VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

OPTIMALIZAČNÍ METODY PRO KNIHOVNU SIMLIB/C++

OPTIMIZATION METHODS FOR SIMLIB/C++ SIMULATION LIBRARY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **JAKUB CHLEBÍK**
AUTHOR

VEDOUCÍ PRÁCE — **Dr. Ing. PETR PERINGER**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/20033/2016/xchleb07

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Chlebík Jakub**
Obor: Informační technologie
Téma: **Optimalizační metody pro knihovnu SIMLIB/C++**
Optimization Methods for SIMLIB/C++ Simulation Library
Kategorie: Modelování a simulace

Pokyny:

1. Seznamte se s problematikou optimalizačních metod a jejich použití při simulaci. Prostudujte existující jednoduché nástroje pro optimalizaci zahrnuté do SIMLIB/C++.
2. Navrhněte rozšíření knihovny SIMLIB/C++ modulem pro optimalizační metody. Zaměřte se na návrh vhodného rozhraní. Navrhněte také rozhraní pro použití externích optimalizačních knihoven.
3. Navržené rozšíření implementujte v C++. Implementujte alespoň 5 optimalizačních metod. Na vhodně zvolených příkladech proveďte základní testování všech implementovaných metod.
4. Zhodnoťte dosažené výsledky a navrhněte další možné vylepšení podpory optimalizačních metod.

Literatura:

- Gosavi A.: *Simulation-Based Optimization*, Kluwer Academic Publishers, Boston, 2003
- http://www.fit.vutbr.cz/~peringer/SIMLIB/

Pro udělení zápočtu za první semestr je požadováno:

- Splnění prvních dvou bodů zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Peringer Petr, Dr. Ing.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L. S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Práce se věnuje metodám optimalizace parametrů simulačních modelů. Seznamuje se základy matematické optimalizace a jejím využitím v operačním výzkumu. Dále navrhuje rozšíření knihovny SIMLIB/C++ modulem pro optimalizační metody. Několik vybraných metod teoreticky popisuje, implementuje v jazyce C++, demonstruje jejich použití na několika příkladech a zhodnocuje jejich úspěšnost.

Abstract

This thesis addresses the topic of parametric optimization of simulation models. It introduces theoretical foundation of optimization and its uses in simulation analysis. Furthermore, it suggests the extension of SIMBLI/C++ library by module for optimization methods. Some of the chosen methods are then theoretically described, implemented in C++ language, demonstrates its uses and evaluates their success.

Klíčová slova

Optimalizace, SIMLIB/C++, simulované žíhání, evoluční algoritmus, metoda sdružených gradientů, line search, Nelder-Mead simplexová metoda, optimalizace mravenčí kolonií

Keywords

Optimization, SIMLIB/C++, simulated annealing, evolution algorithm, conjugate gradient, line search, Nelder-Mead simplex method, ant-colony optimization

Citace

CHLEBÍK, Jakub. *Optimalizační metody pro knihovnu SIMLIB/C++*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Dr. Ing. Petr Peringer

Optimalizační metody pro knihovnu SIMLIB/C++

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Dr. Ing. Petra Peringra. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jakub Chlebík
17. května 2017

Poděkování

Tímto bych chtěl poděkovat Dr. Ing. Petrovi Peringerovi za poskytnutou pomoc a rady, které mi poskytl při psaní této práce a při konzultacích.

Obsah

1 Úvod **2**

2 Přehled problematiky **3**
2.1 Teoretický základ . . . 3
2.1.1 Matematická optimalizace . . . 3
2.1.2 Hodnotící funkce . . . 4
2.1.3 Klasifikace optimalizačních úloh . . . 4
2.1.4 Operační výzkum . . . 4
2.1.5 Hodnotící funkce definována modelem . . . 5
2.2 Rozdělení optimalizačních metod dle vlastností . . . 5
2.2.1 Kritéria pro výběr optimalizační metody . . . 6
2.3 Heuristické metody . . . 6
2.3.1 Simulované žíhání . . . 6
2.3.2 Nelder-Mead simplexová metoda . . . 8
2.4 Iterační metody . . . 12
2.4.1 Metoda sdružených gradientů . . . 12
2.5 Algoritmy inspirované přírodou . . . 15
2.5.1 Evoluční strategie . . . 15
2.5.2 Optimalizace mravenčí kolonií . . . 17

3 Návrh a implementace **20**
3.1 Návrh rozhraní . . . 20
3.1.1 Využití návrhového vzoru Bridge . . . 20
3.1.2 Výsledné veřejné rozhraní . . . 22
3.1.3 Spuštění optimalizace . . . 24
3.2 Implementace metod . . . 24
3.2.1 Simulované žíhání . . . 24
3.2.2 Nelder-Mead simplexová metoda . . . 26
3.2.3 Sdružené gradienty . . . 26
3.2.4 Evoluční strategie . . . 28
3.2.5 Optimalizace mravenčí kolonií . . . 29
3.3 Testovací příklady . . . 30
3.3.1 Popis testovacích funkcí . . . 31
3.3.2 Zhodnocení výsledků . . . 33

