující chování skupin **55**

7.1 Algoritmus PSO 55

7.2 Algoritmus SOMA 64

8 Adaptace pomocí soutěže strategií **70**

9 Literatura **78**

1 Úvod

Tento text je určen studentům volitelného předmětu Stochastické algoritmy pro globální optimalizaci v prezenčním, kombinovaném i v distančním studiu na Ostravské univerzitě. Cílem textu je seznámit studenty se základy problematiky globální optimalizace a algoritmy pro heuristické hledání globálního extrému funkcí. Text je zaměřen především na stochastické algoritmy inspirované živou přírodou, tj. na evoluční algoritmy a na algoritmy modelující chování společenství živočišných jedinců v kolektivu.



Každá kapitola začíná pokyny pro její studium. Tato část je vždy označena jako **Průvodce studiem** s ikonou na okraji stránky.



Pojmy a důležité souvislosti k zapamatování jsou vyznačeny na okraji stránky textu ikonou.



V závěru každé kapitoly je rekapitulace nejdůležitějších pojmů. Tato rekapitulace je označena textem **Shrnutí** a ikonou na okraji.



Oddíl **Kontrolní otázky** označený ikonou by vám měl pomoci zjistit, zda jste prostudovanou kapitolu pochopili a budou i inspirovat k dalším otázkám, na které budete hledat odpověď.



U některých kapitol je připomenuto, že k této části textu je zadána **Korespondenční úloha**. Úspěšné vyřešení korespondenčních úloh je součástí podmínek pro získání zápočtu pro studenty kombinovaného a distančního studia. Korespondenční úlohy budou zadávány v rámci kurzu daného semestru.

V textu budeme užívat následující druhy písma pro různé symboly:

- kurzíva malá písmena, např. $a, b, x_1, y_j, \delta, \varepsilon$ pro *reálná čísla*,
- kurzíva velká písmena, např. X, S, D pro *množiny* reálných čísel nebo vektorů. Pro celou množinu reálných čísel užíváme speciální symbol $\mathbb{R}$,
- kurzíva tučná malá písmena, např. $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\alpha}$ pro *vektory*. Vektor délky n je uspořádaná n-tice reálných čísel, např. $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$,
- kurzíva tučná velká písmena, např. $\boldsymbol{X}, \boldsymbol{S}$ pro *matice*.

Zdrojové texty algoritmů v Matlabu jsou tištěny strojopisným typem, který vypadá např. takto: `y = x + (b-a) .* rand(1,n)`.

Pokud nejste obeznámeni se základy práce s Matlabem, doporučujeme nejdříve absolvovat kurz Základy modelování v MATLABU nebo alespoň prostudovat učební texty k tomuto kurzu [24].

2 Problém globální optimalizace

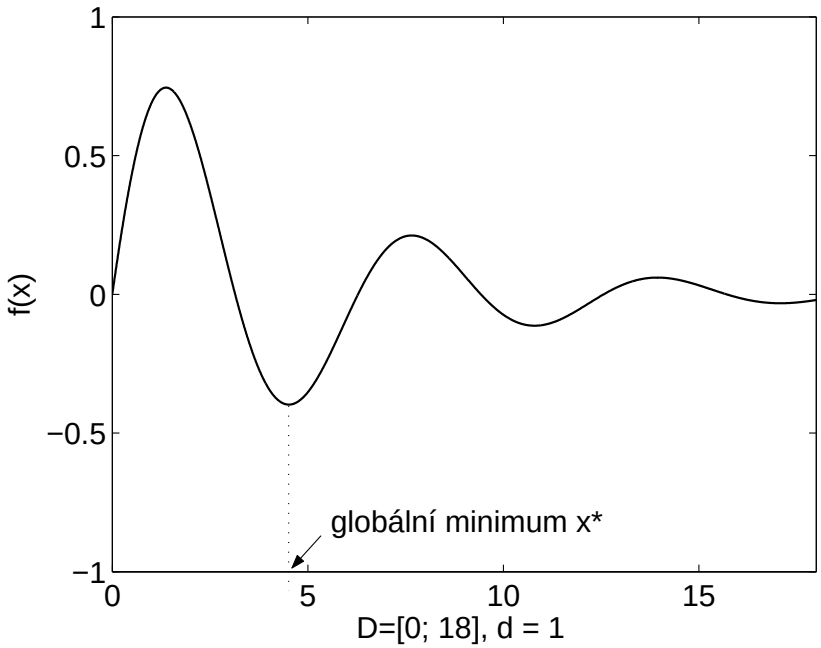
Průvodce studiem:



V této kapitole je zformulován problém globální optimalizace a stručně jsou zmíněny základní myšlenky stochastických algoritmů, které se používají pro nalezení globálního minima účelových funkcí. Počítejte asi se dvěma hodinami studia.

2.1 Formulace problému globální optimalizace

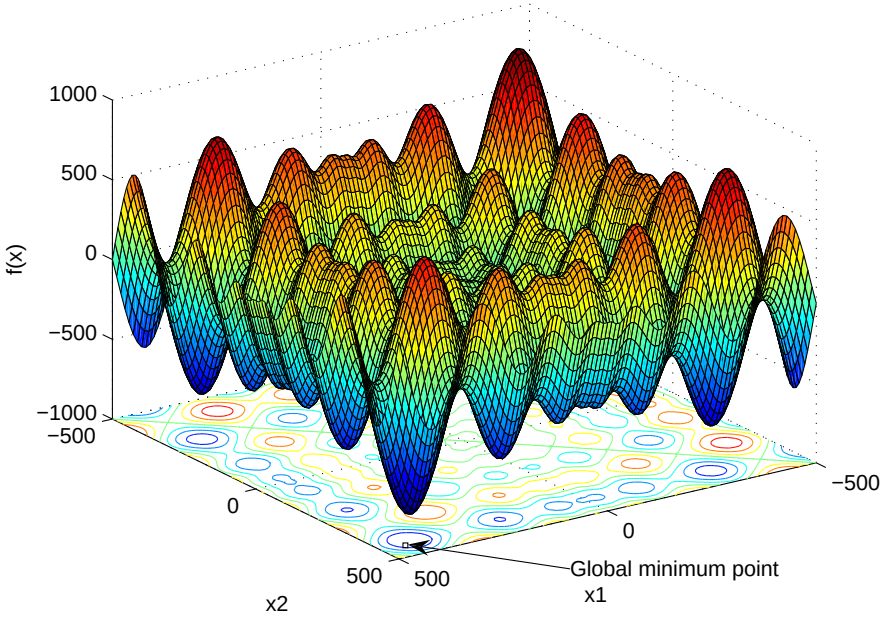
Budeme se zabývat řešením problému globální optimalizace, tj. nalezením souřadnic takového bodu v definičním oboru funkce, ve kterém má funkce extrémní (nejmenší nebo největší) hodnotu. Problém nalezení globálního minima funkce s jedním argumentem (argument je jedno reálné číslo) můžeme ilustrovat jednoduchým obrázkem:



Vidíme, že funkce na obrázku má v oboru $[0,\ 18]$ více minim, ale jen jedno je globální, tj. takové, že funkční hodnota v tomto bodě je nejmenší ze všech hodnot v oboru $[0,\ 18]$.

Ze základního kurzu matematické analýzy známe postup, jak nalézt extrémy funkcí, u kterých existuje první a druhá derivace. Zdálo by se, že úloha nalezení globálního minima je velmi jednoduchá. Bohužel tomu tak není. Nalézt obecné řešení takto jednoduše pochopitelného problému je obtížné, zvláště když účelová funkce má více lokálních minim nebo argument funkce není jedno reálné číslo, ale vektor reálných čísel, viz další obrázek pro funkci se dvěma argumenty. Navíc, ne každá funkce

je diferencovatelná, ale přesto potřebujeme její globální extrém nalézt nebo se mu alespoň přiblížit s přijatelnou přesností.





Úlohu nalezení globálního minima můžeme formulovat takto:

Mějme účelovou funkci

$$f : D \to \mathbb{R}, \quad D \subseteq \mathbb{R}^d. \tag{2.1}$$

Máme najít bod $\boldsymbol{x}^* \in D$, pro který platí, že $f(\boldsymbol{x}^*) \leq f(\boldsymbol{x})$, pro $\forall \boldsymbol{x}, \boldsymbol{x} \in D$. Nalezení bodu $\boldsymbol{x}^* \in D$ je řešením problému globální optimalizace. Bodu $\boldsymbol{x}^*$ říkáme bod globálního minima (global minimum point), definičnímu oboru D se říká doména nebo prohledávaný prostor (domain, search space), přirozené číslo d je dimenze úlohy.

Formulace problému globální optimalizace jako nalezení globálního minima není na úkor obecnosti, neboť chceme-li nalézt globální maximum, pak jej nalezneme jako globální minimum funkce $g(\boldsymbol{x}) = -f(\boldsymbol{x})$.



Analýza problému globální optimalizace ukazuje, že neexistuje deterministický algoritmus řešící obecnou úlohu globální optimalizace (tj. nalezení dostatečně přesné aproximace $\boldsymbol{x}^*$) v polynomiálním čase, tzn. problém globální optimalizace je NP-obtížný.

Přitom globální optimalizace je úloha, kterou je nutno řešit v mnoha praktických problémech, mnohdy s velmi významným ekonomickým efektem, takže je nutné hledat algoritmy, které jsou pro řešení konkrétních problémů použitelné. Algoritmy pro řešení problému globální optimalizace se podrobně zabývá celá řada monografií, např. Torn a Žilinskas [22], Míka [15], Spall [19], kde je možné najít mnoho užitečných poznatků přesahujících rámec tohoto předmětu.

Úloha (2.1) se označuje jako hledání volného extrému funkce (*unconstraint optimization*). Je možné přijatelnost řešení ještě omezit (vázat) nějakou podmínkou, např. nějakými rovnicemi nebo nerovnostmi. Pak jde o problém hledání vázaného extrému (*constrained optimization*).

V tomto kurzu se soustředíme především na problémy, kdy hledáme globální minimum v souvislé oblasti



$$D = \langle a_1, b_1\rangle \times \langle a_2, b_2\rangle \times \ldots \times \langle a_d, b_d\rangle = \prod_{i=1}^{d} \langle a_i, b_i\rangle, \quad a_i < b_i,\ i = 1, 2, \ldots, d, \tag{2.2}$$

a účelovou funkci $f(\boldsymbol{x})$ umíme vyhodnotit s požadovanou přesností v každém bodě $\boldsymbol{x} \in D$. Podmínce (2.2) se říká *boundary constraints* nebo *box constraints*, protože oblast D je vymezena jako d-rozměrný kvádr. Pro úlohy řešené numericky na počítači nepředstavuje podmínka (2.2) žádné podstatné omezení, neboť hodnoty a_i, b_i jsou tak jako tak omezeny datovými typy užitými pro $\boldsymbol{x}$ a $f(\boldsymbol{x})$, tj. většinou reprezentací čísel v pohyblivé řádové čárce. Proto se dosti často takové úlohy označují jako *unconstrained continuous problems*.

Úlohy hledání vázaného extrému *(constrained optimization)* jsou obvykle formulovány takto:

$$\text{Najdi minimum funkce } f(\boldsymbol{x}),\ \boldsymbol{x} = (x_1, x_2, \ldots, x_d)\ a\ \boldsymbol{x} \in D \tag{2.3}$$

$$\begin{aligned} \text{za podmínky:}\quad & g_i(\boldsymbol{x}) \le 0, \quad i = 1, \ldots, p \\ & h_j(\boldsymbol{x}) = 0, \quad j = p+1, \ldots, m. \end{aligned}$$

Řešení je považováno za přijatelné (*feasible*), když $g_i(\boldsymbol{x}) \le 0$ pro $i = 1, \ldots, p$ a $|h_j(\boldsymbol{x})| - \varepsilon \le 0$ pro $j = p+1, \ldots, m$. Pro libovolný bod $\boldsymbol{x} \in D$ a zadané kladné číslo ε můžeme definovat průměrné porušení podmínek (*mean violation*) $\bar{v}$ jako

$$\bar{v} = \frac{\sum_{i=1}^{p} G_i(\boldsymbol{x}) + \sum_{j=p+1}^{m} H_j(\boldsymbol{x})}{m},$$

kde

$$G_i(\boldsymbol{x}) = \begin{cases} g_i(\boldsymbol{x}) & \text{if} \quad g_i(\boldsymbol{x}) > 0 \\ 0 & \text{if} \quad g_i(\boldsymbol{x}) \le 0 \end{cases}$$

$$H_j(\boldsymbol{x}) = \begin{cases} |h_j(\boldsymbol{x})| & \text{if} \quad |h_j(\boldsymbol{x})| - \varepsilon > 0 \\ 0 & \text{if} \quad |h_j(\boldsymbol{x})| - \varepsilon \le 0. \end{cases}$$

Možná jste se s nejjednodušší variantou takových úloh setkali. Omezující podmínkou byla lineární nerovnost a úlohu jste vyřešili tzv. lineárním programováním.

Dalším typem optimalizačních úloh jsou *diskrétní problémy*, kdy prohledáváná oblast D není spojitá, ale diskrétní, např. hodnoty jednotlivých prvků vektoru $\boldsymbol{x}$ mohou být pouze celočíselné. Do této skupiny úloh patří např. problémy hledání optimální cesty v grafu.

2.2 Stochastické algoritmy pro globální optimalizaci

Nemožnost nalézt deterministický algoritmus obecně řešící úlohu globální optimalizace v polynomiálním čase vedla k využití algoritmů stochastických, které sice nemohou garantovat nalezení řešení v konečném počtu kroků, ale často pomohou nalézt v přijatelném čase řešení prakticky použitelné.



Stochastické algoritmy pro globální optimalizaci heuristicky prohledávají prostor D. Heuristikou rozumíme postup, ve kterém se využívá náhoda, intuice, analogie a zkušenost. Rozdíl mezi heuristikou a deterministickým algoritmem je v tom, že na rozdíl od deterministického algoritmu *heuristika nezajišťuje nalezení řešení*. Heuristiky jsou v praktickém životě zcela samozřejmě užívané postupy, jako příklady můžeme uvést hledání hub, lov ryb na udici, výběr partnera, pokus o výhru ve sportovním utkání nebo o složení zkoušky ve škole.



Většina stochastických algoritmů pro hledání globálního minima v sobě obsahuje zjevně či skrytě proces učení. Inspirace k užití heuristik jsou často odvozeny ze znalostí přírodních nebo sociálních procesů. Např. simulované žíhání je modelem pomalého ochlazování tuhého tělesa, tabu-search modeluje hledání předmětu člověkem tak, že v krátkodobé paměti si zapamatovává zakázané kroky vedoucí k již dříve projitým místům. Popis těchto algoritmů najdete např. v knize Kvasničky, Pospíchala a Tiňa [12]. Podobné postupy učení najdeme snad ve všech známých stochastických algoritmech pro hledání globálního minima (s výjimkou slepého náhodného prohledávání).

Značná část stochastických algoritmů pracuje současně s více kandidáty řešení, tj. s více body v prohledávaném prostoru. Tyto body vytvářejí skupinu (populaci), která se v průběhu hledání nějak v prohledávaném prostoru pohybuje a přitom nachází lepší kandidáty řešení. Stochastické algoritmy pro globální optimalizaci jsou příkladem využití soft computingu pro řešení úloh, které hard computingem řešit neumíme.

3 Experimentální porovnávání algoritmů

Průvodce studiem:



V této kapitole je uvedeno několik funkcí, které se užívají k testování stochastických algoritmů pro globální optimalizaci. Z nich si vybereme testovací funkce pro experimenty. Druhá část kapitoly se zabývá testováním stochastických algoritmů. K orientaci v této kapitole budete potřebovat asi dvě až tři hodiny.

3.1 Experimentální ověřování algoritmů

Pro ověřování a vzájemné porovnávání stochastických algoritmů jsou nutné numerické experimenty. Při porovnávání dvou či více algoritmů je samozřejmě nutné, aby testy byly provedeny za stejných podmínek, tj. zejména při stejně vymezeném prohledávaném prostoru D a při stejné podmínce pro ukončení prohledávání.

Základními veličinami pro porovnání algoritmů jsou časová náročnost prohledávání a spolehlivost nalezení globálního minima. Časová náročnost algoritmu se ohodnocuje počtem vyhodnocení účelové funkce v průběhu hledání, takže výsledky jsou srovnatelné bez ohledu na rychlost počítačů užitých k testování. Počet vyhodnocení účelové funkce potřebný k dosažení podmínky ukončení můžeme označit např. `nfe` (*number of function evaluations*). U testovacích funkcí, kdy řešení problému je známé, se někdy také sleduje počet vyhodnocení účelové funkce `nfe_near` potřebný k tomu, aby nejlepší bod populace (skupiny) měl hodnotu nižší než je zadaná hodnota f_{near}, o které se ví, že její dosažení znamená, že se algoritmus nejlepším nalezeným bodem dostatečně přiblížil globálnímu minimu. Někdy se tato hodnota nazývá také *VTR*, což je zkratka anglického termínu value to reach.

Podmínka ukončení bývá formulována jako

$$f_{\max} - f_{\min} < \varepsilon,$$

kde $f_{\max}$ je největší a $f_{\min}$ nejmenší funkční hodnota mezi všemi body skupiny (populace). Prohledávání končí, když rozdíl funkčních hodnot je tak malý, že už nelze očekávat nalezení vhodnějších kandidátů řešení.



Je však praktické formulovat podmínku ukončení tak, abychom předešli abnormálnímu ukončení programu např. v situaci, kdy prohledávání dojde do nekonečného cyklu. Proto je vhodné přidat další vstupní parametr algoritmu – `maxevals`, což

je maximální dovolený počet vyhodnocení funkce na jednu dimenzi prohledávaného prostoru a podmínku ukončení pak formulovat jako

$$f_{\max} - f_{\min} < \varepsilon \ \vee \ \texttt{nfe} \geq \texttt{maxevals} * d,$$

to znamená, že hledání pokračuje tak dlouho, dokud funkční hodnoty v populaci se liší více než požadujeme nebo dokud není dosažen nejvyšší dovolený počet vyhodnocení účelové funkce, tzn. je splněna podmínka

$$f_{\max} - f_{\min} \geq \varepsilon \ \wedge \ \texttt{nfe} < \texttt{maxevals} * d.$$

Místo $f_{\max}$ lze v podmínce ukončení užít i jiný kvantil funkční hodnoty v populaci, např. medián. Tím učiníme podmínku ukončení snadněji dosažitelnou, stačí, aby jen např. polovina populace měla funkční hodnoty dostatečně blízké.

Z uvedených úvah je zřejmé, že pak musíme rozlišovat následující čtyři typy ukončení prohledávání v tetovacích úlohách:

- Typ 1 – korektní ukončení, tj. $f_{\max} - f_{\min} < \varepsilon$ a $f_{\min} \leq f_{\text{near}}$, algoritmus tedy splnil podmínku ukončení před dosažením maximálního dovoleného počtu iterací a současně se dostatečně přiblížil globálnímu minimu.
- Typ 2 – pomalá konvergence, $f_{\min} \leq f_{\text{near}}$, ale prohledávání bylo ukončeno dosažením maximálního dovoleného počtu iterací.
- Typ 3 – předčasné konvergence (*premature convergence*), $f_{\max} - f_{\min} < \varepsilon$, ale $f_{\min} > f_{\text{near}}$, tzn. nebylo nalezeno globální minimum, algoritmus buď skončil prohledávání v lokálním minimu nebo se body populace přiblížily k sobě natolik, že jejich funkční hodnoty jsou velmi blízké.
- Typ 4 – úplné selhání, prohledávání bylo ukončeno dosažením maximálního dovoleného počtu iterací, aniž byl nalezen bod blízký globálnímu minimu.

Po prohledávacích algoritmech globální optimalizace chceme, aby uměly nacházet řešení dostatečně blízké globálnímu minimu $\boldsymbol{x}^*$ co nejrychleji, co nejspolehlivěji a aby prohledávání uměly ukončit ve vhodném stádiu prohledávání. Zcela úspěšné prohledávání je tedy jen takové, jehož ukončení je typu 1, někdy však můžeme považovat za úspěšné i ukončení typu 2.



Spolehlivost (*reliability*, R) nalezení globálního minima můžeme charakterizovat jako relativní četnost ukončení typu 1 (případně typu 1 nebo typu 2)

$$R = \frac{n_1}{n}, \tag{3.1}$$

kde n_1 je počet ukončení typu 1 (případně typu 1 nebo 2) v n nezávislých opakováních. Veličina n_1 je náhodná veličina $n_1 \sim Bi(n, p)$ a R je nestranným odhadem

parametru p, tj. pravděpodobnosti, že algoritmus v daném problému najde globální minimum (přesněji, že konec prohledávání je typu 1). Rozptyl $\mathrm{var}(R) = p(1-p)/n$. Pro větší hodnoty n můžeme rozdělení R považovat za přibližně normální, $R \sim N\left(p, \frac{p(1-p)}{n}\right)$. Pak 100 $(1-\alpha)$-procentní oboustranný interval spolehlivosti pro p je

$$[R - \triangle,\ R + \triangle],$$

kde

$$\triangle = t_{n-1}\left(1 - \frac{\alpha}{2}\right)\ \sqrt{\frac{R(1-R)}{n}}$$

V následující tabulce jsou uvedeny hodnoty $\triangle$ pro $\alpha = 0.05$ a pro různá n a R. Pokud byste vyjadřovali spolehlivost R v procentech, pak je potřeba hodnoty $\triangle$ v tabulce vynásobit stovkou. Z uvedené tabulky je zřejmé, že pro porovnávání spolehlivosti algoritmů potřebujeme velký počet opakování n. Např. pro $n = 100$ je možno za významný považovat až rozdíl v reliabilitě větší než 10 až 20 procent (závisí na hodnotě R).



