Evoluční algoritmy v prostředí WebMathematica

Evolutionary algorithms in WebMathematica

Bc. Jindřich Zdráhal

Diplomová práce
2007

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav aplikované informatiky
akademický rok: 2006/2007

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jindřich ZDRÁHAL**
Studijní program:	**N 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Evoluční algoritmy v prostředí WebMathematica**

Zásady pro vypracování:

1. Vypracujte literární rešerši na téma využití evolučních algoritmů, při řešení globální optimalizace a použitých programovacích jazyků použitých na tyto algoritmy.
2. Navrhněte způsob realizace těchto návrhů s využitím programového prostředí WebMathematica.
3. Realizujte tento návrh a aplikujte evoluční algoritmy na vybrané testovací funkce.
4. Proveďte závěr celé práce.

Rozsah práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

Zelinka, Ivan. Umělá inteligence I. Volume 1. Zlín: Vutium, Brno, 1998, ISBN 80-21-1163-5
Zelinka, Ivan. Umělá inteligence v problémech globální optimalizace, Ben - technická literatura, Praha (2002)
Kvasnička V., Pospíchal J., Tiňo P., Evolučné algoritmy, STU Bratislava, 2000, ISBN 85-246-2000
Lampinen Jouni, Zelinka Ivan, New Ideas in Optimization - Mechanical Engineering Design Optimization by Differential Evolution. London: McGrawHill, 1999, ISBN 007-709506-5
Zelinka Ivan, New Optimization Techniques in Engineering, Springler-Verlag
Koza J.R. Genetic Programming, MIT Press, 1998, ISBN 0-262-11189-6
Koza J.R., Bennet F.H., Andre D., Keane M., Genetic Programming III, Morgan Kaufnamm pub., 1999, ISBN 1-55860-543-6

Vedoucí diplomové práce: **doc. Ing. Ivan Zelinka, Ph.D.**
Ústav aplikované informatiky

Datum zadání diplomové práce: **13. února 2007**

Termín odevzdání diplomové práce: **28. května 2007**

Ve Zlíně dne 13. února 2007

prof. Ing. Vladimír Vašek, CSc.
děkan

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky

doc. Ing. Ivan Zelinka, Ph.D.
ředitel ústavu

ABSTRAKT

Práce by měla přiblížit vybrané evoluční algoritmy a prostředky použité při jejich programování. Taky jsem se pokusil ukázat, jakým přínosem jsou evoluční algoritmy pro optimalizaci. Hlavním cílem mé práce je zpřístupnit tyto algoritmy všem zájemcům prostřednictvím internetu. Jsou použity tyto algoritmy: horolezecký algoritmus, simulované žíhání, rozptylové hledání.

Klíčová slova: evoluční algoritmy, horolezecký algoritmus, simulované žíhání, rozptylové hledání, optimalizace, Mathematica, WebMathematica

ABSTRACT

This work tries to introduce the chosen evolutionary algorithms and means to programming them. Then I tried to show the benefits of EA for the optimization. The main ambition of this work is to make these algorithms available for all interested people. I use these algorithms: Hill-climbing, Simulated Annealing and Scatter Search.

Keywords: evolutionary algorithms, Hill-climbing algorithm, Simulated Annealing, Scatter Search, optimization, Mathematica, WebMathematica

PODĚKOVÁNÍ

Poděkovat bych chtěl všem lidem, kteří mi pomohli s napsáním této práce a to ať už přímo nebo nepřímo. Přímo svou radou a nepřímo svou prací a jedinečnými myšlenkami a vědeckým přínosem. Jsou to jmenovitě tito lidé: Ivan Zelinka, Zuzana Oplatková, Fred Glover, Manuel Laguna, Rafael Martí, Zdeněk Vašíček a Vladimír Kvasnička. Uvádím je všechny bez titulů, není to neúcta k nim, ale všichni jsou to vědečtí pracovníci a časem jim tituly přibývají a mění se, a proto by je mohlo mrzet, že jsem použil titul, který už dávno nepoužívají.

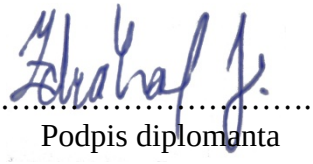
Taky bych chtěl samozřejmě poděkovat všem ostatním, kteří mi sice nepomohli s vědeckou stránkou práce, ale kteří mi pomohli jinak. Jsou to, především moji rodiče, kteří mi umožnili studovat a poskytli mi zázemí pro napsání této práce. Můj bratr, který ač není z oboru, mi poskytl několik dobrých rad a pak všichni okolo mě, že měli se mnou takovou trpělivost.

Pokud jsem na někoho zapomněl, tak se omlouvám, nebyl to úmysl.

Díky všem.

Prohlašuji, že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků, je-li to uvolněno na základě licenční smlouvy, budu uveden jako spoluautor.

Ve Zlíně 28. 5. 2007



…….
Podpis diplomanta

OBSAH

ÚVOD

Už při psaní mé bakalářské práce mě uchvátily evoluční algoritmy a tak je přirozené, že ve své diplomové práci se vracím k této problematice. Na začátku bych vás chtěl trochu uvést do problematiky, to má na starost vlastně celá teoretická část. Kapitoly zabývající se obecně optimalizací a evolučními algoritmy jsou spíše obecné a tak kdo je obeznámen s touto problematikou tak může tyto části přeskočit. Pokud by měl někdo zájem o bližší studium těchto témat, pak doporučuji projít seznam použité literatury, protože jsem z ní čerpal a spousta myšlenek je odtud převzata.

V části věnované jednotlivým algoritmům jsem uvedl jejich popis a princip fungování.

Protože nám EA psaný na papíře není k ničemu, dokud se nenaprogramuje, tak jsem se rozhodl trochu rozvést i programové prostředky použité a používané k programování EA.

V praktické části jsou rozebrány jednotlivé algoritmy, jak jsem je naprogramoval. Můžete zde taky najít simulace jaký vliv má použití jednotlivých metod a některých parametrů.

A to co je alespoň pro mě nejdůležitější a co byl hlavní cíl této práce, zde stejně nenajdete. Jsou to webové stránky a na nich použité algoritmy.

Doufám, že Vás čtení mé práce obohatí alespoň tak jako obohatilo její psaní mě.

I. TEORETICKÁ ČÁST

1 OPTIMLIZACE

Vzhledem k tomu, že se tato práce zabývá evolučními algoritmy, tak považuji za nezbytné uvést, v rámci zasazení problematiky do širších souvislostí, i něco málo o optimalizaci. Nebudeme zde zabíhat do detailů, protože optimalizace je věda, o které by se dala napsat spousta knih a mi tady na to nemáme ani prostor, ale ani není naším cílem objasňovat celou oblast optimalizace.

Optimalizace, alespoň ta část, kterou se budeme zabývat, je věda, která se snaží hledat optimální řešení, ale to už vyplývá z názvu. Je to věda, která samozřejmě vychází z požadavků nejen ostatních vědních oborů a běžného života. Spoustu optimalizačních problémů řešíme sami každý den a vůbec nás nenapadne, že by to byla nějaká věda. Tak třeba, kterou cestou je to do práce/školy nejrychlejší? Jdu na úřady vyřizovat nějaké papíry a mám toho víc, tak kam půjdu nejdřív a kam potom? Chci uspořádat večírek a chci, aby bylo všeho dost, ale stálo to co nejmíň peněz …

Toto bylo několik příkladů toho, že i v běžném životě řešíme spoustu optimalizačních problémů. Takových příkladů jistě najdete sami spoustu. Většina z vás může namítnout, že toto není důvod pro to, abychom zakládali nějakou vědní disciplínu, a že i když to pořádně nevyřešíme, tak se nic nestane. Souhlasím, že tyto příklady nejsou důvodem pro existenci optimalizace, ale co když firma řeší problém jak nejúsporněji vybudovat logistiku? Nebo jak s co nejmenšími náklady mít co největší zisk? Nebo jakým způsobem nejlépe regulovat reaktor? Jak navrhnout nejlepší motor… Tady už jistě každému dojde, že jak se začne jednat o peníze a navíc o velké peníze, tak se jistě začnou hledat cesty, jak by se to dalo udělat a jak by se to dalo udělat co nejpohodlněji a nejrychleji.

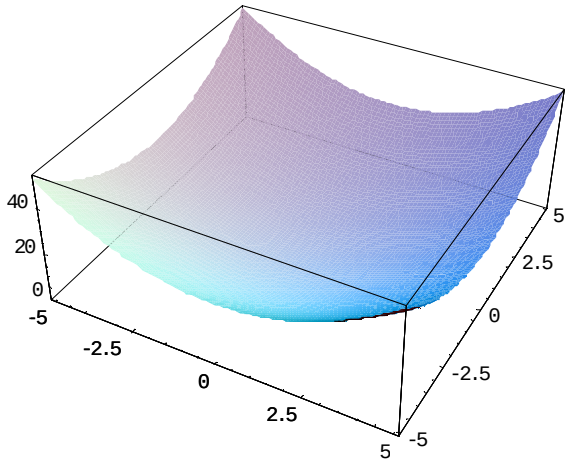
Optimalizace je matematická disciplína, která se snaží najít globální extrémy (minimum nebo maximum) dané funkce v dané množině. Toto je jedna z možných definic, na které se pokusím vysvětlit pojem optimalizace, jak ji budeme potřebovat chápat pro pochopení EA. Za pomoci optimalizace se snažíme najít optimum (globální extrém) dané funkce na dané množině, kde ale vezmeme tu množinu a funkci? To je jednoduché, převedeme reálný problém na matematický model a hned máme potřebnou funkci a množina zase reprezentuje omezení možných řešení daného problému, například, kdy už pro nás řešení nemá význam nebo je fyzikálně nerealizovatelné.

1.1 Základní pojmy

1.1.1 Účelová funkce / funkce vhodnosti

Tato funkce, na které budeme hledat optimum, se nazývá **účelová funkce** (cost function), můžeme se setkat taky s názvem **funkce vhodnosti** (fitness function). Mezi těmito dvěma názvy je určitý rozdíl. Tento rozdíl vzniká z toho, co vlastně v matematickém modelu znamená naše funkce. Jestli je to například zisk, výkon nebo přímo nějaká veličina která nás zajímá, tak hovoříme o účelové funkci. Někdy je pro nás vhodnější zavést funkci, které počítá s více hodnotícími parametry, například nás nezajímá jenom zisk, ale i kvalita. Nebo jsou pro nás některá řešení i při vysoké hodnotě účelové funkce nezajímavá nebo méně zajímavá, tak zavedeme funkci vhodnosti, které s těmito všemi kritérii počítá.

Pro naši práci je to nepodstatný rozdíl. V dalším textu se budu odvolávat na účelovou funkci. Pokud by ale byl problém jinak nadefinovaný nebo namodelovaný, tak klidně můžete používat i funkci vhodnosti.



Obr. 1. Ukázková účelová funkce: „První De Jongova funkce“

1.1.2 Hodnota účelové funkce

Dalším pro nás důležitým pojmem je **hodnota účelové funkce** (cost function value). Je to hodnota, kterou dostaneme, pokud do účelové funkce dosadíme konkrétní řešení. Všechny problémy, se kterými budeme pracovat a počítat si převedeme **na hledání minima účelové funkce,** takže nás bude zajímat pro které řešení je hodnota účelové funkce minimální.

1.1.3 Množina přípustných řešení

Takto je označována množina, kam patří všechna řešení, která nás budou zajímat. Toto množina vznikne po uplatnění všech omezení, která dostaneme z matematického modelu problému. Díky tomu dostaneme ohraničení pro naši n-rozměrnou plochu, která reprezentuje účelovou funkci.

2 OPTIMALIZAČNÍ ALGORITMY

Optimalizační algoritmy mohou být chápány jako samostatný vědní obor, ale nesmíme zapomínat, na to, že jsou to prostředky pro optimalizaci a tudíž z optimalizace vycházejí.

S rostoucí složitostí a náročností problémů nastal okamžik, kdy už nebylo možné buď vůbec, nebo alespoň v rozumné době provést analýzu a přijít na minimum analyticky. Naštěstí stejně jako se objevily složitější problémy, tak se začalo pátrat po tom, jak je řešit a jednou z odpovědí jsou právě tyto algoritmy.

Můžeme si všimnout toho, že tyto algoritmy si za svůj vzor berou přírodu a procesy v ní probíhající. Nevím, koho prvního napadlo inspirovat se přírodou při tvorbě algoritmu, dokonce věřím, že počátky těchto algoritmů měly jiné důvody než optimalizaci, ale naštěstí a díky úsilí mnoha vědeckých kapacit z nich vzniklo to, co máme dneska.

Možná vás zarazí, že tato práce má být o evolučních algoritmech a já tady píšu o optimalizačních algoritmech. Není to žádná mýlka. Jde o to, že evoluční algoritmy jsou podmnožinou optimalizačních algoritmů. Tato podmnožina není nijak ostře ohraničená, určitě se vede spousta sporů o to, které algoritmy do ní patří a které ne. Pojem „Evoluční algoritmy“ se začal používat pro všechny algoritmy, které se snaží kombinací stávajících řešení získat řešení nové. Neřeší se ale už, že předlohou není evoluce, ale třeba jiné procesy.

2.1 Dělení optimalizačních algoritmů

Optimalizační algoritmy můžeme, stejně jako cokoliv jiného, rozdělit podle několika hledisek. Já se budu držet způsobu rozdělení, které uvádí ve své knize pan Zelinka [1]. Pro toto dělení je použit způsob jakým algoritmy pracují. Nicméně některé algoritmy řadím do jiných kategorií.

- **Enumerativní**
- **Deterministické**
  - Horolezecký
  - Gradientní
  - Lokální prohledávání

- Zakázané prohledávání

- **Stochastické**
  - Náhodné prohledávání
  - Monte Carlo
  - Simulované žíhání
- **Smíšené**
  - Diferenciální evoluce
  - Genetický
  - SOMA
  - Stochastický horolezecký algoritmus
  - Rojení částic
  - Rozptylové prohledávání

2.1.1 Enumerativní

Jedná se o postup, kdy vypočteme všechna možná řešení a všechny možné kombinace daného problému a vybereme to nejvhodnější. Sami jistě uznáte, že je to možné provést pouze pro některé problémy. Důležité je, aby řešení byl konečný počet, tzn., argumenty účelové funkce musí být diskrétní. Možných řešení nesmí být velké množství. Toho dosáhneme pouze tak, že se musí omezit rozsah nabývaných hodnot u argumentů. Pokud bychom nedodrželi tyto podmínky, tak bychom na výsledek mohli čekat nekonečně dlouho, což je z ekonomického a lidského hlediska k ničemu.

Tyto algoritmy jsou vývojově nejstarší. Nehodí se na drtivou většinu v praxi řešených problémů, ale najdou své uplatnění. Hlavně v těch jednodušších problémech, kde by nasazení jiných algoritmů bylo jako použít bagr na porytí zahrádky.

2.1.2 Deterministické

Tyto algoritmy jsou postaveny na pevných základech matematiky. Jsou zde použity matematické prostředky, které při svém opětovném použití na jeden problém dají vždy stejný výsledek. Díky použití striktních matematických pravidel mají ovšem řadu omezení a nevýhod. Mezi tyto nevýhody patří zejména podmínky pro úspěšné řešení. Jsou to především tyto podmínky:

- Řešený problém je lineární

- Řešený problém je konvexní
- Prostor možných řešení je malý
- Prostor možných řešení je spojitý
- Účelová funkce je pokud možno unimodální
- Argumenty účelové funkce nemají mezi sebou jiné vazby než lineární
- Problém je definován v analytickém tvaru
- Je možno získat doplňující informace o prostoru řešení (gradient…)

Výhodou tohoto typu algoritmu je, že když je použit správně, tak nám dá jediné řešení. Naneštěstí, ne vždy lze zajistit podmínky pro úspěšnou funkci těchto algoritmů.

Za typický deterministický algoritmus považuji gradientní metodu. Jedná se o to, že ze startovního bodu je minimum hledáno za pomoci gradientů. Stejnou myšlenku mají i ostatní, například horolezecký algoritmus také jde po spádu, nevyužívá však gradient, ale přímo počítá hodnoty účelové funkce sousedních řešení. Jako největší nevýhodu těchto algoritmů uvedu možné uvíznutí v lokálním extrému. Proto je požadavek, aby účelová funkce bylo pokud možno unimodální.

2.1.3 Stochastické

Tyto algoritmy používají na rozdíl od deterministických algoritmů náhodu. Čistě stochastický algoritmus hledá řešení pouze na základě náhody. Mohli bychom říct, že zkusí jisté, předem definované, množství řešení a z nich vybere to nejlepší. Sami jistě vidíte hlavní slabiny těchto algoritmů, jsou to především časová náročnost, nutnost opakování a hlavně to, že bychom museli mít velké štěstí, abychom dostali globální extrém, nebo vlastně jakýkoliv extrém. Na druhou stranu je považuji za velký zlom v oblasti optimalizačních algoritmů díky jejich převratné myšlence použití náhody. Netroufám si odhadnout, jakým způsobem se podařilo v tak přesné vědě, jako je matematika prosadit použití náhody pro výpočet řešení. Byla to ale převratná novinka, která získala ohromné množství odpůrců, ale naštěstí se uchytila.

Možná se divíte, že mezi těmito algoritmy chybí některé, které se sem obvykle řadí, jako například stochastický horolezecký algoritmus, ale po důkladnějším zkoumání jsem jej raději zařadil do kategorie „smíšené“ abych zdůraznil i jejich deterministický charakter.

2.1.4 Smíšené

Tyto algoritmy těží ze všech předchozích skupin, čímž se snaží odstranit jejich nedostatky. Vzhledem, že jsou považovány za aktuální vrchol optimalizačních algoritmů, tak se jim to daří. Používají jak přísně matematických pravidel tak i náhodu.

Vyvinuly se ze stochastických algoritmů. Jejich vývoj byl zapříčiněn tím, že se začalo uvažovat, jak by se daly za pomoci matematických pravidel vylepšit stochastické metody.

Tyto algoritmy mají řadu výhod oproti předchozím skupinám, jsou to především tyto:

- Minimální nebo žádné požadavky na předběžné informace
- Jsou schopny pracovat nezávisle na počátečních podmínkách
- Naleznou kvalitní řešení poměrně rychle
- Nepotřebují analytické zadání problému
- Dokážou nalézt větší množství řešení

Představitelů této kategorie je mnoho, protože většina algoritmů kombinuje jak deterministickou složku, tak i složku stochastickou.

3 POUŽITÉ EVOLUČNÍ ALGORITMY

Cílem mé práce je umožnit přiblížit vybrané evoluční algoritmy, prostřednictvím internetu, širší veřejnosti zájemců. A proto považuji za nezbytné tyto algoritmy popsat a objasnit jejich principy a myšlenkové pochody vedoucí k vytvoření těchto algoritmů.

3.1 Stochastický horolezecký algoritmus

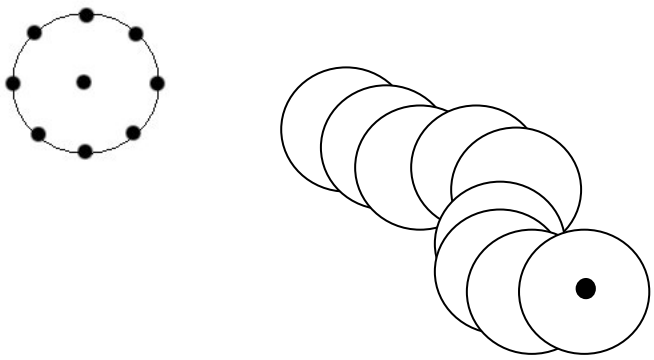
Možná jste si všimnuli, že řadím tento algoritmus mezi smíšené EA, ale vzhledem k tomu, že vychází z deterministického horolezeckého algoritmu, tak považuji za nevhodné řadit jej mezi stochastické algoritmy. I když kdybychom se na to podívali z jiného hlediska, tak by tam rozhodně patřil.

Abychom mohli správně pochopit jeho funkci, tak nejprve vysvětlím horolezecký algoritmus.

Ke vzniku tohoto algoritmu pravděpodobně vedli úlohy jako je tato:

Příklad: Turisté se ztratili v mlze v horách a ví, že v nejnižší části údolí je jejich horská chata.

Tento algoritmus udělá to samé co turisté v příkladu, prohledá okolí a pak se přesune do nejnižšího bodu ze svého okolí. To algoritmus iterativně opakuje, dokud nedosáhne určeného počtu iterací (Obr. 2). Mohli bychom říct, že se jedná a gradientní metodu bez gradientu.

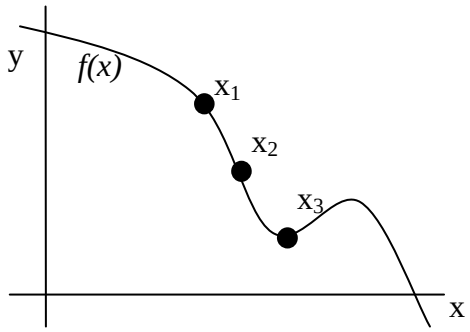


Obr. 2. Schéma tvorby okolí a prohledávání okolí
u horolezeckého algoritmu

Princip algoritmu:

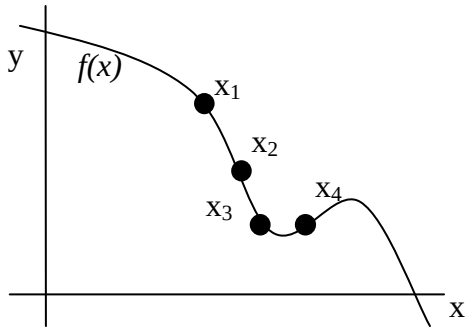
1. Vytvoř počáteční x_0, nastav $i = 0$.
2. Vytvoř okolí δx a spočítej hodnotu účelové funkce (*CV*) pro tyto body.
3. Najdi bod v okolí s nejmenší *CV*.
4. Přesuň se do tohoto bodu, nastav $i = i + 1$.
5. **If** (i = *maximální počet iterací*) **then** jdi na konec **else** jdi na 2.
6. **Konec**

Algoritmus naneštěstí může, stejně jako turisté uvíznout v lokálním minimu, jak nám to ukazuje následující obrázek (Obr. 3).



Obr. 3. Uvíznutí v lokálním extrému

Je snaha, aby k tomuto uvíznutí nedocházelo a proto se při každém kroku neposuzuje výchozí řešení, ale jenom řešení z prohledávaného okolí. Nicméně i tak může dojít k zacyklení (Obr. 4).



Obr. 4. Zacyklení

Proto se vytvořila stochastická verze horolezeckého algoritmu. Stochastičnost můžeme přidat ve dvou principech. Zaprvé bychom mohli celý algoritmus opakovat několikrát a počáteční bod náhodně generovat. Nebo můžeme použít jiné pravidla pro tvorbu okolí, kde uplatníme stochastičnost, čímž minimalizujeme riziko zacyklení.

Tato stochastičnost nám sice sníží pravděpodobnost uvíznutí v lokálním minimu, ale nemůžeme říct, že by ji s jistotou zabránila. Proto pak na principu horolezeckého algoritmu vznikly další algoritmy, které se snažili ještě víc omezit tuto šanci jako například zakázané prohledávání apod., pro nás ale nejsou důležité a tak je nerozebírám.