4 Závěr **40**

Literatura **41**

Kapitola 1

Úvod

Simulační experimenty jsou velice rozšířeným způsobem, jak zjišťovat informace o reálných systémech. Často je třeba najít řešení problémů, které se snaží minimalizovat výrobní rizika, hledání optimálního stavu v chemických procesech apod. K řešení těchto úloh jsou běžně využity prostředky tzv. operačního výzkumu, jehož základ tvoří teorie matematické optimalizace.

Cílem práce je navrhnout rozšíření knihovny SIMLIB/C++ o optimalizační modul s metodami simulovaného žíhání, Nelder-Mead simplexového algoritmu, evoluční strategie, sdružených gradientů a algoritmu optimalizace mravenčí kolonií. Pro tyto metody je navíc třeba navrhnout i vhodné rozhraní, které umožní přidávání dalších metod, včetně těch z externích optimalizačních knihoven. Výsledné řešení je nutno otestovat na vhodně zvolených modelech a funkcích.

Následující kapitola vysvětluje základní pojmy jako matematická optimalizace, operační analýza a hodnotící funkce, je popsán výběr optimalizačních metod a vysvětlen jejich princip a vlastnosti. Kapitola 2 popisuje návrh a implementaci rozhraní nového modulu optimalizačních metod pro knihovnu SIMLIB/C++. Následně dokumentuje implementaci vybraných optimalizačních metod a podsystémů pro jejich funkcionalitu, včetně návodu k jejich použití. V poslední části kapitoly jsou porovnávány výsledky metod nad vybranými reálnými modely, je zhodnocena jejich úspěšnost a jsou navržena případná další rozšíření.

Kapitola 2

Přehled problematiky

Cílem této kapitoly je poskytnout čtenáři základní přehled o problematice matematické optimalizace, vysvětlit několik nezbytných pojmů pro její pochopení a zařadit optimalizaci do kontextu operačního výzkumu. Dále popíše rozdělení optimalizačních metod do tří skupin na základě jejich vlastností a způsobu výpočtu optima. V posledních sekcích kapitola vysvětlí princip pěti vybraných a následně implementovaných optimalizačních metod. Pokud by měl čtenář zájem o hlubší studium problematiky, může se obrátit například na knihy [13] a [11], kde jsou tyto pojmy vysvětleny podrobněji.

2.1 Teoretický základ

Tato sekce si klade za cíl objasnit některé základní definice a zařadit je do kontextu operačního výzkumu.

2.1.1 Matematická optimalizace

Optimalizace [13] je podoborem matematické analýzy a numerické matematiky, zabývající se hledáním extrému zobrazení $f : \mathbb{R}^n \to \mathbb{R}$ v prostoru $\Omega = \mathbb{R}^n$. Přesněji, vyhledává množinu vstupních parametrů $x = x_1, ..., x_n$ problému, které odpovídají minimu, případně maximu, hodnotící funkce $f(x) = f(x_1, ..., x_n)$.

Definice 1. Minimum funkce.
Nechť $f : \mathbb{R}^n \to \mathbb{R}$ je funkce n proměnných. Tvrdíme, že $x^* \in \mathbb{R}$ je minimem f (dále v textu pouze "minimum"), pokud existuje $\delta > 0$ taková, že pro všechna x ve vztahu $\|x - x^*\| \leq \delta$ platí $f(x^*) \leq f(x)$. Analogicky lze odvodit definici maxima.

O takovémto minimu tvrdíme, že je lokálním, pokud je nejmenším bodem v daném okolí. O globálním minimu hovoříme, pokud je takovýto bod nejmenším na celém definičním oboru.

Díky vztahu

$$\max_{x \in \Omega}(f(x)) = -\min_{x \in \Omega}(-f(x)) \tag{2.1}$$

můžeme vždy předpokládat, že optimalizační problém je problémem minimalizačním a maximalizací se dále nemusíme zabývat.

2.1.2 Hodnotící funkce

Hodnotící funkce [11], (někdy také funkce účelová nebo cenová), je zobrazení, které transformuje vstupní parametry optimalizační funkce (tzv. řešení problému), na nějaký výstup (tzv. přípustné řešení). Nezbytným předpokladem je, že dva výstupy funkce mají jasně definované relační operátory a je tedy možné je porovnat a vybrat vhodnější. Optimální řešení je pak takové, které má mezi všemi přípustnými řešeními nejmenší, případně největší, hodnotu hodnotící funkce.