Hodnoty $\triangle$ pro $\alpha = 0.05$

	R					
n	0.50	0.60	0.70	0.80	0.90	0.95
10	0.358	0.350	0.328	0.286	0.215	0.156
25	0.206	0.202	0.189	0.165	0.124	0.090
50	0.142	0.139	0.130	0.114	0.085	0.062
100	0.099	0.097	0.091	0.079	0.060	0.043
200	0.070	0.068	0.064	0.056	0.042	0.030
500	0.044	0.043	0.040	0.035	0.026	0.019

Výsledky testování by vždy měly obsahovat specifikaci testovaných algoritmů včetně hodnot jejich vstupních parametrů, specifikaci testovacích funkcí, počet opakování experimentů, tabulku s průměrnými hodnotami charakterizujícími časovou náročnost, případně i vhodné charakteristiky její variability a spolehlivost R, která může být uvedena v procentech.

Pro názorné porovnání časové náročnosti jsou vhodné krabicové grafy, ve kterých snadno vizuálně porovnáme jak charakteristiky polohy, tak variability. Pokud rozdíly v charakteristikách algoritmů nejsou zjevné z popisné statistiky, použijte vhodné statistické testy. Pro porovnání časové náročnosti to může být dvouvýběrový t -test, analýza rozptylu nebo jejich neparametrické analogie. Pokud chcete porovnávat spolehlivosti dvou algoritmů, můžete pro čtyřpolní tabulku absolutních četností užít Fisherův exaktní test.

U některých algoritmů je zajímavé sledovat i jiné veličiny, které charakterizují průběh prohledávacího procesu. Volba dalších sledovaných veličin je závislá na testovaném algoritmu.

Zajímavou popisnou informací o průběhu vyhledávání je také graf závislosti funkčních hodnot na počtu vyhodnocení účelové funkce. Průběh této závislosti nám umožňuje posoudit rychlost konvergence algoritmu v různých fázích prohledávání. V takových grafech je ovšem žádoucí mít charakteristiky z více opakování a pokud možno nejen průměrné hodnoty, ale i minima a maxima.

K rychlému porovnávání účinnosti algoritmů byly navrženy různé další veličiny integrující časovou náročnost i spolehlivost do jediného kritéria. Jednou z nich je tzv. Q-míra, definovaná jako

$$Q_m = \texttt{nfe}/R,$$

kde `nfe` je průměrný počet vyhodnocení funkce v n opakovaných běžích algoritmu při řešení úlohy a R je spolehlivost definovaná vztahem (3.1). Pak nejúčinnější algoritmus mezi porovnávanými je ten s minimální hodnotou Q_m. Nevýhodou tohoto kritéria je, že jeho hodnota není definována pro $R = 0$. Proto je vhodnější

$$Q_{inv} = R/\texttt{nfe},$$

pro kterou platí, že čím vyšší hodnota Q_{inv}, tím lepší účinnost algoritmu. Kritéria integrující `nfe` a R jsou však vhodná jen pro rychlé přehledné porovnávání celkové účinnosti algoritmů, protože z nich nelze dělat úsudky o variabilitě časové náročnosti algoritmů, ani jednoznačně posoudit jejich spolehlivost.

3.2 Testovací funkce

K testování stochastických algoritmů pro globální optimalizaci se užívá řada funkcí, u kterých je známé správné řešení, tj. globální minimum, které lze najít analyticky. Testovací funkce (*benchmark functions*) jsou často navrženy tak, aby je bylo možné použít pro problémy s různou dimenzí prohledávané domény. Testovací funkce mohou mít různou obtížnost, nejlehčí jsou konvexní funkce, ty mají jen jedno lokální minimum, které je tedy i minimem globálním. Takovým funkcím se říká unimodální. Funkce s více lokálními minimy se nazývají multimodální a nalezení globálního minima je obvykle obtížnější než u unimodálních funkcí. Za lehčí problémy jsou považována hledání globálního minima separabilních funkcí, tj. funkcí, jejichž argumenty jsou nezávislé.

Obecně platí, že funkce splňující podmínku

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} = g(x_i)\ h(\boldsymbol{x})$$

jsou separabilní. Pro takové funkce platí

$$\begin{aligned} \forall i, j \quad i \neq j : \qquad & f(\ldots, x_i^*, \ldots) = opt \ \wedge \ f(\ldots, x_j^*, \ldots) = opt \\ & \Rightarrow f(\ldots, x_i^*, \ldots, x_j^*, \ldots) = opt \end{aligned}$$

Často lze takové funkce vyhodnotit jako

$$f((x) = \sum_{i=1}^{d} f_i(x_i) \tag{3.2}$$

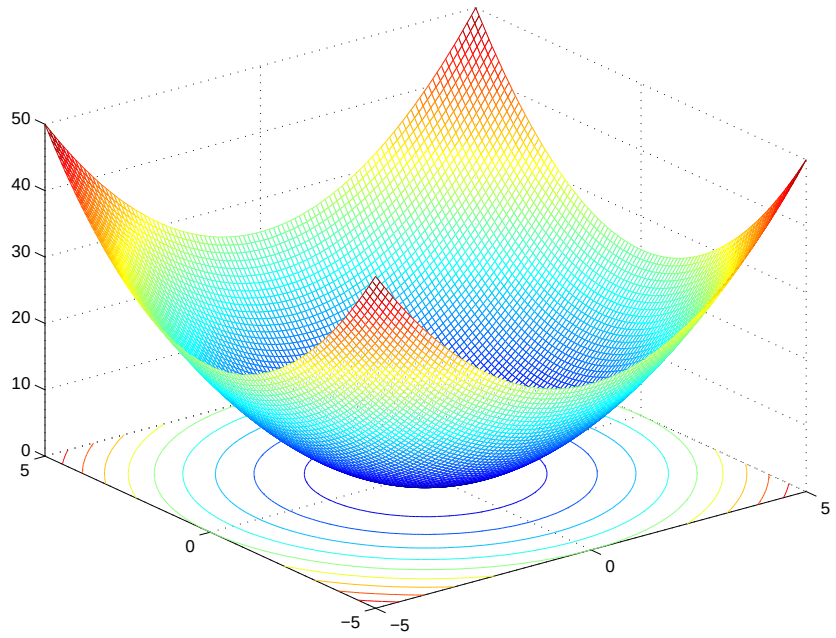
Hledání minima takových separabilních funkcí je snazší, protože poloha x_i^* je nezávislá na poloze x_j^* pro $i \neq j$ a tedy mohou být hledány nezávisle na sobě.

Zde uvádíme několik funkcí (separabilních i neseparabilních), které můžeme užít při experimentální testování algoritmů. Další testovací funkce naleznete např. v [2, 20, 29] nebo na webu. Všechny dále uvedené funkce lze užít pro různé dimenze prohledávané domény pro $d \geq 2$, některé i pro $d = 1$.

První De Jongova funkce je paraboloid

$$f(\boldsymbol{x}) = \sum_{i=1}^{d} x_i^2,$$

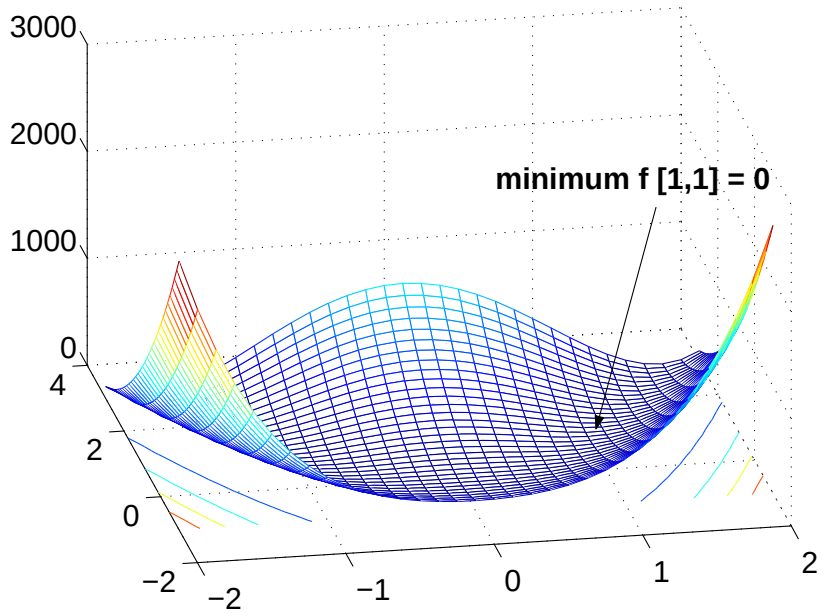
při experimentech prohledávaný prostor je obvykle vymezen podmínkou $x_i \in [-5.12,\ 5.12]$. Je to konvexní separabilní funkce, globální minimum je $\boldsymbol{x}^* = (0, \ldots, 0)$, $f(\boldsymbol{x}^*) = 0$. Je považována za velmi snadnou úlohu globální optimalizace. Pokud algoritmus selhává nebo pomalu konverguje v této úloze, tak nezaslouží další pozornosti. Graf této funkce pro $d = 2$ je na následujícím obrázku.



Druhá De Jongova funkce (Rosenbrockovo sedlo, známá i pod názvem banánové údolí), neseparabilní funkce, je definována takto:

$$f(\boldsymbol{x}) = \sum_{i=1}^{d-1} \left[100(x_i^2 - x_{i+1})^2 + (1 - x_i)^2\right],$$

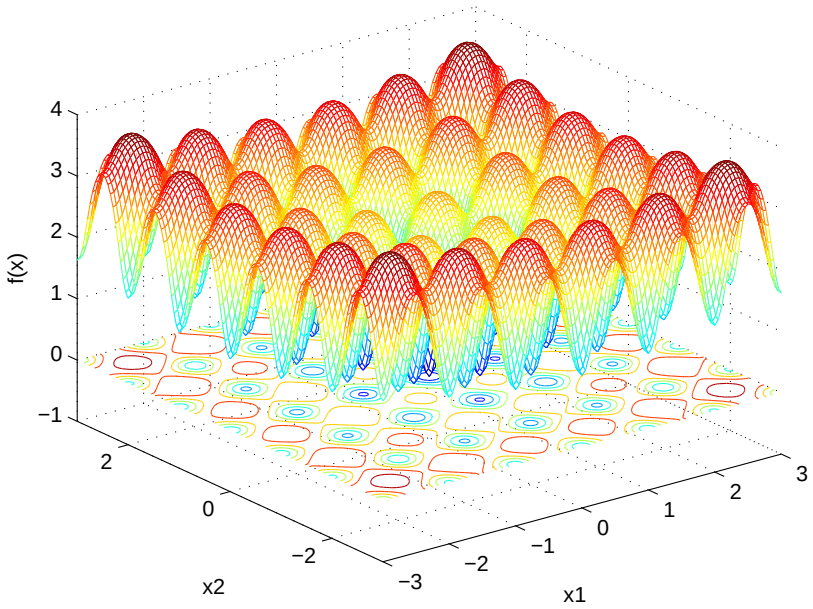
prohledávaný prostor je obvykle vymezen podmínkou $x_i \in [-2.048,\ 2.048]$, globální minimum je $\boldsymbol{x}^* = (1, 1, \ldots 1)$, $f(\boldsymbol{x}^*) = 0$. Ač je to jednomodální funkce (v poslední době se objevily domněnky, že pro některé dimenze má tato funkce nevýrazná lokální minima), nalezení minima iteračními algoritmy není jednoduchá úloha, neboť minimum leží v zahnutém údolí s velmi malým spádem a některé algoritmy končí prohledávání předčasně. Graf této funkce pro $d = 2$ je na následujícím obrázku.



Ackleyho funkce je

$$f(\boldsymbol{x}) = -20 \exp\left(-0.02 \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d} \cos 2\pi x_i\right) + 20 + \exp(1),$$

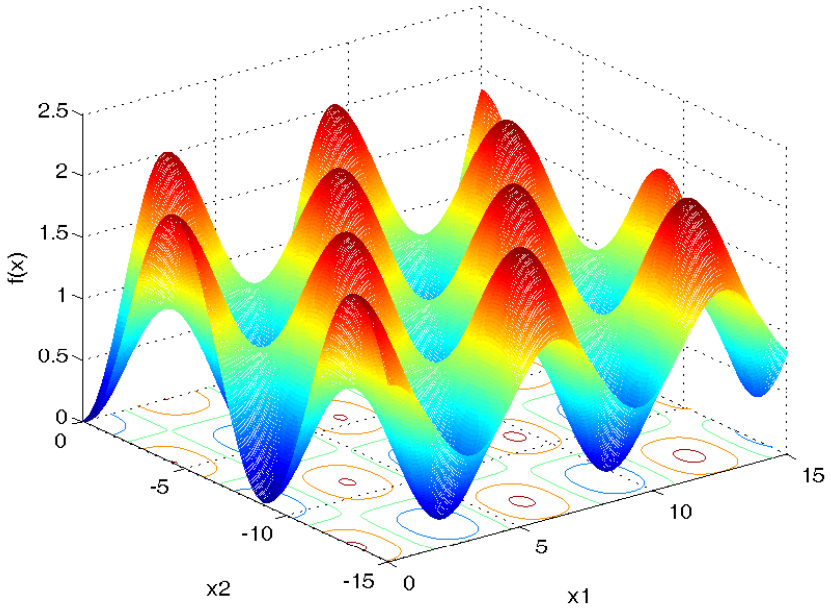
$x_i \in [-30,\ 30]$, $\boldsymbol{x}^* = (0, 0, \ldots 0)$, $f(\boldsymbol{x}^*) = 0$. Je to multimodální separabilní funkce s mnoha lokálními minimy, středně obtížná. Graf této funkce pro $d = 2$, $x_i \in [-3,\ 3]$, je na následujícím obrázku.



Griewankova funkce, multimodální, neseparabilní,

$$f(\boldsymbol{x}) = \sum_{i=1}^{d} \frac{{x_i}^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1,$$

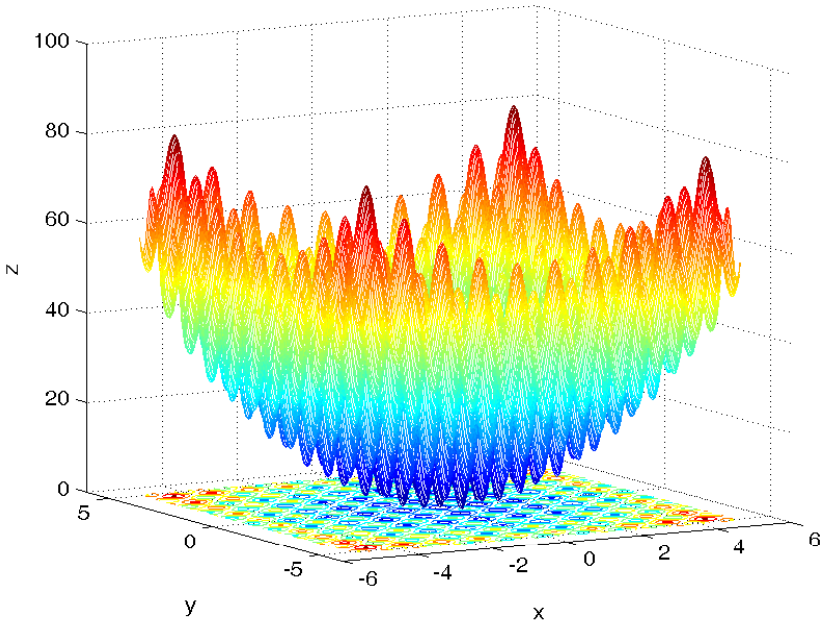
$x_i \in [-400,\ 400]$, $\boldsymbol{x}^* = (0, 0, \ldots 0)$, $f(\boldsymbol{x}^*) = 0$. Nalezení minima této multimodální funkce je považováno za obtížnou úlohu globální optimalizace. Graf této funkce pro $d = 2$, $x_i \in [0,\ 15]$, je na následujícím obrázku.



Rastriginova funkce, multimodální, separabilní,

$$f(\boldsymbol{x}) = 10d + \sum_{i=1}^{d} \left[{x_i}^2 - 10\cos(2\pi x_i)\right]$$

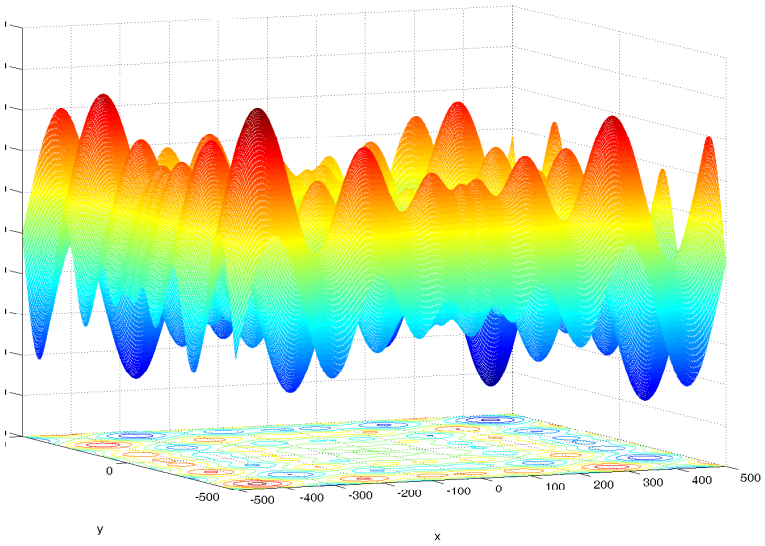
$x_i \in [-5.12,\ 5.12]$, $\boldsymbol{x}^* = (0, 0, \ldots 0)$, $f(\boldsymbol{x}^*) = 0$. Nalezení minima této funkce je považováno za obtížnou úlohu globální optimalizace. Graf této funkce pro $d = 2$ je na následujícím obrázku.



Schwefelova funkce, multimodální, separabilní, je zde uvedena ve tvaru s aditivním členem za rovnítkem, který posouvá hodnoty funkce tak, aby její hodnota v globálním minimu byla rovna nule stejně jako u ostatních zde uvedených testovacích funkcí.

$$f(\boldsymbol{x}) = 418.9828872724338\ d - \sum_{i=1}^{d} x_i \sin\left(\sqrt{|\ x_i\ |}\right),$$

$x_i \in [-500,\ 500]$, $\boldsymbol{x}^* = (s, s, \ldots s)$, $s \doteq 420.968746$, $f(\boldsymbol{x}^*) \doteq 0$, pro $d < 80$ platí, že $0 <= f(\boldsymbol{x^*}) < 1e - 10$. Nalezení minima této funkce je považováno za středně obtížnou úlohu globální optimalizace, její globální minimum je v prohledávaném prostoru velmi vzdálené od druhého nejlepšího lokálního minima. Graf této funkce pro $d = 2$ (bez aditivního členu posouvajícího funkční hodnoty) je na následujícím obrázku.



Někteří autoři oprávněně považují funkce, u kterých je globální minimum ve středu prohledávané domény (z uvedených funkcí je to první De Jongova, Ackleyho, Griewankova a Rastriginova funkce), tj. v těžišti d-rozměrného kvádru D, za nevhodné pro testování stochastických algoritmů, neboť u nich lze najít globální minimum pouhým průměrováním náhodně vygenerovaných bodů v D. Tento nedostatek testovacích funkcí s bodem globálního minima v těžišti domény D lze snadno odstranit. Místo vyhodnocení funkce v bodě $\boldsymbol{x}$ se vyhodnocuje funkce v bodě $\boldsymbol{x} - \boldsymbol{o}$. Bod $\boldsymbol{o} \in D$ znamená posun (*shift*). Globální minimum takové posunuté funkce je pak $\boldsymbol{x}^* = \boldsymbol{o}$. Posun $\boldsymbol{o}$ můžeme zadat předem nebo generovat náhodně před každým spuštěním optimalizace testovací funkce.

Implementace těchto testovacích funkcí je v Matlabu velmi jednoduchá, jak vidíte v následujících zdrojových textech.

```
function y=dejong1(x)
%  First Dejong
y=sum(x.*x);
```

```
function y=rosen(x)
%  Rosenbrockova funkce
n=length(x);
a=(x(1:n-1).^2-x(2:n)).^2;
b=1-x(1:n-1);
y=sum(100*a+b.^2);
```

```
function y=ackley(x)
%  Ackley's  function, <-30,30>, glob. min f(0)=0
d=length(x);
a=-20*exp(-0.02*sqrt(sum(x.*x)));
b=-exp(sum(cos(2*pi*ones(1,d).*x))/d);
y=a+b+20+exp(1);
```

```
function y=griewank(x)
%  Griewank's  function, <-400,400>, glob. min f(0)=0
d=length(x);
a=sum(x.*x)/4000;
j=1:d;
b=prod(cos(x./sqrt(j)));
y=a-b+1;
%
```

```
function y=rastrig(x)
%  Rastrigin's  function, <-5.12,5.12>, glob. min f(0)=0
d=length(x);
sz=size(x);
if  sz(1)==1  x=x'; end
y=10*d+sum(x.*x-10*cos(2*pi*x));
%
```

```
function y=schwefel0(x)
%   Schwefel's  function, <-500,500>,
%   glob. min f(x_star)~0, (for d<80 0<=f(x_star)<1e-10),
%   x_star=[s,s,...,s], s=420.968746
d=length(x);
sz=size(x);
if  sz(1)==1
    x=x';
end
y = 418.9828872724338*d - sum(x.*sin(sqrt(abs(x))));
%
```

Shrnutí:



- Testovací funkce, numerické testování algoritmů
- Různé typy ukončení prohledávání
- Časová náročnost měřená počtem vyhodnocení účelové funkce, spolehlivost nalezení globálního minima
- Vyhodnocení experimentu

Kontrolní otázky:



1. Proč je nutné posuzovat algoritmy nejen z jednoho běhu, ale z více opakování?
2. Jak popsat specifikaci testovacího problému?
3. Jak specifikovat testovaný algoritmus?

4 Slepé náhodné prohledávání



Průvodce studiem:

V této kapitole se seznámíte s nejjednodušším stochastickým algoritmem pro hledání globálního minima. Kromě toho uvidíte i snadnost zápisu takových algoritmů v Matlabu a seznámíte se s pojmem konvergence stochastického algoritmu. Počítejte asi se dvěma hodinami studia.