Tento algoritmus byl vybrán, protože se jedná o typického zástupce EA. Jeho myšlenkový princip je velmi jednoduchý, a tudíž je názorný. Navíc se na něm dá krásně demonstrovat uvíznutí v lokálním extrému.

3.2 Simulované žíhání

Tento algoritmus si bere za svou inspiraci žíhání kovů. Fyzikálně bychom tento postup mohli popsat asi takto:

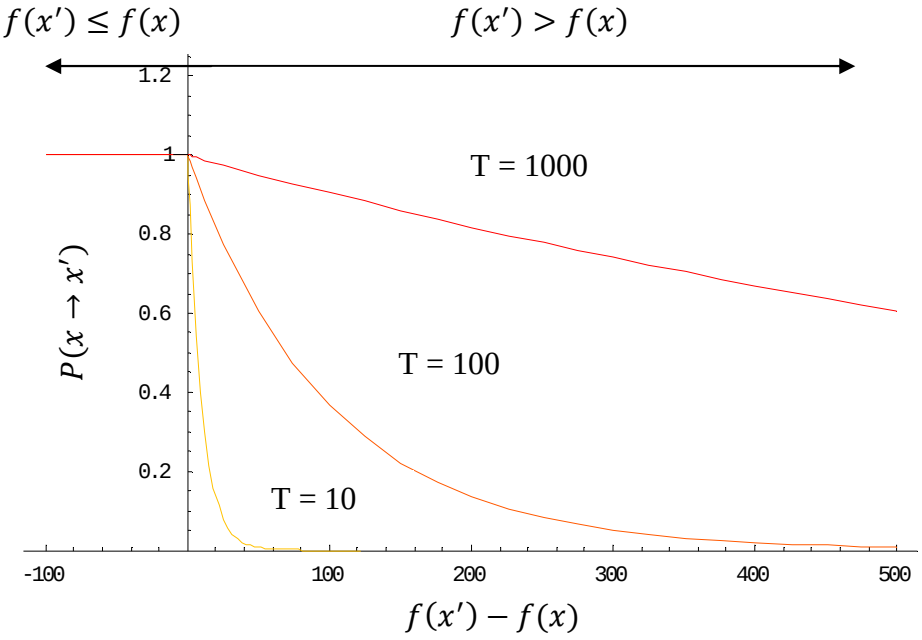
Těleso je vloženo do pece, která je zahřátá na vysokou teplotu, postupným snižováním teploty jsou odstraňovány vnitřní defekty v tělese. Při vysoké teplotě je těleso roztopeno, atomy tělesa jsou umístěny náhodně a postupným ochlazováním je jim dána možnost zaujmout stabilní polohu.

Myslím, že uvádět matematické modely tohoto fyzikálního procesu je prozatím zbytečné. Pokud by někdo měl zájem i o tyto informace, pak mu mohu doporučit knihu pana Kvasničky [2].

U simulovaného žíhání netvoříme a neprohledáváme žádné okolí, ale tvoříme nová náhodná řešení, čímž odstraňujeme problém zacyklení. Budeme uvažovat o funkci $x' = O_{por}(x)$, kde x je původní řešení, x' je nové řešení a O_{por} je stochastický operátor poruchy, teda naše funkce, která nám generuje nová řešení. Abychom neuvízly v lokálním extrému, tak simulované žíhání nám dovolí v určitých případech akceptovat jako nové řešení i řešení s horší hodnotou účelové funkce. K tomuto rozhodování se používá Metropolisova kritéria (1). Kritériu nám udává pravděpodobnost pro přijetí nového řešení oproti starému.

$$P(x \to x') = \begin{cases} 1 & pro\ f(x') \leq f(x) \\ e^{\frac{f(x')-f(x)}{T}} & pro\ f(x') > f(x) \end{cases} \quad (1)$$

Toto kritérium nám teda říká, že pokud je nové řešení lepší než to původní tak pravděpodobnost na jeho přijetí je rovna 1. A pokud je nové řešení horší, tak se pravděpodobnost spočítá jako $e^{\frac{f(x')-f(x)}{T}}$. Všimněte si, že na tuto pravděpodobnost má vliv jak to o kolik je nové řešení horší než původní, tak i teplota. Přehledně nám to může demonstrovat graf (Obr. 5.). Tady se můžeme podívat, že s klesající teplotou velmi klesá pravděpodobnost, že bude akceptováno řešení horší než původní.



Obr. 5. Metropolisovo rozložení

Operátor poruchy také můžeme nechat naprosto stochastický, nebo jej můžeme svázat s teplotou a pak také pro klesající teplotu bude klesat vzdálenost mezi původním řešením *x* a novým řešením *x'*. Toto je důležité rozhodnutí, protože pokud necháme větší váhu náhodě a neustále prohledáváme celou množinu řešení, tak sice máme menší šanci na uváznutí v lokálech, ale taky nevyužíváme plnou sílu evoluce. Na druhou stranu pokud budeme prostor prohledávaných řešení s každým ochlazením neustále zmenšovat, tak plně využíváme evoluce a při každém ochlazení zjemňujeme a zpřesňujeme naše výsledné řešení, ale taky se nám může stát, že uvázneme v lokálním extrému. Několik ukázek si můžeme pak prohlédnout v kapitole o elitismu.

Další co musíme vyřešit, je plán chlazení. Jedná se o způsob, jakým budeme snižovat teplotu. Používají se dva základní postupy. Prvním je metoda skokového ochlazování.

A druhým postupem je chlazení kontinuální. Pak se samozřejmě dá taky používat nějaký adaptivní postup, ale tím se zabývat nebudeme.

Skokové ochlazování se provádí tak, že se od aktuální teploty odečte určité číslo (*krok*) a dostaneme novou teplotu.

$$T' = T - krok \tag{2}$$

Naproti tomu postupné ochlazování využívá *multiplikátoru*.

$$T' = T \times multiplikátor \tag{3}$$

Metoda postupného ochlazování je věrnější svému fyzikálnímu originálu a obecně dává lepší výsledky než metoda skokového ochlazování.

Princip simulovaného žíhání si můžeme znázornit asi takto:

1. Inicializace algoritmu: $\boldsymbol{x} = \boldsymbol{x_0}$, $\boldsymbol{T} = \boldsymbol{T_{max}}$,
2. **While** ($\boldsymbol{T}$ >= $\boldsymbol{T_{min}}$)
   - 2.1. **For** ($\boldsymbol{i}$ = 1 **to** ***počet zkoušených řešení***)
     - 2.1.1. Vytvoříme nové řešení $\boldsymbol{x'} = \boldsymbol{O_{por}(x)}$,
     - 2.1.2. **If** ($\boldsymbol{x'} < \boldsymbol{x}$) **then** $\boldsymbol{x} = \boldsymbol{x'}$ **else**
       - 2.1.2.1. Vygenerujeme náhodné číslo $\boldsymbol{p}$ = **Rand(real, {0,1})**
       - 2.1.2.2. Vypočítáme mez metropolisova kritéria $\boldsymbol{m} = \boldsymbol{e}^{\frac{f(x')-f(x)}{T}}$
       - 2.1.2.3. **If** ($\boldsymbol{p}$ <= $\boldsymbol{m}$) then $\boldsymbol{x} = \boldsymbol{x'}$
     - 2.1.3. Zvýšíme počitadlo iterací $\boldsymbol{i} = \boldsymbol{i}$ +1
   - 2.2. Provedeme chlazení $\boldsymbol{T'}$ = chlazení($\boldsymbol{T}$)
3. **Konec**

Kroky 2.1 – 2.1.3 z předchozího postupu jsou také někdy nazývány Metropolisovým algoritmem.

Zjednodušeně tedy můžeme říct, že simulované žíhání probíhá takto:

1. Inicializace ($\boldsymbol{x} = \boldsymbol{x_0}$, $\boldsymbol{T} = \boldsymbol{T_{max}}$)
2. **I**-krát provedeme ***metropolisův algoritmus*** jako vstup vždy použijeme výstup z předchozího běhu metropolisova algoritmu.
3. Provedeme chlazení, a pokud jsme ještě nedosáhli $\boldsymbol{T} < \boldsymbol{T_{min}}$ pak jdeme znovu na bod 2.

Použití metody pro ochlazování a tvorba nového řešení se řídí podle potřeby jednotlivých řešených úloh. Díky generování nového řešení z celé množiny možných řešení omezuje šanci jak na zacyklení, tak i na uvíznutí v lokálním extrému, ale vyhnout se mu nedokážeme.

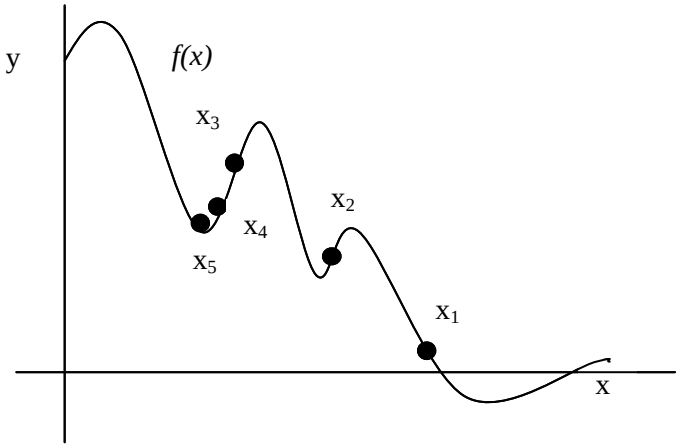
3.2.1 Elitismus

Další možnost zlepšení tohoto algoritmu můžeme spatřovat v zavedení **elitismu**. Jde o zavedení nové proměnné, ve které budeme uchovávat dosavadní nejlepší řešení. Toto nejlepší řešení pak použijeme při inicializaci metropolisova algoritmu v bodě 2 zjednodušeného postupu. Pak bychom tedy tento postup mohli napsat asi takto:

1. Inicializace algoritmu: $x_{opt} = x = x_0$, $T = T_{max}$,
2. **While** (**T** >= T_{min})
   - 2.1. Uplatnění elitismu $x = x_{opt}$
   - 2.2. **For** (i = 1 **to** ***maximální počet iterací***)
     - 2.2.1. Vytvoříme nové řešení $x' = O_{por}(x)$,
     - 2.2.2. **If** ($x' < x$) **then** $x = x'$ **else**
       - 2.2.2.1. Vygenerujeme náhodné číslo p = **Rand(real, {0,1})**
       - 2.2.2.2. Vypočítáme mez metropolisova kritéria $m = e^{\frac{f(x')-f(x)}{T}}$
       - 2.2.2.3. **If** ($p <= m$) then $x = x'$
     - 2.2.3. **If** ($x < x_{opt}$) **then** $x_{opt} = x$
     - 2.2.4. Zvýšíme počitadlo iterací $i = i + 1$
   - 2.3. Provedeme chlazení T' = chlazení(T)
3. x_{opt} prohlásíme za výsledek.

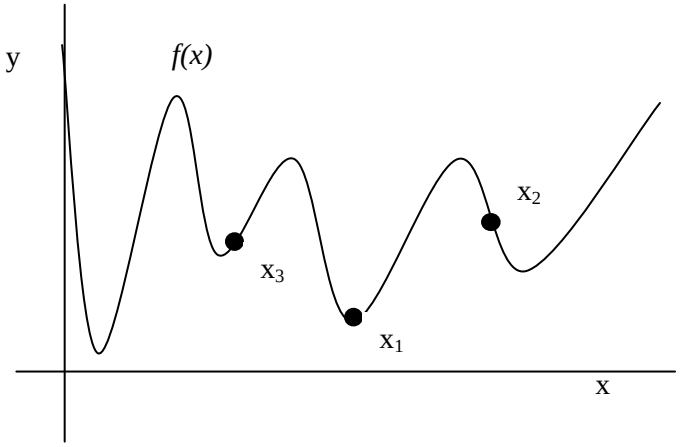
Pro další úvahy vycházejme z toho, že operátor poruchy O_{por} je svázán s teplotou a při každém ochlazení se nám zmenší oblast prohledávaných řešení.

Elitismus nám pomáhá v tom, že se nestane, abychom dostali po ochlazení a aplikaci metropolisova algoritmu řešení s větší hodnotou účelové funkce než před ochlazením. Toto se totiž může díky metropolitovu kritériu stát. Je to výhodné, že se nám nemůže stát, že by se nám řešení postupně, díky stochastičnost procesu a metropolisova kritéria zhoršovala. Dokonce je reálné, při použití algoritmu bez elitismu, že nám řešení postupně náhodně překoná i několik „energetických bariér“ a vyskáče nám do nějakého lokálního minima (Obr. 6).



Obr. 6. Možný problém při Simulovaném žíhání bez elitismu

Elitismus toto nebezpečí minimalizuje, protože při každé inicializaci metropolisova algoritmu použije nejvýhodnějšího řešení. Může nám ale způsobit jiný problém. Ten spočívá v tom, že nás elitismus může vracet do sice zatím nejvýhodnějšího, ale pro další postup ne až tak vhodného řešení. Takže se může například stát, že získáme jako nejlepší řešení x_1, po ochlazení to bude řešení x_2. Znovu ochladíme systém a pro další běh metropolisova algoritmu použijeme jako výchozí bod ten doposud nejlepší a to je x_1. Po dalším dokončení metropolisova algoritmu získáme jako nejlepší bod x_3, ochladíme systém a znovu pro další běh použijeme to nejlepší řešení a to x_1. A z lokálního minima se zase nedostaneme.



Obr. 7. Možný problém při Simulovaném žíhání s elitismem

Tyto problémy se nám samozřejmě vyhnou, pokud použijeme operátor poruchy O_{por} čistě stochastický. Řešení, která tím získáme, budou ale hrubější.

3.3 Rozptylové hledání (Scatter search)

3.3.1 Obecný popis

Dalším algoritmem, který jsem zpracovával je algoritmus rozptylové hledání (Scatter search). V předchozích algoritmech jsme sice mluvili o evoluci ve smyslu vývoj řešení, ale o evoluci ve smyslu populací a křížení jedinců zatím nemohla být ani řeč. S tím se setkáme až při popisu tohoto algoritmu.

Při popisu funkce tohoto algoritmu budu vycházet z toho, že čtenář má alespoň obecnou znalost EA, pokud tomu tak není, pak doporučuji prostudovat některou z knih uvedených v bibliografii. Pokud máte tyto znalosti nebo zkusíte, jestli to lze pochopit i bez nich, tak můžete směle číst dál.

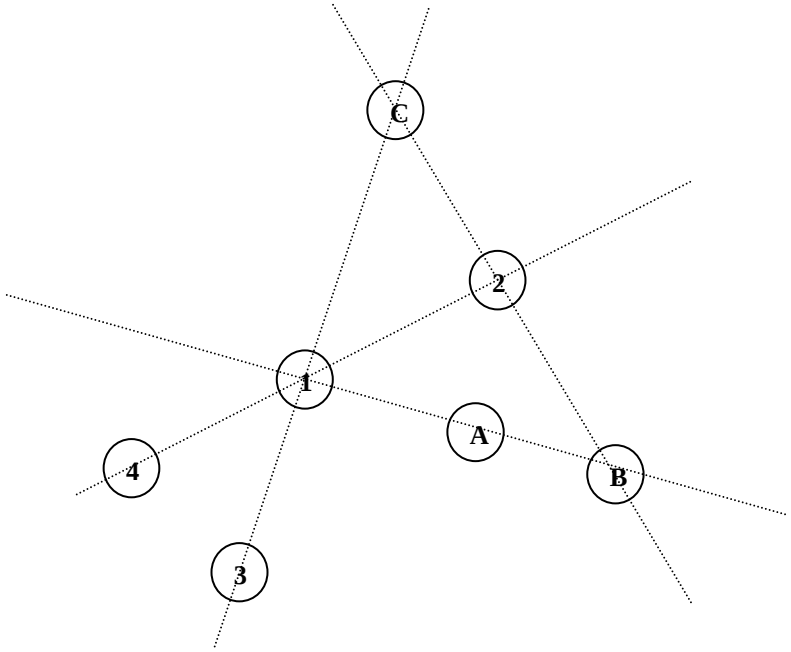
Dalo by se říct, že SS je dalším myšlenkovým krokem, který byl učiněn po vzniku algoritmů horolezeckého a z něj pak vycházejícího algoritmu zakázaného prohledávání (Tabu search). Zakázané prohledávání vychází z horolezeckého algoritmu a snaží se odstranit jeho problém s možným zacyklením. Toto řeší zavedením paměti, kam se zaznamenávají použité transformace a ty které již byly použity, jsou penalizovány a to vede k jejich postupnému vyloučení, a tudíž by nemělo dojít k zacyklení.

Hlavní myšlenkou SS algoritmu je kombinace různých způsobů řešení. Je jasné, že určité postupy nevedou ke zkvalitnění řešení, ale na druhou stranu nám odstraňují různá uváznutí v lokálních extrémech, zacyklení a podobné nehody. I bez znalosti EA můžeme říct, že kombinací různých postupů a metod řešení můžeme dosáhnout lepších výsledků, než pokud bychom používali jediné metody. Síla této myšlenky spočívá v tom, že se sestaví určité množství postupů, jakým způsobem se budou kombinovat stávající řešení. Účelnost jednotlivých pravidel se ohodnotí a tato informace se uchová ve vahách. Když pak přijdeme ke kroku, kde bychom měli získávat nová řešení, tak použijeme tyto váhy pro rozhodování, která pravidla použít.

Po úspěších s tímto postupem se začal podobný princip uplatňovat i na omezení. Začalo se používat kombinování omezení a za pomoci těchto kombinací k tvorbě nových tzv. náhradních omezení.

V SS se požívá množina řešení, se kterými se dále pracuje. V jiných principielně je tato množina shodná populací u genetických algoritmů. U tohoto algoritmu se jí neříká populace, ale referenční množina. Na rozdíl od populace používané u jiných EA je tato

referenční množina docela malá. Z této množiny jsou vybírány prvky (jednotlivá řešení) a na základě použité metody jsou určitým způsobem generována nová řešení. Pokud se použije kombinační metody založené na lineární kombinaci řešení, tak se nám může tato referenční množina vyvíjet například takto (Obr. 8).



Obr. 8. Příklad vývoje referenční množiny u SS algoritmu

Původní množina obsahuje tři řešení (A, B, C). Nejprve se zkombinují řešení A a B za vzniku řešení 1. Následuje kombinace řešení B a C za vzniku řešení 2. Samozřejmě že možných řešení bude víc, ale řekněme, že nejsou pro nás dostatečně vhodná a proto se nedostanou do referenční množiny. Další kombinací C a 1 získáme řešení 3 a kombinací 1 a 2 získáme řešení 4. Myslím, že pro ukázku, jak se lineární kombinací dvou řešení dá získat nové řešení, to stačí.

Další krása SS algoritmu spočívá v tom, že nepracuje s těmito „hrubými“ řešeními, ale použije nějakou metodu lokálního vyhledávání a získá „vylepšená“ řešení. To si můžeme zapsat jako $x \xrightarrow{zlepšení} x^*$. Jakou metodu použijeme se, můžeme rozhodnout na základě řešeného problému nebo osobních zkušeností. Já osobně jsem použil horolezecký algoritmus omezený na několik málo kroků. Můžeme ale použít libovolnou jinou metodu.

Tím ale krásné a účinné myšlenky použité při konstrukci SS algoritmu nekončí. Ještě uvedu minimálně dvě, které mi přijdou jako stěžejní a navíc si nejsem vědomý, že by se ve zvýšené míře vyskytovali u jiných algoritmů.

První z nich je způsob, jakým je generována počáteční množina řešení. V SS se objevuje velký důraz na různorodost počáteční množiny řešení. Pokud budeme brát řešení v tomto algoritmu jako vektory, pak by se dalo říct, že se snažíme, aby tyto vektory byly co nejdál od sebe. K tomu se používají následující postupy. Pro každou složku vektoru je interval přípustných možností rozdělený na 4 intervaly. Pak se za pomoci generátoru náhodných čísel vybere jeden z těchto intervalů, ten je pak penalizován a vybírá se další. Když jsou pro všechny složky všech vektorů vybrány intervaly, tak jsou z těchto intervalů vybrány konkrétní hodnoty.

A ta druhá myšlenka se objevuje při výběru, která řešení se dostanou do referenční množiny. Většina EA se zaměřuje pouze na hodnotu účelové funkce a tu pak používá jako jediné kritérium pro zařazení do referenční množiny. U SS je tomu trošku jinak. Referenční množina je rozdělena na dvě části. V jedné části se ukládají nejvhodnější řešení z hlediska hodnoty účelové funkce. To je normální a používají to všechny EA, ale zajímavostí je druhá část, kam se ukládají nejvhodnější řešení, která ale jako kritérium vhodnosti nemají hodnotu účelové funkce, ale vzdálenost od ostatních vektorů v referenční množině. Tomuto principu se říká princip různorodosti. Myšlenkou pro vznik této různorodosti je snaha o vyloučení uváznutí v lokálním extrému. I pokud by se stalo, že všechna řešení v první části referenční množiny leží ve stejném lokálním minimu, tak při kombinaci s řešením z druhé množiny se dostaneme mimo toto lokální minimum.

3.3.2 Princip fungování SS

Fungování SS se dá, stejně jako fungování ostatních EA, rozdělit do několika kroků. Nejprve tyto kroky popíšu obecně a pak se rozepíšu o tom, jak byly aplikovány u mnou programované verze tohoto algoritmu.

V následujícím popisu budu používat zkratky. Ty se dají najít i na konci práce v seznamu zkratek, ale věřím, že většina lidí by odmítla listovat na konec a pak zpět a tak je uvádím i zde.

Psize metoda	velikost startovní množiny, kterou nám vygeneruje diversifikační
P	množina zkušebních řešení
b	velikost referenční množiny
b1	velikost podmnožiny referenční množiny ve které jsou uložena nejlepší řešení
b2	velikost podmnožiny referenční množiny ve které jsou uložena různorodá řešení
MaxIter	maximální počet iterací
RefSet	referenční množina
RefSet1	podmnožina referenční množiny s nejlepšími řešeními
Refset2	podmnožina referenční množiny s nejrůznějšími řešeními

3.3.2.1 Metoda různorodého generování (diversifikační metoda)

Princip této metody byl již popsaný výše. Zopakuji to podstatné a doplním o další údaje. Důležité je rozdělení intervalu na 4 části při jejich výběru použití penalizací. Výběr konkrétních hodnot se pak z nich provádí již normálně.

Pro demonstraci toho, že dostáváme rovnoměrné rozdělení, použiji tabulku, ze simulací, které jsem prováděl pro svou bakalářskou práci. Generoval jsem čtyři proměnné v rozsahu [-10,10] tyto proměnné si můžeme představit jako čtyři složky požadovaného vektoru. Každou tuto složku jsem generoval 100krát. V tabulce je zachyceno rozdělení do jednotlivých dílčích intervalů (Tab. I).

Tabulka I. Rozložení pravděpodobností do jednotlivých intervalů

Interval	X_1	X_2	X_3	X_4
-10 až -5	26	20	26	26
-5 až 0	23	23	17	32
0 až 5	25	24	24	21
5 až 10	26	33	33	21

Díky tomu je pak rozložení jednotlivých vektorů do plochy rovnoměrnější.

