HLEDÁNÍ OPTIMALIZAČNÍ METODY NA MNOHAROZMĚRNÝCH REÁLNÝCH DOMÉNÁCH

soutěžní práce

autor:

Anna Kučerová

odborné vedení:

Ing. Ondřej Hrstka
Prof. Ing. Zdeněk Bittnar, DrSc.

Praha, listopad 2000

1 Úvodní poznámky

Optimalizační metody, zvláště pak tzv. evoluční výpočetní techniky, patří už delší dobu k nejvíce diskutovaným tématům. Genetické algoritmy, které jsou v této oblasti relativní novinkou (jejich vývoj začal před zhruba deseti nebo patnácti lety) se prosazují ve stále širším spektru optimalizačních úloh a jsou známy již konkrétní praktické aplikace.

V létě 2000 jsem byla zaměstnána na K132, abych podpořila vývoj genetických algoritmů na naší fakultě. Konkrétním impulzem k tomu byla potřeba nalezení dostatečně spolehlivého, efektivního a univerzálního algoritmu, pomocí něhož bychom mohli řešit poměrně nesourodou skupinu optimalizačních úloh, kterými se na K132 zabýváme. Šlo mj. o tyto problémy:

- identifikace parametrů materiálových modelů,
- regresní analýza,
- trénování neuronové sítě, která by se měla používat k přibližnému odhadu parametrů materiálových modelů z experimentů, viz např. [1],
- topologické optimalizace, viz např. [2],
- inženýrské úlohy, např. optimalizace železobetonových nosníků (viz [3]) a jiných konstrukcí nebo konstrukčních prvků.

V tomto textu se nemíním zabývat genetickými algoritmy jako takovými, neboť je k dispozici celá řada velmi podrobných publikací, které se tímto tématem zabývají, např. [4, 5] a další.

Řešení úloh nad reálnými doménami

V té podobě, v jaké byly genetické algoritmy navrženy a vyvinuty, pracují s binárními řetězci (tzv. chromozomy). Jak je zřejmé, veškeré výše uvedené úlohy, které nás nyní zajímají, se ale týkají oboru reálných čísel. Byly sice navrženy metody, které upravují binární genetický algoritmus tak, aby pracoval na reálné doméně, nicméně tento přístup přináší nezanedbatelné problémy. Např. je možné zkonstruovat jakési mapování binárních řetězců na reálná čísla, což jednak zdržuje výpočet a jednak to může způsobit nepříjemný jev, kdy se původně spojitá funkce přestane jako spojitá chovat. Malá změna binárního chromozomu může po namapování do reálného oboru způsobit velkou změnu onoho čísla, na něž se mapuje, a tím i velkou změnu funkční hodnoty. Zkušenosti ukazují, že tento jev velmi negativně ovlivňuje chování algoritmu. Další otázkou je volba přesnosti, která v některých případech (jako je právě zmíněné trénování neuronové sítě) nemusí být vůbec jednoduchá. Z těchto důvodů jsme považovali za nezbytné najít algoritmus, který bude přímo a přirozeným způsobem pracovat s reálnými čísly.

Má pozornost se zaměřila na nejnovější typ evolučního algoritmu, kterým je tzv. diferenciální evoluce.

Úlohy s vysokou dimenzí

Mezi uvedenými problémy jsou i úlohy s mnoharozměrným definičním oborem. Optimalizace železobetonového nosníku má 18 až 21 neznámých parametrů; neuronová síť, která by měla identifikovat parametry materiálových modelů, může mít až okolo 200 neznámých (jde o tzv. synaptické váhy, jejichž hodnoty jsou právě výsledkem trénování sítě). Pochopitelně s růstem počtu dimenzí roste i časová náročnost každé myslitelné optimalizační metody. Jedním z mých hlavních cílů bylo nalezení takové metody, pro niž by tento nárůst byl co nejméně nepříznivý, tj. nikoli horší než lineární. Tímto směrem jsem se zaměřila v první fázi tohoto výzkumu.

2 Diferenciální evoluce

Diferenciální evoluce (viz [6, 7]) vznikla jako způsob řešení Čebyševova polynomického zkušebního problému, který jejímu autorovi Kennethu Priceovi předložil Rainer Storn. Největší průlom nastal, když Kenneth Price přišel s nápadem využít při tvorbě nové generace vektor rozdílů dvou řetězců. Od této prvotní myšlenky se odvíjely nekonečné diskuze mezi autory a desítky experimentálních výpočtů vedly k řadě zajímavých zlepšení, díky nimž se diferenciální evoluce stává velice přizpůsobivým a silným nástrojem.

Tato metoda se od klasického genetického algoritmu velmi liší. Především používá jediný rekombinační operátor, který se jistým způsobem podobá operátoru křížení, známému z klasických algoritmů. Tento operátor pracuje podle následujícího schématu (uvádíme nejjednoduší případ):

1. vybereme cílový chromozom (algoritmus publikovaný R. Stornem vybírá chromozomy pořadě a křížení provede pro každého z nich),
2. zvolíme náhodně dva chromozomy, které od sebe navzájem odečteme, čímž získáme vektor diference,
3. tuto diferenci, přenásobenou jistým koeficientem, použijeme k jakémusi křížení s cílovým vektorem (její složky buď po řadě nebo na přeskáčku přičítáme ke složkám cílového vektoru – první možnost se nazývá exponenciální, druhá binomické křížení), čímž získáme tzv. trial chromozom
4. posoudíme, zda je trial chromozom lepší než původní cílový chromozom; pokud ano, zařadíme ho do populace místo něho.

Uvedený postup při cyklickém spouštění splňuje stejné základní poslání operace křížení jako v případě klasického genetického algoritmu – tím je vznik jedinců nových vlastností, kteří v sobě zároveň nesou jisté vlastnosti svých rodičů.