Často se stává, že optimalizační problém musí vyhovět více než jedné hodnotící funkci [11]. Takovéto řešení často nejsou možná, je tedy třeba k takovémuto problému přistupovat jinak. Možné způsoby řešení jsou například:

- Nahradit všechny účelové funkce jejich váženým průměrem. Tím se z několika účelových funkcí stane funkce jediná. Samozřejmě je problém korektně stanovit váhy jednotlivých účelových funkcí.
- Hledat přípustné řešení, které je nejblíže nějakému ideálnímu (ale nepřípustnému) řešení. Zde je problém jednak v definici ideálního řešení, jednak ve způsobu měření vzdálenosti od tohoto řešení.
- Hledat takové přípustné řešení, které při srovnání s kterýmkoli jiným přípustným řešením je v alespoň jedné účelové funkci lepší nebo alespoň stejně dobré. To ovšem zdaleka nemusí být jednoznačné.

2.1.3 Klasifikace optimalizačních úloh

Dle definičního oboru prostoru Ω účelové funkce se optimalizační úlohy podle literatury [9] klasifikují na :

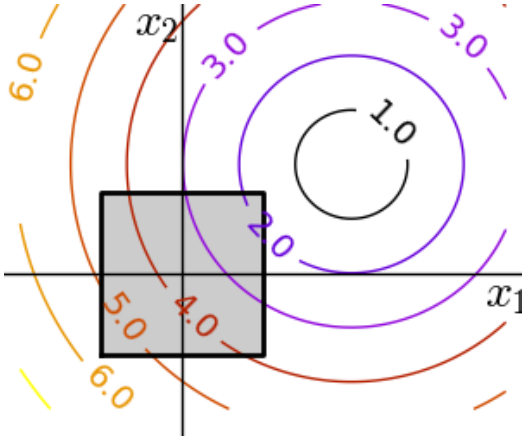
- Úlohy volného extrému. Tento případ nastává, pokud $\Omega = \mathbb{R}^n$.
- Úlohy vázaného extrému - případ kdy $\Omega \subset \mathbb{R}^n$. Zde nás zajímají pouze řešení, která splňují další podmínky, omezující buďto jeden nebo více vstupních parametrů $x_0, ...x_n$ na interval, nebo funkcemi $g_0(x_i, ...x_n) \geq 0$. O takto definovaných úlohách se mluví jako o úlohách matematického programování. Jednoduché příklady omezení vázaného extrému jsou znázorněna na obrázcích 2.1 a 2.2

Názvy volný a vázaný extrém nejsou v matematickém smyslu pevně zavedeny [9]. O volném extrému se tedy mluví i v případě, kdy optimalizovaná funkce $f(x_i)$ není definována na celém prostoru Ω a vyšetřuje se také pouze nad jistou množinou.

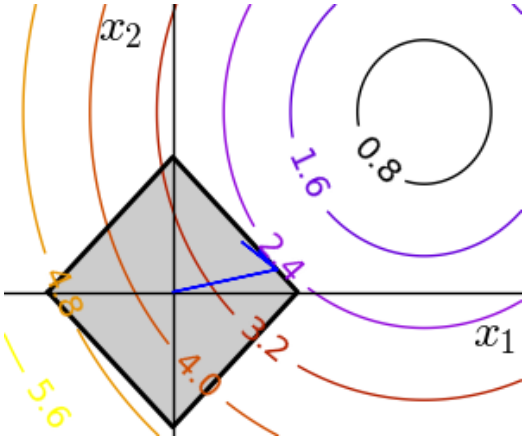
2.1.4 Operační výzkum

Pojmem *operační výzkum* (často označován také jako operační analýza) [13] se všeobecně rozumí převod komplexní inženýrské úlohy reálného světa na matematický, či jiný, model a následné provádění experimentů pro získání informací. Problémy tohoto typu jsou často spojeny s hledáním minima (např. provozní riziko), maxima (např. zisk), či jiného optimálního výsledku. Teoretickým základem operačního výzkumu je tedy matematická optimalizace a k hledání řešení se využívá metody optimalizací, modelování a simulací.

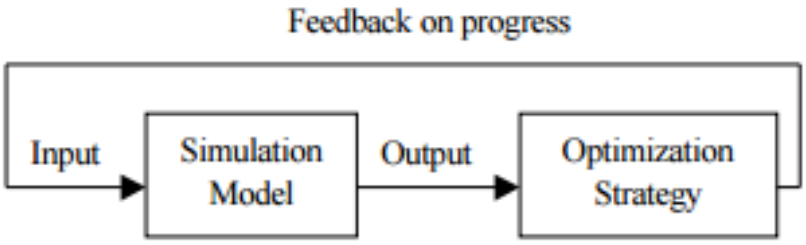
Průběh operačního výzkumu je znázorněn na obrázku 2.3.



Obrázek 2.1: Omezení intervalem.