3.3.2.2 Zlepšující metoda

Zlepšující metoda slouží k transformaci zkušebního řešení na jedno nebo více vylepšených řešení. Ani vstup ani výstup nemusí být řešitelný, nicméně u výstupu se očekává, že bude. Pokud se po provedení této metody nenalezne vylepšené řešení, pak se použije stejné řešení, jako bylo na vstupu. Při programování této metody jsem použil horolezeckého algoritmu. Tento algoritmus v několika málo krocích prohledá okolí a najde nejvhodnější řešení. Pak je ještě řešení zkontrolováno, jestli je vůbec v množině možných řešení, pokud jsme se náhodou dostali mimo tuto množinu, tak je tam řešení vráceno. Původně když jsem programoval tento algoritmus v Mathematice, tak jsem použil jako zlepšující metodu integrovanou funkci `Findminimum`, ale ve webové verzi to dělalo problémy. Tyto problémy byly způsobeny přetížením serveru a vypršením času pro odezvu. Z toho jsem vyvodil, že pro webové účely bude stačit jednoduché horolezecké prohledání, které navíc ukáže, že i takový jednoduchý a neúčinný algoritmus se dá smysluplně využít.

3.3.2.3 Aktualizace referenční množiny

Metoda aktualizace referenční množiny je potřeba k vytvoření a udržování referenční množiny, které se skládá z *b* nejlepších řešení, která byla nalezena (kde *b* je většinou malé číslo, málokdy větší než 20). Tato množina je organizována tak, aby poskytla ostatním metodám co nejlepší přístup k datům v ní obsaženým. Řešení se mohou dostat do této množiny pro jejich kvalitu a různorodost.

Jak jsem již psal, tato referenční množina má dvě části a to *RefSet1*, kde jsou shromažďována nejvhodnější řešení, kde kritérium vhodnosti je hodnota účelové funkce. Tato část bývá větší, ve webovém příkladu je implicitně nastaveno *b1* = 3. Druhou referenční množinou je množina *RefSet2*, kde jsou uložena vhodná řešení, jejichž vhodnost se posuzuje podle vzdálenosti od všech ostatních řešení vyskytujících se v referenčních množinách. Velikost této množiny je nastavena takto *b2* = 2. Sami vidíte, že velikost celé referenční množiny je skutečně malá $b = b1 + b2 = 3 + 2 = 5$.

Pokaždé když jsou vygenerována a zlepšena nějaká nová řešení tak se spustí funkce, která projde obě referenční množiny a upraví je. Pro jednoduchost se dá říct, že se provede $P = RefSet1 \cup P$, z této množiny se pak vybere *b1* nejlepších řešení, které přijdou do *RefSet1* a k nim se pak najde *b2* řešení, které jsou od nich nejvzdálenější. Tato řešení se pak uloží do *Refset2*.

3.3.2.4 Generování podmnožin

Metoda generování podmnožin pracuje na referenční množině a produkuje podmnožiny řešení z této množiny jako podklad pro kombinování řešení. Tyto podmnožiny můžeme generovat různým způsobem. Pro náš příklad stačí ta nejjednodušší. Budeme postupně brát všechny dvojice. To z naší referenční množiny o velikosti 5 prvků udělá 10 dvojic. Ty pak použijeme v kombinační metodě.

3.3.2.5 Kombinování prvků

Metoda kombinace řešení je určena k tomu, aby z podmnožiny řešení dodávané metodou generování podmnožin kombinovala jedno nebo více vektorů řešení. Tady je velká volnost algoritmu, protože způsobů generování nových řešení je nekonečně mnoho. Já jsem se nepouštěl do žádných experimentů a tak jsem použil stejných postupů, jako autoři tohoto algoritmu. Nicméně tady bych viděl možnost pro další vědeckou práci, ale myslím, že by to překročilo rozsah a zadání této mé práce, proto to nechám na někoho jiného.

V této fázi programu se berou dvojice, které jsme si v předcházejícím kroku vygenerovali, a za pomoci následujících lineárních kombinace se hledá nové zkušební řešení. V metodě, kterou jsem použil, se nejprve zjistí, do kterých podmnožin referenční množiny kombinované prvky patří a podle toho se s nimi provedou určité operace. Tyto operace jsou tři a jsou to tyto:

$$C1:\ x = x' - d$$
$$C2:\ x = x' + d$$
$$C3:\ x = x'' + d \qquad (4)$$

Kde $d = \frac{r(x'' - x')}{2}$ a *r* je náhodné číslo v rozsahu 0 až 1.

Pro výběr, které lineární kombinace se použijí, jsou tato pravidla:

- Pokud *x'* a *x''* jsou obě prvky podmnožiny nejlepších řešení (*RefSet1*), tak se provedou C1 a C3 jednou a C2 dvakrát. Tzn., získáme 4 zkušební řešení.

- Pokud je pouze *x'* nebo *x''* prvkem podmnožiny nejlepších řešení *(RefSet1)*, pak se provedou tyto kombinace: C1, C2 a C3 každé jednou. Tím získáme 3 zkušební řešení.
- Pokud ale *x'* ani *x''* nejsou prvky podmnožiny nejlepších řešení (*RefSet1*), pak se provede kombinace: C2 jednou a náhodně se vybere, jestli se použije C1 nebo C3. Tím získáme 2 řešení.

Díky tomuto postupu získáme v našem vzorovém příkladu celkem 32 nových zkušebních řešení.

3.3.2.6 Schéma fungování SS algoritmu

Teď když víme, jaké metody se používají, tak si můžeme zkusit zapsat celý algoritmus v pseudokódu:

1. Začni s $P = \emptyset$, použijeme diversifikační metodu k tvorbě zkušebních řešení x. Toto se opakuje tak aby $|P| = Psize$
2. Použijeme na všechna x zlepšující metodu a dostaneme $x \rightarrow x^*$, tímto prvkem nahradíme původní prvek.
3. Setřídíme řešení v P podle hodnot účelové funkce
   3.1. **For** ($Iter$ = 1 to $MaxIter$)
      3.1.1. Stvoř $Refset = RefSet1 \cup RefSet2$ z P, kde $|RefSet| = b$, $|RefSet1| = b1$ a $|RefSet2| = b2$. Vezmi $b1$ nejlepších řešení z P a vlož je do $RefSet1$, pak pro všechna $x^* \in P$ spočítej jejich vzdálenost od všech prvků $RefSet$. Seřaď tyto prvky podle vzdálenosti a vyber $b2$ nejlepších a vlož je do $Refset2$. $NovePrvky$ = TRUE
      3.1.2. **While** ($NovePrvky$ = TRUE)
         3.1.2.1. Vygeneruj podmnožiny
         3.1.2.2. Zkombinuj prvky a tyto prvky vlož do P
         3.1.2.3. Použij zlepšující metodu $x \rightarrow x^*$
         3.1.2.4. $StaryRefSet = Refset$
         3.1.2.5. $NovePrvky$ = FALSE
         3.1.2.6. **For** (i = 1 **to** $|P|$)
            3.1.2.6.1. **If** ($x_i^* \notin RefSet1$ a hodnota účelové funkce x_i je lepší než hodnota účelové funkce nejhoršího prvku $RefSet1$) **then**

3.1.2.6.1.1. Přidej x_i^* do ***RefSet1*** a smaž nejhorší prvek této množiny

3.1.2.6.1.2. ***NovePrvky*** = TRUE

3.1.2.6.2. **Else**

3.1.2.6.2.1. **If** ($x_i^* \notin$ ***RefSet2*** a vzdálenost x_i^* od prvků je lepší než nejhorší prvek v ***RefSet2***) **then**

3.1.2.6.2.1.1. Přidej x_i^* do ***RefSet1*** a smaž nejhorší prvek této množiny

3.1.2.6.2.1.2. ***NovePrvky*** = TRUE

3.1.1. **If** (***Iter*** < ***MaxIter***) **then**

3.1.1.1. Stvoř novou množinu ***P*** za pomoci diversifikační metody.

3.1.1.2. ***P*** = ***P*** $\cup$. ***RefSet***

4. Vybereme nejlepší prvek z ***RefSet1*** a prohlásíme jej za řešení

Možnosti jednotlivých parametrů hodlám rozebrat v praktické části, takže tady končí popis SS algoritmu.

Pokud by někoho zajímal tento algoritmus více, pak doporučuji oficiální webovou stránku tohoto algoritmu [3], odkud jsem čerpal informace a postřehy.

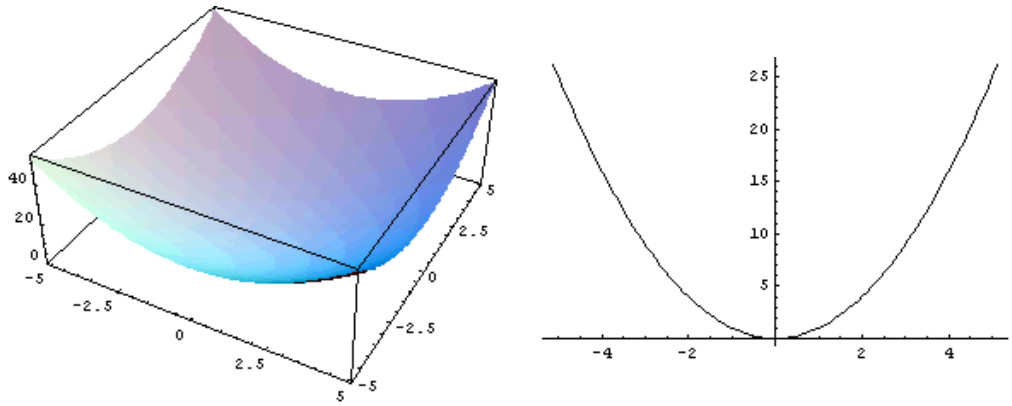
4 TESTOVACÍ FUNKCE

Jak pracují, mnou zpracované algoritmy bychom už nyní měli vědět. Funkčnost a účinnost jednotlivých algoritmů se zkouší na různých příkladech. Na webu má každý možnost zkusit si vybrané algoritmy na testovacích funkcích. Tyto funkce jsem s dovolením použil z webové stránky vedoucího mé práce pana Zelinky [4]. Který se mimo jiné zabývá i sbíráním, těch „nejzajímavějších“ funkcí, takže se zde nashromáždilo několik zajímavých a „zákeřných“ funkcí. Zákeřnost funkcí se posuzuje podle toho, kolik a jakých pastí připraví pro testovaný algoritmus, takže počítám, že se najde několik matematiků, kteří se věnují jenom tomu, že zkouší vymyslet a definovat takovou funkci, na které si i ty nejlepší algoritmy vylámou zuby.

Dále jsou uvedeny dvě testovací funkce, na kterých jsem zkoušel realizaci algoritmů a nastavování jejich parametrů. Pokud má někdo pocit, že se nejedná o dostatečný počet funkcí, tak bych jej rád upozornil, že nebylo mým cílem otestovat chování algoritmů za všech okolností, jen získat hrubou a obecnou informaci o tom, jak se algoritmy chovají na unimodální a multimodální funkci a vyzkoušet změnu jejich parametrů.

4.1 První De Jongova funkce

$$\sum_{i=1}^{Dim} {x_i}^2 \tag{5}$$



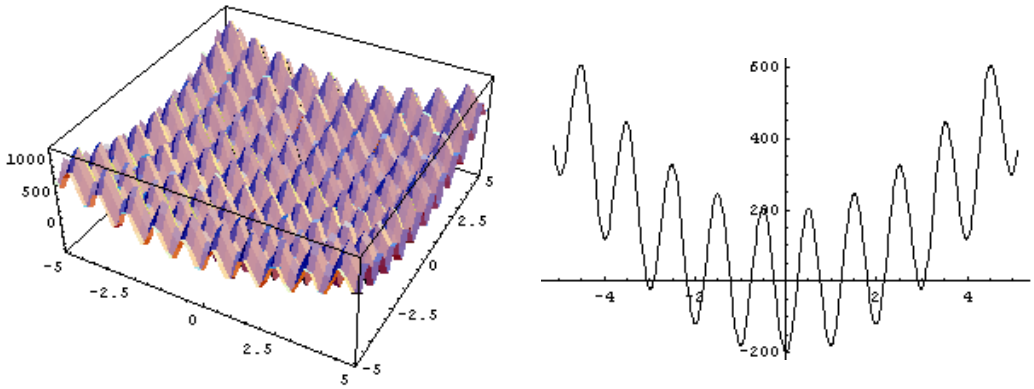
Obr. 9. 3D a 2D graf De Jongovy první funkce

Tahle funkce je základní testovací funkcí, nevyskytují se zde žádné záludnosti. Jak uvidíme v praktické části, tak tahle funkce je velmi vhodná k testování horolezeckého algoritmu, protože neobsahuje žádné lokální minima.

Tuto funkci jsem zvolil za zástupce unimodálních funkcí.

4.2 Rastriginova funkce

$$(20)\sum_{i=1}^{Dim}(x_i{}^2 - 10\cos(2\pi x_i)) \tag{6}$$



Obr. 10. 3D a 2D graf Rastriginovy funkce

Jak můžete sami vidět, tato funkce by se klidně dala nazvat „Vrah gradientních algoritmů“. Přesně tak určitě byla i myšlena. Velké množství lokálních extrémů nevyhovuje ani jiným algoritmům, ale jak si ukážeme dál, tyto algoritmy jsou schopny nám i přesto poskytnout zajímavé výsledky.

Myslím, že jsem neudělal chybu, když jsem si zvolil tuto funkci, jako zástupce multimodálních funkcí.

5 PROGRAMOVÉ PROSTŘEDKY

Protože EA jsou přeci jenom lepší, když jsou realizované, tak si dovolím, krátké zamyšlení nad používanými a použitými prostředky. Stejně jako samotné EA, tak i prostředky se s časem vyvíjely. Vzhledem k tomu, že se jedná o algoritmy, tak nás to přímo vybízí použít na jejich realizaci počítače. Přiznám se, že mi přijde úsměvná, ale ne vyloučená představa, jak nějaký matematik používá EA jenom za pomoci kalkulačky, tužky a papíru. Dovedu si například velmi dobře představit takový horolezecký algoritmus v provedení tužka-papír. Ale na druhou stranu si položme otázku, proč to vůbec dělat, když dnes jsou počítače téměř všude dostupné a pro naše výpočty nepotřebujeme žádný sálový superpočítač, ale stačí nám obyčejný stroj, na kterém například sekretářka píše dopisy a hraje pasiáns.

Abychom mohli použít určité programové prostředí, tak po něm požadujeme tyto podmínky:

- Zvládnutí výpočtu účelové funkce a pomocných výpočtů
- Větvení kódu – podmínky `If, Case`, atd.
- Smyčky a cykly – `For, While, Until` atd.
- Pokud hodláme použít stochastické algoritmy, tak i generátor „náhodných“ čísel

Možná vám to přijde divné, ale pokud se nad tím zamyslíme, tak nám nic nebrání, abychom naprogramovali některý algoritmus například v tabulkovém procesoru některého kancelářského balíku. Bude to sice trošku náročnější na interakci s uživatelem a nebude to tak „čisté“ jako využití některého programovacího prostředí, ale také to půjde.

5.1 Běžné programovací jazyky

Nejprve se na programování EA používaly stejné prostředky jako na programování jiných programů. To má řadu výhod, jako nejdůležitější z nich bych zdůraznil, tyto:

- Existuje dostatečné množství lidí, kteří umí v daném prostředí pracovat
- Prostředí poskytuje všechny nezbytné prostředky pro programování EA
- Kód lze napsat docela optimální, takže provádění EA není tak časově náročné

Vzhledem k tomu, že nic není pouze černé nebo bílé, tak zde nalezneme samozřejmě taky několik nevýhod:

- Pro „běžného“ člověka je složité pochopit některé vazby a způsoby použití syntaxe
- Nutnost definice datových typů
- Nutnost spousty úkonů, které nesouvisí s vlastním algoritmem, ale jsou potřebné pro vlastní program
- Některé matematické operace jsou realizovány velmi krkolomně

Z toho můžeme vyvodit závěr, že běžné programovací prostředky jsou vhodné pro programátory nebo obecně pro lidi, kteří mají čas a chuť k tomu, aby pronikly do tajů daného programovacího jazyka. Na druhou stranu potom umožňují optimálnější a rychlejší kód.

5.2 Mathematica

V následujícím textu uvedu obecnou charakteristiku prostředí Mathematica a některé její přednosti, pokud budete mít pocit, jako by mi platili za reklamu, tak je to pocit mylný, jen se mi s jejich produktem dobře pracuje.

Jedná se o prostředí od firmy Wolfram research, Inc [5]. Jak už sám název napovídá, tak se jedná o prostředí orientované na matematiky a matematiku. Protože programovacích jazyků a programování začaly hojně využívat vědci a matematici, kteří nejsou ochotní, ať už z časových nebo osobních důvodů, studovat určitý programovací jazyk, tak v roce 1988 vznikla první verze tohoto prostředí s označení Mathematica 1.0. Další vývoj tohoto softwaru sledovat nebudeme, ale nynější verze nese označení 5.2. Není to samozřejmě jediný matematický software, který je na trhu, ale vzhledem k tomu, že jsem v něm programoval již bakalářskou práci a navíc nejsem s ostatními blíže obeznámen, tak budu psát o Mathematice.

Tvorbu tohoto typu prostředí považuji za docela úspěšný krok ve vývoji vědy, protože dal mocný nástroj v podobě počítačů a jejich „programování“ i lidem, kteří neznají klasické programovací jazyky.

Ze začátku se vám může zdát, že jde jenom o jakousi chytřejší kalkulačku, ale časem zjistíte, že je o moc chytřejší, než jste předpokládali. K tomuto dojmu také napomáhá její rozšíření mezi uživateli, které má za následek vznik mnoha balíčků dalších

uživatelských funkcí. Následující popis rozdělím na dvě části, nejprve se zmíním o matematických funkcích a pak popíši i programátorské nebo ovládací funkce.

5.2.1 Matematické funkce a proměnné

Protože matematika je zaměřena na výpočty, tak zde najdeme velkou podporu matematických operací, ale začněme pěkně od začátku a tím by měl být zápis. Pokud se s tímto prostředí setká člověk, který není zvyklý na programování, tak uvítá možnost zápisu funkcí přímo ve tvaru, který je běžný pro normální matematický zápis, kde za pomoci nástrojů je možno vkládat normální značky a prvky (Obr. 11).

$$F := (20) \sum_{i=1}^{Dim} (x_i - 10\,Cos[2\pi \times x_i])$$

Obr. 11. Zápis Rastriginovy funkce v Mathematice

Sami vidíte, že zápis se příliš neliší od toho, co jsme zvyklí zapisovat na papír, jediný rozdíl, na který si musíme zvyknout, je používání závorek, kde každý typ závorky má svůj vlastní význam a taky to, že funkce se píší s velkým počátečním písmenem. Toto byl zápis pro matematiky, ale matematika taky počítá s tím, že s ní bude pracovat i člověk, který už umí programovat a tak má své určité návyky, takže stejná rovnice jde napsat i takto (Obr. 12).

```
F := (20) * Sum[(x[[i]] - 10 Cos[2 * PI * x[[i]]]), {i, Dim}]
```

Obr. 12. Jiný styl zápisu Rastriginovy funkce v Mathematice

Sami si můžete všimnout, že zde nejsou použity, žádné symboly, které se nedají vkládat přímo z klávesnice a navíc jsou zde místo toho použity příkazy (kterým se v matematice říká funkce). Za všechny bych uvedl třeba funkci *Sum.* Syntax je jiný, než u ostatních programovacích prostředí, ale smysl a styl zápisu se jim blíží. Na matematice je krásné to, že můžeme používat oba dva styly zápisu a můžeme je klidně i kombinovat. Přiznám se, že i mě je příjemnější zadávat některé matematické operace za pomocí grafických značek, než za pomoci funkcí.

To jsme si ukázaly zápis. Dalším příjemným prvkem je potlačení nutnosti uvádět datový typ, a vůbec inicializace proměnných. Mnozí namítnou, že pak vzniká chaos a programátor pak nemá pod kontrolou tyto proměnné. Ale v tom to je, dalo by se říct,

že matematik to nepotřebuje. Mathematica nám inicializuje proměnnou ve chvíli, kdy ji poprvé použijeme. Datový typ si odvodí sama z typu dat, které do proměnné umístíme. Některým programátorům se jistě začaly ježit hrůzou vlasy na hlavě, ale bát se netřeba. Při své práci jsem s tímhle zatím nenarazil na žádný vážnější problém. Ale pro ty z vás, kteří by z toho mohli mít strach, je tady funkce, která nám vrací typ proměnné. Navíc se zde s typy proměnných pracuje inteligentně. Pokud například nadefinujete, že `a = 1; b = 1.5;` a pak někde použijete `a = a/b;` nestane se katastrofa, ale v proměnné a se objeví číslo `0.666667`, a když se podíváme na typ tak zjistíme, že se `Integer` přepsal na `Real`. Některé typy číselných proměnných jako například *`Rational`*, který je definován jako `Integer / Integer` bychom asi v jiných prostředí hledali marně, ale myslím, že není důležité se o tom více rozepisovat.

Rád bych se ale zmínil o polích a seznamech, které mi přijdou nádherně řešené a práce s nimi je jednodušší než v klasických programovacích jazycích.

Nebudu se rozepisovat o postupech, jak se dá definovat pole nebo vícerozměrné pole v jiných jazycích, ale v Mathematice je to velmi jednoduché. Mějme například *p* pole o 5 prvcích, které budou čísla tytu *integer* 1-5. Toto pole pak nadefinujeme takto: `p={1,2,3,4,5}`. Pokud budeme pak potřebovat větší pole, tak tam jenom jednoduše přidáme prvek a nestaráme se to, jestli musíme pole nějak rozšiřovat.

Pokud budeme potřebovat například matici 3×3 prvky tak ji můžeme nadefinovat takhle: `matice = {{`a_{11}`,`a_{12}`,`a_{13}`},{`a_{21}`,`a_{22}`,`a_{23}`},{`a_{31}`,`a_{31}`,`a_{33}`}}`. Takže sami můžete pozorovat, že práce s poli je velmi intuitivní, což mnohým značně usnadní práci. Stejný způsob zápisu a uchovávání hodnot se používá i pro tabulky a v Mathematice se nepoužívá pojem pole, ale spíše se používá termín seznam nebo tabulka. Tyto seznamy mohou mít i více rozměrů tak například není problém vytvořit seznam podobný tomuto:

`seznam = {{{`a_{111}`,`a_{112}`},{`a_{121}`,`a_{122}`}},{{`a_{211}`,`a_{212}`},{`a_{221}`,`a_{222}`}}}`. Jedná se o třírozměrné pole. A adresování jednotlivých prvků je stejně snadné, takže pokud budeme například potřebovat prvek a_{211} z tohoto seznamu tak jej zavoláme jako `seznam[[2,1,1]]`.