3 Experimentální výpočty

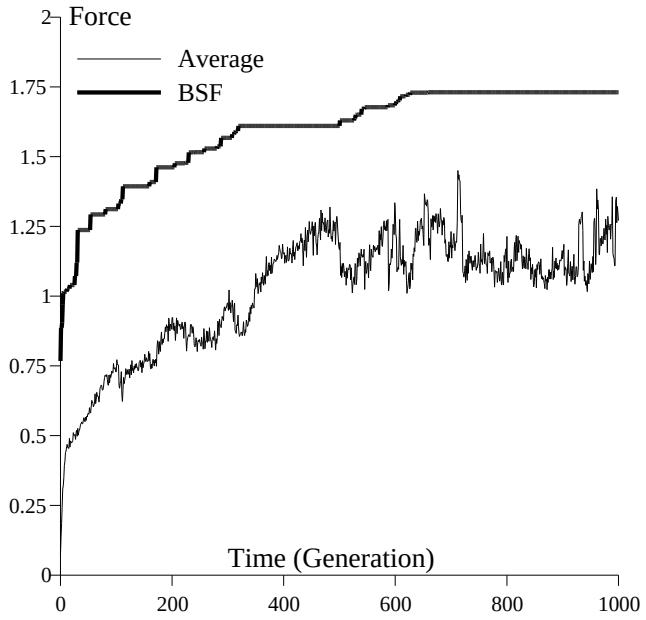
K veškerým testům jsem používala dvě testovací úlohy. Prvním z nich je problém optimalizace železobetonového nosníku, druhým je benchmarkový test, na němž jsem zkoumala chování algoritmu při vysokých dimenzích.

Terminologii z předcházející kapitoly je nutné ještě upřesnit. Konkrétních algoritmů, které lze označit pojmem diferenciálních evoluce je několik. V podstatě jde o různé modifikace uvedeného postupu, navržené pro různé typy optimalizovaných funkcí. Při obou našich testech se nejlépe osvědčila strategie, kterou R. Storn označuje jako *rand_to_best/1/exp*, hodnoty parametrů: $F = 0{,}85$ a $CR = 1$ (viz [7]).

Problém železobetonového nosníku

V roce 2000 publikoval M. Lepš práci [3], v níž se zaměřil na optimalizaci železobetonového nosníku, kterou studoval na příkladě jednoho konkrétního zatěžovacího stavu. Tato úloha patří k těm obtížnějším, protože ohodnocovací funkce je nespojitá a po částech konstantní. Algoritmus, který na řešení této úlohy autor používá, je založen na binárním genetickém algoritmu v kombinaci s metodou simulovaného žíhání (AUSA), čímž se snižuje pravděpodobnost předčasné konvergence. Dále zdokonalil tuto metodu tak, že nejprve vygeneruje deset startovacích populací, z nichž následně vybere po deseti nejlepších jedincích do další populace, kteraá se pak využije jako výchozí pro další výpočet.

Já jsem se pokusila řešit stejnou úlohu diferenciální evolucí. Jelikož je definiční obor v tomto případě diskrétní, je nezbytné souřadnice vektoru řešení před vyhodnocením síly chromozomu zaokrouhlovat a zároveň kontrolovat, zda nepřekročily hranice definičního oboru. Použitá modifikace diferenciální evoluce je sama o sobě velice jednoduchá, přesto vykázala překvapivě dobré výsledky, jak ukazuje tabulka 1.



Obrázek 1: Průběh konvergence při optimalizaci železobetonového nosníku pomocí diferenciální evoluce

V obou případech byl počet volání fitness funkce omezen na 200 000. Diferenciální evoluce ve většině případů nalézala optimum po devadesáti až sto dvaceti tisících volání. Průběh konvergence je zachycen na grafu 1. Je vidět, že algoritmus se několikrát zachytil v nějaké oblasti, kde je pravděpodobně ohodnocovací funkce konstantní, ale dokázal ji překonat sám, bez pomoci dalších technologií.

Metoda	**SGA + AUSA**	**Diferenciální evoluce**
Nejlepší výsledek	579 Kč	574 Kč
Nejhorší výsledek	660 Kč	621 Kč
Průměr	603 Kč	584 Kč

Tabulka 1: Výsledky optimalizace železobetonového nosníku

Studie růstu časové závislosti s počtem dimenzí problému

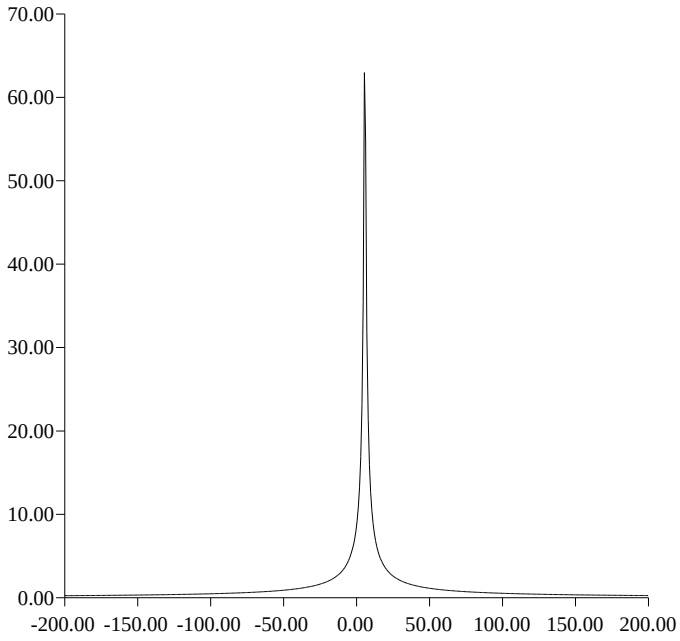
Za nejdůležitější charakteristiku evolučního algoritmu považuji to, jakým způsobem narůstá časová náročnost (obvykle udávaná počtem volání ohodnocovací funkce, potřebným k nalezení globálního optima) s růstem počtu dimenzí problému. To pochopitelně závisí zejména na tom, jaký charakter má ona funkce, jejíž optimum hledáme.

Pro své testovací výpočty (bylo by možno říci benchmarkové testy) jsme v této fázi výzkumu zvolili funkci, která má sice jediný lokální extrém, nicméně na vrcholu úzkého a vysokého peaku;

takovouto funkci označujeme jako typ 0. Její předpis uvádíme zde:

$$f(x) = y_0 \left(\frac{\pi}{2} + \arctan \frac{\|\mathbf{x} - \mathbf{x_0}\|}{r_0} \right).$$

V této rovnici představuje vektor $\mathbf{x}$ proměnnou, $\mathbf{x_0}$ je místo globálního optima (vrchol peaku), y_0 a r_0 jsou parametry, z nichž první ovlivňuje výšku a druhý šířku peaku. Graf funkce je na obrázku 2.