Náhodné prohledávání se také někdy nazývá slepý algoritmus, neboť se v něm opakovaně generuje náhodné řešení (nový bod $\boldsymbol{y}$ v souvislé oblasti $D = \prod_{i=1}^{d} \langle a_i, b_i \rangle$ z rovnoměrného spojitého rozdělení) a zapamatovává se tehdy, když je lepší než předtím nalezené řešení. Náhodné prohledávání je příklad velmi jednoduchého stochastického algoritmu pro hledání globálního minima využívajícího velmi prostou heuristiku, kterou bychom mohli slovně vyjádřit jako "zkus bez rozmyslu cokoliv, co není zakázáno". Slepý algoritmus lze zapsat v Matlabu na pár řádcích:

```
function [x,fx] = blindsearch(fn_name, a, b, t)
% input parameters:
%   fn_name     function to be minized (M file)
%   a, b        row vectors, limits of search space
%   t            iteration number
% output:
%   fx         minimal function value found by the search
%   x          minimum point found by the search
d = length(a);
fx = realmax;
for i=1:t
    y = a + (b - a).*rand(1,d);
    fy = feval(fn_name, y);
    if fy < fx
        x = y;
        fx = fy;
    end
end
```

U tohoto algoritmu lze také poměrně jednoduše dokázat jeho teoretickou konvergenci. Platí, že s rostoucím počtem iterací se nalezené řešení $\boldsymbol{x}$ přibližuje globálnímu minimu $\boldsymbol{x}^*$:



$$\lim_{t\to\infty} P(\parallel (\boldsymbol{x}-\boldsymbol{x}^*) \parallel < \varepsilon \mid t) = 1 \quad \varepsilon > 0, \tag{4.1}$$

kde $\parallel (\boldsymbol{x}-\boldsymbol{x}^*) \parallel$ je vzdálenost nalezeného řešení $\boldsymbol{x}$ od globálního minima $\boldsymbol{x}^*$ (norma vektoru $\boldsymbol{x}-\boldsymbol{x}^*$) a $P(\parallel (\boldsymbol{x}-\boldsymbol{x}^*) \parallel < \varepsilon \mid t)$ je pravděpodobnost jevu $(\parallel (\boldsymbol{x}-\boldsymbol{x}^*) \parallel < \varepsilon)$ za podmínky, že bylo provedeno t iterací.

K tvrzení vyjádřenému rovnicí (4.1) dojdeme následující úvahou: Všechny body splňující podmínku $\parallel (\boldsymbol{x}-\boldsymbol{x}^*) \parallel < \varepsilon$ jsou vnitřními body d-rozměrné koule D_ε o poloměru ε). Jelikož $\varepsilon > 0$, je "objem" této koule kladný, přesněji míra této množiny $\lambda(D_\varepsilon) > 0$. Označme dále míru množiny D jako $\lambda(D)$. Generujeme-li bod $\boldsymbol{y} \in D$ z rovnoměrného rozdělení, pak pravděpodobnost jevu $A \equiv \{\boldsymbol{y} \in D_\varepsilon\}$ je

$$P(A) = \frac{\lambda(D_\varepsilon)}{\lambda(D)}$$

a pravděpodobnost jevu opačného je

$$P(\bar{A}) = 1 - \frac{\lambda(D_\varepsilon)}{\lambda(D)}$$

Pravděpodobnost, že jev $\bar{A}$ nastane v t po sobě jdoucích opakováních je

$$P(\bar{A} \mid t) = \left(1 - \frac{\lambda(D_\varepsilon)}{\lambda(D)}\right)^t.$$

Jelikož $0 < \left(1 - \frac{\lambda(D_\varepsilon)}{\lambda(D)}\right) < 1$, je potom

$$\lim_{t\to\infty} P(\bar{A} \mid t) = \left(1 - \frac{\lambda(D_\varepsilon)}{\lambda(D)}\right)^t = 0,$$

pravděpodobnost jevu opačného je rovna 1, a tedy rovnice (4.1) platí.

Lze ukázat, že nejenom slepé prohledávání, ale dokonce každý stochastický algoritmus, ve kterém se jiný způsob generování nového bodu $\boldsymbol{y}$ střídá s jeho slepým generováním z rovnoměrného rozdělení v prohledávaném prostoru D a toto slepé generování má v každém kroku kladnou pravděpodobnost (přesněji, když tyto pravděpodobnosti pro jednotlivé kroky algoritmu tvoří divergentní řadu, čehož lze snadno dosáhnout volbou konstantní pravděpodobnosti v každém kroku), tak takový kombinovaný algoritmus je konvergentní ve smyslu rov. (4.1).

Bohužel rovnice (4.1) nám neposkytuje žádnou informaci o kvalitě nalezeného řešení po konečném počtu iterací. Jak uvidíme později, s podobným problémem se budeme setkávat i u dalších stochastických algoritmů. Teoretická konvergence podle



rovnice (4.1) je z praktického hlediska velmi slabé tvrzení a pro posouzení konvergence stochastických algoritmů a porovnání jejich efektivity jsou důležitější experimentální výsledky na testovacích funkcích než teoretický důkaz konvergence ve formě rovnice (4.1).



Shrnutí:

- Generování bodu z rovnoměrného rozdělení na D
- Konvergence stochastického algoritmu



Kontrolní otázky:

1. Jakou heuristiku užívá slepý algoritmus pro generování nového bodu?
2. Je ve slepém náhodném prohledávání modelován nějaký proces učení?

5 Řízené náhodné prohledávání

Průvodce studiem:



V této kapitole se poprvé seznámíte s algoritmem, který pracuje s populací jedinců, tj. bodů (kandidátů řešení) v D. Algoritmus řízeného náhodného prohledávání je jednoduchý a efektivní stochastický algoritmus pro hledání globálního minima. Pochopení tohoto algoritmu je dobrým odrazovým můstkem k evolučním algoritmům. Počítejte asi se dvěma hodinami studia a několika hodinami věnovanými implementaci algoritmu v Matlabu a jeho ověření na testovacích funkcích.

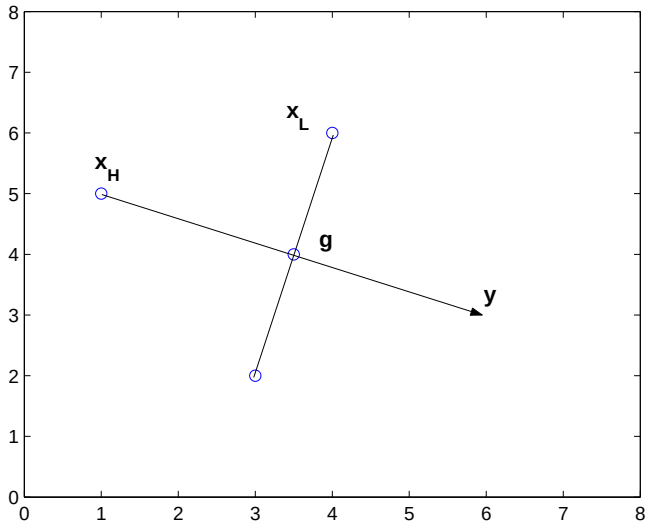
Řízené náhodné prohledávání (controlled random search, CRS) je příkladem velmi jednoduchého a přitom efektivního stochastického algoritmu (čili heuristiky) pro hledání minima v souvislé oblasti D. První verzi algoritmu CRS navrhl Price [18] v sedmdesátých letech minulého století. Pracuje se s populací N bodů – kandidátů řešení v prohledávaném prostoru D a z nich se pomocí nějaké *lokální heuristiky* generuje nový bod $\boldsymbol{y}$, který může být zařazen do populace místo dosud nejhoršího bodu. Počet vygenerovaných bodů populace N je větší než dimenze d prohledávaného prostoru D.

Price jako lokální heuristiku pro generování nového bodu $\boldsymbol{y}$ užíval reflexi simplexu, která je známa z velmi populární simplexové metody hledání minima, kterou navrhli před zhruba 50 léty Nelder a Mead [16]. Simplexová metoda je využita také ve vestavěné funkci `fminsearch` v Matlabu jako prostředek pro hledání minima zadané funkce.

Reflexe simplexu v CRS se realizuje takto: Z populace o velikosti N, $N > d$ se náhodně vybere $d + 1$ bodů tvořících simplex (optimisticky spoléháme na to, že tyto náhodně vybrané body jsou nekomplanární, tj. libovolných d vektorů z těchto $d+1$ vektorů tvoří bázi d-rozměrného prostoru). Pak bod simplexu s největší funkční hodnotou překlopíme kolem těžiště zbývajících bodů simplexu a získáme tak nového potenciálního kandidáta řešení $\boldsymbol{y}$. Reflexe v simplexu je vyjádřena vztahem

$$\boldsymbol{y} = \boldsymbol{g} + (\boldsymbol{g} - \boldsymbol{x}_H) = 2\boldsymbol{g} - \boldsymbol{x}_H, \tag{5.1}$$

kde $\boldsymbol{x}_H$ je bod simplexu s největší hodnotou účelové funkce a $\boldsymbol{g}$ je těžiště zbývajících d bodů simplexu, jehož souřadnice spočítáme jako průměry těchto d bodů simplexu. Graficky je reflexe znázorněna na následujícím obrázku.



Pokud v novém bodu $\boldsymbol{y}$ je funkční hodnota $f(\boldsymbol{y})$ menší než v nejhorším bodu celé populace, pak je tento nejhorší bod nahrazen bodem $\boldsymbol{y}$. Nahrazením nejhoršího bodu populace novým bodem $\boldsymbol{y}$ dosahujeme toho, že populace se koncentruje v okolí dosud nalezeného bodu s nejmenší funkční hodnotou. Algoritmus můžeme jednoduše zapsat ve strukturovaném pseudokódu následujícím způsobem:



generuj populaci P, tj. N bodů náhodně v D
repeat
  najdi $\boldsymbol{x}_{\text{worst}} \in P$ takové, že $f(\boldsymbol{x}_{\text{worst}}) \geq f(\boldsymbol{x}), \quad \boldsymbol{x} \in P$
  repeat
    vyber z P simplex
    $\boldsymbol{y} :=$ reflexe simplexu, $\boldsymbol{y} \in D$
  until $f(\boldsymbol{y}) < f(\boldsymbol{x}_{\text{worst}})$;
  $\boldsymbol{x}_{\text{worst}} := \boldsymbol{y}$;
until podmínka ukončení;

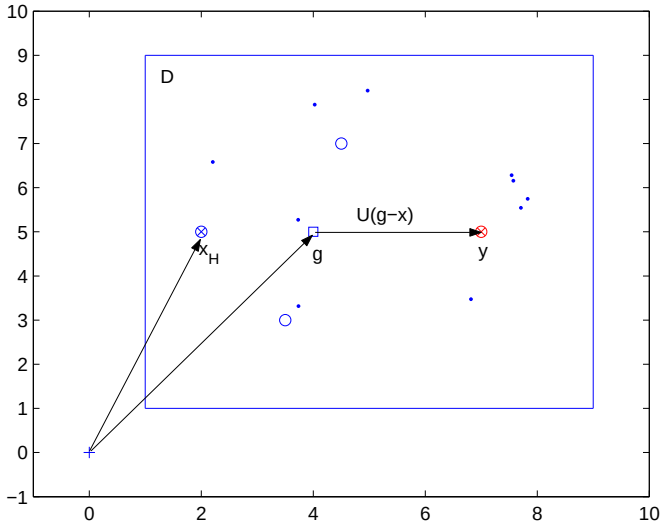
Reflexe simplexu podle rovnice (5.1) samozřejmě není jediná možnost, jak v algoritmu CRS generovat nový bod. Price později navrhl jinou lokální heuristiku, kdy nový bod $\boldsymbol{y}$ se generuje podle (5.1), ale do simplexu je vždy zařazen nejlepší bod populace $\boldsymbol{x}_{\min}$ s funkční hodnotou $f_{\min}$, $f_{\min} \leq f(\boldsymbol{x})$, $\boldsymbol{x} \in P$ a zbývajících d bodů simplexu se pak vybere náhodně z ostatních bodů populace.

Další možnost je znáhodněná reflexe, která byla navržena v 90. létech. Znáhodněná reflexe v simplexu je popsána vztahem



$$\boldsymbol{y} = \boldsymbol{g} + U\ (\boldsymbol{g} - \boldsymbol{x}_H), \tag{5.2}$$

U je náhodná veličina vhodného rozdělení. Graficky je taková reflexe znázorněna na následujícím obrázku.



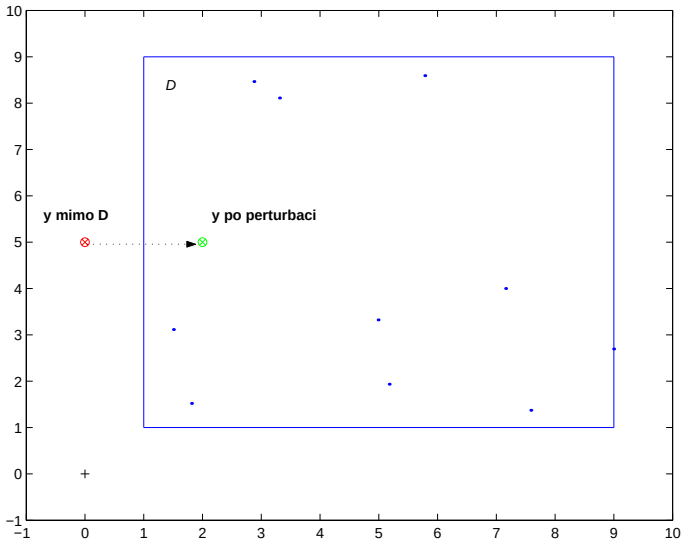
Pokud bychom užili reflexi podle (5.1), byla by vzdálenost bodů $\boldsymbol{g}$ a $\boldsymbol{y}$ byla stejná jako vzdálenost bodů $\boldsymbol{x}$ a $\boldsymbol{g}$, tj. hodnota U v rovnici (5.2) by byla konstantní během prohledávání a rovna jedné. Při užití znáhodněné reflexe je tato vzdálenost určována náhodně podle zvoleného rozdělení náhodné veličiny U.

Na testovacích úlohách bylo ověřováno několik rozdělení náhodné veličiny U. Osvědčilo se rovnoměrné rozdělení na $[0, \alpha)$, kde α je vstupní parametr algoritmu. Podle empirických výsledků testů nejrychleji algoritmus konvergoval na většině testovacích funkcí při hodnotách $\alpha \approx 4$. Střední hodnota je $EU = \alpha/2$.

Za pozornost stojí užití rovnoměrného rozdělení náhodné veličiny U na $[s, \alpha - s)$, kde $\alpha > 0$ a $s \in (0, \alpha/2)$ jsou vstupní parametry této lokální heuristiky. Střední hodnota je opět $EU = \alpha/2$. Je však možno užít i jiná rozdělení produkující nezáporné hodnoty náhodné veličiny U, např. lognormální rozdělení, případně i jiné lokální heuristiky pro generování nového bodu $\boldsymbol{y}$, než je reflexe.

Reflexe podle rovnic (5.1) nebo (5.2) nezaručuje, že nově vygenerovaný bod $\boldsymbol{y}$ bude v prohledávaném prostoru D. Pokud nastane situace, že $\boldsymbol{y} \notin D$, jsou různé možnosti, jak postupovat dále. Buď generujeme nový bod opakovaně tak dlouho, než se podaří vygenerovat $\boldsymbol{y} \in D = \prod_{i=1}^{d} \langle a_i, b_i \rangle$ nebo použijeme výpočetně úspornější postup. Např. je možné vygenerovat souřadnice $y_i \notin \langle a_i, b_i \rangle$ náhodně nebo užít tzv. *zrcadlení* (perturbaci), kdy ty souřadnice $y_i \notin \langle a_i, b_i \rangle$ překlopíme dovnitř prohledávaného prostoru D kolem příslušné strany d-rozměrného kvádru D. Toto zrcadlení znázorňuje následující obrázek, jeho algoritmus je zapsán jako funkce `zrcad` v Matlabu.






```
% zrcadleni, perturbation y into <a,b>
function y = zrcad(y, a, b)
zrc = find(y < a | y > b); % najdi pretekajici dimenze
for i = zrc  % preklop
    while (y(i) <  a(i) ||  y(i) > b(i))
        if y(i) > b(i)
            y(i) = 2*b(i) - y(i);
        elseif y(i) < a(i)
            y(i) = 2*a(i) - y(i);
        end
    end
end
```

Při stochastickém prohledávání podmínku ukončení formuluje uživatel. Obvykle je možné užít podmínku ukončení formulovanou tak, že nejlepší a nejhorší bod populace se ve funkčních hodnotách liší jen málo, tedy

$$f(\boldsymbol{x}_{\text{worst}}) - f(\boldsymbol{x}_{\text{min}}) < \varepsilon, \tag{5.3}$$

kde $\varepsilon > 0$ je vstupní parametr algoritmu. Jinou možností je hledání ukončit při dosažení zadaného počtu vyhodnocení funkce, případně tyto podmínky kombinovat – viz kap. 3.

Algoritmus CRS můžeme celkem rychle napsat v Matlabu takto:

```
function [x_star, fn_star, func_evals]=...
    crs1(fn_name, a, b, N, my_eps, max_evals, alfa, shift)
d = length(a);
P = zeros(N,d+1);
for i = 1:N
    P(i,1:d) = a + (b - a).*rand(1,d);
    P(i,d+1) = feval(fn_name,P(i,1:d) - shift);
end  % 0-th generation initialized
[fmax, indmax] = max(P(:,d+1));
[fmin, indmin] = min(P(:,d+1));
func_evals = N;
while (fmax - fmin > my_eps) && (func_evals < d*max_evals)% main loop
    y = reflrand(P, alfa);
    y = zrcad(y, a, b);     % perturbation
    fy = feval(fn_name,y - shift);
    func_evals = func_evals + 1;
    if fy < fmax  %  y is good
        P(indmax,:) =  [y,  fy];
        [fmax, indmax] = max(P(:,d+1));
        [fmin, indmin] = min(P(:,d+1));
    end
end   % main loop - end
x_star = P(indmin,1:d);
fn_star = fmin;
end
```

Funkce `crs1` kromě zrcadlení `zrcad` volá další dvě funkce, a to `reflrand`, která realizuje znáhodněnou reflexi podle vztahu (5.2) s U rovnoměrně rozděleném na $[0, \alpha)$ a funkci `nahvyb_expt`, která vybírá z N bodů populace náhodně $d+1$ bodů do simplexu. Prostudujte pozorně implementaci všech těchto funkcí, nebot' se v nich můžete seznámit i s různými technikami efektivního využití možností Matlabu, které značně snižují délku zdrojového kódu a tím i pracnost implementace. Funkce `nahvyb_expt` a `zrcad` využijeme i později v některých evolučních algoritmech.

```
function y = reflrand(P, alpha)
%  znahodnena reflexe simplexu
    N = length(P(:,1));
    d = length(P(1,:)) - 1;
    vyb = nahvyb_expt(N, d+1);
    S = P(vyb,:); % simplex S vybran nahodne z P
    [x, indx] = max(S(:, d+1)); % nejhorsi bod v S
    x = S(indx, 1:d);
    S(indx,:)= []; % zrus radek s nejhorsi hodnotou
    S(:,d+1) = []; % zrus posledni sloupec s funkcni hodnotou
    g = mean(S);   % teziste
    y = g + alpha * rand(1) * (g - x);
```

```
% random sample, k of N without repetition,
% numbers given in vector expt are not included
function vyb = nahvyb_expt(N, k, expt)
opora = 1:N;
if nargin == 3
    opora(expt) = [];
end
vyb = zeros(1,k);
for i = 1:k
    index = 1 + fix(rand(1)*length(opora));
    vyb(i) = opora(index);
    opora(index) = [];
end
```

Algoritmus CRS je pro nás prvním algoritmem, který pracuje s populací bodů (potenciálních kandidátů možného hledaného řešení) a tuto populaci nechává vyvíjet v průběhu prohledávacího procesu. Proto někteří autoři algoritmus CRS řadí mezi evoluční algoritmy, i když z principů evolučních algoritmů jsou uplatněny pouze některé, jak uvidíme v následujících kapitolách.

Shrnutí:



- Základní principy algoritmu CRS
- Populace bodů náhodně vygenerovaných v D
- Lokální heuristika generující nový pokusný bod
- Simplex náhodně vybraný z populace
- Reflexe a znáhodněná reflexe
- Podmínka ukončení

Kontrolní otázky:



1. Jak určíte souřadnice těžiště n bodů v D?
2. Co je to zrcadlení a kdy se užívá?
3. Jak lze zformulovat podmínku ukončení?
4. Jak byste experimentálně porovnali účinnost algoritmu CRS a slepého prohledávání na vybraných testovacích úlohách?

6 Evoluční algoritmy



Průvodce studiem:

V této kapitole se stručně seznámíme s historií a principy evolučních algoritmů. Na tuto krátkou kapitolu počítejte asi s hodinou studia.

V posledních desetiletích se s poměrným úspěchem pro hledání globálního minima funkcí užívají stochastické algoritmy zejména evolučního typu. Podrobný popis této problematiky naleznete v knihách Goldgerga [8], Michalewicze [13] nebo Bäcka [2]. Rozvoj evolučních algoritmů je záležitostí posledních desetiletí a je podmíněn rozvojem počítačů a pokroky v informatice. Přelomovými pracemi zavádějícími biologickou terminologii do modelů hledání globálního extrému jsou články Schwefela a Rechenberga ze šedesátých let minulého století o evoluční strategii, práce L. Fogela o evolučním programování a Hollandova kniha o genetických algoritmech [9]. Tím byl odstartován bouřlivý rozsáhlý rozvoj evolučních algoritmů a jejich aplikací.



Evoluční algoritmy jsou ve své podstatě jednoduchými modely Darwinovy evoluční teorie vývoje populací. Charakteristické pro ně je to, že pracují s populací (tvořenou jedinci – kandidáty možných řešení), která se v průběhu hledání vyvíjí (vznikají nové generace). Tento vývoj se uskutečňuje aplikací *evolučních operátorů*, nejdůležitější evoluční operátory jsou tyto:

- *selekce* – silnější jedinci z populace mají větší pravděpodobnost svého přežití i předání svých vlastností potomkům,
- *křížení* (rekombinace) – dva nebo více jedinců z populace si vymění informace a vzniknou tak noví jedinci kombinující vlastnosti rodičů,
- *mutace* – genetická informace zakódovaná v jedinci může být náhodně modifikována.