Tím se konečně dostávám k tomu, na co jsem si už u Mathematicy zvykl a co by mi jinde chybělo. Jedná se o seznam, ve kterém uchovávám jednotlivá řešení EA. Tento

seznam má následující strukturu: `výsledky = {{CV, {`x_1`,`x_2`, … `x_{dim}`}},{CV, {`x_1`,`x_2`, … `x_{dim}`}}, …}`, takže jednotlivé výsledky zavolám `výsledky[[pořadí výsledku]]`, čímž dostanu: `{CV, {`x_1`,`x_2`, … `x_{dim}`}`. Adekvátně k tomu pak voláním `výsledky[[pořadí výsledku,1]]`, dostanu hodnotu účelové funkce a `výsledky[[pořadí výsledku,2]]` mi zavolá vektor řešení, atd. Trochu problém pak nastává při složitější struktuře seznamů, kdy už člověk ztrácí přehled o tom, který prvek je co. Pak se taky musí hlídat, jaký přesný výsledek nám funkce předá, protože `{{1}}` je něco jiného než `{1}` (jinak se volá) a některé funkce si přidávají vlastní rozměry seznamů. S tím je ale počítáno a tak se vždy v dokumentaci můžete dočíst jaký výsledek je poskytován a navíc existuje spousta funkcí, které se zabývá práci se seznamy, jako je jejich seskupování nebo naopak odstraňování vnoření. Je to nádherná hračka, která velmi pomáhá při práci.

5.2.2 Funkce pro řízení běhu programu

Do kategorie těchto funkcí patří příkazy pro podmínky, cykly atd. Na tomto místě, by se potencionální programátor měl rozhodnout, jakým stylem bude programovat. Mathematica, totiž umožňuje vkládat i kód z programovacího jazyka C++ a hlavně umožňuje programovat podobně jako v tomto jazyce. Syntax je sice trošičku odlišná, ale principy a myšlenky se dají použít totožné. Tímto způsobem jsem programoval i svou bakalářskou práci. Hemžilo se to v ní cykly a podmínkami a procedurami. Nyní při programování na této práci jsem zkusil jiný přístup. Mathematica je totiž vymyšlena pro matematiky a ne pro programátory, tudíž se zde dají použít i jiné prostředky a postupy. Pro člověka, který se v tom nevyzná, se to může zdát matoucí, ale alespoň základy se dají pochopit poměrně rychle. Nejprve změna, která se bude zdát malicherná, ale pomůže nám v lepším chápání dalších postupů. V Mathematice nejsou procedury. Říká se jim funkce, což přesněji odráží pohled matematiků. Když pochopíme toto a naučíme se používat několik neobvyklých postupů a prostředků, tak se nám hned bude pracovat lépe. S podmínkami se budeme setkávat běžně a používají se standardně. Navíc zde můžeme definovat nejen to, co se má vykonat pokud je podmínka splněna a co pokud podmínka splněna není, ale i to, co se má vykonat pokud podmínka nelze posoudit. Pokud například máte podmínku `x < y`, ale *x* nemá žádnou hodnotu, pak samozřejmě není ani menší, ale ani větší, takže se provede tato třetí větev programu.

Cykly se dají použít trošičku jinak. Původně jsem používal, tak jako každý cykly `For`, `While` atd., ale postupně jsem objevoval funkce, které toto nahrazují a navíc přímo pracují se seznamy. Například potřebujeme vygenerovat 10 náhodných čísel typu *Integer* v rozsahu (0,10) a uložit je v seznamu *prvky*. Můžeme použít standardní funkci `For`, pak kód může vypadat takto:

```
For[i=0, i<10, i++, prvky = Append[prvky, Random[Integer,{0,10}]]];
```

Nesmíme ale předem ještě zapomenout vytvořit seznam *prvky* protože `Append` je jedna z funkcí, které potřebují, aby již proměnná byla nadefinována. Takže ještě musíme přidat před kód toto: `prvky = {};` čímž vytvoříme prázdný seznam.

Druhý postup je elegantnější a navíc rychlejší a to jak při programování, tak i při zpracování a je to využití funkce `Table`, která stvoří tabulku (seznam). Tady pak nemusíme předem nic definovat nebo vytvářet.

```
prvky=Table[Random[Integer,{0,10}],{10}];
```

Podobných funkcí je k dispozici víc, další, kterou bych vám chtěl ukázat, je funkce `Nest`. Její předpis je `Nest[f, expr, n]`, kde *f* je funkce která se má provést, *expr* je argument této funkce a *n* udává kolikrát se má tato funkce na argument použít, takže například tento zápis: `Nest[f, x, 3]` je ekvivalentní se zápisem `f[f[f[x]]]`. Když si uvědomíme, že funkce používáme místo procedur, tak se tato funkce dá skvěle použít při EA, například se dá (zjednodušeně) použít takto:

```
Nest[horolezeky_algoritmus,vychozi_reseni,pocet_iteraci]
```

Spustíme funkci `horolezeky_algoritmus` s argumentem `vychozi_reseni`. Tím dostaneme nějaké nové řešení. Toto dosadíme znovu do funkce `horolezeky_algoritmus`, a to provedeme `pocet_iteraci`-krát. Přesně toho dosáhneme použitím předchozího příkazu.

Dalším podobným příkazem, který nám nahradí použití cyklů, může být příkaz Map. Předpis tohoto příkazu je následující: `Map[f, expr]` a jeho funkci můžeme pochopit na následujícím příkladě: `Map[f,{a,b,c}]`, který by se dal zapsat jako `f[a]; f[b]; f[c]`. Navíc existují i rychlejší zápisy některých těchto funkcí takže například předchozí zápis se dá taky napsat jako: `f /@ {a,b,c}`. Na jednu stranu, pokud se v tom již orientujeme, tak nám to usnadní práci, ale pokud ne, tak nám to kód znepřehlední.

Podobných funkcí najdeme v Mathematice spousty, ale myslím, že ty které jsem již uvedl, stačí pro to, abychom si mohli udělat představu o tom, že programování je trošku odlišné od běžně používaného. Pokud byste se o tuto problematiku chtěli hlouběji zajímat, tak doporučuji nápovědu přímo v Mathematice, která je velmi hezky a přehledně zpracovaná a to i včetně krátkých příkladů použití. Případně se můžete podívat na dokumentaci přímo na internetové stránce výrobce [5].

5.3 WebMathematica

Vzhledem k rozšíření internetu a potřebě prezentovat matematické výpočty i na www stránkách, eventuelně provádět na stránkách určité výpočty nebo simulace, tak vznikla WebMathematica. Dalo by se říct, že se jedná o rozšíření Mathematicy. Vlastní program se skládá ze serverové aplikace, do které jsou přes www stránky za pomoci skriptů odesílána data a aplikace nám na stránku vrací výpočty, grafy a ostatní výstupy na které jsme zvyklí z klasické Mathematicy. Pokud potřebujete na své webové stránky umístit výpočty, grafy nebo podobné matematické náležitosti, pak je WebMathematica tím pravím řešením.

Vzhledem k tomu, že vychází z Mathematicy, tak nikoho nepřekvapí, že používá téměř stejné funkce a postupy. Největší rozdíl je v předávání si dat mezi uživatelem a výpočetním jádrem aplikace. Když tedy pomineme jiné vstupy a výstupy, potřebu volat jádro a podobné drobnosti, které jsou způsobeny tím, že se výpočty provádí skriptově na jiném počítači, tak můžeme prohlásit, že je to stejné.

Dalo by se prohlásit, že Mathematica je vývojovým prostředím pro WebMathematicu. Proto většina toho, co jsme si řekli v minulé kapitole, platí i zde. Chtěl bych se ale zmínit o některých odlišnostech a postupech, se kterými se setkáme při převádění programů z Mathematicy do WebMathematicy.

5.3.1 Volání WebMathematicy

Pokud vás zajímají spíše nastavení serveru a práce se serverem, pak vás musím odkázat na stránky s dokumentací [6], protože praktické zkušenosti s tím nemám a opisovat dokumentaci mi přijde zbytečné.

Soubor se skriptem musí obsahovat určité náležitosti, aby jej server mohl vykonat. Než se na ně podíváme, tak bych ještě zmínil, že soubor má příponu „jsp“. V souboru můžeme používat standardní HTML příkazy a využívat Javascript.

Na úplném začátku souboru se musí objevit tyto informace:

```
<%@ page language="java" %>
<%@ taglib uri="/webMathematica-taglib" prefix="msp" %>
```

Tyto informace říkají serveru, že se jedná o stránku se skriptem pro WebMathematicu.

Zde se dají ještě doplnit například informace o kódování znakové sady:

```
<%@ page contentType="text/html; charset=windows-1250"%>
```

Data a příkazy se odesílají na server pomocí formuláře, proto se musí funkční část kódu obalit tagy pro formulář:

```
<form action="jméno souboru.jsp" method="post">
…
</form>
```

Formulář se potom odešle tlačítkem, které si vygenerujeme tímto kódem:

```
<input type="submit" name="btnSubmit" value="Pocitej">
```

což je standardní potvrzovací tlačítko.

Proměnné se zadávají za pomoci standardních formulářových položek, jako je `input` a `select`.

Teď už víme, jakým způsobem odeslat data serveru, ale ještě nesmíme zapomenout na několik věcí. Tou první je alokovat a pak uvolnit jádro pro výpočty na serveru. Pro tuto činnost se používá párový tag:

```
<msp:allocateKernel>
…
</msp:allocateKernel>
```

mezi tyto tagy se musí umístit to, vše co budeme chtít, aby nám server provedl. Takže vlastní program budeme mít obalen jak tagy pro alokaci jádra tak i tagy pro formulář. Teď bychom měli představu o tom, jak zasílat serveru proměnné, ale nikde jsem se nezmínil o tom, jak předávat příkazy. Je to velmi jednoduché. Cokoliv zabalíme do tagu:

```
<msp:evaluate>
…
</msp:evaluate>
```

tak bude chápáno jako instrukce pro server a bude provedeno. Do tohoto tagu můžeme dávat přímo náš kód, který jsme si napsali a odladili v Mathematice.

5.3.2 Práce s proměnnými

Proměnné si můžeme rozdělit několika způsoby, nejprve si je rozdělíme na dva druhy. Prvním jsou zadávané proměnné, tím druhým pak pomocné proměnné.

Zadávané proměnné se, jak jsme si už řekli, zadávají standardními formulářovými prvky, pro odlišení začínají znaky `$$`. Inicializovány jsou až po odeslání formuláře a trvají, dokud trvá HTML session. Uvozovací znaky nám přidá přímo jádro, takže pokud máme například vstupní pole `<input type="text" name="proměnná" />`, tak po odeslání formuláře získáme proměnnou: `$$proměnná` s hodnotou jakou jsme jí zadali. Odlišení zadávaných proměnných je důležité, protože dokud neodešleme formulář, tak vlastně neexistují. Proto, si musíme dát pozor, aby se například nevypisovali jako výsledek nebo s ní nepočítali, stránka by nám pak generovala chybové hlášení o neexistující proměnné, dokud bychom neodeslali formulář. Dalším důvodem je fakt, že aby mohl server s touto hodnotou počítat, tak ji musíme přeložit, jinak se bude jednat o data typu string. Pokud ale budeme mít například proměnnou `$$checked` která nám dává informaci o tom, jestli je checkbox zatrhnutý nebo ne, tak ji překládat nemusíme.

Překlad se provádí za pomoci několika vestavěných funkcí WebMathematicy. Jsou to především tyto dvě: `MSPBlock[]` a `MSPToExpression[]`. `MSPBlock[]` pracuje tak, že vše co uzavře, se vykoná až po spuštění skriptu a bude považováno jako vstup. Takže si nemusíme dělat starosti s tím, jakého typu jsou proměnné, protože WebMathematica si to přebere stejně jako by si to přebrala Mathematica.

`MSPToExpression[]` se používá trošku jinak. Vrací přeloženou hodnotu proměnné. Takže je vhodná například k tomu, abychom si do pomocné proměnné uložili hodnotu zadávané proměnné. Možná se to zdá krkolomné, ale pokud potřebujete s jednou proměnnou pracovat víc než jenom jednou, pak je výhodné si její interpretovanou hodnotu uložit jako pomocnou proměnnou: `A = MSPToExpression[$$A]`, tímto jsme hodnotu zadávané proměnné *$$A* vložili do pomocné proměnné *A*, ale zase se nám objeví chybová hláška, že proměnná *$$A* neexistuje. Takže co s tím? Jednoduše doplníme předchozí příkaz o podmínku:

```
A = If[MSPValueQ[$$A], MSPToExpression[$$A]];
```

Tímto jsme nejen pozdrželi vyplnění proměnné *A* přeloženou hodnotou proměnné *$$A* do doby než bude mít nějakou hodnotu, ale zároveň jsme otestovali, proměnnou *$$A*, že má platnou hodnotu.

Stejný postup pak použijeme při vypisování výsledků a kreslení grafů. Použijeme podmínku a budeme testovat, jestli některá proměnná má nebo nemá hodnotu. Vyhneme se tak vypisování chybových hlášek z důvodu toho, že proměnné ještě nebyly inicializovány a program ještě neproběhl.

Pomocné proměnné jsou ty proměnné, jejichž hodnota se nezadává. Tzn., jejich hodnota se buď počítá, nebo jinak stanovuje programem. Nejsou nijak označeny a používají se stejně jako proměnné v Mathematice.

Další dělení proměnných se děje podle délky jejich životnosti. Zde rozlišujeme proměnné stránky a proměnné sessionu. Proměnné stránky mají stejné trvání jako stránka. Tzn., po uzavření stránky se smažou, pokud bychom chtěli předávat nějaká data mezi několika různými stránkami, tak potřebujeme session proměnnou. Ta se používá stejně jako proměnná stránky, ale je třeba ji někde nadeklarovat za pomoci funkce `MSPSessionVariable[proměnná, její hodnota]`. Tímto jasně řekneme, že se má proměnná zachovat v paměti dokud neskončí session.

5.3.3 Výpis proměnných, výsledků a grafů

Řekněme, že máme program, víme jak mu předat proměnné, ale budeme chtít, aby nám vrátil výsledek. V Mathematice, to bylo jednoduché, napsali jsme proměnnou a pokud jsme za ni nedali středník, tak se nám vypsala její hodnota.

Pokud bychom stejný postup použili ve WebMathematice, tak bychom se dočkali nemilého překvapní. A to proto, že jeden blok mezi tagy `<evaluate>` může obsahovat pouze jeden řádek, který není ukončen středníkem a ještě to může být pouze ten poslední. Dalším problémem by bylo to, že server by se pokusil vypsat výsledek hned při načítání stránky a je jasné, že v té době ještě nejsou načteny zadávané proměnné a tak bychom dostali chybovou hlášku. Řeší se to stejně jako s proměnnými. Dá se podmínka, která testuje, jestli některá ze zadávaných proměnných má hodnotu. Pokud ano, pak již byl odeslán formulář a můžeme vypsat i výsledek. Například pokud

bychom měli zadávanou proměnnou *$$A* a výsledek bychom měli v proměnné *A* pak bychom tento výsledek mohli vypsat například takto:

```
If[MSPValueQ[$$A],A]
```

Tento kousek kódu bychom buď umístili na konec `<evaluate>` bloku nebo do samostatného bloku.

Grafy jsou ještě o kousek komplikovanější. V Mathematice nám stačilo zavolat funkci grafu s patřičnými parametry a pokud jsme ji neukončili středníkem, tak se nám graf vykreslil. Ve WebMathematice máme dva postupy, jak graf vykreslit. První je za použití `MSPBlock[]`. Pokud chceme použít tento postup, pak je to možno za využití funkce `MSPBlock[]` a funkce `MSPShow[]`. Pak například graf za použití funkce `Plot[]` se zobrazí následovně:

```
MSPBlock[{},MSPShow[Plot[]]]
```

Nejprve se vykreslí graf bez hodnot a po odeslání formuláře se hodnoty dokreslí.

Druhý postup je trošičku krkolomnější, ale zase se graf objeví až celý. Prvním krokem, bude to, že si do nějaké proměnné uložíme graf: `g = Plot[…];`

Pak jej za použití funkce `MSPShow[]` vykreslíme. Pokud dáme před tuto funkci podmínku stejně jako při vypisování výsledků, pak se celý graf vykreslí až po odeslání formuláře.

Doufám, že to jako úvod do programu WebMathematica stačilo. U jednotlivých algoritmů ještě ukážu kousky kódu, na kterém je možno pochopit více. Pokud Vás Mathematica nebo WebMathematica zaujala, pak Vám jako zdroj dalšího studia doporučuji dokumentaci k těmto programům, kterou můžete najít na stránkách firmy Wolfram [5]. Mathematica má také velmi přehledně a srozumitelně udělaný help. Tyto dva zdroje jsem použil i já při studiu Mathematicy a WebMathematicy.

II. PRAKTICKÁ ČÁST

6 HOROLEZECKÝ ALGORITMUS

O principu fungování horolezeckého algoritmu jsem již psal. Pojďme se nyní podívat na to, jak jsem jej realizoval a hlavně jaký má vliv výběr parametrů na úspěšnost algoritmu.

6.1 Parametry

Chtěl jsem dát uživateli možnost měnit různá nastavení a sledovat, jaký vliv to má na úspěšnost algoritmu. Horolezecký algoritmus nemá parametrů a nastavení tolik, ale stejně se na ně pojďme podívat.

6.1.1 Účelová funkce

Zde se nastavuje účelová funkce, pro kterou se bude algoritmus vykonávat. Je možnost použít již nadefinované testovací funkce, kde se nám pro každou funkci upraví i rozsahy jednotlivých proměnných. Pokud by si uživatel chtěl nadefinovat vlastní funkci, tak samozřejmě může, slouží k tomu vstupní pole nadepsáno „Vaše vlastní funkce“. Zde je ale nutné dodržovat syntaxi používanou v Mathematice. Pokud se Vám podaří napsat funkci špatně, tak se program sice provede, ale nebude umět vypočítat Vaši funkci a proto se nedopočítá k žádnému výsledku.

6.1.2 Vzorový jedinec

Tento způsob zápisu počtu proměnných, jejich typů a omezení jsem převzal z jiných EA. Uvažoval jsem, jestli je vhodné jej použít i pro horolezecký algoritmus. Přehlednosti zápisu a usnadnění realizace algoritmu mě přesvědčila, že je vhodné jej použít. Je nutné přesně dodržet zápis. Druh a množství závorek (svorek). Na začátku i na konci jsou dvě svorky, každá proměnná je pak napsána ve vlastní svorce. Oddělování všech hodnot čárkou a používání desetinné tečky. Proměnná má tvar `{typ,{minimum,maximum}}`. Jako typ se dá použít buď *Int* pro celočíselné proměnné nebo *Re* pro reálné proměnné.

Například budeme potřebovat dvě proměnné. První bude celočíselná od -100 do 180 a druhá reálná od 0,16 do 6,25. Zápis by potom byl tento:

```
{{{Int,{-100,180},{Re,{0.16,6.25}}}}
```

Pokud by se někomu podařilo napsat špatně jedince, tak se algoritmus provede s tímto špatným jedincem a tudíž nám nevrátí smysluplné řešení.

6.1.3 Počet iterací

Tento parametr udává, kolikrát se bude horolezecký algoritmus opakovat. Doporučuji používat s rozmyslem, protože se jedná o webovou aplikaci a pokud by se zadalo velké číslo, tak se server zbytečně přetíží. Server se proti přetížení brání tím, že pokud do určité doby nemá hotový výsledek, tak výpočet ukončí a uživateli pošle oznámení o vypršení časového limitu.

6.1.4 Okolí

Jediné čím můžeme zkusit nějak rozumně ovlivňovat funkčnost horolezeckého algoritmu je sousedství a všechno co s ním souvisí.

Tvorba okolí se dá provádět, v mé realizaci, dvěma způsoby. Než si je uvedeme, tak bych zmínil ty parametry, které se uplatní u obou způsobů.

6.1.4.1 Velikost okolí

Tento parametr udává, jak daleko od původního řešení se budou hledat nová řešení (tzv. sousedé). Pokud například máme proměnnou 10 a okolí nastavené na 2, tak se bude nová proměnná hledat v rozsahu {8,12}. Při simulacích jsem narazil na drobný zádrhel. Pokud máte funkci, kde jsou rozsahy jednotlivých proměnných rozdílné, mám na mysli řádově, tak narážíme na problém. Uvedu příklad: měl jsem optimalizovat funkci, kde bylo mimo jiné $x_1 = \{Int,\{130,210\}\}$ a $x_2 = \{Re,\{0.000005,0.00001\}\}$. Jak teď nastavit správně okolí aby mělo smysl jak pro první proměnnou, tak i pro druhou? Nejprve jsem se rozhodl řešit tento problém možností definice různých okolí pro různé proměnné. Zkusil jsem to a narazil jsem na složitost zadávání parametrů a složitost realizace. Pak mě napadlo jiné řešení. Toto řešení spočívá v aplikaci logiky a matematických pravidel. V definici proměnných jsem napsal $x_2 = \{Re,\{50,100\}\}$ a v účelové funkci jsem místo x_2 používal $x_2 \times 10^{-7}$.

6.1.4.2 Velikost kroku

Udává, po jakých krocích budeme prozkoumávat okolí při tvorbě sousedů. Například když zadáme *velikost okolí* = 1, *velikost kroku* = 0.5 a počáteční bod bude mít hodnotu 4, tak prozkoumáme tyto hodnoty{3,3.5,4,4.5,5}. Vzhledem k tomu, že horolezecký

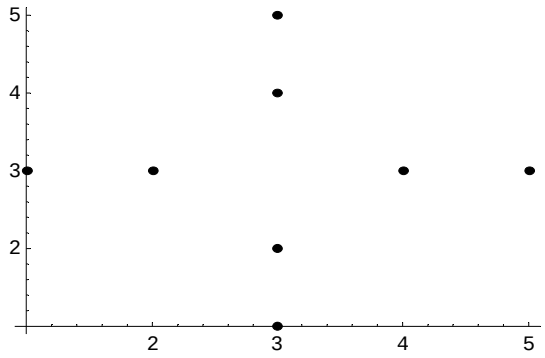
algoritmus už nehodnotí bod, ve kterém se právě nalézá, tak musíme hodnotu 4 vyřadit. Pokud bude navíc uvedeno, že proměnná je celočíselného typu, pak nám vypadnou i hodnoty {3.5,4.5}. Pokud by se někomu povedlo zadat nesmyslné hodnoty, jako například nulové okolí atd., tak algoritmus vrátí nesmyslné výsledky.

6.1.4.3 Tvorba sousedů

Protože se dá použít více různých postupů jakou metodu aplikovat na výběr sousedů z okolí, tak jsem se rozhodl realizovat dvě z nich, které si myslím, že hodně rozšíří použitelnost horolezeckého algoritmu při simulacích. Tyto metody se vybírají z roletky.

6.1.4.3.1 Kříž

Tento postup spočívá v tom, že se vezme jedna proměnná, která se bude měnit. Měnit se bude podle *velikosti okolí* a *velikosti kroku* ostatní proměnné zůstanou stejné. Toto se provede postupně se všemi proměnnými. Kdybychom toto použili při řešení funkce s dvěma proměnnými, tak nám vznikne kříž, proto metoda kříže. Jako je to ukázáno na obrázku (Obr. 13), kde jsme použili *velikost okolí* = 2, *velikost kroku* = 1 a počáteční bod [3,3].