Obrázek 2: Příklad funkce typu 0

Jakkoli má tato funkce jediný extrém, nalezení optima s takovou přesností, jakou jsme při testech vyžadovali (rozdíl funkčních hodnot správného řešení a nalezeného nesměl být větší než 0,001), není zcela triviální záležitostí, a to z několika důvodů. Předně je funkce v okolí optima natolik strmá, že i nepatrná změna souřadnic způsobí velkou změnu funkční hodnoty; při křížení, stejně jako při mutaci může algoritmus i tehdy, posune-li některou souřadnici správným směrem, tedy k vrcholu, snadno „přestřelit“. V takové situaci je pro algoritmus skutečně velmi obtížné určit, který směr je vlastně správný. Dále je peak velmi úzký a projeví se na nepatrném rozsahu definičního oboru; tento nepoměr obrovsky roste s počtem dimenzí problému. Tento typ optimalizované funkce tedy podle našeho názoru velmi dobře prověří schopnost algoritmu adaptovat svoji rozlišovací schopnost; to je další významná výhoda diferenciální evoluce oproti binárnímu genetickému algoritmu, jehož rozlišovací schopnost je dána předem a napevno.

Výsledky uvedené v tabulce 2 ukazují, že metoda se chová velmi dobře na nízkých dimenzích, ale s dalším nárůstem počtu rozměrů se její chování rychle zhoršuje. Vyzkoušeli jsme různé strategie a různá nastavení parametrů, ale chování metody při vyšších dimenzích se zlepšit nepodařilo.[1]

Výpočet probíhal pro dimenze 1 až 40. Pro vyšší dimenze se tento algoritmus nedopočítal k cíli ani po deseti hodinách. Dokonce se mu nepodařilo identifikovat oblast, kde se peak projevuje. Lepších výsledků nedosáhla ani žádná další z modifikací, které jsou uvedeny v [6, 7].

V další fázi jsem se pokusila upravit diferenciální evoluci tak, aby se dosáhlo menšího nárůstu časových nároků s počtem dimenzí.

[1] Uvedené výsledky platí pro stejnou konfiguraci diferenciální evoluce jako při předchozím výpočtu na železobetonovém nosníku.

Dimenze	Nejlepší	Nejhorší	Průměr
1	150	360	266
2	860	1320	1113
3	2250	2940	2581
4	4120	5280	4786
5	6750	8550	7702
6	9900	13380	11780
7	14770	18830	16920
8	20240	25440	23020
9	26820	34200	30490
10	34600	43000	39340
12	52800	67680	61100
14	78120	96180	87340
16	108600	132300	120200
18	142600	169600	157500
20	179400	222600	202000
25	322500	404500	357500
30	549300	754200	653600
35	1063000	1884000	1435000
40	3323000	19740000	9426000

Tabulka 2: Nárůst časových nároků diferenciální evoluce s velikostí dimenze problému

4 Technologie SADE

Jakkoli je klasický genetický algoritmus pracující s binárními chromozomy poměrně jednoduchý, po přechodu na reálné domény jsou už konkrétní dopady rekombinačních operátorů na mapovaná čísla málo průhledné a obtížně představitelné. Totéž lze říci také o chování operátorů navržených pro diferenciální evoluci.

Je třeba říci, že toto považuji za poměrně zásadní problém. Domnívám se totiž, že transparentní a co možná nejnázornější algoritmus má řadu výhod. Především je možno různá zlepšení odvodit racionální úvahou a není třeba postupovat metodou pokusu a omylu; podobně je možno odhalit nedostatky v chování programu nebo implementační chyby. Dále lze říci, že jednoduchý algoritmus se snadno programuje, snadno realizuje, a je-li dobře navržen, bude pravděpodobně i efektivní.

Ze zkušeností vyplývá, že naprosto není nutné snažit se o to, aby navrhovaný evoluční algoritmus ortodoxně kopíroval děje probíhající v přírodě. Z toho důvodu jsem se rozhodla navrhnout algoritmus tak, aby byl co možná nejjednodušší a nejprůhlednější. Výsledkem je technologie SADE[2]. Jde o kombinaci genetického algoritmu klasické osnovy, který využívá rekombinačního operátoru křížení převzatého z diferenciální evoluce, ovšem značně zjednodušeného.

Algoritmus uvádíme zapsaný v jazyce C v podobě poněkud upravené – pro zlepšení čitelnosti – oproti našim zdrojovým textům.[3]

```
void SADE ( void )
{
    double btg,bsf=0.0 ;
    FIRST_GENERATION() ;
```

[2] Simplified Atavistic Differential Evolution

[3] Vycházím z názoru, že zdrojové texty v konkrétním programovacím jazyce (a zejména v C) jsou pro mnoho čtenářů srozumitelnější než často používané pseudoformalizované zápisy nebo prostý slovní popis. Navíc uváděné části zdrojového kódu jsou přímo použitelné, jakkoli nejsou – a nemohou – být úplné.

```
    while ( to_continue ( bsf ) )
    {
        MUTATE() ;
        BOUNDARY_MUTATE() ;
        CROSS() ;
        btg=EVALUATE_GENERATION() ;
        if ( btg>bsf ) bsf=btg ;
        SELECT() ;
    }
}
```

Prvním krokem je tedy jako obvykle vytvoření náhodné první generaci (funkce `FIRST_GENERATION`). Potom tak dlouho, dokud není splněna nějaká podmínka ukončení (funkce `to_continue`), provádíme cyklus, v němž vždy nejprve necháme vzniknout novou generaci postupným voláním funkcí `MUTATE`, `BOUNDARY_MUTATE`, `CROSS` a `SELECT`, kterou ještě ohodnotíme zavoláním funkce `EVALUATE_GENERATION`.