Dále se mezi evoluční operátory počítá *migrace*, která se užívá v tzv. paralelních evolučních algoritmech, kdy se modeluje vývoj více populací vedle sebe a jejich vzájemné ovlivňování, ale takovými algoritmy se v tomto kurzu nebudeme zabývat.

Evoluční algoritmy jsou heuristické postupy, které nějakým způsobem modifikují populaci tak, aby se její vlastnosti zlepšovaly. O některých třídách evolučních algoritmů je dokázáno, že nejlepší jedinci populace se skutečně přibližují ke globálnímu extrému.



Evoluční algoritmy byly a jsou předmětem intenzivního výzkumu a počet publikací z této oblasti je velký. Jedním z hlavních motivů jsou především aplikace v praktických problémech, které jinými metodami nejsou řešitelné. Dalšími motivy pro roz-

voj evolučních algoritmů jsou výzkum umělé inteligence a teorie učení. Tento rozvoj nezastavil ani tzv. "No Free Lunch Theorem"[27], který ukazuje, že nelze najít univerzálně nejlepší stochastický algoritmus pro globální optimalizaci. Porovnáváme-li dva algoritmy, vždy lze najít skupinu problémů, ve kterých bude první algoritmus účinnější a jinou skupinu problémů, ve kterých bude účinnější algoritmus druhý.

Shrnutí:



- Populace a její vývoj
- Evoluční operátory
- No Free Lunch Theorem

Kontrolní otázky:



1. Čím jsou evoluční algoritmy inspirovány?
2. Proč byly evoluční algoritmy navrženy a proč jsou dále zkoumány a rozvíjeny?

6.1 Genetické algoritmy



Průvodce studiem:

Tato stručná kapitola má posloužit k základní orientaci v principech genetických algoritmů a upozornit na postupy, které jsou inspirativní pro jiné evoluční algoritmy. Počítejte asi se dvěma hodinami studia.

Genetické algoritmy od svého vzniku v 70. letech [9] představují mohutný zdroj inspirace pro další evoluční algoritmy, pro zkoumání umělé inteligence a pro rozvoj soft computingu. O genetických algoritmech existuje řada specializovaných monografií, viz např. Goldberger [8], Michalewicz [13] nebo Bäck [2]. Velmi dobře a srozumitelně napsaný úvod do genetických algoritmů najdete v dostupné knize Kvasničky, Pospíchala a Tiňa [12]. Zde se omezíme jen na stručné vysvětlení základních principů těchto algoritmů.

Základní myšlenkou genetických algoritmů je analogie s evolučními procesy probíhajícími v biologických systémech. Podle Darwinovy teorie přirozeného výběru přežívají jen nejlépe přizpůsobení jedinci populace. Mírou přizpůsobení je tzv. "fitness" jedince. V biologii je fitness chápána jako relativní schopnost přežití a reprodukce genotypu jedince. Biologická evoluce je změna obsahu genetické informace populace v průběhu mnoha generací směrem k vyšším hodnotám fitness. Jedinci s vyšší fitness mají větší pravděpodobnost přežití a větší pravděpodobnost reprodukce svých genů do generace potomků. Kromě reprodukce se v populačním vývoji uplatňuje i tzv. mutace, což je náhodná změna genetické informace některých jedinců v populaci.

V genetických algoritmech je fitness kladné číslo přiřazené genetické informaci jedince. Tato genetická informace jedince (chromozóm) se obvykle vyjadřuje bitovým řetězcem. Populaci jedinců pak modelujeme jako populaci chromozómů a každému chromozómu (bitovému řetězci) umíme přiřadit hodnotu účelové funkce a podle vhodného předpisu umíme přiřadit i jeho fitness, podrobněji viz [12]. Obvykle je chromozómem binární vektor konstantní délky k



$$\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_k) \in \{0,1\}^k$$

a populace velikosti N je potom množina takových chromozómů

$$P = \{\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \ldots, \boldsymbol{\alpha}_N\}$$

Pokud hledáme globální minimum účelové funkce f, pak fitness F je nějaké zobrazení, které vyhovuje následující podmínce

$$f(\boldsymbol{\alpha}_i) \leq f(\boldsymbol{\alpha}_j) \Rightarrow F(\boldsymbol{\alpha}_i) \geq F(\boldsymbol{\alpha}_j) > 0,\ i,j = 1,2,\ldots,N$$

Nejjednodušší způsob, jak vyhovět této podmínce, je lineární zobrazení funkčních hodnot na fitness, kdy hodnotu fitness určíme podle vztahu

$$F(\boldsymbol{\alpha}) = \frac{F_{max} - F_{min}}{f_{min} - f_{max}} f(\boldsymbol{\alpha}) + \frac{f_{min}F_{min} - f_{max}F_{max}}{f_{min} - f_{max}},$$

kde f_{min} a f_{max} je nejmenší a největší hodnota funkce v populaci, F_{min} a F_{max} je minimální a maximální hodnota fitness, které obvykle volíme $F_{max} = 1$ a $F_{min} = \epsilon$, kde ϵ je malé kladné číslo, např. $\epsilon = 0.01$. Pak vztah pro fitness má tvar

$$F(\boldsymbol{\alpha}) = \frac{1}{f_{min} - f_{max}} \left[(1-\epsilon) f(\boldsymbol{\alpha}) + \epsilon f_{min} - f_{max}\right].$$

Hodnoty fitness je vhodné renormalizovat, tj. přepočítat tak, aby jejich součet byl roven jedné. Renormalizovaná fitness i-tého jedince je

$$F'(\boldsymbol{\alpha}_i) = \frac{F(\boldsymbol{\alpha}_i)}{\sum_{j=1}^{N} F(\boldsymbol{\alpha}_j)}$$

a tato hodnota pak udává i pravděpodobnost přežití jedince a jeho účasti na reprodukci své genetické informace, tj. vytvoření potomka. Výběr (selekci) jedince s pravděpodobností $F'(\boldsymbol{\alpha}_i)$ ukazuje následující algoritmus nazývaný také ruleta. Vstupním parametrem `fits` je vektor hodnot fitness všech jedinců populace, hodnoty fitness nemusí být renormalizovány. Funkce vrací číslo (index) vybraného jedince, pravděpodobnost výběru je úměrná hodnotě fitness.

```
function res=roulette_simple(fits)
%
% returns an integer from [1, length(fits)]
% with probability proportional
% to fits(i)/ sum fits
h = length(fits);
ss = sum(fits);
cp = zeros(1, h);
cp(1) = fits(1);
for i=2:h
    cp(i) = cp(i-1) + fits(i);
end
cp = cp/ss;
res = 1 + fix(sum(cp < rand(1)));
```

Reprodukce probíhá křížením dvou chromozómů tak, že si vymění své geny (části bitových řetězců). Ze dvou rodičů $\boldsymbol{\alpha}, \boldsymbol{\beta}$ v jednobodovém křížení vzniknou dva potomci podle následujícího schématu:



$$(\alpha_1, \ldots, \alpha_l, \alpha_{(l+1)}, \ldots, \alpha_k) \quad \searrow \quad (\alpha_1, \ldots, \alpha_l, \beta_{(l+1)}, \ldots, \beta_k)$$
$$(\beta_1, \ldots, \beta_l, \beta_{(l+1)}, \ldots, \beta_k) \quad \nearrow \quad (\beta_1, \ldots, \beta_l, \alpha_{(l+1)}, \ldots, \alpha_k)$$

Bod křížení l se volí náhodně. Někdy se také užívá dvoubodové křížení, ve kterém se určí náhodně dvě pozice pro křížení, l_1, l_2, $l_2 > l_1$ a v bitových řetězcích se vymění úseky

$$\alpha_{l_1}, \ldots, \alpha_{l_2} \leftrightarrow \beta_{l_1}, \ldots, \beta_{l_2}$$

Mutace je většinou v genetických algoritmech implementována jako změna hodnoty náhodně vybraného bitu v chromozómu, tj. hodnota 0 se změní na 1, hodnota 1 na 0. Mutací je zajištěno, aby mohla v populaci vzniknout genetická informace, která v ní v předcházejících generací nebyla, nebo se obnovila genetické informace ztracené v průběhu dosavadního vývoje. V důsledku toho pak algoritmus může uniknout z oblasti lokálního minima, ke kterému by směřoval, pokud bychom užívali pouze křížení. Naopak, pokud bychom užívali pouze mutaci, genetický algoritmus by se choval podobně jako slepé náhodné prohledávání. Lze ukázat, že genetický algoritmus konverguje ke globálnímu extrému.



Genetické algoritmy na rozdíl od ostatních algoritmů uváděných v tomto textu reprezentují jedince jako bitový řetězec, nikoliv jako vektor hodnot v pohyblivé řádové čárce. To je často výhodou při řešení tzv. diskrétních úloh globální optimalizace (kombinatorické úlohy jako je problém obchodního cestujícího), ale přináší to kom-

plikace v problémech, kdy je prohledávaná oblast D možných řešení spojitá. Pokud bychom úseky chromozómu interpretovali jako celá čísla (datový typ integer), pak se potýkáme s obtíží, že sousední číselné hodnoty jsou reprezentovány řetězci, které se mohou podstatně lišit, dokonce i ve všech bitech, např. dekadická hodnota 7 je binárně (0111), následující číselná hodnota 8 je (1000). Tuto nepříjemnost je nutno v genetických algoritmech nějak ošetřit, pokud mají být úspěšně používány i pro řešení spojitých problémů globální optimalizace. Je to možno řešit použitím Grayova kódu [12], kdy řetězce reprezentující sousední celočíselné hodnoty se liší jen v jednom bitu, ale přináší to další časový nárok na konverzi do Grayova kódu a zpět. Byly navrženy i varianty genetických algoritmů s reprezentací jedince jako vektoru číselných hodnot v pohyblivé čárce. Pak je ovšem nutné užívat jiné operace křížení a mutace než ty, které se používají pro bitové řetězce. Některé takové operace si ukážeme na dalších evolučních algoritmech v tomto textu.

Shrnutí:



- Evoluce v biologických systémech
- Genetická informace, chromozóm, fitness
- Operátory selekce, křížení a mutace v genetických algoritmech

Kontrolní otázky:



1. Proč při hledání globálního minima nelze užívat jako fitness hodnotu účelové funkce?
2. Čím se liší dvoubodové křížení od jednobodového?
3. Jaká je role mutace v genetických algoritmech?

6.2 Evoluční strategie



Průvodce studiem:

V této kapitole jsou vysvětleny základy evoluční strategie, která patří k nejstarším a stále velmi populárním a inspirujícím evolučním algoritmům. Počítejte asi se třemi až čtyřmi hodinami studia.

Původní nejjednodušší verzi algoritmu navrhli v šedesátých létech minulého století Schwefel a Rechenberg, když potřebovali nalézt optimální tvar obtékaného tělesa. Základní myšlenka je podobná slepému náhodnému prohledávání, ale rozdíl je v tom, že nový bod $\boldsymbol{y}$ se generuje jako mutace bodu $\boldsymbol{x}$ tak, že jednotlivé složky vektoru $\boldsymbol{x}$ se změní přičtením hodnot normálně rozdělených náhodných veličin

$$\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{u}, \quad \boldsymbol{u} \sim N(\boldsymbol{0}, \sigma^2 \boldsymbol{I}), \tag{6.1}$$

tj. $\boldsymbol{u} = (U_1, U_2, \ldots, U_d)$ je náhodný vektor, jehož každý prvek je náhodná veličina $U_i \sim N(0, \sigma^2),\ i = 1, 2, \ldots, d$ a tyto veličiny jsou navzájem nezávislé. Algoritmus lze zapsat v Matlabu takto:

```
function [x,fx] = es1p1(fn_name,a,b,t,sigma)
% input parameters:
%   fn_name    function to be minimized (M file)
%   a, b       row vectors, limits of search space
%   t          number of iterations
%   sigma      standard deviation (const)
% output:
%   fx         the minimal function value found
%   x          minimum found by search
d = length(a);
fx = realmax;
x = a + (b - a).*rand(1,d);
for i = 1:t
    y = x + sigma*randn(1,d);
    y = zrcad(y, a, b);
    fy = feval(fn_name, y);
    if fy < fx
        x = y;
        fx = fy;
    end
end
```

Porovnejte tento zdrojový text se zdrojovým textem algoritmu slepého prohledávání v kap. 4 a uvědomte si rozdíly.

Evoluční terminologií může být tento algoritmus popsán tak, že z rodičovské generace velikosti 1 vzniká generace potomků rovněž velikosti 1, potomek vzniká mutací z jednoho rodiče podle rov. (6.1) a operátorem selekce je výběr lepšího jedince z dvojice (tzv. turnajový výběr). Tento algoritmus je v literatuře označován zkratkou ES(1+1).



Příklad 6.1 Porovnáme účinnost této nejjednodušší varianty evoluční strategie se slepým náhodným prohledáváním. Jak ze srovnání zdrojového kódu obou algoritmů vidíme, jsou velmi podobné, evoluční strategie má o jeden vstupní parametr víc, a to o směrodatnou odchylku `sigma`, která je shodná pro všechny složky nového generovaného bodu $\boldsymbol{y}$ a konstantní pro celé prohledávání. V implementaci funkce `es1p1` je navíc užito zrcadlení, aby nově vygenerovaný bod nebyl vně prostoru D.

Pro spouštění experimentálního porovnání si napíšeme následující skript, který uspoří pracné opakované zadávání příkazů z klávesnice a umožní i zaznamenat dosažené výsledky z jednotlivých běhů do souboru, který pak bude vstupem pro statistické zpracování výsledků. Výběr úlohy k řešení zařídíme vždy odstraněním znaku % na začátku zvoleného řádku a "vyprocentováním" ostatních řádků se jménem funkce a dalšími vstupními údaji. Podobný skript si můžete snadno vytvořit i pro spouštění slepého prohledávání.

Testy provedeme na šesti testovacích funkcích uvedených v kap. 3.2, dimenze problému $d = 2$, funkce jsou užity bez posunu ($\boldsymbol{o} = (0, 0)$, viz kap. 3.2), pro všechny funkce je prohledávání ukončeno po 10 000 vyhodnocení funkce a pro každou úlohu se provede 10 opakování. Volba hodnoty `sigma` je zřejmá ze zdrojového kódu spouštěcího skriptu.

```
% start
tol=1e-4; % tolerance f(x) pro  f_near
cfn=0;
s=420.968746;  % pro schwefel0
% fn_name='ackley'; b=30*ones(1,2);   cxst=zeros(1,2); f_near=cfn+tol;
% fn_name='ackley'; b=30*ones(1,5);   cxst=zeros(1,5); f_near=cfn+tol;
% fn_name='ackley'; b=30*ones(1,10);  cxst=zeros(1,10); f_near=cfn+tol;
% fn_name='ackley'; b=30*ones(1,30);  cxst=zeros(1,30); f_near=cfn+tol;
 fn_name='rosen'; b=[2.048 2.048];    cxst=ones(1,2); f_near=cfn+tol;
% fn_name='rosen'; b=2.048*ones(1,5);  cxst=ones(1,5);f_near=cfn+tol;
% fn_name='rosen'; b=2.048*ones(1,10); cxst=ones(1,10);f_near=cfn+tol;
% fn_name='rosen'; b=2.048*ones(1,30); cxst=ones(1,30);f_near=cfn+tol;
% fn_name='dejong1'; b=5.12*ones(1,2); cxst=zeros(1,2);f_near=cfn+tol;
```

```
% fn_name='dejong1'; b=5.12*ones(1,5); cxst=zeros(1,5);f_near=cfn+tol;
% fn_name='dejong1'; b=5.12*ones(1,10);cxst=zeros(1,10);f_near=cfn+tol;
% fn_name='dejong1'; b=5.12*ones(1,30);cxst=zeros(1,30);f_near=cfn+tol;
% fn_name='griewank'; b=400*ones(1,2);  cxst=zeros(1,2);f_near=cfn+tol;
% fn_name='griewank'; b=400*ones(1,5);  cxst=zeros(1,5);f_near=cfn+tol;
% fn_name='griewank'; b=400*ones(1,10); cxst=zeros(1,10);f_near=cfn+tol;
% fn_name='griewank'; b=400*ones(1,30); cxst=zeros(1,30);f_near=cfn+tol;
% fn_name='schwefel0';b=500*ones(1,2); cxst=s*ones(1,2);f_near=cfn+tol;
% fn_name='schwefel0';b=500*ones(1,5); cxst=s*ones(1,5);f_near=cfn+tol;
% fn_name='schwefel0';b=500*ones(1,10);cxst=s*ones(1,10);f_near=cfn+tol;
% fn_name='schwefel0';b=500*ones(1,30);cxst=s*ones(1,30);f_near=cfn+tol;
% fn_name='rastrig'; b=5.12*ones(1,2); cxst=zeros(1,2);f_near=cfn+tol;
% fn_name='rastrig'; b=5.12*ones(1,5); cxst=zeros(1,5);f_near=cfn+tol;
% fn_name='rastrig'; b=5.12*ones(1,10);cxst=zeros(1,10);f_near=cfn+tol;
% fn_name='rastrig'; b=5.12*ones(1,30);cxst=zeros(1,30);f_near=cfn+tol;
max_evals=5000; a=-b; d=length(a);
evals= d * max_evals;
sigma=(b - a) / 10;
sigma = mean(sigma);
fid=fopen('es11.002','a');
disp(fn_name);
for krok=1:10
    disp('krok'); disp(krok);
    [xmin,fnmin] = es1p1(fn_name, a, b, evals, sigma);
    fprintf(fid,'%-15s', 'ES11');
    fprintf(fid,'%-10s %4.0f', fn_name, d);
    fprintf(fid, '%4.0f  ', krok);
    fprintf(fid, '%8.4e  ', sigma);
    fprintf(fid,'%8.0f', evals);
    fprintf(fid,' %22.11e', fnmin);
    fprintf(fid,'%1s\n',' ');
end
fclose(fid);
```

V každé úloze vyhodnocujeme nalezenou průměrnou hodnotu funkce spočítanou z 10 opakování. Výsledky jsou uvedeny v následující tabulce.

	blindsearch		ES(1+1)	
	fprum	std	fprum	std
ackley	0.6214	0.2528	0.2719	0.1937
dejong1	0.0069	0.0080	0.0002	0.0001
griewank	0.1155	0.0497	0.0291	0.0213
rastrig	0.3306	0.3900	0.0531	0.0654
rosen	0.0049	0.0083	0.0004	0.0004
schwefel0	4.3406	3.3183	130.5089	87.5699

Vidíme, že s výjimkou Schwefelovy funkce dosáhla i tato jednoduchá varianta evoluční strategie výrazně lepších výsledků. Průměr nalezených funkčních hodnot je výrazně menší a také směrodatná odchylka nalezených funkčních hodnot je podstatně menší než u slepého prohledávání. Horší výsledky u Schwefelovy funkce jsou způsobeny tím, že prohledávání končí v oblasti některého z lokálních minim a tuto oblast není schopno opustit. Algoritmus ES(1+1), který využívá své "znalosti" získané během hledání (byť jen velmi omezeně), je účinnější než úplně slepé a bezhlavé prohledávání.



Konec příkladu.

Zajímavou otázkou je to, jak volit hodnoty směrodatných odchylek pro mutaci. V rov. (6.1) jsou všechny hodnoty σ_j, $j = 1, 2, \ldots, d$ shodné a konstantní po celé vyhledávání. To samozřejmě není nutné, každá dimenze může mít svou hodnotu směrodatné odchylky, tedy budeme pracovat s vektorem směrodatných odchylek

$$\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \ldots, \sigma_d)$$

Hodnoty směrodatných odchylek mohou být určeny jako poměrná část velikosti hrany d-rozměrného kvádru D, tj.

$$\boldsymbol{\sigma} = c(\boldsymbol{b} - \boldsymbol{a}),$$

kde $0 < c < 1$ je zadaná konstanta. Je zřejmé, že čím je hodnota c větší, tím důkladněji bude doména D prohledávána, ale za cenu pomalejší konvergence, naopak příliš malá hodnota c zvýší riziko předčasné konvergence algoritmu v lokálním minimu.

Navíc, vektor směrodatných odchylek nemusí být v každém iteračním kroku stejný, ale hodnoty jeho prvků se mohou *adaptovat* podle průběhu procesu vyhledávání. Rechenberg studoval vliv velikosti mutace při generování potomka, tj. hodnot σ na rychlost konvergence algoritmu a na dvou jednoduchých funkcích odvodil tzv. *pravidlo jedné pětiny* úspěšnosti. Empiricky pak byla ověřena užitečnost tohoto pravidla i pro jiné funkce. Hodnoty směrodatných odchylek se v i-té iteraci hledání upravují podle pravidla vyjádřeného rovnicí



$$\sigma_i = \begin{cases} c_1\ \sigma_{i-1} & \text{když } \varphi(n) < \frac{1}{5} \\ c_2\ \sigma_{i-1} & \text{když } \varphi(n) > \frac{1}{5} \\ \sigma_{i-1} & \text{jinak} \end{cases} \tag{6.2}$$

$c_1 < 1$ a $c_2 > 1$ jsou vstupní parametry, kterými se řídí zmenšování či zvětšování hodnot směrodatných odchylek podle relativní četnosti úspěchu $\varphi(n)$ v předcházejících n krocích. Úspěchem se rozumí to, že $f(\boldsymbol{y}) < f(\boldsymbol{x})$. Počet kroků n je vstupní parametr. Obvykle se volí hodnoty $c_1 = 0.82$ a $c_2 = 1/c_1 \doteq 1.22$.

Pod vlivem rozvoje jiných evolučních algoritmů bylo dalším krokem v evoluční strategii zavedení populace velikosti větší než jedna. Označíme-li velikost rodičovské populace μ a velikost populace potomků λ, $\lambda > \mu$, pak můžeme uvažovat o dvou variantách algoritmu evoluční strategie, v literatuře obvykle označovaných jako ES(μ+λ) a ES(μ, λ):



- ES($\mu + \lambda$) - generace potomků je vytvořena μ jedinci s nejmenšími funkčními hodnotami ze všech $\mu + \lambda$ jedinců jak rodičovské populace, tak populace potomků. V této variantě se dodržuje tzv. *elitismus*, tj. nejlepší dosud nalezený jedinec vždy přežívá a přechází do nové generace.
- ES(μ, λ) - generace potomků je vytvořena μ jedinci s nejmenšími funkčními hodnotami z λ jedinců populace potomků. Rodičovská generace je tedy kompletně nahrazena nejlepšími jedinci z populace potomků.