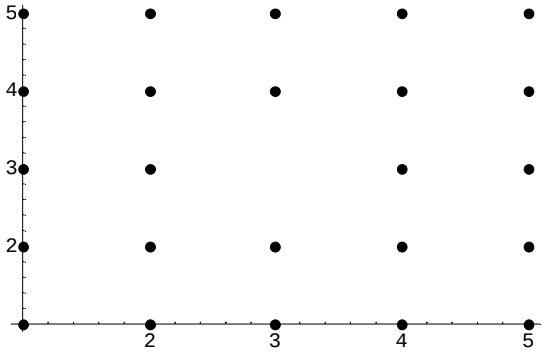
Obr. 13. Sousedé vygenerovaní metodou „Kříž“

Je samozřejmě možné nastavit *velikost okolí* shodnou s *velikostí kroku* a pak bychom prohledávali jenom hranice okolí.

6.1.4.3.2 Náhodný čtverec

Myšlenka této metody generování sousedů spočívá v tom, že se bude prohledávat „celé“ okolí a náhodně se z něj vybere několik sousedů. Protože horolezecký

algoritmus si ani neklade za cíl, aby byl velmi přesný, proto jsem použil zase *velikost okolí* a *velikost kroku*. Tito náhodní sousedé se pak vygenerují jenom ze čtverce, který můžeme vidět na obrázku (Obr. 14). Parametry jsou nastaveny jako v předchozí ukázce.



Obr. 14. Možní sousedé vygenerovaní metodou „náhodný čtverec“

Z tohoto čtverce se vyberou sousedé. V této metodě se uplatní nový parametr *maximální počet sousedů,* tento parametr je nezbytný pro webovou aplikaci, aby se omezil počet prohledávaných řešení a tím i možnost přetížení serveru. Maximální proto, že například kdybychom použili předchozí zadání a chtěli bychom najít 26 náhodných řešení, tak bychom je našli, ale některá by se opakovala a pro nás by tudíž neměla žádnou hodnotu a tak je vyřadíme.

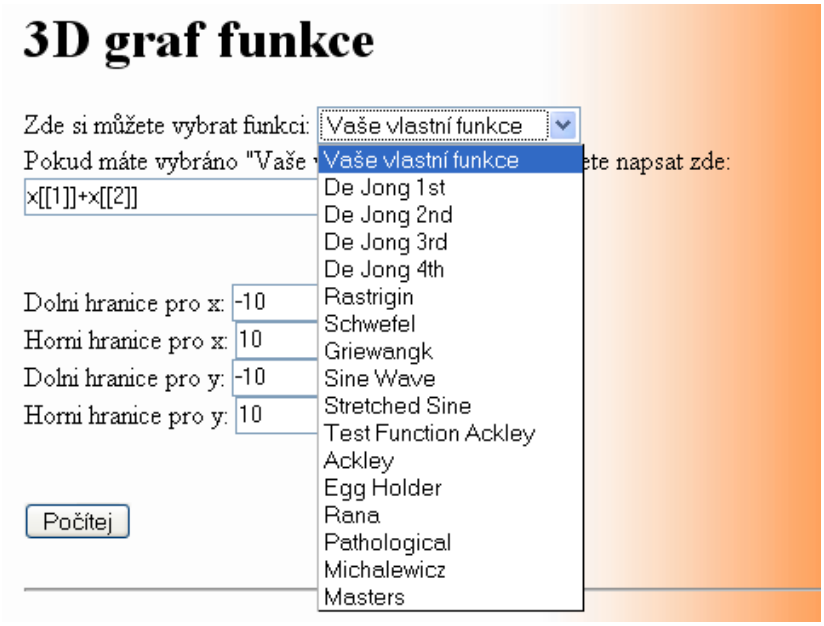
6.2 Realizace

V této části, bych chtěl předvést, jakým způsobem jsem realizoval zadaný algoritmus a jak tato realizace vypadá po grafické stránce. Předem bych ozřejmit, že design, který najdete na stránkách [7] není mým dílem a nemůžu si za něj připsat žádné zásluhy. Bylo totiž potřeba stránky s různými algoritmy sjednotit pod jeden design a můj design nevyhrál konkurz.

6.2.1 Graf účelové funkce

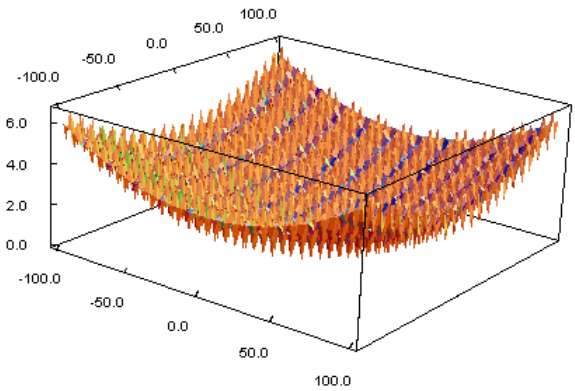
Ještě než se pustím do popisu realizace algoritmu, tak bych chtěl představit jedenu z funkcí, kterou do WebMathematicy přidal Martin Kraus [8]. Jmenuje se LiveGraphics3D a používá se k zobrazování 3D grafiky v prostředí WebMathematica. Výhoda této funkce spočívá v tom, že uživatel si může za pomoci myši otáčet grafem. Tato funkce mě natolik zaujala, že jsem se rozhodl dát uživatelům možnost zobrazit si

grafy testovacích účelových funkcí, popřípadě vlastní funkce. Na obrázku se můžete podívat na obrazovku, přes kterou se zadávají parametry pro tento graf (Obr. 15).



Obr. 15. Zadávání hodnot pro 3D graf

Kromě funkce se zadávají ještě dolní a horní meze pro obě proměnné, je jasné, že jdou vygenerovat jenom grafy účelových funkcí se dvěma argumenty. Pokud vybere jednu z předdefinovaných funkcí, tak se samozřejmě změní hranice argumentů. Výsledný graf si sice můžete prohlédnout (Obr. 16), ale neoceníte možnost prostorového otáčení, ani přibližování. Jako ukázkovou funkci jsem si zvolil Griewangkovu funkci.



Obr. 16. 3D graf - Griewangkova funkce

6.2.2 Vlastní algoritmus

Na názorných ukázkách bych chtěl vysvětlit, jak jsem řešil některé problémy při realizaci a na závěr přidám obrázek, jak to celé vypadá.

Horolezecký algoritmus by se dal rozložit do několika kroků, které se realizují za pomoci jednotlivých funkcí. Na následujících řádcích bych chtěl ukázat, jak se tento algoritmus poskládá dohromady. Nebudu uvádět, jak se proměnné zadávají a jak překládají, ale doufám, že tato ukázka pomůže pochopit blíž, jak byl algoritmus realizován. Taky upozorňuji na to, že v zájmu přehlednosti jsem v ukázkách přejmenoval některé proměnné.

Nejprve potřebujeme **počáteční bod**, ze kterého by algoritmus mohl vyjít. Ten získáme pomoci následující funkce:

```
StartPoint =

    Table[Random[Specimen[[1, i, 1]], Specimen[[1, i, 2]]],{i,

        Dimensions[Specimen[[1]]][[1]]}];
```

Tímto získáme při zavolání funkce `StartPoint` tabulku, která bude mít tolik prvků, kolik je proměnných nadefinovaných v proměnné *Specimen* (náš vzorový jedinec). Tyto prvky budou náhodně vygenerovány a za parametry pro funkci `Random` nám budou sloužit přímo prvky z proměnné *Specimen*.

Tímto získáme počáteční bod, účelovou funkci k němu získáme velmi snadno zavoláním předem nadefinované `CostFunction`. Tuto uspořádanou dvojici (*CV*,*Bod*) si uložíme do proměnné *SSS*, kam budeme ukládat všechna řešení.

Nyní je nezbytné vygenerovat okolí bodu, ve kterém budeme zkoumat hodnotu účelové funkce. Při realizaci jsem použil dvě metody pro tvorbu tohoto okolí. Myslím, že jsem je dostatečně rozepsal v předchozím textu, pojďme se nyní podívat na to, jak jsou realizovány.

```
Okoli[PC_] := Module[{P, cv, Sousede, k, AAA, s, a, i},
 P = Table[Union[Table[CheckTyp[PC[[i]] + j, i], {j, -NH, NH, Step}]]
,{i,Dimensions[PC][[1]]}];
 If[M == SousedeRand,
        SousedeRand =
            Append[Table[
                Table[P[[i,
```

```
                    Random[Integer,{1,  Dimensions[P[[i]]][[1]]}]]],
{i,Dimensions[P][[1]]}], {k, NS}], PC]; ];
      If[M == SousedeAllCross,
        SousedeAllCross =
            Append[Union[
                Flatten[
                  Table[Table[s = PC; s[[a]] = P[[a, i]];
                      AAA = s, {i, Dimensions[P[[a]]][[1]]}], {a,
                      Dimensions[P][[1]]}], 1]], PC];];
      Sousede = #1 & [M];
      Sousede = Complement[Sousede, PC];
      cv = CostFunction /@ Sousede;
      Return[MapThread[{#1, #2} &, {cv, Sousede}]];];
```

Tato funkce nám vytvoří tabulku *P*, kde jsou pro jednotlivé proměnné účelové funkce uloženy všechny možnosti, které mohou nabývat z tohoto bodu za použití kroku a omezení velikostí okolí. Například, pokud se nacházíme v bodě [1;1], *okolí = 1* a *velikost kroku = 0.5*. Pak bude v tabulce toto *P* = {{0,0.5,1,1.5,2},{0,0.5,1,1.5,2}}.

Na zvolené metodě bude záležet, co s tabulkou provedeme. Pokud použijeme metodu „kříž“, tak se nejprve za první souřadnici budou dosazovat čísla z tabulky a druhá zůstane původní a pak se za druhou souřadnici budou dosazovat čísla z tabulky a první zůstane původní. Pokud se použije metoda Náhodný čtverec, tak se všechny souřadnice náhodně vygenerují z tabulky, těchto bodů bude tolik, kolik je nadefinováno v parametru počet sousedů. Pokud by bylo v zadání, že některá z proměnných bude typu *Integer*, tak je výsledek zaokrouhlen. Pokud by se stalo, že bychom dostali číslo, které se dostalo mimo omezení vyplývající z účelové funkce, tak toto číslo by bylo vráceno zpět. O to se stará funkce `CheckTyp` již při tvorbě tabulky možných hodnot. Všimněte si použití funkce `Union` (sjednocení), která nám vyloučí zdvojené řešení a nechá vždy jen jedno. Pak se ještě za pomoci funkce `Complement` (doplněk) vyloučí původní bod. Nakonec se spočítá pro každý nový bod jeho hodnota účelové funkce a utvoří se tabulka, která se vrátí za pomoci `Return`.

Můžeme tedy říct, že pokud zavoláme `Okoli[bod]` tak jako výsledek získáme tabulku okolních bodů s jejich hodnotami účelové funkce.

Pak za pomoci funkce `FindBest`:

```
FindBest[Oko_] := Module[{CVo, Nejlepsi},
      CVo = Table[Oko[[i, 1]], {i, Dimensions[Oko][[1]]}];
```

```
Nejlepsi = Oko[[Position[CVo, Min[CVo]][[1, 1]]]];
SSS = Append[SSS, Nejlepsi];
Return[Nejlepsi[[2]]];
];
```

Získáme z této tabulky bod s nejmenší hodnotou účelové funkce, který je jak zapsán do tabulky *SSS*, tak i vrácen funkcí.

Toto označíme za jeden „skok“ a sloučíme do jedné funkce:

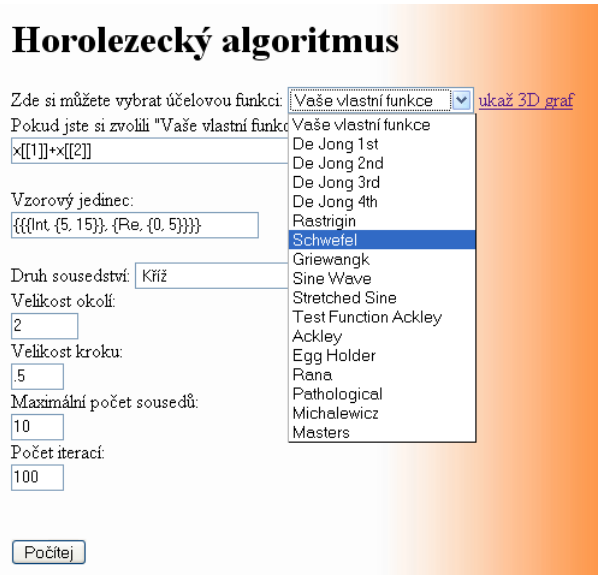
```
Skok[x_] := FindBest[Okoli[x]];
```

Všimněte si, že pokud zavoláte funkci skok s parametrem aktuálního bodu, tak jako výsledek vám vrátí bod, do kterého by se měl algoritmus přesunout.

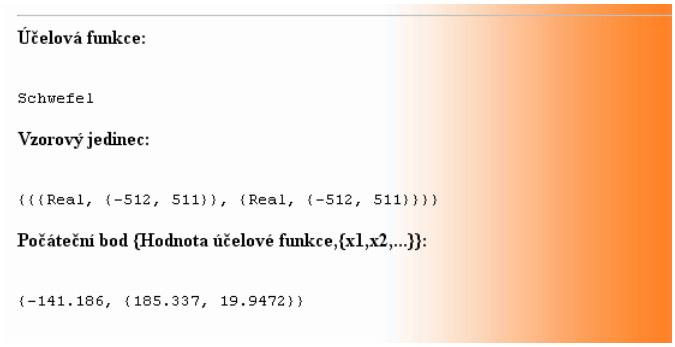
A aby se tento skok, provedl tolikrát, kolik iterací máme nastaveno, tak se funkce zavolá takto: `Nest[Skok, CP, Iterace];`

kde *CP* je bod vygenerovaný na začátku. Tato funkce nám vrátí bod, ve kterém se algoritmus zastavil. V tabulce *SSS* pak můžeme sledovat, jak algoritmus postupoval.

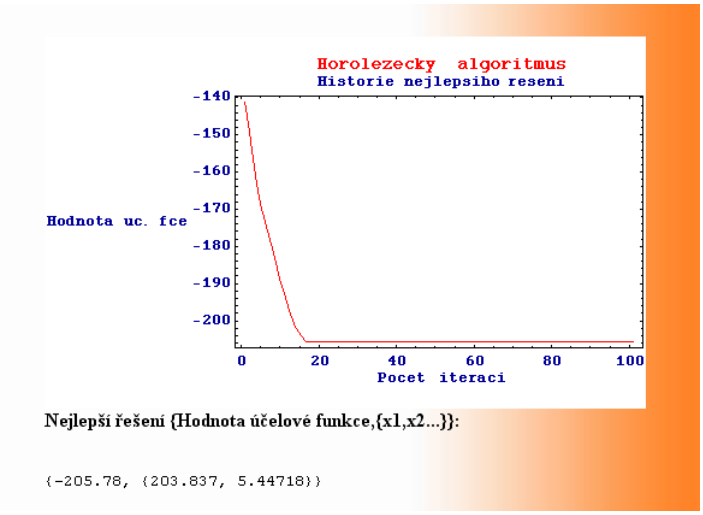
Následující obrázky by nám mohli ukázat, jak to celé vypadá na internetu. Na prvním obrázku (Obr. 17) můžete vidět, přes jaké rozhraní se zadávají vstupní parametry. Další obrázky (Obr. 18, 19) ukazují, v jakém tvaru získáme výsledek.



Obr. 17. Zadávání parametrů do horolezeckého algoritmu



Obr. 18. Výstup z horolezeckého algoritmu, první část.



Obr. 19. Výstup z horolezeckého algoritmu, druhá část.

6.3 Simulace

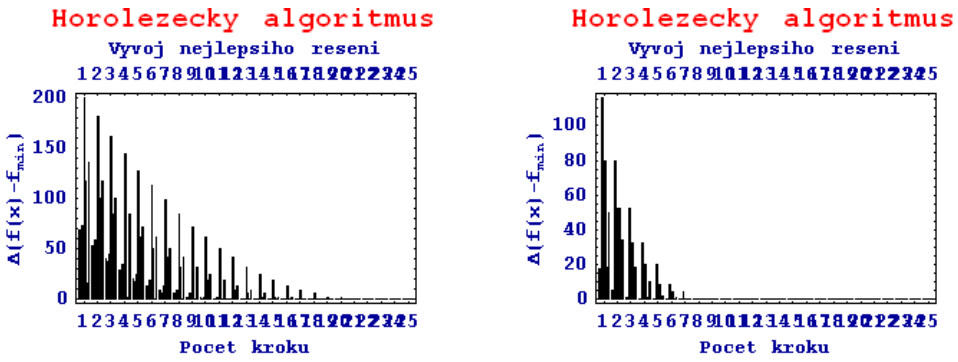
V rámci simulací bych chtěl ukázat funkčnost horolezeckého algoritmu a především vliv některých parametrů.

6.3.1 Unimodální funkce

Nejprve bych se zaměřil na velikost prohledávaného okolí a velikost kroku. Použiji nejprve První De Jongovu funkci, která nepředstavuje pro horolezecký algoritmus žádný problém.

Při simulaci použiji tohoto vzorového jedince {{{Integer,{-10,10}},{Integer,{-10,10}}}}. Jako metodu tvorby sousedů použiji metodu kříže. *Velikost okolí* nastavím na 1 a *krok* také na 1. *Počet iterací* nám bude stačit 25. Algoritmus postupně spustím

10krát a výsledky se nám ukážou v prvním grafu (Obr. 20). Z grafu je nám zřejmé, že všechny algoritmy se nakonec dostali do minima, počet iterací, za které minimum dosáhly, se liší podle počátečního bodu, který byl vygenerován náhodně. Tvar křivky, po které se nám dostanou do minima, kopíruje tvar funkce (Obr. 9). Pokud bychom použili větší okolí tak se k minimu dopracujeme rychleji, to vidíme na 2. grafu (Obr. 20), tady jsme použili dvojnásobnou velikost okolí a původní velikost kroku. Pokud bychom použili i *velikost kroku* = 2 pak bychom už počet iterací nutných pro dosažení minima nezmenšili. Čas ano, ale počet iterací ne. Navíc by spousta algoritmů nedosáhla do minima, protože by jim to znemožnila délka kroku.

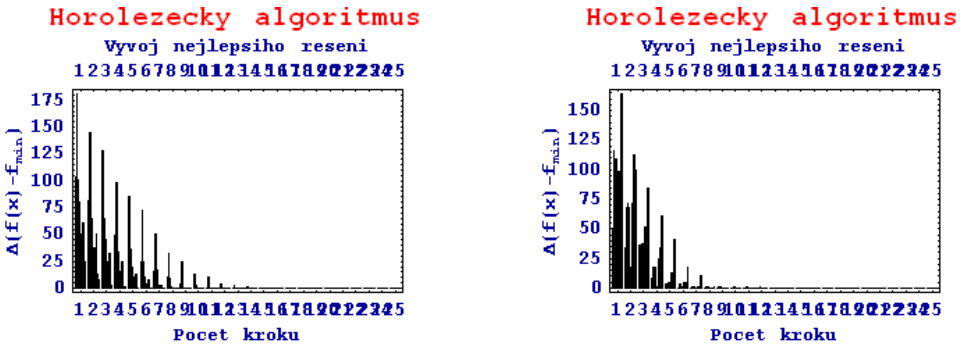


Obr. 20. Horolezecký algoritmus – vliv velikosti okolí (*velikost okolí* = 1 a *velikost okolí* = 2, *metoda pro tvorbu sousedů* = kříž)

Můžeme zkusit zobecnit a říct, že na unimodálních funkcích má velikost okolí vliv na počet iterací potřebných pro dosažení optima. Čím větší velikost okolí, tím míň je potřeba iterací na dosažení optima. Velikost kroku nám na počet iterací vliv nemá, ale má vliv na čas. Také pokud se zvolí příliš velký krok, tak dostaneme hrubý výsledek, tzn., nemusí se nám podařit dosáhnout optima, pokud bychom ale velikost kroku zvolili příliš malou, tak se nám čas potřebný pro výpočet hodně zvýší.

Zkusme provést tentýž pokus s náhodnou generací sousedů (Obr. 21). Opět vidíme, že s velikostí okolí klesá počet iterací nutných k dosažení optima. Vidíme, ale že se nám v grafech objevila nepravidelnost. Tato nepravidelnost, je způsobena stochastickým výběrem sousedů. Objeví se nám tehdy, pokud nevygenerujeme souseda s menší hodnotou účelové funkce. Pak se nám pokles pro tuto iteraci zastaví. Pokud ale vygenerujeme pouze sousedy s vyšší hodnotou účelové funkce, tak se nám hodnota účelové funkce dokonce zvýší. Další změna je v tom, že neprohledáváme okolí pouze paprskovitě vždy po jedné proměnné, ale ve čtverci. Nevím, jestli je dostatečně patrné,

ale při velikosti okolí nastavené na 2 se nám u některého algoritmu projevila tato stochastičnost i v tom, že se řešení několik iterací drželo na stejné hodnotě účelové funkce. Mohlo by se dokonce stát, že by se do optima nikdy nedostalo. Stejně tak by se tato šance zvýšila, i pokud bychom použili větší krok. Můžeme tedy říct, že pro unimodální funkce nemá stochastický způsob tvorby sousedů velký vliv, ale může se díky němu stát, že nedosáhneme globálního optima.



Obr. 21. Horolezecký algoritmus – Unimodální funkce (*velikost okolí* = 1 a *velikost okolí* = 2, *metoda pro tvorbu sousedů* = kříž)

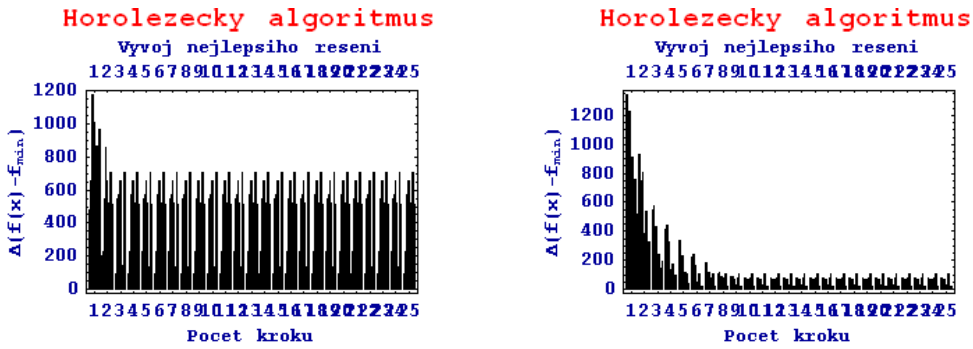
6.3.2 Multimodální funkce

Jako zástupce multimodálních funkcí jsem zvolil Rastriginovu funkci (Obr. 10). Zkusíme stejnou sadu simulací, jako jsme použili na unimodální funkci.

Pro simulace s touto funkcí jsme si zvolili vzorového jedince {Re, {-5.12,5.11}}, {Re, {-5.12,5.11}}}}. Z toho vyplívá, že budeme muset zvolit i jinou počáteční velikost okolí, abychom zhruba zachovali poměr velikosti okolí a rozsah proměnných, tak zvolím *velikost okolí* = 0.5 a *velikost kroku* = 0.25. Tyto parametry použijeme při první simulaci, při druhé simulaci zvolíme *velikost okolí* = 1. Jak nám ukazují grafy (Obr. 22).