Samotné chromozomy budou v dalších ukázkách zdrojového kódu značeny `CH`, i-tý chromozom je tedy `CH[i]` (proměnná `CH` je typu `double **`), délka chromozomu má název `Dim`, protože je odvozena od počtu proměnných řešené funkce, tedy od dimenze prostoru jejích řešení. Další důležité proměnné jsou `SelectedSize`, `PoolSize` a `ActualSize`. Jejich význam je následující: v průběhu výpočtu dochází periodicky ke změnám aktuálního počtu „živých“ chromozomů, jejich počet v každém okamžiku výpočtu je roven `ActualSize`; na začátku cyklu je vždy tato hodnota rovna `SelectedSize`; poté genetické operátory `MUTATE` a `BOUNDARY_MUTATE` tento počet o něco zvýší a operátor `CROSS` ho dorovná do hodnoty `PoolSize`. Následně všechny nově vzniklé chromozomy ohodnotí funkce `EVALUATE_GENERATION` a na základě těchto hodnot zredukuje operace `SELECT` jejich počet na původních `SelectedSize`. Genetické operátory tedy probíhají jen na vybrané části populace. Tím není nutno zavádět tzv. pářecí rybníček, často uváděný v literatuře.

Co se týče hodnot těchto proměnných, ukazuje se toto: Jde-li především o rychlost algoritmu, je lepší volit hodnotu `SelectedSize` spíše nízkou, např. 10, naproti tomu, chceme-li se vyhnout předčasné konvergenci, pokrýt větší prostor a zvýšit pravděpodobnost nalezení globálního extrému při složitějších úlohách, je třeba tento počet zvyšovat na 200 až 500 i více, v závislosti na počtu proměnných. Hodnotu `PoolSize` běžně nastavujeme na dvojnásobek `SelectedSize`.

Funkci `to_continue` nebudu na tomto místě podrobně rozvádět. Pouze uvedu, že pracuje s proměnnou `bsf` (čili Best So Far), do níž jsme předtím uložili hodnotu dosud nejlepšího nalezeného řešení.

Dále předpokládám jakousi funkci `new_point`, která vygeneruje náhodný chromozom (vektor řešení) z definičního oboru optimalizované funkce. Operace `FIRST_GENERATION` je víceméně triviální, jen je nutné zmínit, že nejprve vygeneruje `PoolSize` chromozomů, které jsou následně ohodnoceny funkcí `EVALUATE_GENERATION` a jejichž počet je pak zredukován funkcí `SELECT` na hodnotu výchozí pro operátor `MUTATE`.

```
void FIRST_GENERATION ( void)
{
    int i ;
    for ( i=0 ; i<PoolSize ; i++ )
    {
        CH[i]=new_point() ;
    }
    ActualSize=PoolSize ;
    btg=EVALUATE_GENERATION() ;
    if ( btg>bsf ) bsf=btg ;
    SELECT() ;
```

```
}
```

Podobně i funkce `EVALUATE_GENERATION` je zcela jednoduchá. Používá volání ohodnocovací funkce `fitness`, která pro daný vektor řešení `CH[i]` vrátí jeho funkční hodnotu. Tu ukládáme do pole `Force` a průběžně vyhledáváme nejlepší hodnotu této generace, která se ukládá do proměnné `btg` (čili Best This Generation).

```
double EVALUATE_GENERATION ( void )
{
    double btg=0.0 ;
    int i ;
    for ( i=0 ; i<ActualSize ; i++ )
    {
        Force[i]=fitness( CH[i] ) ;
        if ( Force[i]>btg ) btg=Force[i] ;
    }
    return btg ;
}
```

V následujících podkapitolách se budu o něco podrobněji věnovat jednotlivým genetickým operátorům v takové podobě, jakou získaly v průběhu vývoje mého algoritmu, a pokusím se zdokumentovat jejich nynější vlastnosti.

Selekce

Funkce `SELECT` představuje jádro evolučního algoritmu. Jejím hlavním úkolem je zajistit postupné zlepšování populace, což se děje prostřednictvím redukování počtu chromozomů zároveň se zachováváním těch lepších. Výběr jedinců, kteří budou zachováni, je možné provádět různými způsoby. Dva nejzásadnější postupy jsou ruletový a turnajový výběr.

Obecný princip turnajového výběru je následující: Náhodně zvolíme dva jedince a lepší z nich určíme pro přežití. To opakujeme tak dlouho, až získáme potřebný počet jedinců. Tento postup jsem ve svém programu realizovala tak, že celou novou generaci, která čítá `PoolSize` chromozomů, redukujeme na počet `SelectedSize`: vybereme z populace náhodně dva chromozomy, porovnáme je a horšího zavrhneme – to se děje tak, že v poli, kde udržujeme všechny chromozomy, zavrženého jedince prohodíme s posledním, přičemž proměnnou `ActualSize` zmenšíme o 1.

```
void SELECT ( void )
{
    double *h ;
    int i1,i2,dead,last ;
    while ( ActualSize>SelectedSize )
    {
        i1=random_int( 0,ActualSize ) ;
        i2=random_int( 1,ActualSize ) ;
        if ( i1==i2 ) i2-- ;
        if ( Force[i1]>=Force[i2] ) dead=i2 ;
        else dead=i1 ;
        last=ActualSize-1 ;
        h=CH[last] ;
        CH[last]=CH[dead] ;
        CH[dead]=h ;
        Force[dead]=Force[last] ;
```

```
        ActualSize-- ;
    }
}
```

Turnajový výběr má oproti ostatním řadu výhod, takže se v průběhu vývoje stal nejpoužívanějším selekčním prostředkem. Jeho hlavní přednosti jsou:

- jednoduchost a výpočtová nenáročnost; např. ruletový výběr potřebuje náročný výpočet tzv. rulety,
- zachovává diverzitu populace díky tomu, že i špatný jedinec má poměrně velkou šanci na přežití; to má obvykle příznivý vliv na kvalitu konečného výsledku,
- v souvislosti s předcházejícím zároveň klesá i tendence algoritmu předčasně konvergovat do lokálního extrému.

Velkým nedostatkem tohoto obecného principu je možnost, že nepřežije nejlepší jedinec, pokud se nedostane do turnaje. Postup, který uvádím zde, odstraňuje tento problém tím, že turnajovým způsobem vybírá jedince, kteří nepřežijí; takže ať už se nejlepší jedinec turnaje zúčastní nebo ne, v každém případě přežije.