Obrázek 2.2: Omezení funkcí.



Obrázek 2.3: Zjednodušený náhled na proces operačního výzkumu

2.1.5 Hodnotící funkce definována modelem

Pro potřeby operačního výzkumu je v první řadě třeba vytvořit simulační model, nad kterým budou prováděny experimenty. Při hledání optima těchto modelů je třeba zajistit, že budou mít stejné vlastnosti jako reálné funkce [13] — a to:

- simulační model má jasně definované vstupy a zobrazuje je na výstupy.
- dva výstupy z tohoto modelu mají jasně definované relační operátory. (V této práci se předpokládá, že výstupy modelu jsou z definičního oboru $\mathbb{R}$)
- pro použití gradientních metod musí definovat derivace ve všech bodech $x_1, ..., x_n$ na celém svém definičním oboru a to analyticky, či numericky.

2.2 Rozdělení optimalizačních metod dle vlastností

Metody pro optimalizace funkce lze dle jejich vlastností rozdělit na tři hlavní podskupiny tak, jak je rozdělili Nocedal a Wright v knize [11]:

- *Analytické algoritmy*, které využívají v zásadě prostředky diferenciálních počtu, jsou vysoce přesné a relativně rychle. Díky tomu jsou vhodné spíše pro individuální ruční výpočet. Tato práce se jimi nebude nadále příliš zabývat.
- *Iterační algoritmy*, které vytváří posloupnost bodů na definičním oboru. Tyto body jsou vytvářeny takovým způsobem, že postupně konvergují k hledanému extrému.

Iterační skupina algoritmů také využívá diferenciálního počtu, ovšem často pouze k určení následujícího bodu algoritmu, nikoliv na celém povrchu modelu. Tyto metody jsou pro účely operačního výzkumu použitelné, vyžadují ovšem více specifický model. Pro nalezení globálního optima je třeba správně zvolit počáteční bod, navíc nejsou tyto algoritmy často schopny dobře pracovat s omezeními daného modelu.

- *Metaheuristické algoritmy.* Do této skupiny patří algoritmy inspirované přírodou a metody využívající heuristických funkcí pro nalezení minima. Pro řešení problémů operační analýzy je tato rodina — díky svým vlastnostem — využívána nejčastěji. Při výběru správné metody pro specifický problém mohou být i rychlejší než metody jiné rodiny a přitom stejně přesné. Heuristické algoritmy mají ovšem tendenci spouštět experimenty vícekrát, případně potřebují řešit i experimenty s parametry, které se od řešení vzdalují. Metody této rodiny jsou často schopny nalézt i globální optimum.

2.2.1 Kritéria pro výběr optimalizační metody

Jak již bylo zmíněno, pro nalezení optima modelu lze použít mnoho algoritmů. Vhodnost takto vybrané metody na optimalizovaný problém je často nejzásadnější otázkou a je tedy nutno vybrat správně [13]. Je třeba vzít v úvahu vlastnosti metod a modelů — některé postupy nejsou například schopny překonat lokální minimum. Jiné naopak vyžadují první, případně další, derivace, nebo provádí nadměrně vysoké množství experimentů. Dále je třeba zvolit i podle charakteru optimalizovaného modelu. Fakta ke zvážení mohou být například :

- Jak časově a výpočetně náročné je experimentování nad modelem.
- Jedná se o model diskrétní či spojitý.
- Spojitý model často obsahuje lokální minimum, navíc je třeba zvážit, zda model podléhá omezením.
- Model lze reprezentovat jako graf a nalezení řešení je tedy redukováno na hledání optimálního průchodu grafem.

2.3 Heuristické metody

Heuristika [12] je strategie, jak lidé i stroje mohou řešit problémy s použitím dostupných – i když jen volně aplikovatelných – informací. Používá se nejčastěji tam, kde neexistuje přesný deterministický algoritmus, jak řešení nalézt. Nebo postup řešení znám je, ale jeho provedení není kvůli jeho náročnosti (většinou časové) přijatelné. Heuristické algoritmy jsou založeny na určitém způsobu prohledávání prostoru přípustných řešení. Tento způsob často vychází z náhody, intuice, analogie a zkušenosti. Heuristiky obecně nezaručují nalezení optimálního (nejlepšího) řešení. V praxi však často dávají dostatečně dobrá řešení v přijatelném čase.

2.3.1 Simulované žíhání

Metoda *simulovaného žíhání* [8] patří mezi heuristické optimalizační algoritmy, které, jak již je zřejmé z názvu samotného algoritmu mají základ ve fyzice, na rozdíl od jiných stochastických optimalizačních algoritmů, které mají většinou svůj základ v biologii. Ve fyzice žíhání označuje takový proces, při kterém je těleso umístěné do pece vyhřáté na vysokou

teplotu a postupným pomalým ochlazováním se odstraňuji vnitřní defekty tělesa. Při vysoké teplotě je těleso roztopené, což znamená, že částice dané látky jsou náhodně uspořádané v prostoru. Při postupném snižování teploty se částice dostávají do rovnovážné polohy, tj. celková energie tělesa se snižuje.