Doporučuje se, aby velikost populace potomků λ byla volena několikanásobně větší než velikost rodičovské populace μ. Také se uvádí, že varianta ES(μ, λ) obvykle konverguje pomaleji než ES(μ+λ), ale s menší tendencí ukončit prohledávání v lokálním minimu.

Implementace algoritmu ES(μ+λ) s adaptací směrodatných odchylek podle pravidla jedné pětiny v Matlabu je uvedena v následujícím výpisu:

```
function [x, fx, func_evals] =...
    es_mpl(fn_name, a, b,...
    M, k, lfront, my_eps, max_evals, shift)
```

```
d = length(a);
sigma = (b - a) / 6;        % initial values of sigma
fr = rand(1, lfront) < 0.2; % initial values in last lfront steps
% fr je vektor delky lfront
% s prvky 1 nebo 0 indikujicimi uspech nebo neuspech v poslednich
% lfront generovani bodu y
L = M*k;  % k>1, integer, velikost generace potomku
P = zeros(M, d+1);  % generace rodicu
Q = zeros(L, d+1);  % generace potomku
for i=1:M
    P(i,1:d) = a + (b-a).*rand(1,d);
    P(i,d+1)= feval(fn_name, P(i,1:d)- shift);
end  % 0-th generation initialized
P = sortrows(P, d+1);
func_evals = M;
while ((P(M,d+1) - P(1,d+1)) > my_eps)&&...
        (func_evals <= d*max_evals - L) % main loop
    for t = 1:k
        for i = 1:M
            x = P(i, 1:d);
            y = x + sigma.*randn(1,d);
            y = zrcad(y,a,b);
            fy = feval(fn_name, y - shift);
            func_evals = func_evals + 1;
            Q((t-1)*M + i, :) = [y fy];
            fr(1)= [];
            if fy < P(i,d+1)  % indikator uspechu a neuspechu
                fr(end + 1) = 1;
            else
                fr(end + 1) = 0;
            end
        end
        % adaptace sigma podle cetnosti uspechu a neuspechu
        % v poslednich lfront krocich
        if sum(fr) / lfront < 0.2
            sigma = sigma * 0.82;
        elseif sum(fr) / lfront > 0.2
            sigma = sigma * 1.22;
        end
    end   % vytvorena generace potomku velikosti L
```

```
    PQ =[ P; Q]; % spojeni generace rodicu a potomku
    PQ = sortrows(PQ, d+1);
    P = PQ(1:M,:);   % nova rodicovska generace
end   % main loop - end
x  = P(1,1:d);
fx = P(1,d+1);
```

Autoři algoritmu evoluční strategie v minulosti museli čelit výtkám, že evoluční strategie z osvědčených evolučních operátorů vůbec nevyužívá křížení. Proto byly navrženy pokročilejší varianty evoluční strategie, ve kterých je křížení obsaženo. Základní idea takového křížení spočívá v tom, že u každého jedince v populaci je kromě souřadnic v prostoru D uchováván i jeho vektor směrodatných odchylek. Vektor směrodatných odchylek potomka se pak generuje křížením s náhodně vybraným jiným jedincem. Máme-li dva rodiče $\boldsymbol{r}$, $\boldsymbol{s}$ s vektory směrodatných odchylek $\boldsymbol{\sigma}^r$, $\boldsymbol{\sigma}^s$, pak vektor směrodatných odchylek jejich potomka je určován např. jako

$$\boldsymbol{\sigma} = \frac{\boldsymbol{\sigma}^r + \boldsymbol{\sigma}^s}{2},$$

což je tzv. křížení průměrem, nebo podle nějakého podobného pravidla pro křížení.

Pro funkce, u kterých se dá očekávat globální minimum v protáhlém údolí, jehož směr není rovnoběžný se žádnou dimenzí prostoru D, je vhodné užít ještě sofistikovanější variantu evoluční strategie, v níž u každého jedince v populaci se kromě souřadnic v prostoru D a jeho vektoru směrodatných odchylek uchovávají i mimodiagonální prvky kovarianční matice rozměru $(d \times d)$. Nový bod vzniká mutací, tj. přičtením náhodného vektoru $\boldsymbol{u} \sim N(\mathbf{0}, \boldsymbol{\Sigma})$, kde $\boldsymbol{\Sigma}$ je kovarianční matice vektorů souřadnic bodů v populaci. Také křížení může probíhat nejen na vektorech směrodatných odchylek, ale na celé kovarianční matici. Takové modifikace evoluční strategie jsou v literatuře označovány zkratkou CMES (Covariance Matrix Evolution Strategy). Jsou však nejenom implementačně podstatně náročnější, ale také mají větší paměťové nároky i větší časovou spotřebu na jeden iterační krok. Proto se užívají spíše jen u problémů, kdy vyhodnocení účelové funkce je časově náročné a snížení počtu vyhodnocení funkce uspoří více času, než který je spotřebován na opakované vyhodnocování kovarianční matice a složitější křížení. Místo kovariancí dvojic souřadnic se mohou užívat směrové úhly, které lze z kovarianční matice snadno vyhodnotit.



Porovnáme-li evoluční strategii ES $(\mu + \lambda)$ s algoritmem CRS, mohli bychom CRS označit jako evoluční algoritmus typu $(\mu + 1)$, neboť nová generace má jen jednoho jedince, který může nahradit nejhoršího jedince v populaci. Stejně jako v ES$(\mu + \lambda)$ se spojuje stará a nová generace a z této spojené populace přežívá μ nejlepších jedinců, tedy užívá se elitismus. Na rozdíl od evoluční strategie se nový bod v CRS negeneruje mutací přičtením normálně rozděleného náhodného vektoru, ale lokální

heuristikou, např. reflexí náhodně vybraného simplexu, kterou můžeme považovat za kombinaci mutace a křížení.

Shrnutí:



- mutace v evoluční strategii, její řídicí parametry
- algoritmus ES (1+1)
- adaptace hodnot směrodatných odchylek
- ES($\mu + \lambda$) a ES(μ, λ), elitismus

Kontrolní otázky:



1. Co je pravidlo jedné pětiny a k čemu se užívá?
2. Porovnejte výhody a nevýhody variant ES(μ, λ) a ES($\mu + \lambda$).
3. Jak je pravidlo jedné pětiny implementováno ve variantě ES($\mu + \lambda$) v Matlabu a kterým parametrem řídíme délku sledovaného počtu předchozích iterací?
4. Jak byste porovnali účinnosti algoritmu ES($\mu+\lambda$) a ES (1+1) experimentálně? O kterém z těchto algoritmů se domníváte, že bude účinnější?

6.3 Diferenciální evoluce



Průvodce studiem:

V této kapitole se seznámíte s algoritmem diferenciální evoluce. Tento algoritmus byl navržen nedávno a poprvé publikován v roce 1995. Během několika let se stal velmi populární, byl podrobně studován na mnoha pracovištích a je často aplikován na problémy hledání globálního minima. Je to příklad jednoduchého heuristického hledání, ve kterém se užívají evoluční operátory. Počítejte asi se třemi až pěti hodinami studia.

Diferenciální evoluce (DE) je postup heuristického hledání minima funkcí, který navrhli R. Storn a K. Price [20] v devadesátých letech. Experimentální výsledky z testování i zkušenosti z četných aplikací ukazují, že často konverguje rychleji než jiné stochastické algoritmy pro globální optimalizaci.



Algoritmus diferenciální evoluce vytváří novou generaci Q tak, že postupně pro každý bod $\boldsymbol{x}_i$ ze staré generace P vytvoří jeho potenciálního konkurenta $\boldsymbol{y}$ a do nové populace z této dvojice zařadí bod s nižší funkční hodnotou. Algoritmus můžeme v pseudokódu zapsat takto:

generuj počáteční populaci P (N bodů náhodně v D)
repeat
  for i := 1 **to** N **do**
    generuj vektor $\boldsymbol{u}$ mutací
    vytvoř vektor $\boldsymbol{y}$ křížením $\boldsymbol{u}$ a $\boldsymbol{x}_i$
    if $f(\boldsymbol{y}) < f(\boldsymbol{x}_i)$ **then** do Q zařaď $\boldsymbol{y}$
      else do Q zařaď $\boldsymbol{x}_i$
  endfor
  $P := Q$
until podmínka ukončení

Generování vektoru $\boldsymbol{u}$ představuje v DE mutaci, tento vektor-mutant je pak jedním z rodičovských vektorů pro křížení, druhým rodičem je aktuální protějšek ve staré generaci, nový zkusmý bod $\boldsymbol{y}$ je tedy produktem evolučních operátorů mutace a křížení, selekce se provádí turnajem s jeho protějškem ve staré generaci.

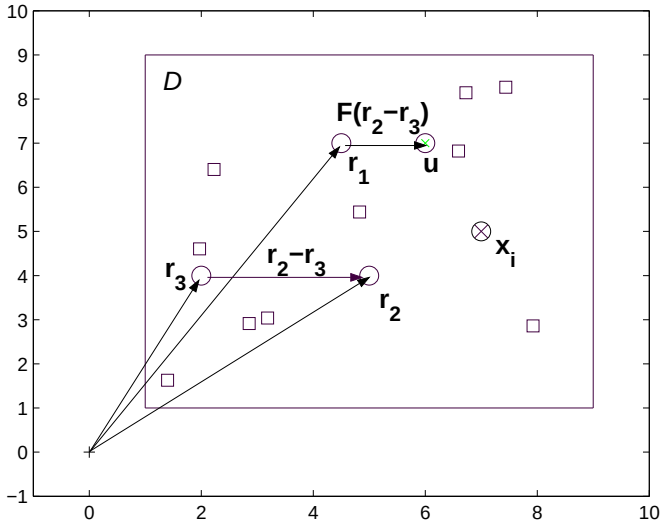


Je několik možných způsobů mutace a dva druhy křížení. Varianty těchto různých strategií diferenciální evoluce se často označují zkratkou DE/m/a/k, kde m znamená užitý způsob mutace, a je počet rozdílů (diferencí) vektorů v operaci mutace a k je typ křížení.

Nejčastěji užívanou mutací v DE je postup označovaný *rand/1/*, který generuje bod $\boldsymbol{u}$ ze tří bodů ze staré populace podle vztahu

$$\boldsymbol{u} = \boldsymbol{r}_1 + F(\boldsymbol{r}_2 - \boldsymbol{r}_3), \tag{6.1}$$

$\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3$ jsou navzájem různé body náhodně vybrané z populace P různé od aktuálního bodu $\boldsymbol{x}_i$, $F > 0$ je vstupní parametr. Generování bodu $\boldsymbol{u}$ podle rovnice (6.1) je graficky znázorněno následujícím obrázkem.



Další často užívaný druh mutace využívá nejlepší bod z populace P a čtyři další náhodně vybrané body podle vztahu

$$\boldsymbol{u} = \boldsymbol{x}_{\text{best}} + F(\boldsymbol{r}_1 + \boldsymbol{r}_2 - \boldsymbol{r}_3 - \boldsymbol{r}_4), \tag{6.2}$$

kde $\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{r}_3, \boldsymbol{r}_4$ jsou navzájem různé body náhodně vybrané z populace P různé od aktuálního bodu $\boldsymbol{x}_i$ i od bodu $\boldsymbol{x}_{\text{best}}$ s nejnižší funkční hodnotou v populaci P. $F > 0$ je opět vstupní parametr. V (6.2) vidíme, že k vektoru $\boldsymbol{x}_{\text{best}}$ se přičítají dva rozdíly náhodně vybraných vektorů násobené F, proto je tato mutace označována *best/2/*.

Kaelo a Ali [11] navrhli modifikovat mutaci *rand/1/* tak, že vektor $\boldsymbol{r}_1$ v (6.1) je ten s nejmenší funkční hodnotou mezi vektory $\boldsymbol{r}_1$, $\boldsymbol{r}_2$ a $\boldsymbol{r}_3$. Označují takovou mutaci jako náhodnou lokalizaci (*random localization*) a tuto mutaci můžeme označit jako *randrl/1/*. V rozsáhlých experimentálních testech ukázali, že oproti *rand/1/* tato mutace snižuje počet vyhodnocení potřebných k dosažení podmínky ukončení až o třetinu a většinou je zachována spolehlivost dobrého přiblížení ke globálnímu minimu.

V poslední době se v některých variantách DE užívá mutace *current-to-best/2/*, která generuje vektor $\boldsymbol{u}$ podle vztahu

$$\boldsymbol{u} = \boldsymbol{x}_i + F(\boldsymbol{x}_{\text{best}} - \boldsymbol{x}_i) + F(\boldsymbol{r}_1 - \boldsymbol{r}_2) \tag{6.3}$$

Tato mutace má podobně jako (6.2) tendenci zrychlovat konvergenci algoritmu a zvyšovat riziko předčasné konvergence v lokálním minimu.

Kromě uvedených druhů mutace se v literatuře objevuje i celá řada dalších, ale mutace *rand/1/* je stále v aplikacích DE nejčastěji užívanou.

Nový vektor $\boldsymbol{y}$ vznikne *křížením* vektoru $\boldsymbol{u}$ a vektoru $\boldsymbol{x}_i$. V diferenciální evoluci se užívají dva typy křížení, a to *binomické* křížení a *exponenciální* křížení. Oba typy křížení byly navrženy v prvních publikacích o DE [20].

V binomickém křížení vznikne vektor $\boldsymbol{y}$ tak, že kterýkoli prvek tohoto vektoru může roven buď odpovídajícímu prvku vektoru $\boldsymbol{x}_i$ nebo odpovídajícímu prvku vektoru $\boldsymbol{u}$. Pravděpodobnost, že ve vektoru $\boldsymbol{y}$ bude prvek vektoru $\boldsymbol{u}$ je úměrná hodnotě zadaného vstupního parametru CR. Binomické křížení probíhá podle následujícího pravidla:

$$y_j = \begin{cases} u_j & \text{když } R_j \leq CR \quad \text{nebo } j = I \\ x_{ij} & \text{když } R_j > CR \quad \text{a } j \neq I \ , \end{cases} \tag{6.4}$$

kde I je náhodně vybrané celé celé číslo z $\{1, 2, \ldots, d\}$, $R_j \in (0, 1)$ jsou voleny náhodně a nezávisle pro každé j a $CR \in [0, 1]$ je vstupní parametr. Z pravidla (6.4) vidíme, že nejméně jeden prvek vektoru $\boldsymbol{u}$ přechází do nového bodu zkusmého $\boldsymbol{y}$, dokonce i při volbě $CR = 0$. Strategie diferenciální evoluce užívající binomiální křížení se označují zkratkou DE/m/a/*bin*.



V exponenciálním křížení se vyměňuje náhodně určený počet *sousedních* prvků vektorů. V tom je exponenciální křížení podobné dvoubodovému křížení popsaném v genetických algoritmech. Počáteční pozice pro křížení se vybere náhodně z $\{1, \ldots, d\}$ a pak L následujících prvků (počítáno cyklicky, tj. za d-tým prvkem následuje první prvek vektoru) se vloží z vektoru $\boldsymbol{u}$. Jeden prvek vektoru $\boldsymbol{u}$ se vloží vždy, pravděpodobnost vložení dalších prvků pak klesá exponenciálně. Strategie diferenciální evoluce užívající exponenciální křížení se označují zkratkou DE/m/a/*exp*. Popsané exponenciální křížení lze snadno implementovat využitím cyklu `while` pro zvětšování počtu prvků z vektoru $\boldsymbol{u}$, jak je ukázáno v následujícím zdrojovém textu funkce `rand1_exp`, která vytvoří nový pokusný bod $\boldsymbol{y}$ strategií DE/*rand*/1/*exp*.

```
%  rand/1/exp
function y = rand1_exp(P, F, CR, expt)
N = length(P(:, 1));
d = length(P(1,:)) - 1;
y = P(expt(1),1:d);          % expt(1) je index akt. radku v P
vyb = nahvyb_expt(N, 3, expt);  % three random points without expt
r1 = P(vyb(1),1:d);
r2 = P(vyb(2),1:d);
r3 = P(vyb(3),1:d);
u = r1 + F*(r2 - r3);
nah_pozice = 1 + fix(d*rand(1));
pridej = 0;
while rand(1) < CR && pridej < d - 1
    pridej = pridej + 1;
end
delka_nazacatku = nah_pozice + pridej - d;
if delka_nazacatku <= 0
    indexy_zmeny = nah_pozice: nah_pozice + pridej;
else
    indexy_zmeny = [1: delka_nazacatku, nah_pozice:d];
end
y(indexy_zmeny)= u(indexy_zmeny);
```

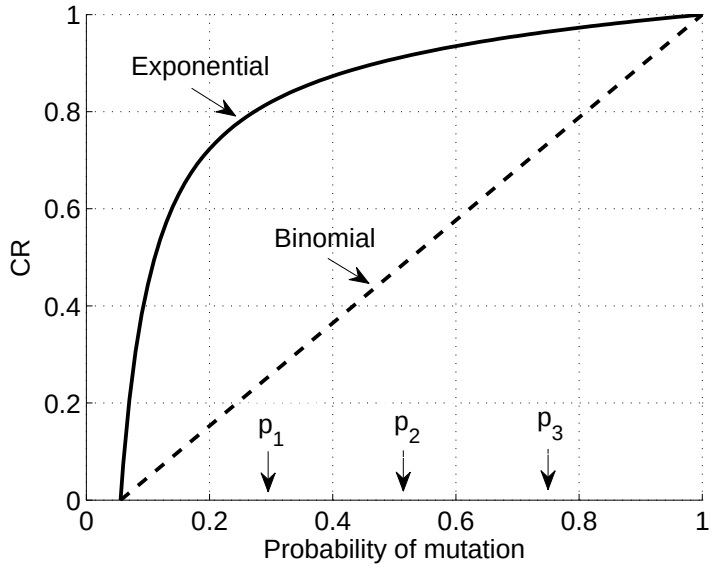
Pravděpodobnost p_m zařazení prvků mutačního vektoru $\boldsymbol{u}$ do zkusmého vektoru $\boldsymbol{y}$ je možno definovat jako střední hodnotu relativní četnosti těchto prvků, tj. $p_m = E(L)/d$. Vztah mezi touto pravděpodobností p_m a řídicím parametrem CR podrobně studovala Zaharie [28]. U binomického křížení je tato závislost lineární,

$$p_m = CR(1 - 1/d) + 1/d, \tag{6.5}$$

zatímco pro exponenciální křížení je tato závislost silně nelineární a odchylka od linearity se zvětšuje s rostoucí dimenzí prohledáváného prostoru.

$$p_m = \frac{1 - CR^d}{d(1 - CR)}, \quad \text{pro} \quad CR < 1. \tag{6.6}$$

Graf závislosti CR a p_m pro $d = 18$ je na následujícím obrázku. Nelinearitu této závislosti pro exponenciální křížení je nutno vzít v úvahu při zadávání hodnoty parametru CR. Např. pro tuto dimenzi problému, chceme-li dosáhnout $p_m \doteq 0.5$, je nutno zadat $CR \doteq 0.9$.



Výhodou algoritmu diferenciální evoluce je jeho jednoduchost a výpočetně nenáročné generování nového bodu $\boldsymbol{y}$, které lze navíc velmi efektivně implementovat. Pro zvolenou strategii DE/m/a/k je potřeba zadat jen tři hodnoty řídicích parametrů, a to velikost populace N, hodnotu parametru F pro mutaci a hodnotu parametru CR pro křížení. Nevýhodou je poměrně velká citlivost algoritmu na nastavení hodnot parametrů F a CR. Storn a Price doporučují volit hodnoty $N = 10d$, $F = 0.8$ a $CR = 0.5$ a pak tyto hodnoty modifikovat podle empirické zkušenosti z pozorovaného průběhu hledání. To je ovšem doporučení velmi vágní a časově náročné. Důležitá je i zkušenost a dobrá intuice řešitele problému. Většinou lze užít hodnoty $0.5 \leq F \leq 1$ a hodnoty CR z celého oboru, $0 \leq CR \leq 1$. Také velikost populace N často postačí menší než $N = 10d$, tím je možné zvýšit rychlost konvergence, ale současně se při menší velikosti populace zvyšuje riziko předčasné konvergence.

Citlivost algoritmu DE na nastavení hodnot řídicích parametrů vedla k různým návrhům adaptivních verzí DE, ve kterých se hodnoty řídicích parametrů mění adaptivně podle řešeného problému v průběhu prohledávání, takže uživatel pouze nastaví jejich počáteční hodnoty nebo dokonce se nemusí o nastavení parametrů starat vůbec. Příkladem takového postupu je adaptivní nastavení parametru F, který navrhli Ali a Törn. Hodnota parametru F se v každé generaci mění podle adaptivního pravidla

$$F = \begin{cases} \max(F_{\min}, 1 - |\frac{f_{\max}}{f_{\min}}|) & \text{if } |\frac{f_{\max}}{f_{\min}}| < 1 \\ \max(F_{\min}, 1 - |\frac{f_{\min}}{f_{\max}}|) & \text{jinak,} \end{cases} \tag{6.7}$$

kde $f_{\min}$, $f_{\max}$ jsou minimální a maximální funkční hodnoty v populaci P a $F_{\min}$ je vstupní parametr, který zabezpečuje, aby bylo $F \in [F_{\min}, 1)$. Autoři doporučují volit hodnotu $F_{\min} \geq 0.45$. Předpokládá se, že tento způsob výpočtu F udržuje prohledávání diverzifikované v počátečním stádiu a intenzivnější v pozdější fázi prohledávání, což obvykle zvyšuje spolehlivost hledání i rychlost konvergence. Některé

další možnosti adaptace vyhledávací strategie pro hledání globálního minima ukážeme v závěrečné kapitole tohoto učebního textu.

Porovnáme-li diferenciální evoluci s algoritmem CRS, vidíme, že v diferenciální evoluci se nenahrazuje nejhorší bod v populaci, ale pouze horší bod ve dvojici, což zabezpečuje větší diverzitu bodů populace po delší období, takže při stejné velikosti populace DE ve srovnání s CRS má většinou menší tendenci ukončit prohledávání v lokálním minimu, ale za to platíme pomalejší konvergencí při stejné podmínce ukončení.