Mutace

Operátor mutace u klasického genetického algoritmu ve své obecné podobě nahrazuje u náhodně vybraného chromozomu některou jeho souřadnici jinou, náhodně zvolenou z definičního oboru. Už jistou modifikací je varianta, kdy se proces mutace neaplikuje na jednu souřadnici, ale na všechny, resp. náhodně zvolený chromozom z populace je nahrazen zcela novým, náhodně vytvořeným prvkem definičního oboru. V této podobě jsem tento operátor implementovala a dosahovala jsem dobrých výsledků.

Při pokusech s optimalizací železobetonového nosníku (tato ohodnocovací funkce má charakter, který by se dal postihnout výrazem „bradavičnatý kopec“, jsem dospěla k další modifikaci: náhodně vybraný chromozom není nahrazen novým náhodně zvoleným prvkem z definičního oboru, ale je k němu posunut o jistou část jejich vzdálenosti. Tato úprava vedla pro uvedený typ funkce ke značnému zlepšení, přičemž se na ostatních funkcích, na kterých jsem algoritmus také testovala, chování nezhoršilo. Samozřejmě, že efektivita této úpravy je přímo závislá na tom, o jak velkou část se chromozom k náhodnému prvku posune. To udává parametr `mutation_rate`.

Dalším parametrem je pravděpodobnost mutace, kterou nazýváme radioaktivita (`radioactivity`) – ta udává, kolik chromozomů v dané populaci bude mutováno. Je-li tedy radioaktivita např. 1 % a počet chromozomů, kterých se mutace může týkat, 100, pak v každé generaci bude mutován právě jeden chromozom. Je vidět, že pokud je radioaktivita takto nízká, vzniká jistý problém při malých velikostech populace. Mělo by tomu být tak, že je-li radioaktivita 1 % a velikost populace 20, pak by měl být mutován jeden chromozom co pět generací. To řešíme tak, že stanovíme maximální počet chromozomů, které mohou být mutovány v jedné generaci (v tomto případě jeden – `mutants`=1), a zredukujeme radioaktivitu pro tento jeden chromozom. Má být mutován co pět generací, upravená radioaktivita (`reduced_radioactivity`) musí tedy být 20 %.

Operátor `MUTATE` má tedy v mém algoritmu tuto podobu:

```
void MUTATE ( void )
{
    double p,x[Dim] ;
    int i,j,index ;

    for ( i=0 ; i<mutants ; i++ )
```

```
    {
        p=random_double( 0,1 ) ;
        if ( p<=reduced_radioactivity )
        {
            index=random_int( 0,SelectedSize ) ;
            x=new_point() ;
            for ( j=0 ; j<Dim ; j++ )
            {
                CH[ActualSize][j]=CH[index][j]+mutation_rate*( x[j]-CH[index][j] ) ;
            }
            ActualSize++ ;
        }
    }
}
```

Z pokusných výpočtů vyplývá optimální hodnota `radioactivity` mezi 0,5 a 5 %. Hodnoty proměnných `mutants` a `reduced_radioactivity` dopočítává algoritmus takto:

```
double proposed_mutants=radioactivity*SelectedSize ,
mutants=( int )ceil(proposed_mutants ) ,
reduced_radioactivity=proposed_mutants/( double )mutants .
```

Hodnotu parametru `mutation_rate` volíme obvykle 50 %.

Ještě je nutné dodat, což ostatně vyplývá z uvedeného zdrojového textu, že v případě mého algoritmu není vhodné používat termín „nahrazení chromozomu“. Ve skutečnosti se nově vzniklý chromozom přidá do populace, jejíž počet se tím zvýší, a jedinec, který byl při mutaci použit, má šanci dál přežít.

Hraniční mutace

Jak už bylo naznačeno, jednou z nesporných výhod turnajové selekce je zachovávání diverzity populace a snížení tendence předčasně konvergovat. V praxi to zároveň znamená celkové zpomalení konvergence, což při vysokých dimenzích problému může znamenat neúsnosné prodloužení výpočtu. Zavedla jsme proto operátor hraniční mutace, jehož smyslem je rychleji dohledávat nejbližší lepší řešení (i kdyby šlo o lokální extrém), aniž by přitom docházelo k nadměrné homogenizaci populace.

Tento operátor vybírá z populace náhodně jedince a určitou modifikací z nich vytváří nové, přičemž jejich počet závisí na parametru `radioactivity` a na parametrech z něho odvozených, stejně jako v předchozím případě u operátoru `MUTATE`. Modifikací se myslí změnit všechny souřadnice daného chromozomu o náhodnou hodnotu v zadaném rozmezí. Toto rozmezí udává parametr `mutagen`. Smyslem tohoto procesu je podrobnější prohledávání prostoru v blízkém okolí chromozomů, což má velký význam zejména ve fázi stoupání k „vrcholku“. Z toho vyplývá, že `mutagen` by se měl volit jako relativně malé číslo.

```
void BOUNDARY_MUTATE ( void )
{
    double p,dCH ;
    int i,j,index ;
    for ( i=0 ; i<mutants ; i++ )
    {
        p=random_double( 0,1 ) ;
        if ( p<=reduced_radioactivity )
        {
```

```
            index=random_int( 0,SelectedSize ) ;
            for ( j=0 ; j<Dim ; j++ )
            {
                dCH=random_double( -mutagen,mutagen ) ;
                CH[ActualSize][j]=CH[index][j]+dCH ;
            }
            ActualSize++ ;
        }
    }
}
```

Určení velikosti parametru `mutagen` je poměrně komplikovaná otázka – jednou z možností je zadávat ho relativně vzhledem k rozsahu definičního oboru, ale jako mnohem příhodnější se zdá určit ho ad hoc víceméně experimentálně. Je možné zformulovat jistá pravidla, která tuto volbu usnadňují: např. má-li některá souřadnice zadanou nebo požadovanou přesnost (rozlišovací schopnost), pak je vhodné zvolit `mutagen` právě jako tuto přesnost.