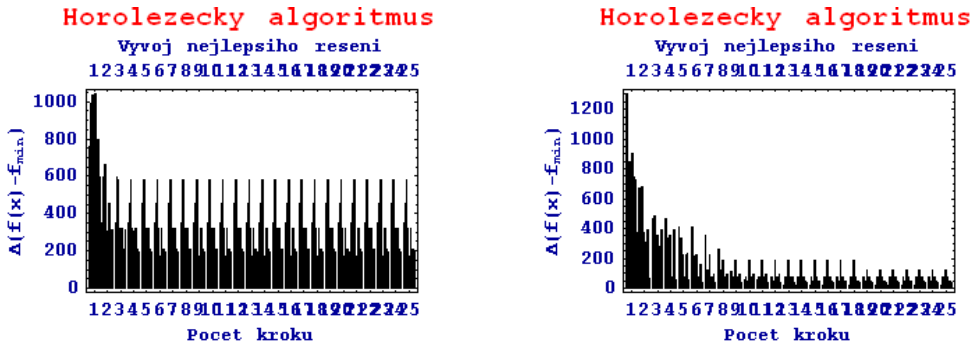
Zde se projevuje to, že horolezecký algoritmus není ideální na multimodální funkce. Sami můžete vidět, že dochází uváznutí v lokálních extrémech. Dokonce ani jednou se nám nepodařilo dosáhnout optima, které má *CV* = -400. Všimněte si prosím, že při využití většího okolí se nám algoritmus dostal sice do lokálních minim, ale s lepšími *CV*. To se nám podařilo díky tomu, že při větším okolí je větší šance, že algoritmus „přeskočí“ lokální minimum. Pokud bychom okolí ještě zvětšili, tak jo to samozřejmě patrnější. Zvyšuje se tím, ale také časová náročnost řešení. Pokud bychom zvětšovali i

krok, tak nám zase řešení „zhrubnou“ a nedostaneme se kolikrát ani do lokálního minima.



Obr. 22. Horolezecký algoritmus – Multimodální funkce (*velikost okolí* = 0.5 a *velikost okolí* = 1, *metoda pro tvorbu sousedů* = kříž)

Pojďme se nyní podívat, jak se nám bude dařit, když použijeme tytéž parametry, ale stochastickou verzi generování sousedů. V grafech (Obr. 23) můžeme vidět, že sice jsme se opět nedostali do optima, ale máme lepší výsledky než v předchozí simulaci. Při generaci „náhodných sousedů“ se totiž potencionálně důkladněji prohledává okolí. Jde opět o to, že nehledáme v kříži, ale prohledáme sousedy v čtverci a máme tudíž další šanci jak se dostat z lokálního minima.



Obr. 23. Horolezecký algoritmus – Multimodální funkce (*velikost okolí* = 0.5 a *velikost okolí* = 1, *metoda pro tvorbu sousedů* = náhodný čtverec)

6.3.3 Shrnutí

Horolezecký algoritmus lze aplikovat na unimodální funkci, kde dosáhne optima. Velikost okolí je nepřímo úměrné počtu iterací, které potřebujeme k dosažení tohoto optima, velikost kroku je zase přímo úměrná hrubosti řešení. Tudíž je vhodnější

používat menší kroky, i když je to náročnější na výpočet. Stochastičnost nemá až takový velký vliv na práci tohoto algoritmu.

Na multimodální funkci tento algoritmus přesně podle očekávání selhal. Problémem je zde uvíznutí v lokálním minimu. Při použití většího okolí jsme se dostali k lepším výsledkům. Při použití metody generování sousedů – náhodný čtverec jsme se dostali k ještě lepším výsledkům, ale opět jsme optimum nenašli.

7 SIMULOVANÉ ŽÍHÁNÍ

Stejně jako předchozí algoritmus, tak i tento jsem dostatečně popsal v teoretické části a proto se tím již nebudu zdržovat.

7.1 Parametry

7.1.1 Účelová funkce a vzorový jedinec

Jsou identicky zadávány jako v předchozím algoritmu a tak pokud předchozí kapitolu přeskočili, tak Vám nezbývá nic jiného než se tam vrátit.

7.1.2 Parametry související s chlazením

7.1.2.1 Počáteční teplota

Zde se nastaví počáteční teplota. Čím vyšší tato teplota bude, tím nám algoritmus připustí horší řešení. Je dobré nastavit tuto teplotu dostatečně velkou, abychom z počátku prohledali větší část množiny možných řešení. V odborné literatuře [2], [9] jsem se dočetl, že vhodná počáteční teplota je velmi zásadním parametrem. Pokud bude zvolena vysoká, tak se algoritmus chová více stochasticky a hledá možná řešení náhodně. Pokud bychom naopak zvolili počáteční teplotu příliš malou, pak bychom vyloučili stochastičnost a funkce algoritmu by se přiblížila k horolezeckému algoritmu. Vhodné je zvolit takovou teplotu, aby se akceptovalo zhruba 50% horších řešení. To bude samozřejmě záležet na tom, o kolik se může maximálně lišit hodnota účelové funkce stávajícího řešení od původního. To bude samozřejmě záležet na řešeném problému. Pokud se podíváme na graf (Obr. 5), tak zjistíme, že pro T_{max} = 1000 akceptoval metropolitův algoritmus s 50% pravděpodobností řešení s hodnotou účelové funkce horší až o 500. Myslím, že nic nebrání nastavit tuto hodnotu algoritmu jako implicitní. Pokud by Vás náhodou zajímaly grafy, které ukazují pravděpodobnost akceptování horšího řešení na teplotě, tak doporučuji náš web [7].

7.1.2.2 Konečná teplota

Tato teplota udává kdy, se celý algoritmus zastaví. Je rozdílná pro různé metody chlazení a proto se o ní zmíním až u metod chlazení. Pokud byste ji chtěli vyhledat bez čtení celých metod chlazení, tak je umístěna vždy na konci.

7.1.2.3 Metoda chlazení

7.1.2.3.1 Skoková metoda

Tato metoda je ve své podstatě velmi jednoduchá. Při každém chlazení se od teploty odečte určitý krok: $T := T - krok$. Tato metoda ale neodpovídá fyzikální předloze a navíc se s ní nedosahují takové výsledky. Použil jsem ji jenom jako ukázku.

S touto metodou souvisí parametr *krok*. Doporučuji jej zvolit tak, abychom získali dostatečný počet chladících cyklů. Počet cyklů si dovede každý spočítat sám, ale pro pořádek je to:

$$počet\ cyklů = (T_{max} - T_{min})/krok \tag{7}$$

Ještě zmíním konečnou teplotu T_{min} tato teplota je implicitně nastavena na 0. Můžete s ní experimentovat, ale pokud zadáte teplotu menší než 0 nebo větší než T_{max}, tak vám ji program vrátí zpět do mezí.

7.1.2.3.2 Postupná metoda

Tato metoda více kopíruje fyzikální předlohu. Její princip je také jednoduchý. Od teploty se v každém cyklu neodečítá krok, ale teplota je vynásobena multiplikátorem. Z toho je zřejmé, že multiplikátor musí být číslo menší než 1, jinak se nejedná o chlazení, ale o ohřev. Pokud bychom však na tom trvali, tak nám to algoritmus umožní, nicméně výsledky tomu budou odpovídat. Jako multiplikátor je vhodné zvolit číslo v rozmezí [0,6; 0,95]. Experimentům se ale meze nekladou. počet cyklů pak získáme ze vztahu:

$$počet\ cyklů = \log_{krok} \frac{T_{min}}{T_{max}} \tag{8}$$

Ze vzorce je jasné, že T_{min} nemůže být 0 jako u předchozí metody, protože pak bychom dostali *počet cyklů* = ∞ a to bychom se pak výsledku nedočkaly. Experimentálně jsem přišel na to, že hodnota T_{min} nastavená na 1 je vyhovující, protože je nechána poměrně dlouhá doba na jemné doladění výsledku, ale není příliš dlouhá. Pokud se někde dále setkáte se zkratkou *pk*, pak je to s velkou pravděpodobností počet cyklů.

7.1.2.4 Počet řešení pro každý cyklus chlazení

Tento parametr udává, kolik se vyzkouší různých řešení mezi dvěma chlazeními. V literatuře [2], [9] se uvádí jako optimální $10^3 – 10^5$, ale to mi přišlo zbytečně mnoho a navíc to zatěžuje server, proto jsem si dovolil implicitně nastavit tuto hodnotu na 10. Budu jej značit jako *Rep*

7.1.3 Elitismus

Dalším parametrem je *elitismus.* Vzhledem k tomu, že se jedná o checkbox, tak má možné hodnoty pouze zapnuto / vypnuto. Udává, jestli se při provádění algoritmu provede nebo neprovede elitismus.

7.2 Realizace

Stejně jako u předchozího algoritmu i zde trochu rozepíšu, jak je vlastně algoritmus realizován.

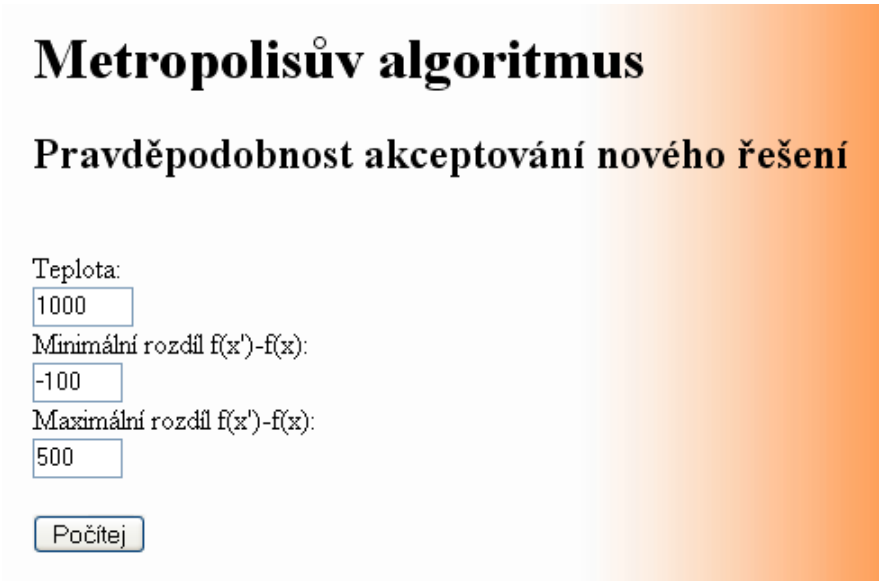
Počáteční bod se vygeneruje stejně jako v předchozím algoritmu a tak se o tom nebudu dále rozepisovat.

Vygenerujeme počáteční řešení a uložíme si jej do seznamu *B* spolu s jeho *CV*. Pak toto řešení použijeme jako parametr pro funkci `Skok`:

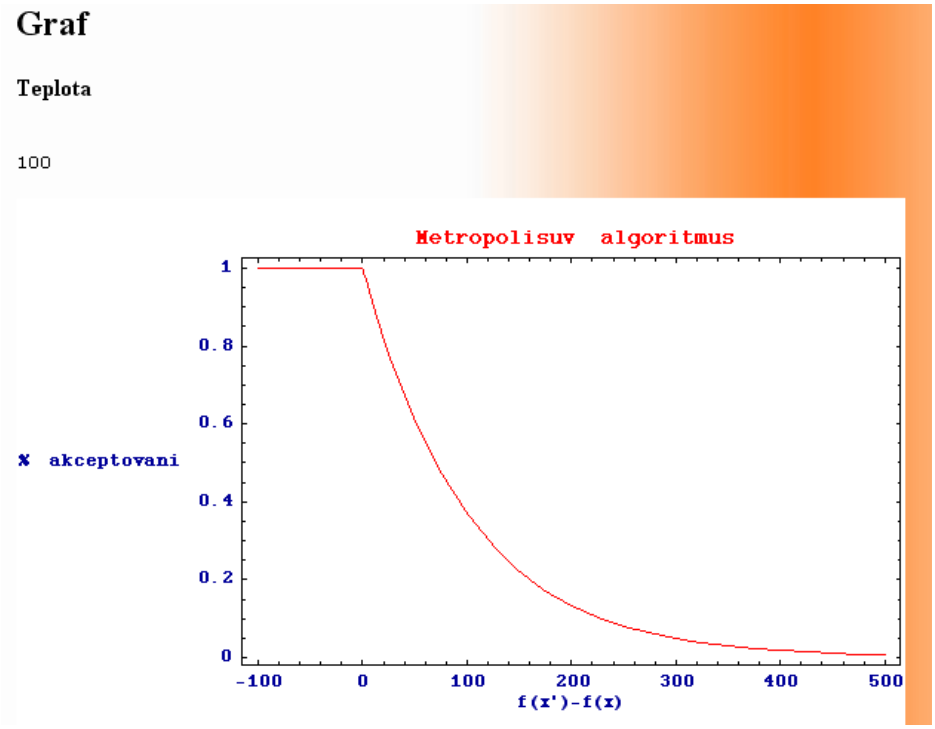
```
Skok[PC_] := Module[{k, n, C},
      n = {};
      n = StartPoint;
      k = {CostFunction[n], n};
      C = Porovnani[B[[-1]], k];
      B = Append[B, C];
      If[B[[-1,  1]]  <  Optim[[-1,  1]],  AppendTo[Optim,B[[-1]]],
AppendTo[Optim, Optim[[-1]]]];
      Return[C];
      ];
```

která nám vygeneruje jiné řešení a tato dvě řešení za pomoci funkce Porovnani porovná podle metropolisova algoritmu a do seznamu *C* nám zapíše nové řešení. Toto řešení pak porovná s posledním řešení v seznamu *Optim* a lepší z těch dvou řešení přidá do tohoto pole. Akceptované řešení se pak ještě zapíše do seznamu *B*. Seznam Optim se nám bude hodit později, až budeme uvažovat o elitismu. Zde si ještě dovolím napsat krátkou vsuvku. Považoval jsem za vhodné udělat stránku, která ukáže

pravděpodobnost akceptování nového řešení. Jako výstup nám dá graf podobný tomu, co jsme mohli vidět v teorii jako metropolitovo rozložení (Obr. 5). Jako vstup pro tento graf máme teplotu a minimální a maximální rozdíl hodnot *CV* stávajícího a nového řešení (Obr. 24). Graf nebude úplně stejný, ale podobný, podívat se můžete níž (Obr. 25).



Obr. 24. Zadávání hodnot pro graf metropolisova algoritmu



Obr. 25. Graf metropolisova algoritmu

Vraťme se ale k samotnému algoritmu. Dalo by se říct, že funkce `Porovnani` je vlastně metropolisův algoritmus. Sami se můžete přesvědčit, že je tomu tak. Tato funkce nedělá nic jiného než že

```
Porovnani[x_, y_] := Module[{reseni},
      If[x[[1]] > y[[1]] || Random[Real, {0,1}] <= E^((x[[1]] -
      y[[1]])/T),
        reseni = y, reseni = x];
      Return[reseni];
     ];
```

Když toto provedeme opakovaně, tak získáme jeden cyklus. Teď by mělo na řadu přijít chlazení. To se realizuje podle použité metody.

```
OchlazeniSkok[PP_] := Module[{},
      Nest[Skok, PP, rep];
      TC = Append[TC, T];
      T = T - krok;
      If[T <= 0, T = 0.00001];
      If[Elite == on,  {
      Delete[B, -1];
      AppendTo[B, Optim[[-1]]];
      };];
      AppendTo[Res, B[[-1]]];
      Return[B[[-1]]];
      ];

OchlazeniPostupne[PP_] := Module[{},
      Nest[Skok, PP, rep];
      TC = Append[TC, T];
      T = T*krok;
      If[T <= 0, T = 0.00001];
      If[Elite == on,  {
      Delete[B, -1];
      AppendTo[B, Optim[[-1]]];
      };];
      AppendTo[Res, B[[-1]]];
      Return[B[[-1]]];
      ];
```

Všimněte si, že každá metoda má hned na začátku cyklus, který *rep*-krát provede skok. Pak obě metody zapíší aktuální teplotu do seznamu *TC*. Pak se provede samotné chlazení, zbytek je opět stejný. V tomto zbytku se testuje, jestli se používá elitismus a jestli ano, pak dá na poslední místo seznamu *B* poslední prvek ze seznamu *Optim*. Pokud ne tak nic nemění. Funkce navíc tento prvek vrací jako svou hodnotu. Možná jste si všimnuli, že na konci ještě přidá poslední hodnotu z chladícího cyklu do seznamu *Res*, ten se využije potom při vykreslování grafů, pro graf, kde máme pouze hodnoty řešení na konci chladícího cyklu.

Pokud tuto funkci zavoláme *pk*-krát, kde *pk* si spočítáme, jak jsme si uvedli výše, tak máme celý algoritmus simulovaného žíhání.

Takže i s nastavením počátečních prvků a seznamů *B*, *Optim* a *Res* nám vlastně program bude vypadat takto:

```
CP = StartPoint;
B = {{CostFunction[CP], CP}};
AppendTo[Optim, B[[1]]];
AppendTo[Res, B[[1]]];
Nest[#1, CP, pk] &[M];
```

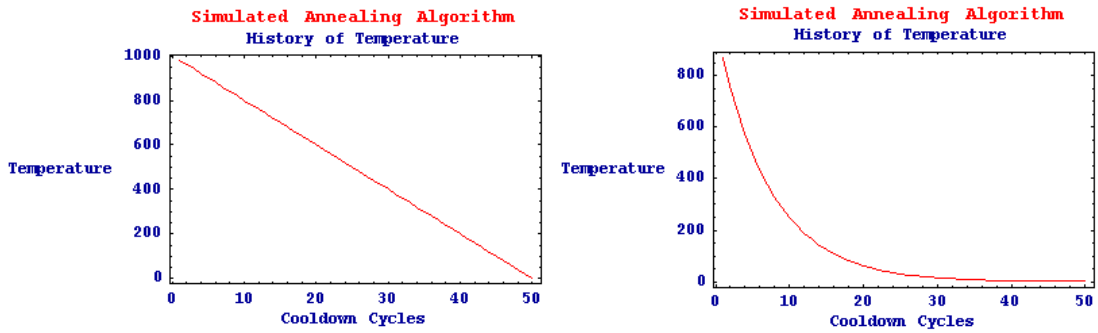
Pokud by vás zajímalo, jak bude vypadat stránka s tímto algoritmem tak musím říct, že hodně podobně jako s předchozím, jediné, co stojí za to ukázat, budou výstupy. Začátek bude i zde stejný, ale za zajímavé považuji hlavně grafy. Tyto grafy jsou ale přesně stejné jako ty zobrazené níže a proto vás nebudu nudit stejnými grafy dvakrát, kdo se na ně chce podívat tak může tam, kde jsou. Jsou to grafy, které ukazují průběh teploty (Obr. 26), vývoj všech řešení (Obr. 27), které můžeme najít v seznamu *B* a řešení po chladícím cyklu, která jsou v seznamu Res (Obr. 28).

7.3 Simulace

Nejprve bych se v simulacích zaměřil na průběhy teplot při použití různých metod chlazení. Pak bych zkusil předvést, jaké výsledky nám bude algoritmus poskytovat při použití těchto metod chlazení při aplikaci na unimodální a multimodální funkci. A nakonec bych to celé zkusil ještě jednou s využitím elitismu.

7.3.1 Metody chlazení

Pro obě metody chlazení použijeme T_{max} = 1000. T_{min} bude pro skokovou metodu 0 a pro postupnou metodu 1. Krok se bude také lišit, pro skokovou metodu bude 20 a pro postupnou 0.87 možná vás udiví tyto hodnoty, ale byly zvoleny tak, aby obě metody vygenerovaly *pk* = 50.



Obr. 26. Vývoj teploty u simulovaného žíhání při využití skokové metody (vlevo) a postupné metody (vpravo)

Vzhledem k tomu, že se *počet cyklů* určuje podle zadaných parametrů, tak při změně ostatních parametrů se změní i *počet cyklů* a průběh bude podobný. Je jasné, že pokud u skokové metody budeme zvětšovat *krok* tak se nám musí zmenšit *počet cyklů* a naopak. Pokud budeme u postupné metody zvyšovat *multiplikátor* (limitně až k 1) tak se nám bude *počet cyklů* zvyšovat a naopak (limitně až k 0).

7.3.2 Simulované žíhání bez elitismu

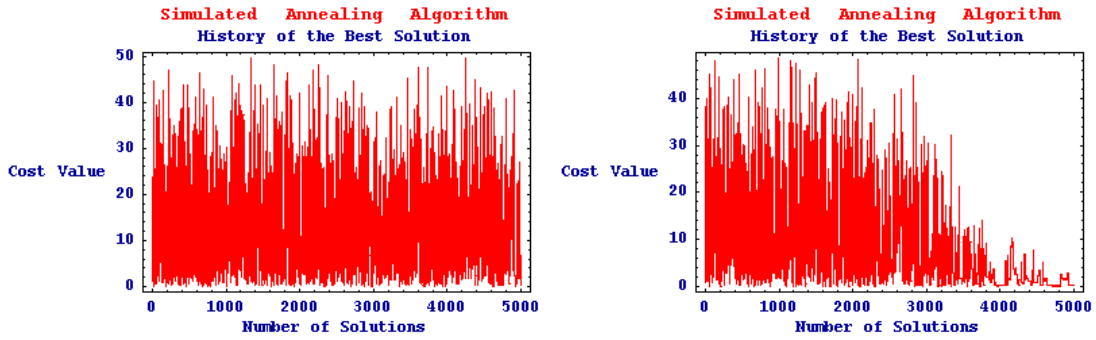
Při simulacích budu využívat nastavení, které bylo uvedeno již při simulaci průběhu teploty. Jenom ještě uvedu, že *pk* = 100

7.3.2.1 Unimodální funkce

Opět použiji De Jongovu první funkci, kdo by se na ni chtěl podívat tak graf (Obr. 9) je k dispozici výše.

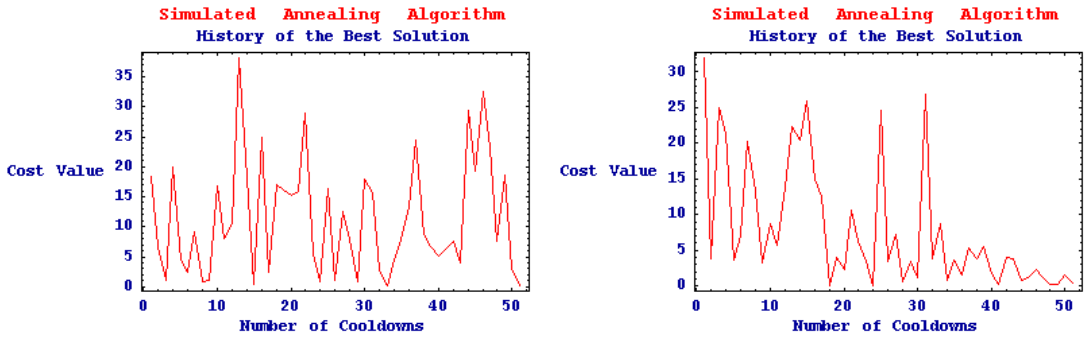
Původně jsem sem chtěl dát opět grafy, kde by se ukázalo, jak si algoritmy vedly při jednotlivých opakování, ale tyto grafy jsou velmi nepřehledné a neposkytují téměř žádnou informaci, proto uvádím pouze grafy pro jeden průběh. Na to bych rád upozornil, protože pokud to nebudeme mít neustále na zřeteli, tak nás grafy mohou svádět k tomu, že tyto hodnoty budeme považovat za průměrné a budeme si nalhávat,

že algoritmus se vždy ustálí na jediné hodnotě. Není tomu tak, tyto grafy jsou pouze reprezentativní grafy pro jediné opakování. Tyto grafy nám umožní udělat si obrázek mezi jednotlivými metodami. Když se podíváme na graf (Obr. 27), tak vlevo uvidíme s využitím skokového ochlazení a vpravo pak s postupným ochlazováním. Tento graf nám ukazuje všechna vyzkoušená řešení, další graf (Obr. 28) nám pak ukazuje pouze konečná řešení před dalším chlazením.