Nejčastěji užívanou variantou diferenciální evoluce je DE/*rand*/1/*bin*. Jak uvidíme v následujícím zdrojovém textu, její naprogramování v Matlabu je velmi snadné a lze ji zapsat na zhruba 25 řádcích, pokud využijeme funkce `zrcad` a `nahvyb_expt` popsané v předcházejících kapitolách a funkci `rand1_bin`, která vytvoří nový zkusmý bod $\boldsymbol{y}$ mutací *rand*/1/ a binomickým křížením, zdrojový text této funkce následuje.

```
function y = rand1_bin(P, F, CR, expt)
N = length(P(:,1));
d = length(P(1,:)) - 1;
y = P(expt(1),1:d);
vyb = nahvyb_expt(N, 3, expt);  % three random points without expt
r1 = P(vyb(1),1:d);
r2 = P(vyb(2),1:d);
r3 = P(vyb(3),1:d);
u = r1 + F*(r2 - r3);
change = find(rand(1,d) < CR);
if isempty(change) % at least one element is changed
    change = 1 + fix(d*rand(1));
end
y(change) = u(change);
```

```
function [x_star,fn_star,func_evals] = ...
    de_rand1_bin(fn_name, a, b, N, my_eps, max_evals, F, CR, shift)
% input parameters:
%   a, b        row vectors, limits of search space, a < b
%   N           size of population
%   my_eps      small positive value for stopping criterion
%   max_evals   max. evals per one dimension of search space
%   F,CR        mutation parameters
% output:
%   fn_star         the minimal function value found by MCRS
%   x_star          global minimum found by search
%   func_evals      number of func_evals for reaching stop
d = length(a);
P = zeros(N,d+1);
for i=1:N
    P(i,1:d) = a + (b - a).*rand(1,d);
    P(i,d+1) = feval(fn_name, P(i,1:d)-shift);
end  % 0-th generation initialized
fmax = max(P(:,d+1));
[fmin, indmin] = min(P(:,d+1));
func_evals = N;  Q = P;
while (fmax-fmin > my_eps) && (func_evals < d*max_evals) % main loop
    for i=1:N  %  in each generation
        y = rand1_bin(P ,F, CR, i);
        y = zrcad(y, a, b);     % perturbation
        fy = feval(fn_name, y-shift);
        func_evals = func_evals + 1;
        if fy < P(i,d+1)  % trial point y is good
            Q(i,:) = [y fy];
        end
    end % end of generation
    P = Q;
    fmax = max(P(:,d+1));
    [fmin,indmin] = min(P(:,d+1));
end   % main loop - end
x_star = P(indmin,1:d);
fn_star = fmin;
```

Shrnutí:



- Mutace jako přičtení rozdílu vektorů náhodně vybraných z populace
- Křížení výměnou náhodně vybraných prvků vektoru
- Binomické křížení, exponenciální křížení
- Výběr lepšího z dvojice (selekce turnajem)

Kontrolní otázky:



1. Jaké jsou hlavní odlišnosti řízeného náhodného výběru a diferenciální evoluce?
2. Čím se liší binomické a exponenciální křížení?
3. Proč musí být $F \neq 0$? Co by se stalo, kdybychom zadali $F < 0$?
4. Co způsobí zadání $CR = 0$ nebo $CR = 1$? Bude se pak lišit binomické křížení od exponenciálního?

6.4 Experimentální porovnání jednoduchých evolučních algoritmů



Průvodce studiem:

V této kapitole si ukážeme experimentální porovnání tří jednoduchých evolučních algoritmů na šesti testovacích funkcích při dvou úrovních dimenze úloh. Počítejte asi se dvěma hodinami studia.

Máme porovnat účinnost tří jednoduchých algoritmů na 6 testovacích funkcích z kapitoly 3.2 pro dimenze problému $d = 2$ a $d = 10$. U čtyřech funkcí s globálním minimem v bodě $\boldsymbol{x}^* = (0, 0, \ldots, 0)$ užijeme náhodný posun popsaný v kapitole 3.2. Testovány jsou tyto algoritmy

- Řízené náhodné prohledávání (CRS) se znáhodněnou reflexí simplexu podle (5.2), $\alpha = 4$, velikost populace byla $N = 10d$.
- Evoluční strategie ES(μ, λ) s adaptací velikosti směrodatných odchylek podle pravidla jedné pětiny, $\mu = 10d$ a $\lambda = 4\ \mu$.
- Diferenciální evoluce DE/*rand*/1/*bin* s parametry $F = 0.5$ a $CR = 0.5$. Velikost populace byla $N = 5d$.

Pro každou úlohu bylo provedeno 100 opakování. Ve všech úlohách byla stejná podmínka ukončení, hledání bylo ukončeno, když rozdíl mezi největší a nejmenší funkční hodnotou v populaci byl menší než $1e{-}6$ nebo počet vyhodnocení funkcí dosáhl hodnotu 20000 d. Za úspěšné hledání bylo považováno nalezení bodu s funkční hodnotou menší než $1e - 4$.

Sledována byla spolehlivost algoritmů v nacházení správného řešení tj. počet běhů, kdy nalezená minimální hodnota funkce je menší než $1e - 4$ (v tabulkách je hodnota spolehlivosti uvedena ve sloupci R) a časová náročnost algoritmů vyjádřená jako průměrný počet vyhodnocení funkce potřebný k dosažení podmínky ukončení, v tabulkách ve sloupci $\overline{nfe}$. Tabulky následují dále v textu.

V úlohách s $d = 2$ měl nejvyšší spolehlivost algoritmus CRS, nejrychleji podmínky ukončení dosahoval algoritmus DE/*rand*/1/*bin* se spolehlivostí srovnatelnou s algoritmem ES(μ, λ), ovšem tento algoritmus měl podstatně vyšší časové nároky než diferenciální evoluce. Povšimněme si výsledků u Rosenbrockovy funkce. Tam s daleko nejmenšími časovými nároky dosahoval spolehlivosti 100 (všechny běhy nalezly správné řešení), zatímco u jiných funkcí byly jeho časové nároky srovnatelné s ES(μ, λ) a výrazně vyšší oproti DE/*rand*/1/*bin*. Je to jedna z ukázek důsledku No

free lunch teorému, nějaký algoritmus je efektivní jen pro nějakou skupinu problémů, v jiných problémech jsou jiné algoritmy efektivnější.

$d = 2$

Algoritmus →	CRS		ES(μ, λ)		DE/*rand*/1/*bin*	
Funkce↓	R	$\overline{nfe}$	R	$\overline{nfe}$	R	$\overline{nfe}$
ackley	100	2053	87	2988	94	810
dejong1	100	735	100	1715	98	421
griewank	80	3893	27	2284	70	1234
rastrig	94	1768	80	2158	84	654
rosen	100	947	83	8438	66	1554
schwefel0	94	1293	96	2360	76	636
průměr	95	1781	79	3324	81	885

$d = 10$

Algoritmus →	CRS		ES(μ, λ)		DE/*rand*/1/*bin*	
Funkce↓	R	$\overline{nfe}$	R	$\overline{nfe}$	R	$\overline{nfe}$
ackley	19	12230	6	32212	100	15756
dejong1	99	7911	100	26068	100	7728
griewank	19	15650	0	34072	100	37392
rastrig	1	53392	0	30188	100	30920
rosen	75	53019	0	199700	0	198601
schwefel0	0	200000	0	34024	99	13644
průměr	36	57034	18	59377	83	50673

V úlohách s $d = 10$ je jasným vítězem algoritmus DE/*rand*/1/*bin*, který v pěti testovacích úlohách dosáhl téměř 100% spolehlivosti většinou i s nejmenšími časovými nároky. Ovšem kompletně selhal v hledání minima Rosenbrockovy funkce, kde byl opět nejúspěšnější algoritmus CRS. Testovaná varianta evoluční strategie zcela selhala na 4 testovacích funkcích a dokonce i k řešení nejjednoduššího problému (De Jongova první funkce) potřebovala nesrovnatelně větší čas než ostatní dva algoritmy.

Experimenty ukázaly, že i velmi jednoduché varianty stochastických algoritmů jsou schopny vyřešit některé úlohy, ale rozhodně nemůžeme spoléhat na to, že vyřeší

každou úlohu. To ostatně nemůžeme očekávat od žádného stochastického algoritmu. Téměř jistě by bylo možno i pro jednoduché algoritmy v tomto testu najít jiné nastavení hodnot řídicích parametrů, pro které by pak dosáhly lepších výsledků. Ale takové hledání vhodných hodnot řídicích parametrů pro řešený problém je časově náročné a jeho jediný smysl je získat zkušenosti v tomto nastavování, které pak lze v omezené míře využít i pro jiné úlohy. Vhodnější cesta je hledání adaptivních algoritmů, které si mění hodnoty řídicích parametrů v průběhu řešení konkrétní úlohy samy, a tak dosahují vyšší účinnosti pro širší třídu optimalizačních problémů.



Shrnutí:

- Experimentální porovnání účinnosti stochastických algoritmů
- Spolehlivost, časová náročnost algoritmu
- Popis experimentu, prezentace a interpretace výsledků



Kontrolní otázky:

1. Co je nutné specifikovat v popisu experimentu?
2. Které veličiny charakterizují účinnost algoritmu?
3. Lze dosažené výsledky nějak odůvodnit?

7 Algoritmy modelující chování skupin

Mezi stochastickými algoritmy pro řešení problému globální optimalizace mají významné místo algoritmy, které modelují sociální chování skupiny (hejna, smečky, kolonie) živých jedinců. Někdy jsou označovány jako modely *particle swarm intelligence* [6], což bychom mohli přeložit jako kolektivní inteligence skupiny. Tyto algoritmy se podobají evolučním algoritmům v tom, že jde o modely interakce více kandidátů řešení, ale liší se ve způsobu modelování interakce mezi jedinci. V evolučních algoritmech máme populaci, která se vyvíjí tím, že probíhá výběr silnějších jedinců, jejich křížení a mutace jejich genetického kódu. U algoritmů popisovaných v této kapitole máme skupinu (do jisté míry je to analogie populace), ale její pohyb v prohledávaném prostoru řešení se řídí pravidly skupinového chování. Mezi nejpopulárnější algoritmy této kategorie patří Particle Swarm Optimization (PSO), Self-Organizing Migration Algorithm (SOMA) a algoritmus mravenčí kolonie (Ant Colony). S principy některých takových algoritmů se seznámíme v této kapitole a ukážeme si také, jak mohou být implementovány v Matlabu.

7.1 Algoritmus PSO

Průvodce studiem:



V této kapitole se seznámíte s principy algoritmu PSO, který vznikl jako model chování skupiny (hejna) ryb nebo ptáků. Počítejte asi se čtyřmi až šesti hodinami studia.

Algoritmus Particle Swarm Optimization (PSO), který jste se už možná poznali v předmětu Úvod do soft computingu [21], byl inspirován chováním hejn ryb a ptáků. Jedinci v takových hejnech mají své individuální chování a současně jsou ovlivňováni ostatními členy hejna. Představme si, že celé hejno tvořené N jedinci hledá nějaký společný cíl a pohybuje se po etapách směrem k tomuto cíli. Změna pozice jedince v dané etapě je dána jeho rychlostí (jak víte ze základní školy, rychlost je dráha uražená za jednotku času). Nechť tedy etapa trvá jednu časovou jednotku a etapy postupného pohybu jedinců hejna očíslujeme $0, 1, 2, \ldots$. Pozici i-tého jedince na začátku následující etapy určíme tak, že k jeho současné pozici $\boldsymbol{x}_i$ přičteme dráhu, kterou v nynější etapě jedinec urazí (tj. rychlost násobena délkou trvání etapy, ale už jsme se dohodli, že etapa trvá jednotkový čas, takže násobení jedničkou můžeme vynechat). Pak nová pozice i-tého jedince v etapě $t + 1$ je dána vztahem

$$\boldsymbol{x}_i(t + 1) = \boldsymbol{x}_i(t) + \boldsymbol{v}_i(t + 1). \tag{7.1}$$

Rychlost jedince v aktuální etapě je ovlivňována jednak minulostí tohoto jedince, jednak minulostí celého hejna. Jak jedinec, tak i ostatní členové hejna získali v minulosti nějaké znalosti o prohledávaném prostoru. Nejjednodušeji můžeme vyjádřit rychlost i-tého jedince v etapě $t+1$ takto:

$$\boldsymbol{v}_i(t+1) = \omega \boldsymbol{v}_i(t) + \varphi_1 \boldsymbol{u_1} \otimes (\boldsymbol{p}_i - \boldsymbol{x}_i) + \varphi_2 \boldsymbol{u_2} \otimes (\boldsymbol{g} - \boldsymbol{x}_i) \tag{7.2}$$

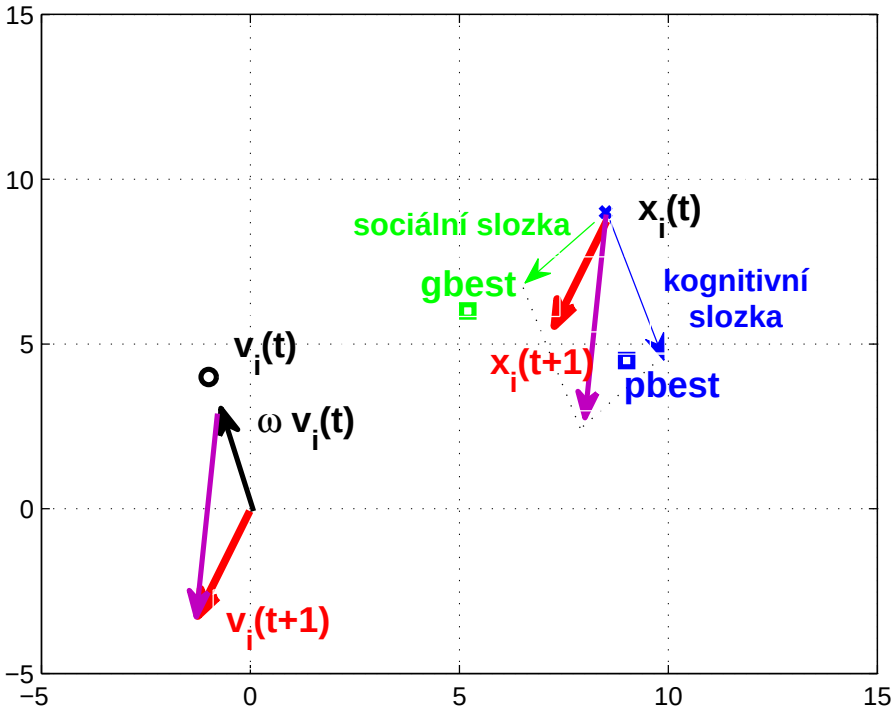
Koeficienty $\omega,\ \varphi_1,\ \varphi_2$ jsou vstupní řídicí parametry algoritmu, $\boldsymbol{u_1},\ \boldsymbol{u_2}$ jsou vektory dimenze d, jejichž prvky jsou nezávislé náhodné hodnoty rovnoměrně rozdělené na intervalu $[0, 1)$, který v Matlabu můžeme vygenerovat příkazem `rand(1,d)`. Symbol $\otimes$ značí násobení dvou vektorů po složkách, v Matlabu se pro tuto operaci užívá operátor `.*`.

Vidíme, že rychlost i-tého jedince v etapě $t+1$ je tvořena třemi složkami

- Rychlostí i-tého jedince v minulé etapě $\boldsymbol{v}_i(t)$, tedy *setrvačná složka* vyjadřující směr, kterým je jedinec rozběhnut. Tato rychlost je násobena váhou setrvačnosti ω, která se obvykle volí $\omega < 1$. Čím menší je hodnota ω, tím více jedinec "zapomíná" svou rychlost z minulé etapy.
- Druhý člen $\varphi_1 \boldsymbol{u_1} \otimes (\boldsymbol{p}_i - \boldsymbol{x}_i)$ je určován minulostí jedince, $\boldsymbol{p}_i$ je poloha bodu s nejmenší funkční hodnotou, kterou i-tý jedinec našel od začátku prohledávání. Hodnota $\boldsymbol{p}_i$ bývá v pracích zabývajících se PSO označována jako *personal best (pbest)*. Tento člen odráží to, co se jedinec ve své minulosti naučil, označuje se jako *kognitivní složka* rychlosti, někdy také jako *nostalgie*, protože vyjadřuje sklon jedince k návratu do nejlepšího bodu své minulosti.
- Třetí člen $\varphi_2 \boldsymbol{u_2} \otimes (\boldsymbol{g} - \boldsymbol{x}_i)$ je určován minulostí celého hejna, bod $\boldsymbol{g}$ *(global best, gbest)* je poloha bodu s nejmenší funkční hodnotou, která byla nalezena hejnem od začátku prohledávání. Tento člen je *sociální složka* rychlosti, neboť modeluje vliv společnosti (hejna) na chování jedince, tedy něco jako podlehnutí vlivu okolí nebo davové psychóze.

Geometrický význam skládání vektorů podle vztahů (7.1) a (7.2) je znázorněn na následujícím obrázku. Povšimněme si, že vektor kognitivní složky $\varphi_1 \boldsymbol{u_1} \otimes (\boldsymbol{p}_i - \boldsymbol{x}_i)$ "nemíří" přímo do bodu `pbest` a podobně ani vektor sociální složky $\varphi_2 \boldsymbol{u_2} \otimes (\boldsymbol{g} - \boldsymbol{x}_i)$ "nemíří" přímo do bodu `gbest`. Je to způsobeno tím, že v obou těchto složkách je vektor rozdílu násoben náhodným vektorem, takže směr je ovlivňován i náhodou a je pouze přibližně do bodů `pbest`, resp. `gbest`.

Takto stručný popis nám stačí k tomu, abychom mohli implementovat základní nejjednodušší verzi algoritmu PSO. Pozice jedinců v nulté etapě se vygenerují náhodně rovnoměrně rozdělené v prohledávaném prostoru D. Podobně i počáteční rychlosti se vygenerují náhodně. Zápis tohoto algoritmu v Matlabu následuje za obrázkem.




```
function [xmin, fmin, func_evals, evals_near]=...
    pso1(fn_name, a, b, N, fi1, fi2, omega, ...
    my_eps, max_evals, shift, fnear)
%
d=length(a);
P=zeros(N,d+1); % alokace pameti pro P
v=zeros(N,d);   % alokace pameti pro v
vmax=(b-a)/6;   % pro generovani pocatecnich rychlosti
size(vmax)
for i=1:N
    P(i,1:d)=a+(b-a).*rand(1,d);
    P(i,d+1)=feval(fn_name,P(i,1:d)-shift);
    v(i,:)= -vmax + 2*vmax.*rand(1,d);
end  % 0-th generation initialized
func_evals=N; evals_near=0;
pbest=P; % nejlepsi pozice jedincu na pocatku = pocatecni pozice
[gbest_fun,indgbest]=min(pbest(:,d+1));
gbest=P(indgbest,:);
fmax=max(P(:,d+1));
fmin=min(P(:,d+1));
while (fmax-fmin > my_eps) && (func_evals < d*max_evals)
    for i=1:N  %  in each generation
```

```
        x=P(i,1:d); % pozice i-teho jedince
        v(i,:)=omega*v(i,:)+fi1*rand(1,d).*(pbest(i,1:d)-x)...
            +fi2*rand(1,d).*(gbest(1:d)-x);
        y = x + v(i,:);  % nova pozice
        y=zrcad(y,a,b); % zabranuje opustit  definicni obor D
        fy=feval(fn_name,y-shift);
        P(i,:)=[y,fy];  % zapis novou pozici a funkcni hodnotu
        func_evals=func_evals+1;
        if fy < pbest(i,d+1)  % aktualizace pbest
            pbest(i,:)= [y,fy];   % prepis  pbest
        end
        if fy < gbest(d+1)  % aktualizace gbest
            gbest=[y, fy];
        end
    end % end of generation
    fmax=max(P(:,d+1));
    fmin=min(P(:,d+1));
    if evals_near==0 && fmin<fnear
        evals_near=func_evals;
    end
end   %  end while
fmin=gbest(d+1);
xmin=gbest(1:d);
```

Tato základní verze algoritmu PSO byla v průběhu posledních let mnohokrát zdokonalována. Jedním z takových kroků bylo zavedení *koeficientu sevření (coefficient of constriction)*. Tím se drobně změní vztah (7.2) pro určení rychlosti jedince v etapě $t+1$:

$$\boldsymbol{v}_i(t+1) = \chi\left[\boldsymbol{v}_i(t) + \varphi_1 \boldsymbol{u_1} \otimes (\boldsymbol{p}_i - \boldsymbol{x}_i) + \varphi_2 \boldsymbol{u_2} \otimes (\boldsymbol{g} - \boldsymbol{x}_i)\right], \tag{7.3}$$

kde koeficient sevření χ je určen vztahem

$$\chi = \frac{2\kappa}{\varphi - 2 + \sqrt{\varphi^2 - 4\varphi}},$$

kde $\varphi = \varphi_1 + \varphi_2$ a κ je vstupní parametr ovlivňující velikost koeficientu sevření. Obvykle se volí $\kappa \approx 1$. Vzhledem k tomu, že je nutné, aby $\varphi > 4$, je potřeba tomuto požadavku podřídit volbu vstupních parametrů φ_1 a φ_2. Pro v literatuře doporučované hodnoty $\varphi_1 = \varphi_2 = 2.05$ a $\kappa = 1$ vyjde hodnota koeficientu sevření $\chi \doteq 0.73$.