Křížení

Díky tomu, že radioaktivita bývá obecně nízká, vytvoří oba předchozí operátory obvykle jen málo nových chromozomů, typicky jeden až dva v každé generaci při `SelectedSize` = 50. Zbytek do počtu `PoolSize` pak doplní operátor křížení (`CROSS`).

Jak už jsem uvedla na začátku této kapitoly, je tento operátor založen na stejném principu jako diferenciální evoluce. Je ovšem podstatně jednodušší a je popsán v následujících bodech:

1. z populace o velikosti `SelectedSize` (populace, která nezahrnuje chromozomy vzniklé mutací) vybere náhodně dva chromozomy,
2. vypočte jejich rozdíl (v každé souřadnici) a vynásobí ho koeficientem `cross_rate`; tak získá modifikovaný vektor diference,
3. tento vektor přičte ke třetímu, rovněž náhodně vybranému chromozomu,
4. takto získaný nový chromozom přidá do populace (na rozdíl od původní diferenciální evoluce, která jím nahradí nějaký stávající chromozom).

Parametr `cross_rate` v porovnání s ostatními pravděpodobně nejvíce ovlivňuje chování algoritmu jako takového. Intuitivně je zřejmé, že chceme-li urychlit konvergenci, musíme tento parametr snižovat až řekněme na 5 %, naopak chceme-li zvýšit rozsah prohledávání prostoru a tím zlepšit pravděpodobnost, s níž algoritmus najde globální extrém mezi větším počtem lokálních, je třeba tento parametr zvyšovat. Z většího počtu experimentů vyplývá, že nejvyšší hodnota, kdy se ještě algoritmus chová smysluplně, je asi 40 % a nejlepší hodnota je v rozmezí 10 až 20 %.

```
void CROSS ( void )
{
    int i1,i2,i3,j ;
    while ( ActualSize<PoolSize )
    {
        i1=random_int( 0,SelectedSize ) ;
        i2=random_int( 1,SelectedSize ) ;
        if ( i1==i2 ) i2-- ;
        i3=random_int( 0,SelectedSize ) ;
```

```
        for ( j=0 ; j<Dim ; j++ )
        {
            CH[ActualSize][j]=CH[i3][j]+cross_rate*( CH[i2][j]-CH[i1][j] ) ;
        }
        ActualSize++ ;
    }
}
```

Tento postup křížení se podstatně liší od toho, který využívají binární algoritmy a který funguje v přírodě:

- vzniká pouze jeden nový jedinec,
- operace vždy zasahuje celý chromozom, mění všechny jeho proměnné,
- nový jedinec po svých rodičích nedědí konkrétní hodnoty některých proměnných, ale své geometrické umístění,
- v souvislosti s předchozím platí, že pokud je stanovena konkrétní oblast, ve které jsou jedinci počáteční populace rovnoměrně rozptýleni, potomek náhodných dvou jedinců hranici této oblasti s jistou pravděpodobností překročí. Maximální vzdálenost od hranice, kam až se může dostat, je rovna součinu diference mezi jeho rodiči a parametru `cross_rate`. Pravděpodobnost tohoto překročení je dána rovnicí:

  $$p = 1 - \left(1 - \frac{cr}{3}\right)^{Dim},$$

  kde cr značí parametr `cross_rate` a Dim počet proměnných neboli dimenzi problému. Z rovnice je patrné, že při vyšších dimenzích může být pravděpodobnost překročení poměrně vysoká. Např. pro $cr = 0{,}1$ a $Dim = 50$ vychází $p = 81{,}6\,\%$, pro stejnou hodnotu cr a $Dim = 200$ vychází dokonce $p = 99{,}9\,\%$. Je tedy vždy nutné si rozmyslet, jak nakládat s jedinci mimo definovanou oblast.

Testování metody SADE

Pro tento algoritmus jsme provedli stejnou studii růstu časových nároků v závislosti na počtu dimenzí, jako pro diferenciální evoluci. Parametry algoritmu jsme stanovili takto:

```
SelectedSize=10
PoolSize=20
radioactivity=5%
mutation_rate=50%
mutagen=1.0
cross_over_rate=10%
```

Výpočet probíhal pro rozsah dimenzí 1 až 200, pro každou hodnotu dimenze byl spuštěn celkem stokrát, aby se zamezilo vlivu náhodnosti.

Výsledky jsou v tabulce 3. Pro každou dimenzi problému uvádíme nejlepší, nejhorší a průměrný počet volání ohodnocovací funkce z oněch 100 spuštění[4].

Řešení jsme nacházeli s přesností 10^{-4}; rozsah definičního oboru v každém směru byl 4.10^2. V případě, že bychom s podobnou požadovanou přesností hledali řešení s použitím mapování binárních chromozomů, toto představuje rozsah prohledávaného prostoru 4.10^6 pro každou dimenzi,

[4] Jak patrno, výpočet byl spuštěn celkem 3100-krát. Takovýto test trvá na počítači s procesorem Xeon 550MHz a 1GB operační paměti něco málo přes pět hodin.

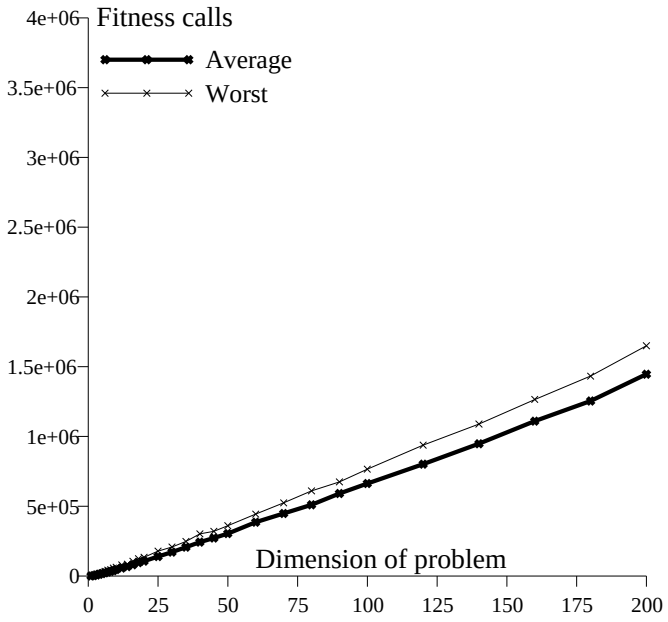
což při deseti dimenzích znamená cca 10^{66} možných řešení. Přitom průměrný počet volání funkčních hodnot při těchto dimenzích byl něco málo přes 50 000. Výpočty jsme prováděli až do dimenze 200, kde rozsah odpovídajícího binárního prohledávaného prostoru je v řádu 10^{1320}.