Algoritmus

Pseudokód algoritmu simulovaného žíhaní [8] využívající metodu Metropolis je definován v algoritmech 1 a 2 :

Algoritmus 1 Algoritmus simulovaného žíhání

Input: $T_{max}, T_{min}, k_{max}, \alpha, f(x), c(x)$ ▷ f(x) — hodnotící funkce, c(x) — funkce omezující optimalizovaný model
Output: x_{best}
  $T := T_{max}$;
  x_{best} := náhodně vygenerovaný stav ;
  while $T > T_{min}$ **do**
    $x_{best} := Metropolis(x_{ini}, k_{max}, T)$;
    $T := T * \alpha$;
  end while
  return x_{best};

Algoritmus 2 Metropolis

Input: $x_{best}, k_{max}, T, f(x), c(x)$ ▷ f(x) — hodnotící funkce, c(x) — funkce omezující optimalizovaný model
Output: x_{new}
  $k := 0$;
  $x_{new} := x_{ini}$;
  while $k < k_{max}$ **do**
    $k := k + 1$;
    x_{new} := posuň řešení náhodným směrem;
    $P := \min(1, exp(-(f(x_{new}) - f(x_{best}))/T))$;
    if $random() \leq P$ **then**
      $x := x_{new}$;
    end if
  end while
  return x_{new};

Rozvrh chlazení

Většina parametrů simulovaného žíhání — cenová funkce, definiční obor, výběr následujících stavů, atd. — jsou dány již v definici a v průběhu se nemění. Tedy jedním z nejdůležitějších parametrů pro nalezení optima je výběr správného chladícího rozvrhu [8]. Běžně nejvíce používaný chladící proces je popsán následujícím vztahem :

$$T(i) = T_0 * \alpha^i$$

Takto definovaný rozvrh nejdříve „roztaví“ systém při vysokých teplotách a poté, opakovaným snižováním o konstantní koeficient α ($0 < \alpha < 1$), systém chladí.

Algoritmus poté pro každou teplotu T_i spouští modifikovanou metodu Monte Carlo, zvanou Metropolis, která zajistí rovnovážný stav systému.

Rovnovážnost systému

Vychází z metody Metropolis [8], přesněji Metropolis-Hasting, která je modifikací algoritmu Monte Carlo. Metropolis generuje sekvenci náhodných vzorků tak, že čím více vzorků je vybráno, tím více se rozložení těchto vzorků podobá danému pravděpodobnostnímu rozložení. V kontextu simulovaného žíhaní funguje následovně:

Nechť je daný aktuální stav systému, jenž je určený polohou částic tělesa, potom je malá náhodná porucha (nové řešení) generovaná tak, že jsou částice jemně posunuté. Proces generování poruchy se nazývá perturberancí [7]. (Tato porucha musí být symetrická, tj. pravděpodobnost toho, že malou poruchou se stav A změní na stav B, musí být stejná jako změna stavu B na A).

Pokud je nový stav blíže minimu, potom proces přijme nový porušený stav jako aktuální. V opačném případě je pravděpodobnost přijetí porušeného stavu určena vztahem [7].

$$e^{-(\Delta E/T_i)}$$

Perturberance systému

Generování sousedního stavu [8] nehraje zdaleka tak velkou roli, jako výběr chladícího mechanismu, ovšem i tak stojí za zmínku. Sousední stav je generován vztahem

$$x_{i+1} = x_i + u * |x^{max} - x^{min}| * T/T^{max}$$

kde u je náhodné číslo rovnoměrného rozložení mezi -1 a 1.

Tento vztah zajišťuje symetrii zmíněnou dříve a tedy i dokončuje snahu o zajištění rovnovážného systému.

2.3.2 Nelder-Mead simplexová metoda

Nelder-Mead algoritmus [15] je jednou z nejlepších známých algoritmů pro více-dimenzionální neomezenou optimalizaci bez použití derivací. Metoda by neměla být zaměňována s Dantzigovou simplexovou metodu, která se věnuje problémům lineárního programování. Základní algoritmus je vcelku jednoduchý a přesný, a (nejen) z těchto důvodů je velice populární ve vědeckých oborech — především chemii a medicíně. Metoda nevyžaduje znalost derivací a je schopná řešit problémy nehladkých, nespojitých či nekonvexních funkcí.