Další modifikací algoritmu PSO je jiný způsob výpočtu sociální složky rychlosti jedince. Vytvoříme model, kdy jedinec není ovlivňován celou skupinou, ale jenom její částí. Motivaci a neformální vysvětlení k této úpravě algoritmu jste si mohli přečíst v učebním textu [21], kdy rozptýlená velká skupina turistů hledá nejvyšší vrchol Beskyd a její členové spolu komunikují vysílačkami omezeného dosahu. Informaci o výšce a pozici jimi dosud nalezeného vrcholu mohou sdělit jen té části skupiny, která je v dosahu jejich vysílačky. Zde si ukážeme tzv. statickou topologii zvanou *local best (lbest)*, kdy jedinci jsou rozděleni do skupin, tyto skupiny se během celého prohledávání nemění a sociální složka rychlosti jedince není ovlivňována bodem `gbest`, ale bodem `lbest`, což je nejlepší bod nalezený skupinou, do které daný jedinec patří. Rychlost jedince v etapě $t+1$ je pak určena vztahem

$$\boldsymbol{v}_i(t+1) = \chi\left[\boldsymbol{v}_i(t) + \varphi_1 \boldsymbol{u_1} \otimes (\boldsymbol{p}_i - \boldsymbol{x}_i) + \varphi_2 \boldsymbol{u_2} \otimes (\boldsymbol{l_j} - \boldsymbol{x}_i)\right], \tag{7.4}$$

kde $\boldsymbol{l_j}$ je `lbest` j-té skupiny, tj. souřadnice nejlepšího bodu dosud nalezeného skupinou j, do níž patří jedinec i.

Ukážeme si jednu z možností, jak můžeme implementovat statickou topologii *lbest*. Pro zadanou velikost hejna N a velikost skupin k určíme počet skupin $s = \lfloor N/k \rfloor$, kde symbol $\lfloor \cdot \rfloor$ značí celou část. Pokud N je dělitelné k beze zbytku, jsou všechny skupiny stejně velké, jinak poslední skupina je větší. Na začátku jsou jedinci do skupin přiděleni náhodně (jelikož souřadnice jedinců hejna jsou inicializovány náhodně, pak přiřazení prvních k jedinců do první skupiny atd. znamená náhodné rozdělení skupin). Ze zkušenosti vyplývá, že v průběhu procesu hledání se většinou skupiny shluknou tak, aby jedinci z téže skupiny si byli i geometricky blízcí. Pokud bychom v tomto algoritmu zadali $k = N$ (přísně vzato, stačí, aby $k > N/2$), pak by se topologie změnila na `gbest`.

Zdrojový kód algoritmu PSO s koeficientem sevření a se statickou topologií `lbest` je uveden v následujícím výpisu:

```
function [xmin, fmin, func_evals, evals_near]=...
    pso_lbest(fn_name, a, b, N, fi1, fi2, kappa, k,...
    my_eps, max_evals, shift, fnear)
%
%  k je velikost skupin, pokud k=N, pak je to gbest
%
pocet_skupin=fix(N/k);
% skupiny velikosti k, posledni skupina muze byt vetsi
prislusnost=zeros(N,1);
for ii=1:pocet_skupin-1
    skup=ii+zeros(k,1);
```

```
    prislusnost((ii-1)*k +1:(ii-1)*k+k) = skup;
end
prislusnost((pocet_skupin-1)*k +1:end) = pocet_skupin;
d=length(a);
P=zeros(N,d+1);
v=zeros(N,d);
vmax=(b-a)/6; % pro generovani pocatecnich rychlosti
for i=1:N
    P(i,1:d)=a+(b-a).*rand(1,d);
    P(i,d+1)=feval(fn_name,P(i,1:d)-shift);
    v(i,:)= -vmax + 2*vmax.*rand(1,d);
end  % 0-th generation initialized
pbest=P; % nejlepsi pozice jedincu na pocatku = pocatecni pozice
lbest=zeros(pocet_skupin,d+1);
for ii=1:pocet_skupin
    ind_skup= prislusnost==ii;
    pom=P(ind_skup,:);
    [tmp, ind_lmin]=min(pom(:,d+1));
    lbest(ii,:)=pom(ind_lmin,:);
end
fmax=max(P(:,d+1));
fmin=min(P(:,d+1));
func_evals=N; evals_near=0;
fi=fi1+fi2;
chi=2*kappa / (fi-2+sqrt(fi*(fi-4)));
% chi je  koef. sevreni (constriction coeff.)
while (fmax-fmin > my_eps) && (func_evals < d*max_evals)
    for i=1:N  %  in each generation
        x=P(i,1:d); % pozice i-teho jedince
        v(i,:)=chi*(v(i,:)+fi1*rand(1,d).*(pbest(i,1:d)-x)...
            +fi2*rand(1,d).*(lbest(prislusnost(i),1:d)-x));
        y = x + v(i,:);  % nova pozice
        y=zrcad(y,a,b); % zabranuje opustit definicni obor
        fy=feval(fn_name,y-shift);
        P(i,:)=[y,fy];  % zapis novou pozici a funkcni hodnotu
        func_evals=func_evals+1;
        if fy < pbest(i,d+1)  % aktualizace pbest
            pbest(i,:)= [y,fy];
            % prepis  pozici a funkcni hodnotu pbest
        end
```

```
        if fy < lbest(prislusnost(i),(d+1))  % aktualizace gbest
            lbest(prislusnost(i),:)=[y, fy];
        end
    end % end of generation
    fmax=max(P(:,d+1));
    fmin=min(P(:,d+1));
    if evals_near==0 && fmin<fnear
        evals_near=func_evals
    end
end   % main loop - end
[fmin, indmin]=min(P(:,d+1));
xmin=P(indmin,1:d);
```

Intuice nám možná napovídá, že správnější by bylo jedince ve skupinách v průběhu hledání přeskupovat tak, aby v každé skupině byli vždy jen jedinci navzájem nejbližší (aby byla celá skupina v dosahu signálu , ale jiné skupiny už signál nemohly přijímat, viz [21]). To by znamenalo neužívat statickou, ale dynamickou topologii. Takové přeskupování je však výpočetně velmi náročné. Optimální přeskupování je NP-obtížný problém a i jeho heuristické řešení mnohokrát opakované v průběhu hledání představuje většinou neúnosnou časovou zátěž. Proto je použití dynamické topologie v implementacích algoritmu PSO velmi omezeno.

Obě varianty algoritmu PSO, jejichž zdrojový kód v Matlabu jsme uvedli, byly experimentálně porovnány na šesti testovacích funkcích uvedených v kap. 3.2, dimenze problému byla $d = 2$. Pro každý algoritmus a problém bylo provedeno 10 opakování. V obou algoritmech byla velikost hejna zvolena $N = 20$, podmínka ukončení definovaná vstupními parametry `max_evals` $= 20000$, `my_eps` $= 1e-4$ a řídicí parametry `fi1` $= 2.05$ a `fi2` $= 2.05$. V algoritmu `pso1` bylo zadáno `omega` $= 0.8$, v algoritmu `pso_lbest` byly řídicí parametry nastaveny na `kappa` $= 1$ a velikost skupin `k` $= 5$ (hejno bylo rozděleno do čtyř stejně velkých skupin). Dále byla sledována nejen celková časová náročnost daná počtem vyhodnocení funkce *nfe* potřebných k dosažení podmínky ukončení, ale i počet vyhodnocení funkce potřebné k tomu, aby se nalezené minimum přiblížilo k hodnotě v globálním minimu testované funkce tak, že se liší o méně než 1×10^{-4}. Tento počet vyhodnocení je v tabulce označen jako *nfenear* a je to průměrná hodnota časové náročnosti úspěšných řešení. Výsledky porovnání jsou v následující tabulce, počty vyhodnocení funkce *nfe* jsou průměry z 10 opakování.

	PSO1				PSO_lbest			
fce	nfe	R	nfenear	std	nfe	R	nfenear	std
ackley	40000	7	15323	11160	32886	10	2260	1176
dejong1	40000	9	1571	722	24776	10	824	927
griewank	40000	3	19093	11498	40000	5	6716	8182
rastrig	40000	5	6700	2420	40000	7	1777	788
rosen	40000	9	9584	8516	15548	10	2952	1325
schwefel0	40000	5	8396	3496	40000	7	2946	3098

Vidíme, že algoritmus `pso_lbest` byl v hledání globálního minima těchto funkcí úspěšnější. Zatímco `pso1` ani jednou neukončil běh jinak než dosažením maximálního zadaného počtu vyhodnocení funkce (40000), `pso_lbest` ve třech ze šesti řešených problémů ukončil úspěšně hledání před dosažením maximálního zadaného počtu vyhodnocení funkce. Navíc, v těchto třech problémech nalezl vždy přijatelnou aproximaci globálního minima, ve sloupci R je zaznamenám počet takových úspěšných řešení. Rovněž vidíme, že u všech testovaných funkcí algoritmus `pso_lbest` byl v nalezení dobrého řešení úspěšnější častěji než `pso1` (srovnej sloupce R) a pokud správné řešení nalezl, tak to bylo s menší časovou náročností (srovnej sloupce *nfenear*), navíc mají tyto hodnoty menší variabilitu než u `pso1`, jak vidíme ze srovnání hodnot jejich směrodatných odchylek ve sloupcích označených *std*.

Z výsledků tohoto srovnání by mohl vzniknout dojem, že i tak jednoduché verze algoritmu PSO jsou schopny řešit obtížné problémy globální optimalizace s vysokou úspěšností. Bohužel tomu tak není. Vyzkoušejte si tyto algoritmy na stejných problémech, ale s větší dimenzí, třeba $d = 5$ nebo $d = 10$ a zjistíte, že úspěšnost bude podstatně nižší. Podobně nepříznivě pro PSO dopadne i srovnání s výsledky jednoduchých evolučních algoritmů v kapitole 6.4. Navzdory této skutečnosti je algoritmus PSO spolu s diferenciální evolucí považován za velmi nadějnou heuristiku pro řešení problémů globální optimalizace a je v posledních létech intenzivně studován. Byly navrženy jeho mnohé sofistikované varianty, s některými se můžeme seznámit např. v monografii [6], další se průběžně objevují v odborných časopisech.

Shrnutí:



- Principy algoritmu PSO
- Setrvačná, kognitivní a sociální složka rychlosti
- Koeficient sevření, topologie `gbest` a `lbest`

Kontrolní otázky:



1. V čem se liší topologie `gbest` a `lbest`? Porovnejte výhody a nevýhody těchto strategií při aplikacích algoritmu PSO.
2. Porovnejte experimentálně na šesti testovacích funkcích z kap. 3.2 pro $d = 2$ a dvě různé velikosti skupin v algoritmu pso_lbest.

7.2 Algoritmus SOMA



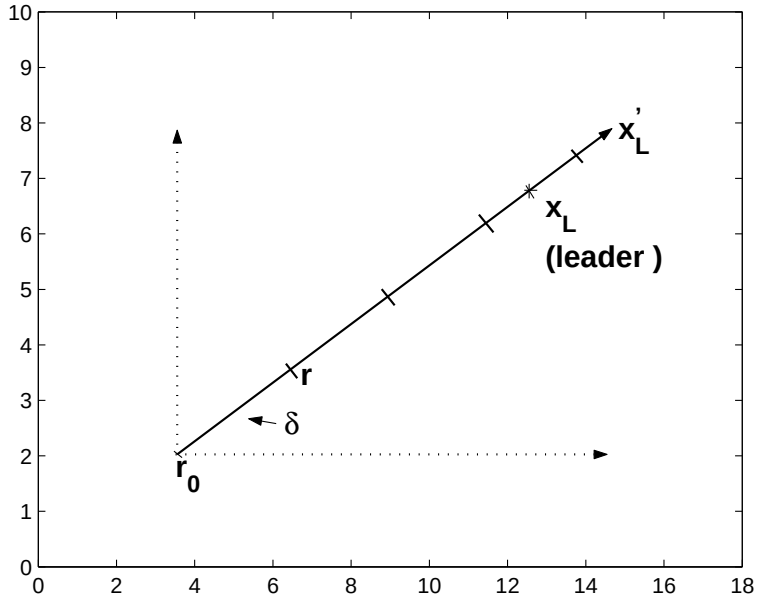
Průvodce studiem:

V této kapitole se stručně seznámíte s algoritmem SOMA, který modeluje chování smečky pronásledující kořist. Počítejte asi se třemi až čtyřmi hodinami studia.

Algoritmus SOMA v roce 2000 navrhli Zelinka a Lampinen, takže jedním z jeho autorů je náš krajan. SOMA je zkratka poměrně dlouhého jména algoritmu (Self-Organizing Migration Algorithm), které však vystihuje jeho základní principy. Na algoritmus můžeme nahlížet jako na jednoduchý model lovící smečky. Ve smečce je N jedinců a ti se pohybují (migrují) po území D tak, že každý jedinec v každém migračním kole doskáče přímým směrem k vůdci smečky, prověří místo každého doskoku na této cestě a zapamatuje si nejlepší jím nalezené místo do dalšího migračního kola. Vůdcem je většinou jedinec s nejlepší hodnotou účelové funkce v celé smečce a rovněž kvalita míst (bodů v D) navštívených jedincem při jeho skocích za vůdcem se posuzuje hodnotou účelové funkce.

Pohyb migrujícího jedince nemusí probíhat ve všech dimenzích prostoru D, ale jen v některých dimenzích, tedy v podprostoru s dimenzí menší než d. Migraci jedince k vůdci ukazuje následující obrázek. Jedinec z výchozího bodu $\boldsymbol{r}_0$ skáče skoky velikosti $\boldsymbol{\delta}$ směrem k vůdci, případně ho může i o trochu přeskočit. Pohyb jedince je buď ve směru naznačeném plnou šipkou, tedy v prostoru dimenze d nebo jen v prodprostoru menší dimenze, tedy v některém ze směrů vyznačených tečkovaně.

Kromě vstupních parametrů obvyklých u všech stochastických algoritmů, tj. specifikace problému (funkce, prohledávaný prostor, podmínka ukončení) a velikosti smečky (populace) N má algoritmus SOMA ještě tři vstupní parametry definující způsob pohybu jedince ve směru za vůdcem.



Pohyb jedince je řízen těmito vstupními parametry (jejich označení ponecháme shodné s tím, které užívá ve své knize Zelinka [29] a jak se také často označují v jiných publikacích):

- `mass` – relativní velikost přeskočení vůdce

$$\texttt{mass} = \frac{\| \boldsymbol{x}_l' - \boldsymbol{r}_0 \|}{\| \boldsymbol{x}_l - \boldsymbol{r}_0 \|},$$

  kde $\boldsymbol{r}_0$ je výchozí pozice jedince v daném migračním kole, $\boldsymbol{x}_l$ je pozice vůdce, $\| \boldsymbol{x}_l - \boldsymbol{r}_0 \|$ je norma vektoru $(\boldsymbol{x}_l - \boldsymbol{r}_0)$, podobně $\| \boldsymbol{x}_l' - \boldsymbol{r}_0 \|$ je norma vektoru $(\boldsymbol{x}_l' - \boldsymbol{r}_0)$. Tyto normy vektoru jsou vlastně délky úseček s koncovými body $\boldsymbol{r}_0, \boldsymbol{x}_l$, resp. $\boldsymbol{r}_0, \boldsymbol{x}_l'$. Geometrický význam je také zřejmý z obrázku. Doporučeny jsou hodnoty $\texttt{mass} \in [1.1;\ 3]$.

- `step` – určuje velikost skoků $\boldsymbol{\delta}$, tj. velikost směrového vektoru skoku podle následujícího vztahu

$$\boldsymbol{\delta} = \texttt{step} * (\boldsymbol{x}_l - \boldsymbol{r}_0),$$

  tedy čím je hodnota parametru `step` menší, tím podrobněji je prostor na cestě k vůdci prozkoumáván. Doporučené hodnoty jsou $\texttt{step} \in [0.11;\ \texttt{mass}]$, autoři algoritmu SOMA rovněž doporučují volit hodnotu parametru `step` tak, aby $1/\texttt{step}$ nebylo celé číslo, tzn. aby jedinec na své cestě za vůdcem nenavštívil zbytečně místo obsazené a prozkoumané vůdcem.

- `prt` – parametr pro určení směru pohybu jedince za vůdcem, $\texttt{prt} \in [0; 1]$. Je to pravděpodobnost výběru dimenze prostoru D, ve které bude pohyb jedince probíhat. Pro směr pohybu se vyberou náhodně ty dimenze, pro které $U_i <$

`prt`, $i = 1, 2, \ldots, d$. Hodnoty $(U_1, U_2, \ldots, U_d)$ tvoří náhodný vektor velikosti d, jehož složky jsou nezávislé náhodné veličiny rovnoměrně rozdělené na intervalu $[0,\ 1)$. Při implementaci algoritmu se tyto hodnoty generují funkcí `rand`. Pokud by žádná složka U_i nesplňovala podmínku $U_i <$ `prt` (např. při volbě `prt` $= 0$), vybere se jedna dimenze náhodně z množiny $\{1, 2, \ldots, d\}$. S podobným postupem jsme se setkali u binomického křížení v diferenciální evoluci.

Směr pohybu, tzn. dimenze, ve kterých probíhají skoky, se volí vždycky pro každé migrační kolo jedince k vůdci znovu, zatímco velikost skoků, tedy $\boldsymbol{\delta}$ je stejná pro celý běh algoritmu SOMA.

Dosud popsaný algoritmus je v literatuře označován jako varianta SOMA *all-to-one*, kdy v každém migračním kole všichni jedinci (kromě vůdce) putují za vůdcem. Implementace této varianty v Matlabu je uvedena v následujícím textu této kapitoly. Pohyb jedince směrem k vůdci je implementován funkcí `go_ahead`:

```
% find the best point y with function value fy
% on the way from r0 to leader
function  yfy = go_ahead(fn_name, r0, delta,...
                 pocet_kroku, prt, a, b, shift)
d = length(r0); tochange = find(rand(d,1) < prt); if
isempty(tochange)
    tochange = 1 + fix(d*rand(1)); % at least one is changed
end C = zeros(pocet_kroku,d+1); r = r0; for i=1:pocet_kroku
    r(tochange) = r(tochange) + delta(tochange);
    r = zrcad(r, a, b);
    fr = feval(fn_name, r - shift);
    C(i,:) = [r, fr];
end [fmin, indmin] = min(C(:,d+1));
yfy = C(indmin,:);  % row with min. fy is returned
```

Tuto funkci pak volá algoritmus SOMA, který implementován jako funkce `soma_to1`, jejíž zdrojový text následuje na další stránce.

```
% SOMA all to 1
function [x_star,fn_star,func_evals, evals_near] =...
    soma_to1(fn_name, a, b,...
    N,my_eps,max_evals,mass,step,prt,fnear, shift)
%
%   fn_name     function to be minized (M file)
%   a, b        row vectors, limits of search space, a < b
%   N           size of population
%   my_eps      small positive value for stopping criterion
%   max_evals   max. evals per one dimension of search space
%   mass,step,prt        tunnig parameters of SOMA
%
d=length(a);
P=zeros(N,d+1);
for i=1:N
    P(i,1:d)=a+(b-a).*rand(1,d);
    P(i,d+1)= feval(fn_name,P(i,1:d) - shift);
end  % 0-th generation initialized
P=sortrows(P,d+1);
pocet_kroku=fix(mass/step);
func_evals=N;
evals_near=0;
while ((P(N,d+1) - P(1,d+1)) > my_eps) ...
        && (func_evals < d*max_evals) % main loop
    leader=P(1,1:d);
    for i=2:N
        r0=P(i,1:d); % starting position of individual
        delta = (leader - r0)*step; % delta is vector (1,d)
        yfy = go_ahead(fn_name, r0, delta, pocet_kroku,...
            prt, a, b, shift);
        func_evals = func_evals + pocet_kroku;
        if yfy(d+1) < P(i,d+1)  % trial point y is good
            P(i,:) =  yfy;
        end
    end
    P=sortrows(P,d+1);  % fmin is in the 1st row
    if evals_near == 0 && P(1,d+1) < fnear
        evals_near=func_evals;
    end
end   % main loop - end
```

```
x_star = P(1,1:d);
fn_star = P(1,d+1);
```

Dále jsou v literatuře zmiňovány další dvě varianty algoritmu SOMA:

- *all-to-all* – v každém migračním kole se vůdcem stávají postupně všichni jedinci a ostatní putují za nimi. Populace se aktualizuje nově nalezenými body až po skončení migračního kola, tzn. po dokončení putování všech ke všem. Oproti předchozí variantě je prohledávání prostoru důkladnější, ale časově náročnější, neboť nejsou tolik využity znalosti získané z předcházejícího hledání o poloze bodu s nejmenší funkční hodnotou ve smečce.
- *all-to-all-adaptive* – podobně jako u předchozí varianty v každém migračním kole se vůdcem stávají postupně všichni jedinci a ostatní putují za nimi, ale jedinec v populaci je nahrazen bezprostředně po ukončení jeho migrace za vůdcem, pokud při této migraci byl nalezen bod s lepší funkční hodnotou, takže následující jedinci v průběhu tohoto migračního kola už putují k tomuto novému vůdci.



Příklad 7.1 Jako v předchozí kapitole, i zde uvedeme experimentální porovnání algoritmu SOMA při dvou různých hodnotách řídicího parametru `prt` na stejných šesti testovacích funkcích uvedených v kap. 3.2, dimenze problému opět byla $d = 2$. Ostatní řídicí parametry algoritmu SOMA byly pro obě varianty shodné, `mass = 2` a `step = 0.11`. Pro každou variantu a problém bylo provedeno 10 opakování. V obou variantách byla velikost smečky zvolena $N = 20$, podmínka ukončení definovaná vstupními parametry `max_evals`= 20000 a `my_eps`= $1e - 4$. Podobně jako v předchozí kapitole byla sledována nejen celková časová náročnost daná počtem vyhodnocení funkce *nfe* potřebných k dosažení podmínky ukončení, ale i počet vyhodnocení funkce potřebné k tomu, aby se nalezené minimum funkce přiblížilo hodnotě v globálním minimu testované funkce tak, že se liší o méně než 1×10^{-4}. Tento počet vyhodnocení je v tabulce označen jako *nfenear* a je to průměrná hodnota časové náročnosti úspěšných řešení, směrodatné odchylky jsou ve sloupcích označných *std*. Výsledky porovnání jsou v následující tabulce, počty vyhodnocení funkce *nfe* jsou průměry z 10 opakování.

	prt = 0.5				prt = 1			
fce	nfe	R	nfenear	std	nfe	R	nfenear	std
ackley	3987	10	2380	340	5697	2	2414	967
dejong1	2756	10	1183	177	1525	10	1217	242
griewank	7202	10	2619	825	6689	1	2072	
rastrig	3474	10	2004	270	12366	3	1730	592
rosen	6586	10	2072	773	4740	5	1798	1065
schwefel0	3508	8	1987	242	5253	2	5663	1209

Vidíme, že varianta s prt = 0.5 byla v hledání globálního minima těchto funkcí spolehlivější (srovnej sloupce R) a navíc, v polovině testovaných funkcí i rychlejší.