Dimenze problému	**Nejlepší**	**Nejhorší**	**Průměr**
1	160	2190	465
2	970	6560	3185
3	3400	13920	7453
4	6670	19450	12491
5	4800	24660	17605
6	14530	33560	23258
7	13130	42280	28964
8	13330	44790	33791
9	17600	57090	38767
10	25280	63190	46956
12	28540	79810	57190
14	42940	85320	67635
16	34080	105990	80774
18	55890	124360	96078
20	75090	134520	106695
25	69230	178580	139670
30	112180	208080	171539
35	130510	248000	206948
40	161670	302380	243026
45	110360	320410	272374
50	177340	360880	304327
60	275950	443860	385751
70	282980	524350	448214
80	207540	610230	510494
90	328340	674590	591505
100	324300	765980	663084
120	494410	937760	801670
140	598430	1089360	948197
160	644250	1265410	1110287
180	709960	1432700	1254312
200	896470	1650110	1446545

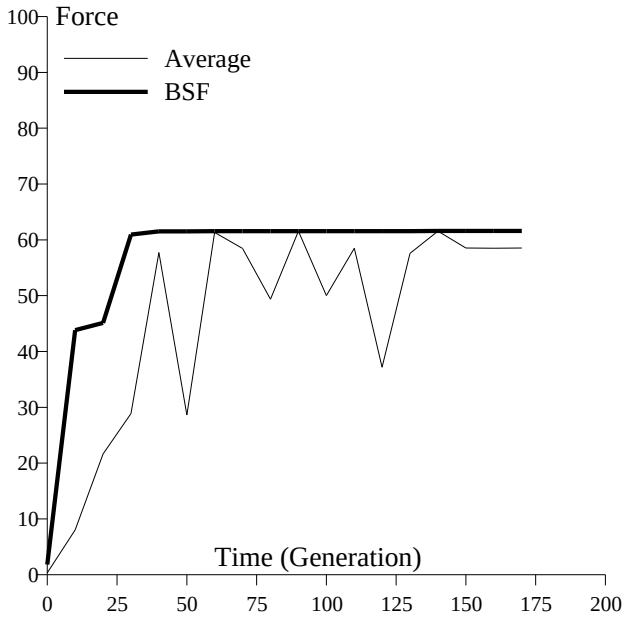
Tabulka 3: Tabulka hodnot ke grafu 3

Výsledky jsou dále vyneseny do grafu 3, kde na vodorovné ose je dimenze problému a na ose svislé je časová náročnost, vyjádřená počtem volání ohodnocovací funkce. Na grafu je vidět, že nárůst časové náročnosti algoritmu je téměř čistě lineární, jak pokud jde o průměr ze stovky testovacích výpočtů pro každou dimenzi problému, tak pokud jde o nejhorší hodnotu.

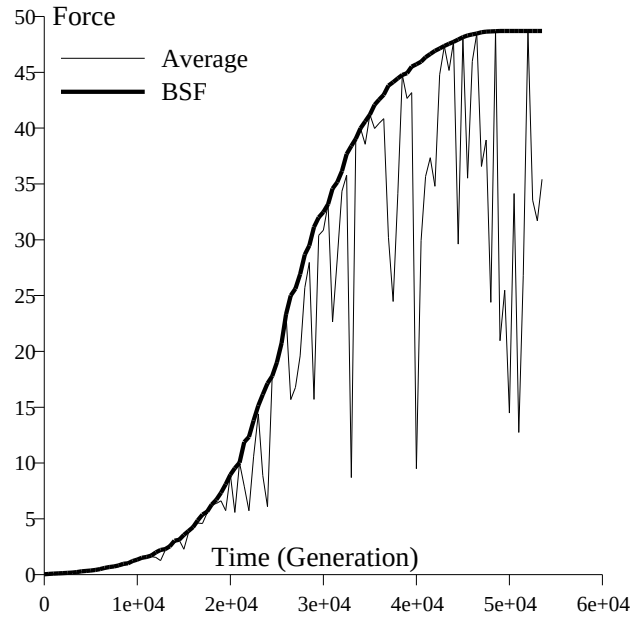
Průběh konvergence ukazují další grafy. V grafu 4 je vynesen průběh výpočtu pro dimenzi problému rovnou dvěma, v grafu 5 pro dimenzi rovnou 50. Jak je vidět, při malých dimenzích algoritmus najde peak velmi rychle a většinu času stráví dohledáváním přesného výsledku, při větších dimenzích je pak konvergence poměrně plynulá.



Obrázek 3: Nárůst časové náročnosti algoritmu v závislosti na velikosti dimenze problému



Obrázek 4: Konvergence při dimenzi problému 2



Obrázek 5: Konvergence při dimenzi problému 50

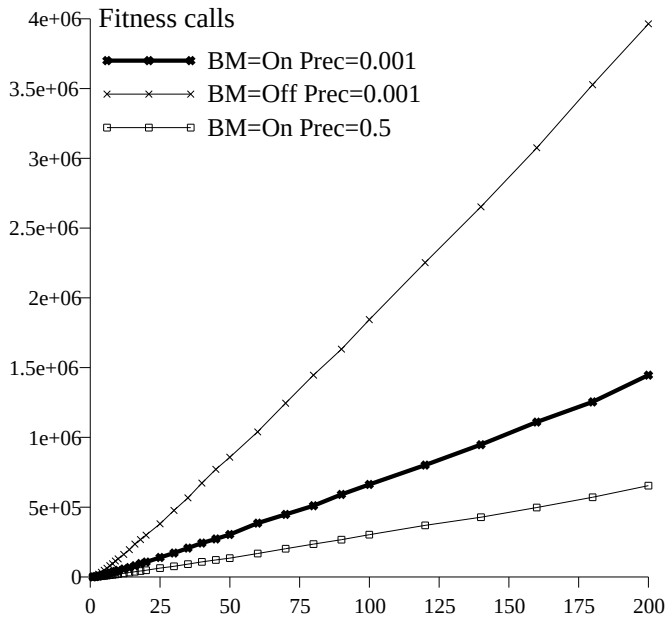
Dimenze	Průměr	Průměr	Průměr
1	465	577	196
2	3185	7315	1392
5	17605	47889	7692
10	46956	129635	20817
20	106695	299841	48182
50	304327	858272	134951
100	663084	1844576	303218
200	1446545	3963486	654503
Hraniční mutace	Zapnutá	Vypnutá	Zapnutá
Přesnost	0,001	0,001	0,5