Obr. 27. Průběh simulovaného žíhání, všechna řešení, vlevo skokové ochlazování, vpravo postupné ochlazování

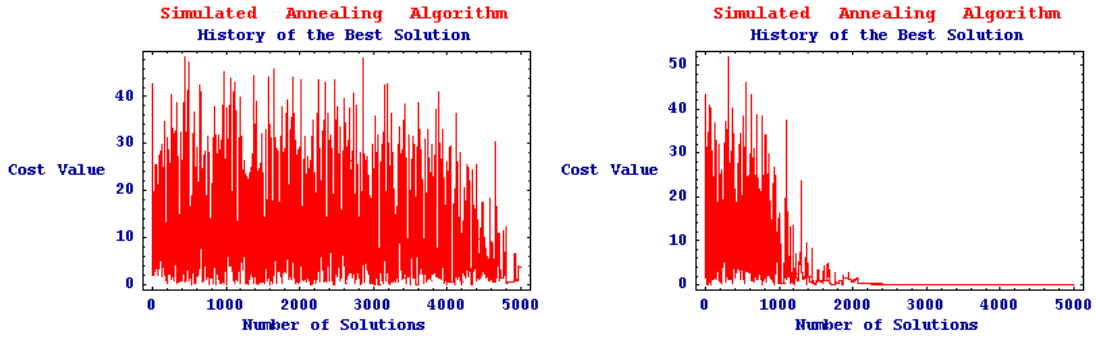
Z grafu (Obr. 27) bychom mohli získat pocit, že nám skoková metoda nedala použitelný výsledek, ale když se podíváme na další graf (Obr. 28) tak zjistíme, že to není tak úplně pravda, poslední 3 chladící cykly nám poskytly docela rozumné řešení, u jiných simulací, se ale stejně občas stalo, že se algoritmus nedobral optima. U metody postupného chlazení je patrné, že zhruba posledních 10 cyklů nám zjemňuje již získané řešení, ale i zde se občas dostaneme k horším výsledkům než v předchozím cyklu.



Obr. 28. Průběh simulovaného žíhání, pouze řešení před dalším ochlazováním, vlevo skokové ochlazování, vpravo postupné ochlazování

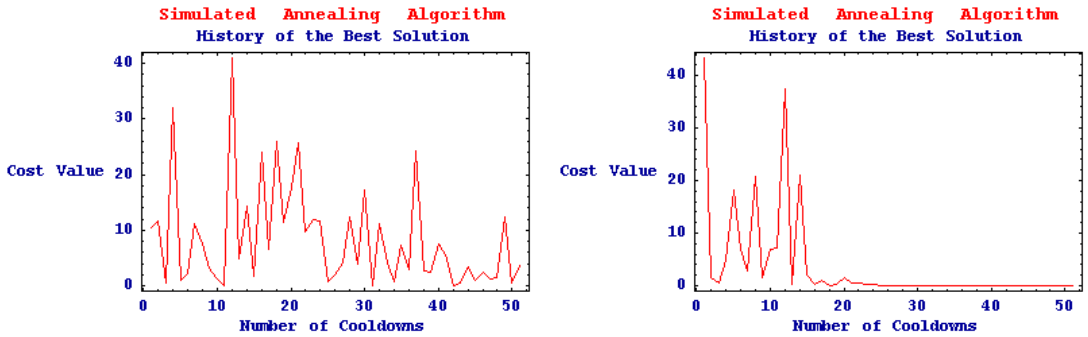
Postupné ochlazování nám dalo výsledky lepší, ale stejně můžeme pozorovat, že se často v dalším chladícím cyklu objevilo horší řešení, které bylo akceptováno, je to díky tomu, že máme malé rozpětí *CV*.

Pokud bychom chtěli zkvalitnit tato řešení, pak bychom museli změnit počáteční teplotu a krok. Jak nám to ukazuje další sada grafů. Nastavení jsem změnil takto, *Tmax* = 100, *Tmin* u skokového chlazení je 0 u postupného 0,001, *krok* je 2 a *multiplikátor* zaokrouhleně 0,8. Snahou bylo, abychom dosáhli stejného *pk*, aby se dalo porovnat nastavení parametrů.



Obr. 29. Průběh simulovaného žíhání, všechna řešení, vlevo skokové ochlazování, vpravo postupné ochlazování, snížena T_{max} a *krok*/*multiplikátor*

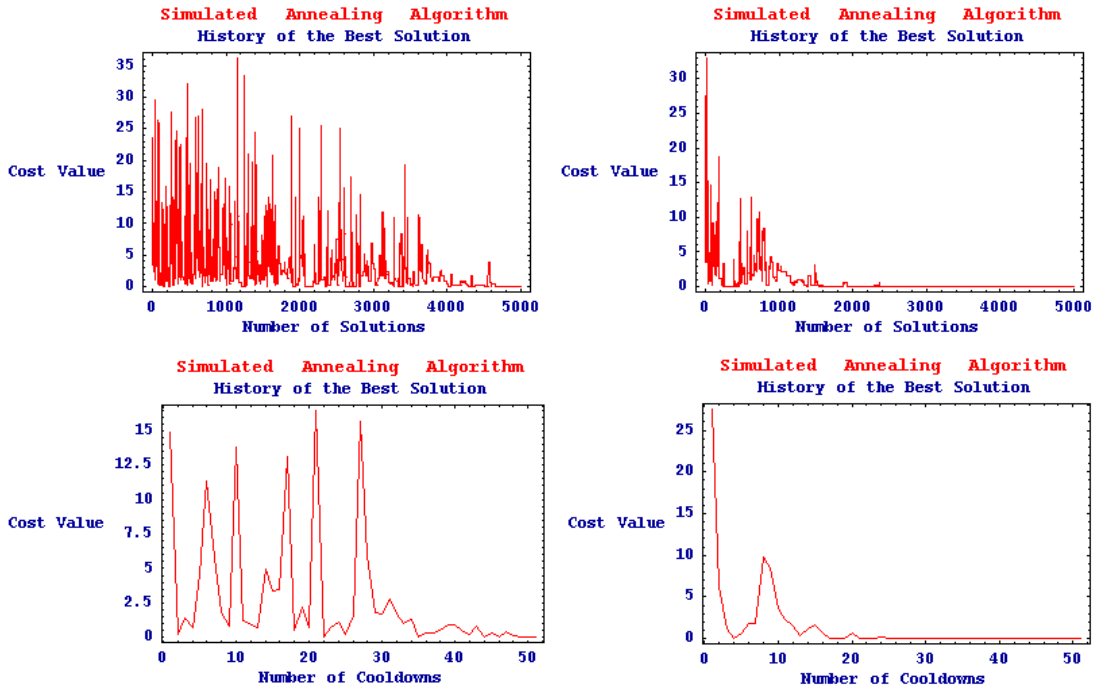
Zjistíme, že toto zjemnění skokové metodě sice pomohlo, ale nijak zvláště. Většinu času znovu prohledává náhodně celou množinu řešení a pouze v posledních 5 krocích, zjemňuje získané řešení, ale opět se můžeme přesvědčit, že i zde dostáváme i horší řešení než v předchozích cyklech. Postupné ochlazování si vede nesrovnatelně lépe, neboť zhruba posledních 30 cyklů je věnováno jenom zjemňování řešení (Obr. 30).



Obr. 30. Průběh simulovaného žíhání, pouze řešení před dalším ochlazováním, vlevo skokové ochlazování, vpravo postupné ochlazování, snížena T_{max} a *krok*/*multiplikátor*

Zkoušel jsem pak ještě řadu nastavení, kde cílem bylo najít takové, aby i skoková metoda poskytovala slušné výsledky. Povedlo se mi to při nastavení T_{max} = 10 a *krok* = 0.2, multiplikátor = 0,83 Pak nám algoritmus poskytl tyto výsledky (Obr. 31).

Ze simulací můžeme říct, že snižování počáteční teploty a zjemňování kroku má vliv na to, po jaké době se začne upřednostňovat získané řešení. Stochastická část je potřebná proto, abychom se vyhnuli uváznutí v lokálním extrému, což u unimodální funkce nehrozí, proto můžeme nastavit tyto hodnoty.

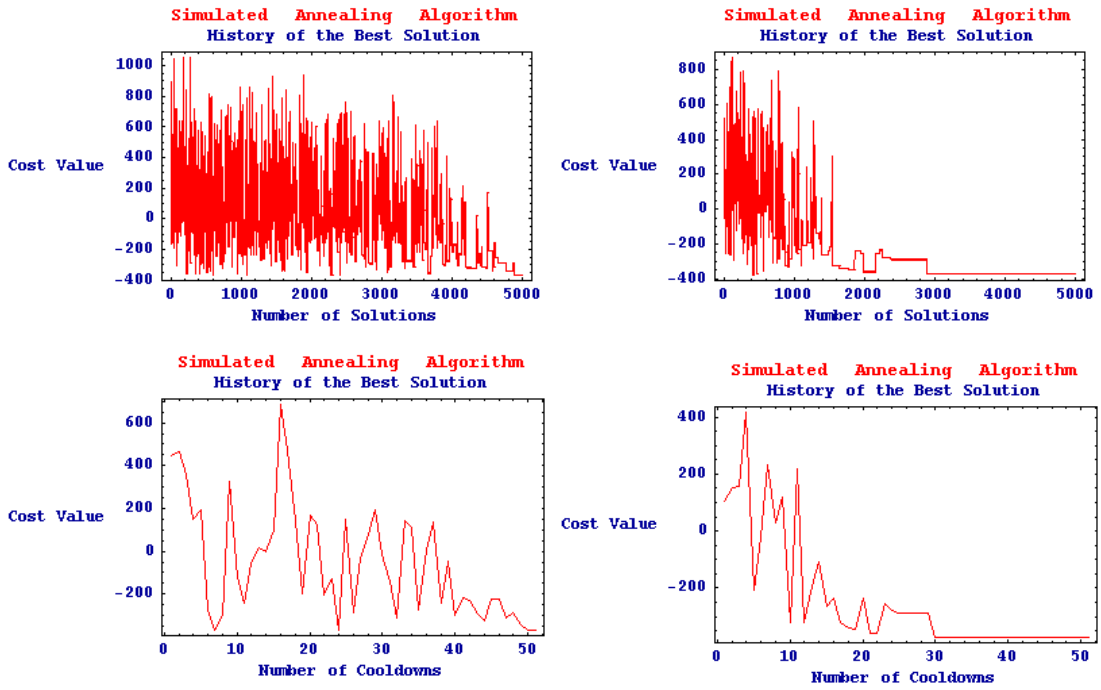


Obr. 31. Průběh simulovaného žíhání, opět snížena T_{max} a *krok*/*multiplikátor*, vlevo skokové ochlazování, vpravo postupné ochlazování, nahoře všechna řešení dole pouze řešení před ochlazováním

7.3.2.2 Multimodální funkce

Opět jsem použil Rastriginovu funkci (Obr. 10). Parametry jsem po předchozích simulacích nasadil tyto: *Tmax* = 1000, *Tmin* u skokového chlazení je 0, u postupného 1, *krok* je 20 a *multiplikátor* zaokrouhleně 0,87.

Následující grafy (Obr. 32) nám ukazují, jaké výsledky nám algoritmus poskytnul. Výsledky jsou opět ve známém tvaru.



Obr. 32. Průběh simulovaného žíhání, multimodální funkce

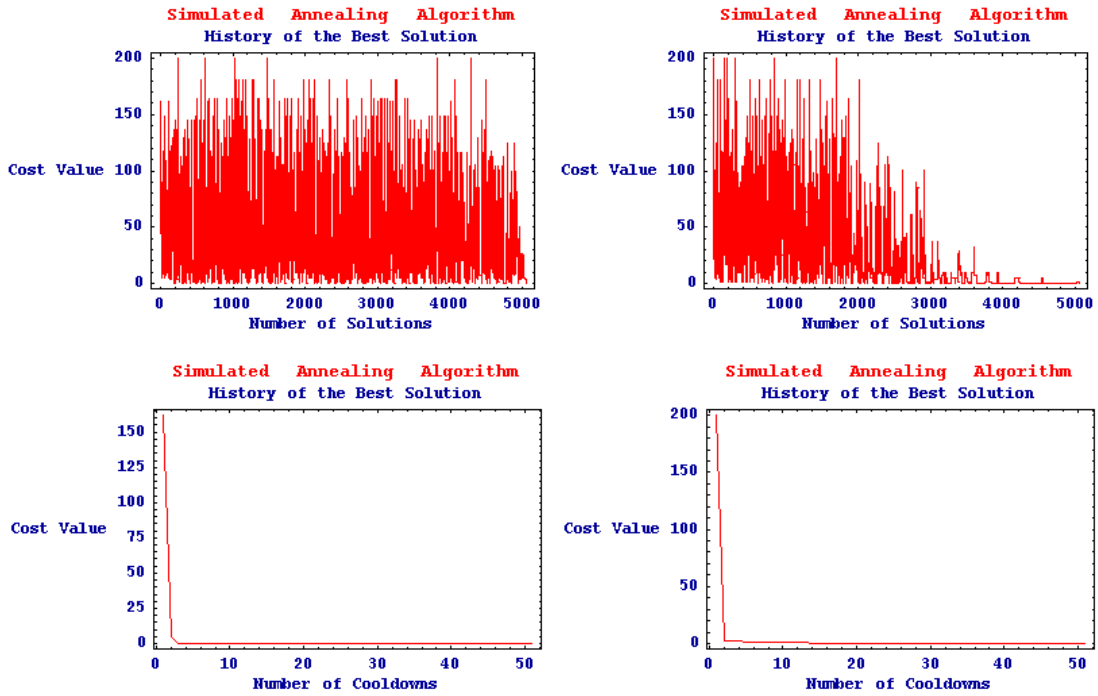
Z grafů je zřejmé, že tentokrát jsme teplotu a ostatní parametry vhodně zvolili již na počátku. Skoková metoda opět většinu času prohledává náhodně, ale na konci se přiblíží podobným řešení jako postupná metoda. Za všimnutí stojí, že algoritmu nevadí multimodální funkce. Pokud bychom chtěli srovnávat s unimodální funkcí (Obr. 29,30), tak zjistíme, že průběhy jsou podobné. Ještě bych chtěl upozornit na cykly 22-30 u postupné metody, kde došlo ke krátkodobému uvíznutí v lokálním extrému, odkud se algoritmus ale dostal. Dá se říct, že správné nastavení parametrů má na algoritmus větší vliv než průběh funkce.

7.3.3 Simulované žíhání s elitismem

Zavedení elitismu by teoreticky mělo vyhladit průběh grafů řešení a navíc by nám mělo poskytnout kvalitnějších řešení. Zkusme se o tom přesvědčit

7.3.3.1 Unimodální funkce

Mějme shodné parametry jako při simulacích bez elitismu. Tzn., *Tmax* = 1000, *Tmin* = 0 a 1, *krok* = 20 a multiplikátor = 0.87. A shodnou funkci, kterou je De Jongova první funkce (Obr. 9).

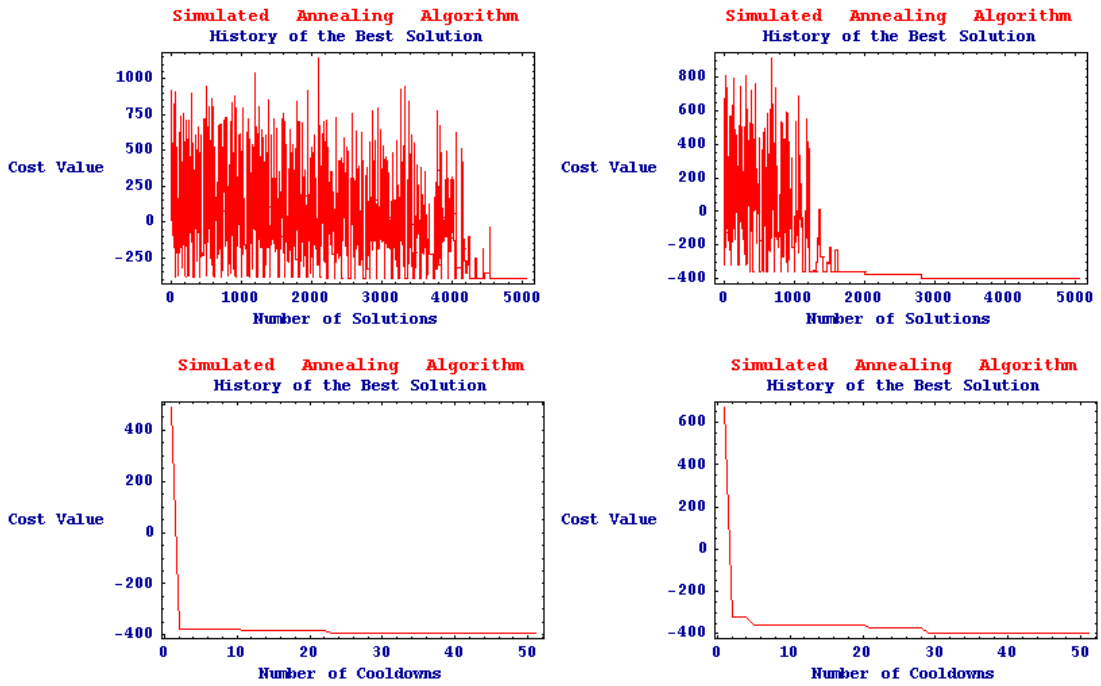


Obr. 33. Průběh simulovaného žíhání, unimodální funkce, elitismus

Pokud se vrátíme zpět a prohlédneme si grafy, které jsme získali se stejně nastavenými parametry (Obr. 27, 28), tak zjistíme, že při porovnávání všech zkoumaných řešení (horní grafy) nezjistíme téměř žádnou odchylku, ale při porovnávání průběhu řešení před ochlazováním zjistíme diametrální rozdíl. Můžeme z toho vyvodit závěr, že i pokud nezvolíme parametry vhodně, pak, za použití elitismu získáme kvalitní řešení (Obr. 33).

7.3.3.2 Multimodální funkce

Opět mějme stejné parametry a funkci jako při zkoumání multimodální funkce bez využití elitismu a pouze nyní využijme elitismus: Rastriginova funkce (Obr. 10). Parametry tyto: *Tmax* = 1000, *Tmin* u skokového chlazení je 0, u postupného 1, *krok* je 20 a *multiplikátor* 0,87.



Obr. 34. Průběh simulovaného žíhání, multimodální funkce, využití elitismu

Průběhy všech řešení (Obr. 34) jsou opět podobné jako u simulací bez využití elitismu, ale rozdílné průběhy pro řešení před ochlazením. Pokud si všimnete, tak algoritmus několikrát uváznul v lokálním extrému, odkud se ale zase dostal. Za zmínku stojí fakt, že spojité ochlazování mělo těchto uvíznutí vždy více než skokové. Je to způsobeno tím, že skokové ochlazování prohledávalo „vzdálenější“ řešení.

7.3.4 Shrnutí

Pokud nebudeme využívat elitismus, pak si musíme lépe pohlídat parametry, teda pokud nechceme, aby většina práce algoritmu vyšla nazmar a počítalo se pouze s posledními 3-10 cykly. Pokud elitismus použije, tak algoritmus není tak citlivý na nastavení parametrů, Algoritmus podává dobré výsledky na multimodální funkci stejně jako na funkci unimodální. Nedosáhne však vždy optima, což je pochopitelné.

8 ROZPTYLOVÉ HLEDÁNÍ

Při realizaci tohoto algoritmu jsem sice vycházel z toho, co jsem naprogramoval již dříve na bakalářskou práci, ale byl jsem nucen většinu kódu přepsat, protože s tím, jak jsem se naučil trochu lépe pracovat v Mathematice, tak se mi původní kód zdál nedostatečný. Další rozdíl spočívá v tom, že původně jsem používal pro zlepšování řešení (lokální optimalizace) funkci *findminimum* která je obsažena přímo v Mathematice. Vzhledem k tomu, že jsem nyní programoval více algoritmů, tak nyní jsem využil pro tuto optimalizaci horolezeckého algoritmu, jak jsem vám jej představil již výše. K tomuto kroku jsem přistoupil hlavně proto, že mi přijde demonstrativnější použít něco, s čím se případný zájemce může seznámit než nějakou zabudovanou funkci.

8.1 Parametry

Parametry u realizace tohoto EA si můžeme rozdělit na dvě hlavní skupiny. V první skupině jsou parametry samotného SS a do druhé skupiny jsem umístil parametry pro algoritmus lokální optimalizace (horolezecký).

8.1.1 Základní parametry

8.1.1.1 Účelová funkce a vzorový jedinec

Tyto dva parametry již nebudu rozvádět. Jsou stejné jako v předchozích dvou algoritmech.

8.1.1.2 Počáteční populace

Tento parametr udává, jak má být velká počáteční populace. Ve shodě s tvůrci tohoto algoritmu [3] jsem nechal implicitně nastavenou hodnotu 10, což je pro tento algoritmus dostatečné, jak jsme se již dozvěděli v teoretické části.

8.1.1.3 Počet iterací

Zde můžeme nastavit, kolikrát se má provést tvorba nových zkušebních řešení. Jistě je každému jasné, že čím výš nastaví tento parametr, tím lepší řešení může získat, ale také se tím prodlouží čas potřebný pro běh programu.

8.1.1.4 Velikost referenční množiny

Jak si možná pamatujete z teoretické části, tak referenční množina se skládá ze dvou částí. První část je referenční množina nejlepších řešení podle *CV* a druhá část je referenční množina nejlepších řešení podle vzdálenosti od ostatních řešení. Oba tyto parametry můžeme měnit. Jako implicitní jsem nastavil 3 pro první část referenční množiny a 2 pro druhou část množiny.