Shrnutí:



- Model pohybu jedince za vůdcem a parametry `mass`, `step`, `prt`
- Varianta `all-to-one`
- Varianta `all-to-all`
- Varianta `all-to-all-adaptive`

Kontrolní otázky:



1. Proč má být parametr `step` volen tak, aby 1/`step` nebylo celé číslo?
2. Jak se liší varianta `all-to-all` od `all-to-all-adaptive`?

8 Adaptace pomocí soutěže strategií



Průvodce studiem:

Tato kapitola je věnována jedné z možností, jak implementovat adaptivní stochastické algoritmy, tj. algoritmy, u kterých není nutné pracně nastavovat jejich řídicí parametry podle právě řešeného problému. Počítejte asi s pěti hodinami studia.

V minulých kapitolách jsme se seznámili s několika stochastickými algoritmy a víme už, že v různých algoritmech se užívá různá strategie hledání nebo v jednom algoritmu užívajícím jednu strategii hledání může mít tato strategie různě zvolené hodnoty jejích řídicích parametrů.

Pokud v jednom stochastickém algoritmu nabídneme několik takových střídajících se strategií a mezi nimi vybíráme podle jejich dosavadní úspěšnosti hledání s tím, že preferujeme ty úspěšnější strategie, dáváme tak algoritmu možnost přednostně vybírat strategii vhodnou pro právě řešený problém.

Podrobněji vysvětlíme principy této soutěže na zobecněném algoritmu řízeného náhodného prohledávání (CRS).

generuj populaci P, tj. N bodů náhodně v D;
repeat
  najdi $\boldsymbol{x}_{\max} \in P$, $f(\boldsymbol{x}_{\max}) \geq f(\boldsymbol{x}), \quad \boldsymbol{x} \in P$;
  repeat
    užij lokální heuristiku k vygenerování nového bodu $\boldsymbol{y} \in D$;
  until $f(\boldsymbol{y}) < f(\boldsymbol{x}_{\max})$;
  $\boldsymbol{x}_{\max} := \boldsymbol{y}$;
until podmínka ukončení;

Nový bod $\boldsymbol{y}$ se v klasické variantě algoritmu CRS generuje reflexí simplexu, viz kapitola (5). Samozřejmě je však možné ke generování bodu $\boldsymbol{y}$ užít i jinou lokální heuristiku, jak už bylo zmíněno v kapitole o algoritmu CRS, např. znáhodněnou reflexi podle (5.2). Bod $\boldsymbol{y}$ však lze generovat jakoukoli jinou heuristikou. Volbou lokální heuristiky volíme strategii hledání v algoritmu CRS.

Odtud už je jen krůček k užití více lokálních heuristik v rámci jednoho algoritmu CRS a tyto heuristiky střídat. Při tomto střídání budeme preferovat ty heuristiky, které byly v dosavadním průběhu hledání úspěšnější. V takovém algoritmu heuristiky soutěží o to, aby byly vybrány pro generování nového bodu. Protože dáváme přednost úspěšnějším heuristikám, je naděje, že bude vybrána vhodná strategie pro řešený problém a algoritmus bude konvergovat rychleji. Takto využíváme učení algo-

ritmu v průběhu řešení daného problému. Vybíráme přednostně tu heuristiku, která má větší fitness ohodnocovanou dosavadní úspěšností.

Mějme tedy v algoritmu k dispozici h strategií (v případě algoritmu CRS lokálních heuristik) a v každém kroku ke generování nového bodu vybíráme náhodně i-tou heuristiku s pravděpodobností $q_i,\ i = 1, 2, \ldots, h$. Na začátku všechny pravděpodobnosti nastavíme na shodné hodnoty $q_i = 1/h$. Pravděpodobnosti q_i měníme v závislosti na úspěšnosti i-té heuristiky v průběhu vyhledávacího procesu. Heuristiku považujeme za úspěšnou, když generuje takový nový bod $\boldsymbol{y}$, že $f(\boldsymbol{y}) < f(\boldsymbol{x}_{\max}))$, tj. nový bod $\boldsymbol{y}$ je zařazen do populace.



Pravděpodobnosti q_i můžeme vypočítat jako relativní četnost úspěchů i-té heuristiky. Pokud n_i je dosavadní počet úspěchů i-té heuristiky, pravděpodobnost q_i je úměrná tomuto počtu úspěchů

$$q_i = \frac{n_i + n_0}{\sum_{j=1}^{h}(n_j + n_0)}, \tag{8.1}$$

kde $n_0 > 0$ je vstupní parametr algoritmu. Nastavením $n_0 > 1$ zabezpečíme, že jeden náhodný úspěch heuristiky nevyvolá příliš velkou změnu v hodnotě q_i. Algoritmus užívající k hodnocení úspěšnosti heuristik vztah (8.1) je jednou z možných variant hodnocení fitness heuristiky.

Jinou možností, jak ohodnotit úspěšnost heuristiky, je vážit úspěšnost relativní změnou v hodnotě funkce. Váha w_i se určí jako

$$w_i = \frac{f_{\max} - \max(f(\boldsymbol{y}), f_{\min})}{f_{\max} - f_{\min}}, \tag{8.2}$$

kde $f_{\max}$ a $f_{\min}$ jsou největší, resp. nejmenší funkční hodnota v populaci. Hodnoty w_i jsou tedy v intervalu $(0,\ 1)$ a pravděpodobnost q_i se pak vyhodnotí jako

$$q_i = \frac{W_i + w_0}{\sum_{j=1}^{h}(W_j + w_0)}, \tag{8.3}$$

kde W_i je součet vah w_i v předcházejícím hledání a $w_0 > 0$ je vstupní parametr algoritmu.

Aby se zabránilo degeneraci rozdělení pravděpodobností q_i např. přílišnou preferencí heuristiky, které byla úspěšná na začátku hledání, lze zavést vstupní parametr δ a klesne-li kterákoli hodnota q_i pod hodnotu δ, jsou pravděpodobnosti výběru heuristik nastaveny na jejich počáteční hodnoty $q_i = 1/h$.

Další prostor pro tvorbu různých adaptivních variant algoritmu je volba soutěžících strategií (v případě algoritmu CRS lokálních heuristik), které jsou zařazeny mezi h soutěžících lokálních heuristik. Příklady možných lokálních heuristik jsou reflexe a

znáhodněná reflexe v simplexu popsané už v kapitole 5. Další možností jsou lokální heuristiky vycházející z diferenciální evoluce, kdy bod $\mathbf{u}$ se generuje jako v diferenciální evoluci a nový vektor $\boldsymbol{y}$ vznikne křížením vektoru $\boldsymbol{u}$ a vektoru $\boldsymbol{x}$ *náhodně* vybraného z populace.

Heuristiky inspirované evoluční strategií mohou genererovat nový bod $\boldsymbol{y}$ podle následujícího pravidla

$$y_i = x_i + U, \quad i = 1, 2, \ldots, d,$$

kde x_i je i-tá souřadnice náhodně vybraného bodu $\boldsymbol{x}$ z populace a U je náhodná veličina, $U \sim N(0, \sigma_i^2)$. Zatímco v evoluční strategii je hodnota parametru σ_i^2 adaptována v průběhu procesu vyhledávání podle tzv. pravidla 1/5 úspěchu, zde může být aktuální hodnota tohoto parametru odvozována z aktuální populace, např. jako

$$\sigma_i = k\ (\max_{\boldsymbol{x} \in P} x_i - \min_{\boldsymbol{x} \in P} x_i) + \varepsilon, \quad i = 1, 2, \ldots, d,$$

operátory max, min znamenají největší, resp. nejmenší hodnotu i-té souřadnice v aktuální populaci P, vstupní parametr $\varepsilon > 0$ (v implementaci můžeme užít např. $\varepsilon = 1 \times 10^{-4}$) zabezpečuje, že hodnota σ_i je kladná, k je vstupní řídicí parametr heuristiky, $k > 0$.

Lokální heuristikou v adaptivním algoritmu CRS může být samozřejmě také náhodné generování bodu $\boldsymbol{y}$ ze spojitého rovnoměrného rozdělení na D, které se užívá ve slepém náhodném prohledávání. Tím můžeme dokonce dosáhnout toho, že u takového algoritmu lze teoreticky dokázat, že je konvergentní (o teoretické konvergenci algoritmu viz kapitola 4).



Algoritmus se soutěžícími strategiemi je konvergentní, když

(a) mezi soutěžícími strategiemi je alespoň jedna zaručující kladnou pravděpodobnost vygenerování nového bodu $\boldsymbol{y} \in S$, kde S je libovolná otevřená podmnožina D mající Lebesgueovu míru $\lambda(S) > 0$. Takovou strategií v případě CRS se soutěžícími lokálními heuristikami je náhodné generování bodu $\boldsymbol{y}$ ze spojitého rovnoměrného rozdělení na D užívané ve slepém náhodném prohledávání.

(b) hodnoty pravděpodobnosti výběru takové strategie q_k v každém z k kroků hledání tvoří divergentní řadu ($\sum_{k=1}^{\infty} q_k = \infty$). To je zajištěno volbou vstupního parametru $\delta > 0$.

(c) nejlepší bod staré populace vždy přechází do nové populace (v algoritmu se užívá tzv. elitismus, což v CRS je dodržováno).

Možná, že ve vás text této kapitoly vzbudil dojem, že implementace algoritmů se soutěží strategií je obzvlášť obtížná. Tento dojem je ale zbytečný. Výběr strategie podle její dosavadní úspěšnosti lze snadno realizovat funkcí `ruleta`, kterou známe už

z kapitoly o genetických algoritmech. Tato funkce dostane na vstupu vektor četností dosavadní úspěšnosti jednotlivých strategií a vrátí číslo "vylosované" strategie spolu s minimální hodnotou pravděpodobnosti jednotlivých strategií. Implementace této funkce je v následujícím výpisu zdrojového textu.

```
function [hh, p_min]=ruleta(cutpoints)
% returns an integer from [1, length(cutpoints)]
% with probability proportional
% to cutpoints(i)/ summa cutpoints
h  = length(cutpoints);
ss = sum(cutpoints);
p_min = min(cutpoints)/ss;
cp = zeros(1, h);
cp(1) = cutpoints(1);
for i = 2:h
    cp(i) = cp(i-1) + cutpoints(i);
end
cp = cp/ss;
hh = 1 + fix(sum(cp<rand(1)));
```

Implementace soutěže lokálních heuristik v algoritmu CRS je naznačena následujícím fragmentem zdrojového textu, ve kterém je ukázáno volání funkce `ruleta` a výběr "vylosované" heuristiky příkazem `switch`.

```
while (fmax-fmin > my_eps)&(func_evals < d*max_evals) %main loop
    [hh p_min]=ruleta(ni);
    if p_min<delta
        ni=zeros(1,h)+n0; % reset
    end
    switch hh
        case 1
            y = lokalni heuristika1
        case 2
            y = lokalni heuristika2
            .....
        case 8
            y = lokalni heuristika8
    end
    y=zrcad(y,a,b);     % perturbation
    fy=feval(fn_name,y);
    func_evals=func_evals+1;
```

```
    if fy < fmax  % trial point y is good
        ni(hh)=ni(hh)+1;    % zmena prsti qi
        .....
    end
end   % main loop - end
```



Podobně lze snadno navrhnout a implementovat adaptivní verze diferenciální evoluce, kde bude soutěžit několik různých strategií. Strategie se mohou lišit druhem mutace, typem křížení nebo různým nastavením řídicích parametrů F a CR. Různé adaptivní verze diferenciální evoluce byly v posledních létech intenzivně zkoumány a byly navrženy verze, které se osvědčily jak na testovacích problémech, tak i v aplikacích. Důvodem pro hledání adaptivních algoritmů je především to, aby se při jejich užití usnadnilo nastavování řídicích parametrů, případně vyloučilo úplně, a přitom aby takové verze algoritmů umožňovaly řešit širokou skupinu problémů s dobrou spolehlivostí a v přijatelném čase. Samozřejmě, že existují i jiné způsoby adaptace řídicích parametrů, než je uvedený mechanismus soutěže strategií.

Připomeneme si algoritmus diferenciální evoluce:

generuj počáteční populaci P (N bodů náhodně v D)
repeat
  for $i := 1$ **to** N **do**
    vytvoř vektor $\boldsymbol{y}$ mutací a křížením
    if $f(\boldsymbol{y}) < f(\boldsymbol{x}_i)$ **then** do Q zařaď $\boldsymbol{y}$
      else do Q zařaď $\boldsymbol{x}_i$
  endfor
  $P := Q$
until podmínka ukončení

Chceme-li implementovat diferenciální evoluci se soutěží různých strategií, pak jenom potřebujeme zařídit, aby se v příkazu "vytvoř vektor $\boldsymbol{y}$ mutací a křížením" vybíralo ruletou z h nabídnutých strategií lišících se typem mutace nebo křížení nebo hodnotami parametrů F a CR. Implementovat to můžeme pomocí příkazu `switch` a musíme doplnit načítání hodnot úspěšnosti jednotlivých strategií pro výpočet hodnot pravděpodobnosti q_i analogicky, jako bylo ukázáno u algoritmu CRS.



Příklad 8.1 Význam a účinnost adaptivních algoritmů ilustrují výsledky experimentálního srovnání tří takových algoritmů na 6 testovacích funkcích z kapitoly 3.2. Dimenze problému byla pro všechny funkce $d = 10$ a pro každou funkci bylo provedeno 100 opakování. Ve všech úlohách byla stejná podmínka ukončení, hledání bylo ukončeno, když rozdíl mezi největší a nejmenší funkční hodnotou v populaci byl

menší než $1e-6$ nebo počet vyhodnocení funkce dosáhl hodnotu $20000d$. Za úspěšné hledání bylo považováno nalezení bodu s funkční hodnotou menší než $1e-4$.

Porovnávány byly tyto algoritmy:

- *jde* [5] – adaptivní diferenciální evoluce DE/*rand*/1/*bin*, ve které se hodnoty parametrů F a CR nastavují evolucí v průběhu procesu hledání.
- *b6e6rl* [26] – diferenciální evoluce, ve které se strategie hledání adaptují soutěží. Jako mutace je ve všech strategiích užita /*randrl*/1/, soutěží 6 různých dvojic nastavení hodnot parametrů F a CR pro binomické křížení a 6 různých dvojic nastavení hodnot parametrů F a CR pro exponenciální křížení.
- *crs4hc* [25] – CRS se soutěží 4 lokálních heuristik.

Jediný řídicí parametr, který je nutno zadávat, je velikost populace. Pro obě varianty diferenciální evoluce bylo $NP = 40$, pro *crs4hc* $NP = 100$. Parametry řídící adaptaci byly ve všech algoritmech ponechány na jejich implicitním nastavení doporučovaném autory.

Výsledky jsou v následující tabulce:

Algoritmus →	jde		b6e6rl		crs4hc	
Funkce↓	R	$\overline{nfe}$	R	$\overline{nfe}$	R	$\overline{nfe}$
ackley	100	16316	100	14899	100	18207
dejong1	100	8367	100	7109	100	10017
griewank	98	31897	98	26099	75	69602
rastrig	100	18664	100	14058	25	186353
rosen	100	68774	99	24081	100	18532
schwefel0	99	13296	100	11770	100	33685
průměr	99.5	26219	99.5	16336	83.3	56066

Porovnejte tyto výsledky s výsledky, které byly dosaženy jednoduchými neadaptivními evolučními algoritmy pro stejné problémy, které jsou uvedeny v kapitole 6.4. Pro pohodlnější porovnání je zde tabulka s výsledky zopakována. Vidíme, že adaptivní algoritmy ve většině problémů nacházejí dobré přiblížení globálnímu minimu s podstatně vyšší spolehlivostí a většinou i v kratším čase.

$d = 10$						
Algoritmus →	CRS		ES(μ, λ)		DE/*rand*/1/*bin*	
Funkce↓	R	$\overline{nfe}$	R	$\overline{nfe}$	R	$\overline{nfe}$
ackley	19	12230	6	32212	100	15756
dejong1	99	7911	100	26068	100	7728
griewank	19	15650	0	34072	100	37392
rastrig	1	53392	0	30188	100	30920
rosen	75	53019	0	199700	0	198601
schwefel0	0	200000	0	34024	99	13644
průměr	36	57034	18	59377	83	50673

Shrnutí:



- Adaptace algoritmu soutěží strategií hledání
- Soutěž lokálních heuristik v algoritmu CRS, pravidla pro posuzování dosavadní úspěšnosti
- Podmínky konvergence algoritmu
- Soutěž strategií v diferenciální evoluci

Kontrolní otázky:



1. Co rozumíme pojmem “lokální heuristika” v algoritmu CRS?
2. V čem se liší pravděpodobnosti výběru strategie podle vztahu (8.1) a (8.3)?
3. Jakou kombinaci strategií navrhnete, aby algoritmus CRS se soutěží lokálních heuristik splňoval podmínku konvergence?
4. Čím se mohou lišit soutěžící strategie v adaptivním algoritmu diferenciální evoluce?

9 Literatura

[1] Ali M. M., Törn A., Population set based global optimization algorithms: Some modifications and numerical studies, *Computers and Operations Research*, **31**, 2004, 1703–1725.

[2] Bäck, T., *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

[3] Bäck, T., Schwefel, H.P., An Overview of Evolutionary Algorithms for Parameter Optimization, *Evolutionary Computation* **1**, 1993, 1–23.

[4] Bäck, T., Fogel, D.B., Michalewicz, Z. (eds). *Evolutionary Computation 2 - Advanced Algorithms and Operators*, Institute of Physics Publishing, Bristol and Philadelphia, 2000.

[5] Brest, J., Greimer, S., Boškovič, B., Mernik, M., Žumer, V., Self-adapting Control Parameters in Differential Evolution: A Comparative Study on Numerical Benchmark Problems. *IEEE Transactions on Evolutionary Computation*, **10**, 2006, 646–657.

[6] Engelbrecht A. P., *Computatinal Intelligence: An Introduction.* Chichester: John Wiley & sons, 2007.

[7] Feoktistov V., *Differential Evolution in Search of Sotution.* Springer, 2006.

[8] Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, Addison Wesley, 1989.

[9] Holland J. H., *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, MI, 1975.

[10] Hynek J., *Genetické algoritmy a genetické programování*, Grada, 2008

[11] Kaelo P., Ali M. M., A numerical study of some modified differential evolution algorithms, *European J. Operational Research*, **169**, 2006, 1176–1184.

[12] Kvasnička, V., Pospíchal, J., Tiňo, P., *Evolučné algoritmy.* STU, Bratislava, 2000.

[13] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs.* Berlin, Springer Verlag, 1992.

[14] Michalewicz, Z., Fogel, D.B. *How to Solve It: Modern Heuristics*. Springer-Verlag, Berlin Heidelberg New York, 2000.

[15] Míka, S., *Matematická optimalizace*. vydavatelství ZČU, Plzeň, 1997

[16] Nelder, J.A., Mead, R., A Simplex Method for Function Minimization, *Computer J.*, **7**(1), 1964, 308–313.

[17] Price, K. V., Storn, R. and Lampinen, J., *Differential Evolution: A Practical Approach to Global Optimization*. Springer, 2005.

[18] Price, W. L., A Controlled Random Search Procedure for Global Optimization. *Computer J.*, 20, 1977, 367–370.

[19] Spall J. C., *Introduction to Stochastic Search and Optimization*, Wiley-Intersience, 2003.

[20] Storn, R., Price, K., Differential Evolution – a Simple and Efficient Heuristic for Global Optimization. *J. Global Optimization*, **11**, 1997, 341–359.

[21] Štěpnička M., Vavříčková L., *Úvod do soft computingu*, Učební text pro distanční studium, Ostravská universita, 2010

[22] Törn, A., Žilinskas A., *Global Optimization*, Lecture Notes in Computer Science, No. 350, Springer, 1989.

[23] Tvrdík, J., *Evoluční algoritmy*. učební text pro kombinované studium, Ostravská universita, 2010.
http://albert.osu.cz/tvrdik/down/vyuka.html

[24] Tvrdík, J., Pavliska V., Bujok P., *Základy modelování v MATLABU*. učební text pro distanční a kombinované studium, Ostravská universita, 2010.

[25] Tvrdík J., Křivý I., Mišík L., Adaptive Population-based Search: Application to Estimation of Nonlinear Regression Parameters, *Computational Statistics and Data Analysis* **52**, 2007, 713-724.

[26] Tvrdík J., Self-adaptive Variants of Differential Evolution with Exponential Crossover. *Analele Universitatii de Vest, Timisoara. Seria Matematica-Informatica*, **47**, 2009, 151–168. Reprint available online:
http://albert.osu.cz/tvrdik/down/global_optimization.html

[27] Wolpert, D. H., Macready, W.G., No Free Lunch Theorems for Optimization, *IEEE Transactions on Evolutionary Computation* **1** (1), 1997, 67–82.

[28] Zaharie D., Influence of crossover on the behavior of Differential Evolution Algorithms. *Applied Soft Computing*, **9**, 2009, 1126–1138.

[29] Zelinka, I., *Umělá inteligence v problémech globální optimalizace*. BEN, Praha, 2002

[30] Zelinka I., Oplatková Z. Šeda M., Ošmera P. Včelař F., *Evoluční výpočetní techniky*, BEN, 2009

WWW stránky:

http://albert.osu.cz/oukip/optimization/
Knihovna algoritmů v Matlabu a C++ k volnému využití.

http://www.mat.univie.ac.at/~neum/glopt.html
Obsáhlé stránky o globální optimalizaci a evolučních algoritmech, mnoho dalších užitečných odkazů na související témata včetně matematiky a statistiky, adresy stránek mnoha autorů článků a knih s touto problematikou atd.

http://www.cs.sandia.gov/opt/survey/main.html
přehled, odkazy na základní články z 90. let a dřívější doby, od roku 1997 nejsou tyto stránky aktualizovány