Jelikož simplexová metoda nevyužívá mechaniky line search či trust region pro určení směru a ani není založena na zjednodušení modelu, patří tedy do skupiny přímých algoritmů [13]. Vzhledem ke svým vlastnostem není ovšem vždy schopná přesně, v porovnání s jinými metodami, konvergovat k optimu nebo nalézt dostatečně přesné optimum [6]. Je tedy vhodná pro řešení úloh, u kterých není vysoká přesnost nezbytná.

K nalezení minima algoritmus vytváří nové body simplexu 2, kterými nahrazuje body staré. Konstrukce těchto bodů je řízena heuristickými funkcemi tak, aby co nejlépe konvergovala k lokálnímu optimu. Nejvíce používanými jsou heuristiky reflexe, expanze, kontrakce a přiblížení simplexu, které tento algoritmus odlišují od běžného simplexu.

Veškeré obrázky, definice, heuristiky a procesy práce se simplexy v této podkapitole jsou převzaty z [15].

Definice 2. Simplex
. Simplex $\mathbb{S} \in \mathbb{R}^n$ je definován jako konvexní obal o $n+1$ bodech $x_0, ..., x_n \; x \in \mathbb{R}^n$. Pro $\mathbb{R}^2$ se jedná o trojúhelník, pro $\mathbb{R}^3$ o čtyřstěn.

Algoritmus

Pseudokód Nelder-Mead algoritmu, tak jak je vysvětlen v [6] je znázorněn v algoritmu 3

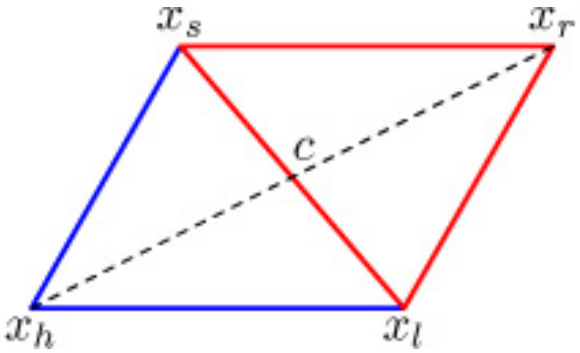
Algoritmus 3 Nelder-Mead simplexový algoritmus

Input: $\{x_i\}$, pro $i...n$ — počáteční simplex a hodna cenové funkce v každém bodě, $f(x), c(x)$ ▷ f(x) — hodnotící funkce, c(x) — funkce omezující optimalizovaný model
Output: x^*, lokální minimum cenové funkce f

```
k := 0;
while STOPING_CRITERIA & & k < k_max do
    h := arg max_i f(x_i);
    l := arg min_i f(x_i);
    x̄ := Centroid(x_i);
    x' := (1 + α) * x̄ − α * x_h;                          ▷ α > 0 je koeficient reflexe
    if f(x') < f(x_l) then
        x'' := (1 + γ) * x' − γ * x̄;                      ▷ γ > 1 je koeficient expanze
        if f(x'') < f(x_l) then
            x_h := x'';
        else
            x_h := x';
        end if
    else if f(x') > f(x_i), ∀i ≠ h then
        if f(x') ≤ f(x_h) then
            x_h := x';
        end if
        x'' := (1 − β) * x̄ + β * x_h;                     ▷ 0 < β < 1 je koeficient kontrakce
        if f(x'') > f(x_h) then
            x_i := (x_i + x_l) * δ,    ∀i ∈ [0, n];       ▷ 0 < δ < 1 je koeficient zmenšení
        else
            x_h := x'';
        end if
    else
        x_h := x';
    end if
    k := k + 1;
end while
return x_l;
```

Seřazení

V prvním kroku algoritmu je třeba určit indexy h, s, l — nejhorší, druhý nejhorší a nejlepší bod aktuálního simplexu.



Obrázek 2.4: Reflexní bod simplexu $n = 3$

Centroid

Další fází metody je výpočet centroidu nejlepší strany simplexu c. Vztah pro tento výpočet je následovný :

$$c = \frac{1}{n} * \sum_{j \neq h} x_j$$

Reflexe

Dále je nutno vypočítat reflexní bod simplexu. Výpočet je založen na skutečnosti, že funkční hodnota se zmenšuje na hraně vedoucí z bodu x_h do x_l a hraně z x_h do x_s. Znázorněno na obrázku 2.4.

Pro výpočet reflexního bodu x_r je běžně používán následující vztah. (Koeficient $\alpha = 1$)

$$x_r = c + \alpha * (c - x_h)$$

Pokud platí, že $f(x_l) \leq f(x_r) < f(x_s)$, nahradí se x_h za x_r a ukončí se iterace.