8.1.2 Parametry pro lokální optimalizaci

Zde se zadávají parametry stejně jako v horolezeckém algoritmu, proto zde uvedu jen hrubý výčet:

- Metoda výběru sousedství
- Velikost sousedství
- Velikost kroku
- Počet iterací

8.2 Realizace

Jak jsme si již řekli, tak tento algoritmus velmi dbá na různorodost a nejinak je tomu i množiny počátečních řešení. Proto je potřeba specielní generátor řešení:

```
StartPopulation := Module[{freq, ifrek, ctvrt, dolnih, kv, A, B},
      ctvrt =
        Table[N[(Specimen[[1, i, 2, 2]] - Specimen[[1, i, 2, 1]])/4],
{i,Length[Specimen[[1]]]}];
      dolnih    =    Table[Specimen[[1,    i,    2,    1]],    {i,
Length[Specimen[[1]]]}];


      gen := {
          freq[[5]] = Apply[Plus, Delete[freq, 5]];
       ifrek = Table[freq[[5]] - freq[[i]], {i, 4}];
        AppendTo[ifrek, Apply[Plus, ifrek]];
          RandomSeed[];
          r = Random[Integer, {1, ifrek[[5]]}];
          P = Table[r = r - ifrek[[i]], {i, 4}];
          Table[If[P[[i]] > 0, P[[i]] = 1000], {i, 4}];
          r = Position[Abs[P], Min[Abs[P]]][[1, 1]];
          freq[[r]]++;
          AppendTo[kv, r];
```

```
          };

      freq = ifrek = Table[0, {5}];
      kv = {};
      RandomSeed[];
      r = Random[Integer, {1, 4}];
      freq[[r]]++;
      AppendTo[kv, r];
      Table[gen, {Dim*pop - 1}];

      A =
        Table[Table[
            dolnih [[i]] + (kv[[i + j*Dim]] - 1)*ctvrt[[i]] +
              Random[Real, {0, ctvrt[[i]]}], {i, Dim}], {j, pop -
1}];
      Table[
        If[Specimen[[1, j, 1]] == Integer,
          Table[A[[i, j]] = Round[A[[i, j]]], {i, pop - 1}]], {j,
          Dim}];
      B = Table[ {CostFunction[A[[i]]], A[[i]]}, {i, pop - 1}];
      Return[B];
      ];
```

Zkusím jej trochu popsat. Jistě jste si všimli proměnných *freq* a *ifrek* tak můžeme začít od nich. Jak jsem již psal. Nejprve jsou vygenerována kvadranty a potom k nim jsou teprve degenerovány přesné hodnoty. Každý interval rozdělíme na čtyři intervaly, velikost těchto intervalů se nám jako tabulka zapíše do proměnné *ctvrt* do proměnné *kv* uložíme tolik těchto kvadrantů, aby nám vystačily pro všechny parametry všech počátečních řešení. Pokud Dim bude počet argumentů účelové funkce a pop velikost počáteční populace, tak jich vygenerujeme *Dim* × *pop*. Již zmíněné proměnné freq a ifrek nám pomohou s tím, aby bylo pokud možno rovnoměrné rozdělení všech čtyř kvadrantů. Pracuje to následovně. Na začátku jsou *ifrek* i *freq* stejné jedná se o pole o pěti prvcích, kde všechny prvky jsou rovny 0. Vygenerujeme první kvadrant jako Integer od 1 do 4. Pak inkrementujeme tolikátý prvek pole *freq*, kolikátý kvadrant jsme získali. Řekněme, že jsme vygenerovali číslo 3, pak se původní hodnota *freq* změní na {0,0,1,0,0}. Na páté místo umístíme součet prvních čtyř a tak získáme *freq* = {0,0,1,0,1}. Teď si spočítáme *ifrek*. Na 1-4 místě tohoto pole bude rozdíl posledního prvku pole *freq* – stejného pole *freq* jako počítáme *ifrek*. Možná je to trošku

krkolomněji napsané, ale pokud to provedeme, tak získáme *ifrek*={1,1,0,1,0}, na poslední místo opět umístíme součet předchozích prvků a tak získáme *ifrek*={1,1,0,1,3}. Teď budeme generovat číslo typu integer v rozsahu od 1 do 3. Toto číslo si označíme *r*. Uděláme tabulku, která se jmenuje *P*, a umístíme do ní rozdíl mezi *r* a daným prvkem *ifrek*, je to ještě komplikovanější, protože na další místo se nedostane celé *r*, ale pouze tento rozdíl, to provedeme pouze pro první 4 prvky. Řekněme, že se nám podařilo vygenerovat číslo 2. Pak tabulka *P* bude vypadat takhle *P*={1,0,0,-1}. Nyní dáme všechny prvky větší než 0 rovny 1000 a dáme všechny prvky do absolutní hodnoty. Takže získáme *P*={1000,0,0,1}. Teď budeme hledat první nejmenší číslo, vidíme, že je na 2 pozici. Tím máme vygenerován kvadrant 2. Vidíme, že jsme neměli šanci vygenerovat kvadrant 3, protože ten už byl vybrán jako první a byl penalizován. Možná je to popsáno trošku nemotorně, ale doufám, že s kódu je to jasné, najdete to jako funkci `gen`.

Tím máme vygenerovány kvadranty s velmi slušným rozdělením. Teď se bude generovat, o jaké přímo číslo se v kvadrantu jedná, tím získáme už přímo argument účelové funkce a uložíme je seznamu *A*. Ještě je otestujeme, jestli se náhodou nejedná o typ integer a pokud ano, tak je zaokrouhlíme. Přidáme k nim jejich *CV* a zapíšeme je do seznamu *B*.

Tím máme vygenerovanou počáteční populaci. Každý bod této populace vložíme jako počáteční bod pro horolezecký algoritmus a ten nám vrátí seznam se zlepšenými řešeními.

Odtud vybereme *b1* nejlepších řešení a vložíme je do *RefSet1*. Pak za pomoci funkce následující funkce:

```
Vzdalenost[x_, y_] :=
    Sqrt[ Sum[Power[ x[[i]] - y[[i]], 2], {i, Dimensions[x][[1]]}]];
```

Spočítáme vzdálenost mezi prvky v *RefSet1* a ostatními. Ten s největší vzdáleností umístíme do *RefSet2*, znovu přepočítáme a přidáme ještě vzdálenost k tomuto prvku v *RefSet2*. Prvek s největší vzdáleností znovu přidáme do *RefSet2*, to opakujeme, dokud těchto prvků nebude *b2*.

Tím jsme získali referenční množinu nejlepších řešení.

Pak vezmeme všechny možné dvojice z této množiny a budeme je kombinovat. Pravidla pro kombinaci nám ukáže následující kousek programu:

```
Kombinuj[r_] := Module[{rp1, rp2, v},
      rp1 = Position[Bref, r[[1]]] [[1, 1]];
      rp2 = Position[Bref, r[[2]]] [[1, 1]];
      If[rp1 <= Ref1 && rp2 <= Ref1,
        AppendTo[Pomocna, {C2[r], C1[r], C3[r], C2[r]}],
        If[rp1 <= Ref1 || rp2 <= Ref1,
          AppendTo[Pomocna, {C1[r], C2[r], C3[r]}] ,
          If[rp1 > Ref1 && rp2 > Ref1,
            AppendTo[
              Pomocna, {C2[r],
                If[Random[Integer, {0, 1}] == 0, C1[r],C3[r] ]}]]]];
      ];
```

Tato funkce dostane jako parametr seznam, ve kterém se bude vyskytovat tato dvojice řešení jako seznam dvou vektorů včetně účelové funkce. Zjistí, jestli jsou obě z *RefSet1* jestli ano pak s nimi provede 2×C2, 1×C1 a 1×C3, z kódu můžete sami zjistit, co se stane v ostatních případech a pokud se vám to nechce zjišťovat, tak si můžete nalistovat zpět a v teorii si to najít.

Co ale dělají funkce C1 – C3? To sice můžete také najít v teorii, ale taky v následujícím kódu:

```
DD[reseni_] := Random[Real, {0, 1}]*(reseni[[2]] - reseni[[1]])/2;
C1[res_] := res[[1]] - DD[res];
C2[res_] := res[[1]] + DD[res];
C3[res_] := res[[2]] + DD[res];
```

Na této množině řešení opět provedeme zlepšení za pomoci horolezeckého algoritmu. Vzniklá řešení sjednotíme s *RefSet1* a *RefSet2,* tím se tyto množiny vyprázdní. *b1* nejlepších opět poputuje do *RefSet1* a pro zbytek se spočítají vzdálenosti podle nich se opět *b2* nejlepších umístí do *RefSet2* Když tento postup budeme opakovat *Iterace*-krát, tak máme hotový algoritmus.

8.3 Simulace

Tento algoritmus by měl být ze všech tří algoritmů nejúčinnější. Zkusíme jej aplikovat na stejné účelové funkce jako předchozí algoritmy, u některých těchto funkcí zkusíme i změnit parametry a podíváme se na to, co nám to udělá.

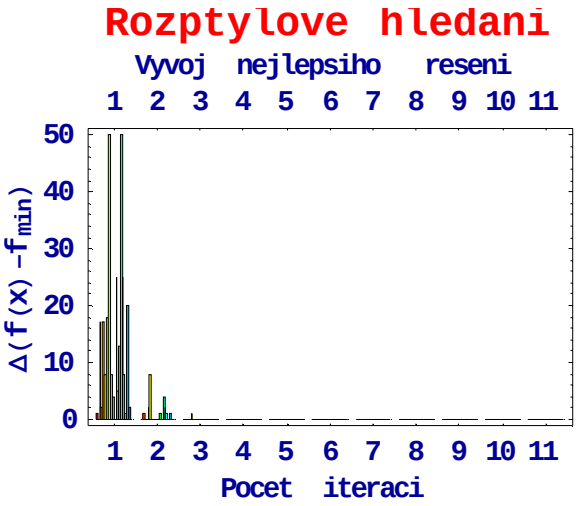
8.3.1 Unimodální funkce

S touto funkcí (Obr. 9) jsme se již setkali v předchozích EA, kde nám reprezentovala unimodální funkce. Nejinak tomu bude i zde.

SS algoritmus spustíme na této funkci 25krát, a budeme sledovat, jak si s ní poradí. Parametry algoritmu budou tyto: P_{size} = 10, b_1 = 3, b_2 = 2, *MaxIter* = 10 a pro lokální optimalizaci to bude *metoda* = kříž, *sousedství* = 1 a *krok* = 0,5.

Jak si s tím algoritmus poradí, se můžeme přesvědčit na grafu (Obr. 35).

Sami jistě vidíte, že tato funkce není pro náš algoritmus žádným oříškem a v optimu se nalézáme již po 3. iteraci. Další simulace s touto funkcí považuji za zbytečné.

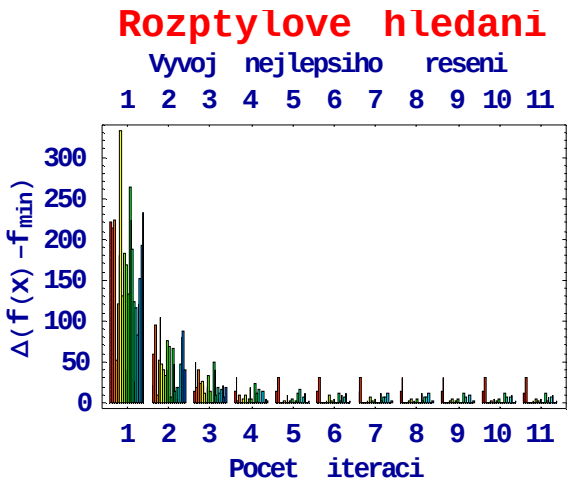


Obr. 35. Úspěšnost SS algoritmu při optimalizace De Jongovy 1. funkce

8.3.2 Multimodální funkce

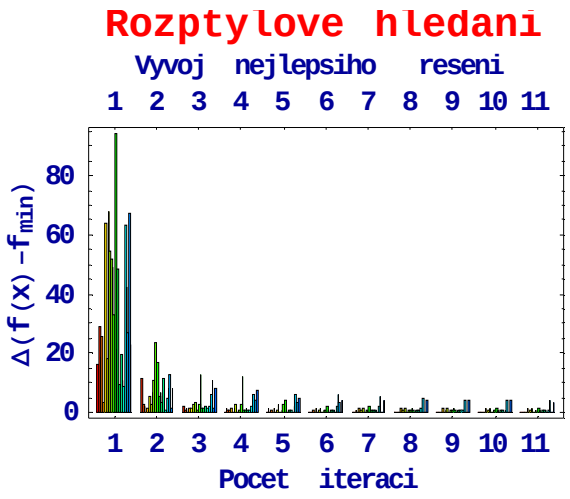
I s touto funkcí (Obr. 10) jsme se již při simulacích setkali, zastupovala nám multimodální funkce.

Parametry algoritmu necháme nastaveny stejně, jako v předchozí simulaci a můžeme se podívat, jak si poradí s touto funkcí.



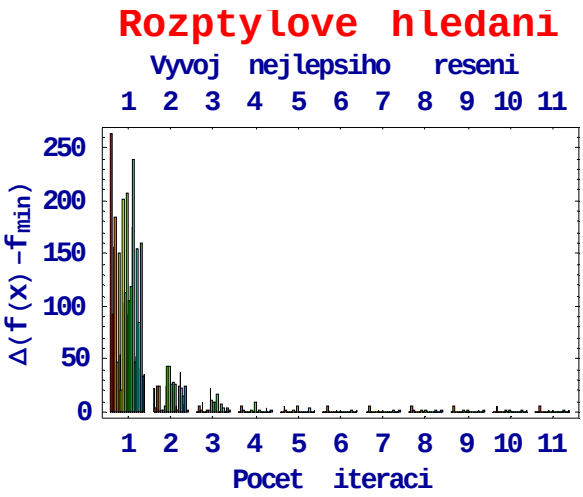
Obr. 36. Úspěšnost SS algoritmu při optimalizace Rastriginovy funkce, 1. nastavení

Sami můžete vidět (Obr. 36), že při všech běžích se dostaneme k dost slušným řešením, ale stejně ne všechny spuštění programu nám dosáhnou optimum. Zkusíme tedy změnit parametry pro lokální optimalizaci, nastavíme počet iterací na 5. Z grafu (Obr. 37) vidíme, že nám to nijak nepomohlo.



Obr. 37. Úspěšnost SS algoritmu při optimalizace Rastriginovy funkce, zvýšený počet iterací lokální optimalizace

Nyní zkusme nastavit tuto hodnotu zpět na 1, ale zkusme nastavit hodnotu kroku pro lokální optimalizaci na 0,1. Což by nám mělo zjemnit prohledávání. Můžeme si na grafu (Obr. 38) všimnout, že jsme dostali lepší výsledky než v předchozí simulaci.

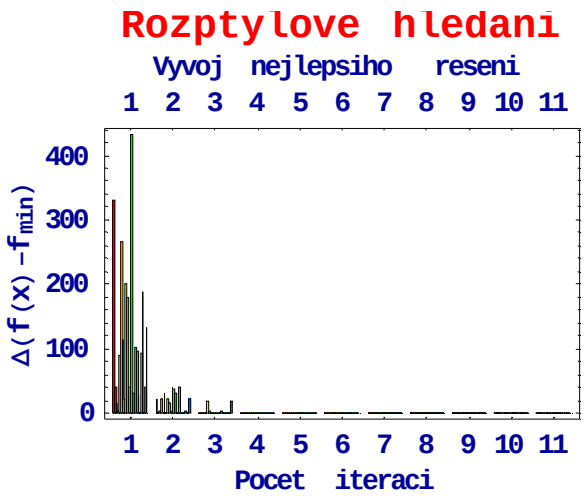


Obr. 38. Úspěšnost SS algoritmu při optimalizace Rastriginovy funkce, zmenšena hodnota kroku, pro lokální optimalizaci.

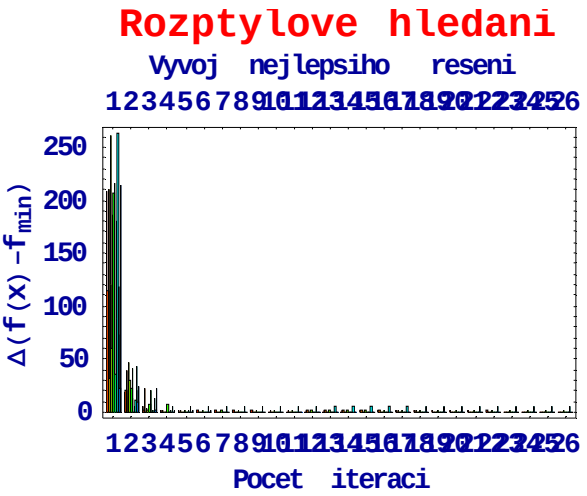
S těmito výsledky bychom mohli být spokojeni. Téměř všechny běhy programu doputovaly buď do minima, nebo alespoň velmi blízko.

Zajímalo mě, ale jestli se mi povede nastavit takové parametry, aby algoritmus našel minimum ve všech bězích. Zkusil jsem tyto hodnoty: *velikost okolí* = 1 *krok* = 0,01 a *počet opakování* = 1. Podařilo se nám získat takové výsledky, kde každý běh algoritmu dorazil do optima (Obr. 39). Za to jsme ale zaplatili delší dobou výpočtu.

Další mou úvahou bylo, jestli by se nedal zachovat původní krok a nestačilo zvýšit počet iterací celého algoritmu. Opět jsem nastavil *krok* = 0.1 a *počet iterací celého algoritmu* = 25. I zde jsme získali poměrně dobré výsledky (Obr. 40). Nebyly ale tak kvalitní jako v předchozím případě. Můžeme tedy říct, že *krok* lokální optimalizace má na výsledky větší vliv než počet iterací celého algoritmu.



Obr. 39. Úspěšnost SS algoritmu při optimalizace Rastriginovy funkce, *krok* = 0.01



Obr. 40. Úspěšnost SS algoritmu při optimalizace Rastriginovy funkce, zvýšen počet iterací algoritmu

8.3.3 Shrnutí

Při využití algoritmu na unimodální funkci jsme se nedostali do problémů při využití implicitních parametrů, dokonce již zhruba při třetí iteraci se dostalo 100% běhů algoritmu do optima.

Při využití na multimodální funkci nám, při využití implicitních parametrů, dělaly problém dvě věci. První zádrhelem bylo nevhodné nastavení lokální optimalizace. Stejně jako jsme zjistili už při zkoumání horolezeckého algoritmu, tak i zde je implicitní nastavení okolí a kroku příliš hrubé. Nebo naopak by se dalo říct, že nastavení je v pořádku, ale testovaná funkce má nastavené rozsahy proměnných na malou hodnotu. Jako druhý zádrhel bych uvedl, že algoritmus občas uvíznul v lokálním minimu, čemuž se dalo pomoct zmenšením kroku lokální optimalizace.

ZÁVĚR

Závěrem bych chtěl říci, že za největší přínos mé práce považuji realizaci výše uvedených algoritmů v prostředí WebMathematica a jejich umístění na internet, kde jsou dostupné pro širokou veřejnost. Na těchto stránkách má každý možnost vyzkoušet a doufám, že i pochopit, jakým způsobem EA pracují. Doufám, že se vyplní původní záměr a tyto stránky budou plnit jistou úlohu při výuce EA a optimalizace, neboť to byla myšlenka, která dala vzniknout této práci.

Nicméně i tato „literární“ část mé práce měla a má určitý smysl. Pro mě osobně je tím smyslem, to, že jsem měl možnost a povinnost při psaní pochopit i jemné nuance jednotlivých algoritmů. Nejednou se mi stalo, že po zpracování textové části jsem přišel na to, co by se dalo realizovat lépe, než jak jsem to měl předtím. Tak jsem to přepracoval a znovu jsem přepracovával i textovou část. A v téhle práci by se dalo pokračovat dál a dál, protože vždy se najde něco, co by šlo udělat ještě lépe.

CONCLUSION

The main purpose of this work is, by my opinion, the realization evolutionary algorithms in WebMathematica and their accessing at internet for common people. This is occasion to try EA on these web pages and I hope that no only try but also understand their principles and functions. If all is right these web pages, will be used for the education of the optimization and the evolutionary algorithms. This was the main idea for working at this work.

And this “literary” part had and has also of purpose. This purpose is a chance to a better grasp of these problems. I had to reorganize and remake my realization of EA many times. When I wrote a part of the literary part, then I understood what could be better in my programming part and I had to rewrite it. Then I had to revise the literary part. This can be going in cycles to infinity. Then I hope in your margin of error, because all could be better than it is.

SEZNAM POUŽITÉ LITERATURY

[1] Zelinka Ivan, Umělá inteligence v problémech globální optimalizace, Ben – technická literatura, Praha 2002, ISBN 80-7300-069-5

[2] Kvasnička V., Pospíchal J., Tiňo P., Evolučné Algoritmy, STU Bratislava, 2000, ISBN 80-227-1377-5

[3] Scatter Search [online]. [cit. 20. 5. 2007] Dostupný z URL: < http://leeds-faculty.colorado.edu/laguna/research.htm>

[4] Soma homepage [online]. [cit. 20. 5. 2007] Dostupný z URL: <http://www.ft.utb.cz/people/zelinka/soma/>

[5] Wolfram research, Inc. [online]. [cit. 20. 5. 2007] Dostupný z URL: <http://www.wolfram.com/>

[6] Wolfram research, Inc. – WebMathematica documentation [online]. [cit. 20. 5. 2007] Dostupný z URL: <http://documents.wolfram.com/webmathematica/>

[7] Evoluční algoritmy [online]. [cit. 20. 5. 2007] Dostupný z URL: < http://zelinka-mathematica.utb.cz:8080/webMathematica/evoluce/>

[8] LiveGraphics3D Homepage [online]. [cit. 20. 5. 2007] Dostupný z URL: <http://www.vis.uni-stuttgart.de/~kraus/LiveGraphics3D/>

[9] Zdeněk Vašíček – Simulované žíhání [online]. [cit. 20. 5. 2007] Dostupný z URL: < http://www.stud.fit.vutbr.cz/~xvasic11/projects/msi.pdf >

[10] Zelinka Ivan. Umělá inteligence I., Volume 1., Zlín: Vutium, Brno 1998, ISBN 80-21-1163-5

[11] Lampinen J., Zelinka I., New Ideas in Optimization – mechanical Engineering Design Optimization by Differential Evolution, McGrawHill, London 1999, ISBN 007-709506-5

[12] Zelinka Ivan. New Optimization Techniques in Engineering, Springler-Verlag

[13] Koza J.R., Genetic Programming, MIT Press, 1998, ISBN 0-262-11189-6

[14] Koza J.R., Bennet F.H., Andre D., Keane M., Genetic Programming III, Morgan Kaufnamm pub., 1999, ISBN 1-55860-543-6

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

b	Velikost referenční množiny
b1	Velikost podmnožiny referenční množiny, ve které jsou uložena nejlepší řešení podle velikosti CV
b2	Velikost podmnožiny referenční množiny, ve které jsou uložena nejlepší řešení podle vzdálenosti od ostatních řešení
CV	Hodnota účelové funkce.
EA	Evoluční algoritmus, popřípadě evoluční algoritmy.
F(x)	Účelová funkce, též hodnota účelové funkce
Iter	Iterace
MaxIter	Maximální počet iterací
$O_{por}(x)$	Operátor poruchy
P	Množina zkušebních řešení
$P(x \rightarrow x')$	Pravděpodobnost akceptování nového řešení
P_{size}	Velikost startovní množiny, kterou nám vygeneruje diversifikační metoda u SS.
RefSet	Referenční množina
RefSet1	Podmnožina referenční množiny, ve které jsou uložena nejlepší řešení podle velikosti CV
RefSet2	Podmnožina referenční množiny, ve které jsou uložena nejlepší řešení podle vzdálenosti od ostatních řešení
SS	Scatter search (algorithm)- (Algoritmus) rozptylové hledání.
T	Teplota
T´	Nová teplota
x	Argumenty účelové funkce, možné řešení, aktuální řešení, bod v n-rozměrném prostoru
x*	Řešení po provedení lokální optimalizace
x´	Nové řešení

δx	Okolí bodu x

SEZNAM OBRÁZKŮ

SEZNAM TABULEK