Tabulka 4: Srovnání výsledků

Studie vlivu operátoru hraniční mutace a studie vlivu požadované přesnosti

Z celé řady dalších pokusných výpočtů uvádíme dva, které považujeme za nejzajímavější a nejlépe vypovídající o chování našeho algoritmu. Velmi zajímavý je vliv operátoru hraniční mutace. Testovací výpočet č. 2 probíhal za stejných podmínek jako v případě 1, až na to, že tento operátor byl zcela vypnutý (`mutagen=0.0`). Ukázalo se, že doba potřebná k nalezení řešení vzrostla zhruba na trojnásobek bez ohledu na dimenzi problému, nicméně nárůst je stále lineární. V testu č. 3 jsme zmírnili požadovanou přesnost z 10^{-3} na 0,5. V takovém případě klesnou časové nároky zhruba na polovinu ve srovnání s testem č. 1.

Výsledky (průměrné hodnoty) všech tří studií jsou uvedeny v grafu 6. a ve zkrácené tabulce 4.



Obrázek 6: Nárůst časové náročnosti algoritmu s počtem dimenzí problému

5 Shrnutí výsledků

Během vývoje jsem vyzkoušela celou řadu různých evolučních technologií, z nichž v této práci uvádím ty, které dosahovaly nejlepších výsledků.

Diferenciální evoluce, vyvinutá R. Stornem a K. Pricem, se ukázala jako nečekaně silná metoda pro značně komplikované úlohy s ohodnocovací funkcí nespojitou a po částech konstantní. Jak se zdá, dokáže si s těmito obtížemi poradit, aniž k tomu potřebuje pomoc zvenčí. Z různých pozorování, které jsem v průběhu výzkumu prováděla, usuzuji, že velkou výhodou diferenciální evoluce je dostatečně pomalá konvergence. Pokud ovšem průměrná vzdálenost mezi jedinci klesne pod určitou mez, je schopna jen minimálního pohybu a téměř ztrácí možnost řešení najít. Při rostoucí dimenzi problému a současně i velikostí prohledávaného prostoru se rychlost její konvergence nemění, a tak od určitých dimenzí konverguje rychleji, než by bylo třeba k nalezení extrému. Vzhledem k dimenzi problému roste její časová náročnost exponenciálně a pro úlohy o vyšších dimenzích se stává prakticky nepoužitelnou.

Ze všech metod, které jsem testovala, se zatím jedině SADE, nejjednodušší varianta diferenciální evoluce, rozšířená o operátory mutace a hraniční mutace a využívající modifikované turnajové selekce převzaté z klasických genetických algoritmů, ukazuje jako vhodná pro úlohy s velkým po-

čtem neznámých (jako je např. v úvodu zmiňovaná neuronová síť). Provedla jsem několik experimentů, které ukázaly, že bez operátoru mutace vykazuje SADE podobné chování jako diferenciální evoluce. Tento operátor zajišťuje pohyblivost populace i poté, co v důsledku konvergence vytvoří shluk a současně s operátorem hraniční mutace brání naprosté homogenizaci populace v této fázi výpočtu. Srovnání růstu časových nároků s počtem dimenzí problému pro obě uvedené metody je v tabulce 5.

Dimenze	**SADE**	**DE-Storn**
1	465	266
2	3185	1113
3	7453	2581
4	12491	4786
5	17605	7702
10	46956	39340
20	106695	202000
30	171539	653600
40	243026	9426000
100	663084	–
200	1446545	–

Tabulka 5: Srovnání růstu časových nároků metody SADE a diferenciální evoluce

Pravým důvodem, proč se SADE nedaří uspokojivým způsobem vyřešit problém železobetonového nosníku, si stále nejsem jistá. Myslím si, že určitým problémem jsou právě konstantní části ohodnocovací funkce, kde mutace nemusí zajistit dostatečnou diverzitu, a algoritmus tak ztrácí schopnost orientace a nalezení správného směru k extrému. Také se ovšem může jednat o lokální extrémy. Obě tyto možnosti se projeví stejným způsobem, a to stagnací algoritmu. Proto považuji za vhodné řešit oba problémy současně.

Na závěr se chci krátce zmínit o otázce lokálních extrémů. Domnívám se, že ideální univerzální strategie by se měla snažit co možná nejrychleji nalézt aspoň nějaké (řekněme přijatelně dobré) řešení, zapamatovat si ho a pak hledat dále nějaké lepší. Tento názor podporují i praktické ohledy: půjde-li o velmi složitý problém, máme takto mnohem větší naději, že v konečném čase získáme alespoň nějaké použitelné řešení. Tato zpráva se zabývá první částí hledání takové strategie, tedy algoritmu, který dokáže rychle hledat právě ono alespoň přijatelně dobré řešení. V současné době pokračuji na další fázi výzkumu, kterou je hledání co nejjednodušší a co nejprůhlednější strategie na překonávání lokálních extrémů.

Reference

[1] G. Yagawa and H. Okuda, *Neural networks in computational mechanics*, CIMNE, 1996

[2] B. Hassani and E. Hinton, *Homogenization and Structural Topology Optimization*, Springer, 1999

[3] M. Lepš and M. Šejnoha, *New approach to optimization of reinforced concrete beams*, CIVIL-COMP, 2000

[4] D. E. Goldberg, *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley, 1989

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

[6] R. Storn, *On the usage of differential evolution for function optimization*, NAPHIS, 1996

[7] `http://www.icsi.berkeley.edu/~storn/code.html`