Expanze

V případě, kdy $f(x_r) < f(x_l)$, si můžeme dovolit reflexní bod x_r expandovat ještě o něco dále. Takto expandovaný bod počítá vztah :

$$x_e = c + \gamma * (x_r - c)$$

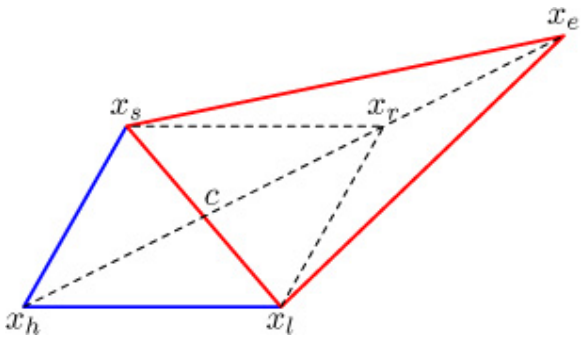
Koeficient γ je empiricky běžně určen na hodnotu 2.

Nyní, pokud platí vztah $f(x_e) < f(x_r)$, přijmeme x_e jako nové x_h a ukončíme iteraci. V opačném případě přijmeme pouze x_r. Princip je znázorněn na obrázku 2.5.

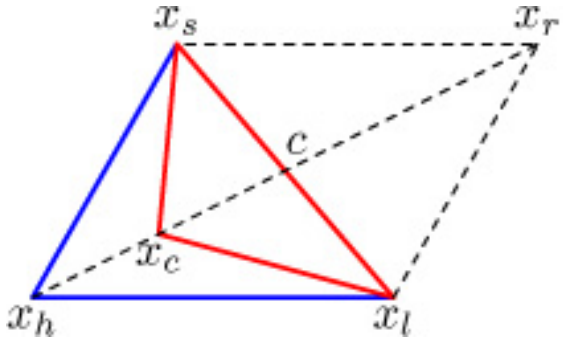
Kontrakce

Kontrakce simplexu nastane v případě, kdy reflexní bod je horší nebo roven druhému nejhoršímu bodu, tedy $f(x_r) \leq f(x_s)$. Tato situace nastává například v případě, kdy optimum leží uvnitř nově vzniklého simplexu. Kontrakce může být dvojího typu — a to vnitřní nebo vnější — jak je znázorněno na obrázcích 2.6 a 2.7. Koeficient kontrakce β je, stejně jako ostatní koeficienty, běžně získán empiricky, a to na hodnotu $\frac{1}{2}$

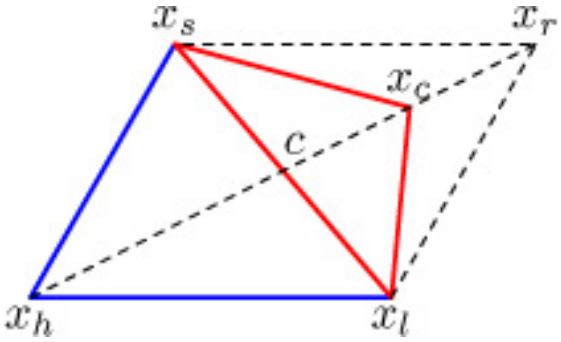
Vnitřní kontrakce — tedy pokud platí $f(x_s) \leq f(x_r) < f(x_h)$ — se určí vztahem



Obrázek 2.5: Expanzní bod simplexu



Obrázek 2.6: Vnitřní kontrakce simplexu.



Obrázek 2.7: Vnější kontrakce simplexu.

$$x_c = c + \beta * (x_r - c)$$

Pokud je kontrakční bod lepším, než bod x_r, nahradí x_h a ukončí iteraci.

Vnější kontrakce — $f(x_r) \geq f(x_h)$ — pozmění heuristiku na

$$x_c = c + \beta * (x_h - c)$$

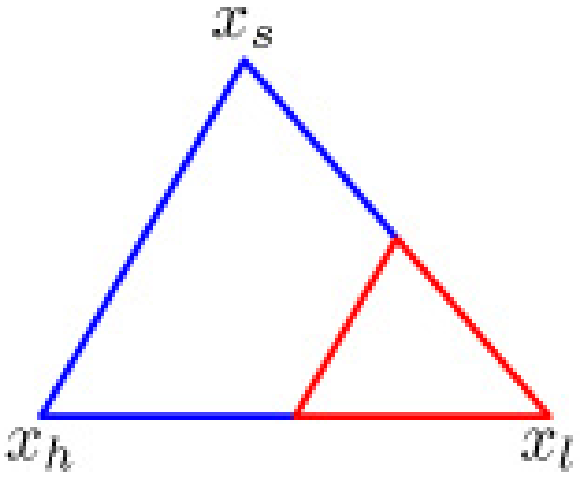
Pokud je kontrakční bod lepším, než bod x_h, nahradí jej momentální iterace se ukončí.

Pokud tyto nerovnosti neplatí, přechází se na poslední operaci — přiblížení.

Přiblížení

Posledním možným stavem algoritmu Nelder-Mead je funkce přiblížení simplexu. Jedná se o stav jednoho bodu simplexu mnohem dále od optima funkce, než ostatní. V případě „nestandardního“ zaoblení údolí by kontrakce daný bod posunula ještě dále od hledaného optima. Operace zmenšení je znázorněna na obrázku 2.8, využívá koeficient $\delta = \frac{1}{2}$ a je určena funkcí :

$$x_j = x_l + \delta * (x_j - x_l), \qquad pro \quad j = 0, ..., n \quad j \neq l$$



Obrázek 2.8: Zmenšení simplexu

Ukončující podmínky

Pro nalezení řešení v konečném čase a zajištění konce algoritmu je ještě třeba vytvořit ukončující podmínky. Běžně jsou používány tzv. podmínky konvergence, které jsou popsány ve [15] takto:

- Doménová konvergence — tato podmínka je splněna, pokud aktuální simplex $\mathbb{S}$ je dostatečně malý (některé, nebo všechny body jsou si dostatečně blízko).
- Konvergence funkčních hodnot — běžný konvergenční test, který projde v případě, kdy jsou si funkční hodnoty bodů dostatečně blízké.
- Fail test — tzv. ne-konvergující test je splněn, pokud počet iterací, či vyhodnocení funkce, přesáhne danou povolenou hranici.

2.4 Iterační metody

Za iterační jsou označovány ty algoritmy, které vytváří posloupnosti řešení úlohy takovým způsobem, že s každou iterací nalezený bod konverguje blíže k řešení [14]. K získání těchto bodů pracuje metoda s hodnotou řešení předchozího, která je modifikována deterministickými algoritmy. Tato skutečnost, spolu s kritérii, která musí řešené úlohy splňovat, poté zaručí nalezení řešení ve spočetném počtu kroků. Bohužel, již zmíněná kritéria jsou velmi specifická. Toto, dohromady s faktem, že deterministické metody vyžadují často větší, než malé množství vyhodnocení hodnotící funkce a počítání diferenciálních počtů, má za následek, že iterační metody mnohdy nelze využít a je třeba přistoupit k metodám metaheuristiky.

2.4.1 Metoda sdružených gradientů

Jedná se o nejpopulárnější metodu pro řešení velkých systémů lineárních rovnic [14]. Navazuje na metodu největšího spádu, která určuje směr sestupu podle záporné hodnoty gradientu v aktuálním bodu. Metoda poté v tomto směru postupuje tak dlouho, dokud se funkční

hodnota zmenšuje a je tímto „zaslepena“. Metoda sdružených gradientů tuto myšlenku rozšiřuje tím, že při určování směru nevyužívá jen lokální informace, ale i paměť (směry posunu v předešlých iteracích), díky čemuž je schopná často optimum nalézt rychleji.

Od doby jejího vzniku na začátku minulého století došlo ke spoustě modifikací; jednou z nich je nelineární iterační metoda sdružených gradientů, kterou lze použít pro účely optimalizace nelineárních funkcí a modelů.

Vzhledem ke skutečnosti, že metoda není schopna překonat lokální minima se často volí varianta restartování algoritmu s různými počátečními body a uchování nejlepšího z výsledků.

Tato práce se nebude zabývat dokazováním konvergence metody. Pokud má čtenář zájem dozvědět se o problematice konvergence gradientních metod více, doporučuji obrátit se na literaturu [14].

Algoritmus

Algoritmus 4 popisuje princip metody sdružených gradientů. Je nutno podotknout, že funkce $CalculateBetaCoeff()$ z této definice se mění podle implementace dané metody a má vysoký podíl na konečné konvergenci.

Algoritmus 4 Metoda sdružených gradientů

Input: $x_0, k_{max}, \epsilon, f(x), g(x)$ ▷ f(x) — hodnotící funkce, g(x) — funkce vracející gradient v bodě x
Output: x^*

$\Delta x_0 := -\nabla_x f(x_0)$;
$\alpha_0 := \arg\min_{\alpha} f(x_0 + \alpha * \Delta x_0)$;
$x_1 := x_0 + \alpha_0 * \Delta x_0$;
$n := 0$;
while $n < n_{max}$ **do**
  $\Delta x_n := -\nabla_x f(x_n)$;
  $\beta_n = CalculateBetaCoeff()$;
  $s_n = \Delta x_n + \beta_n * s_{n-1}$;
  $\alpha_n := \arg\min_{\alpha} f(x_n + \alpha * \Delta s_n)$; ▷ proveď tzv. line search
  $x_{n+1} = x_n + \alpha_n * s_n$;
  $n := n + 1$;
end while

Koeficienty α a β

V algoritmu sdružených gradientů hrají roli dva koeficienty. Koeficient β, který ovlivňuje směr sestupu a koeficient α, který určuje, jak dlouhý krok daným směrem se má provést.

β je vybíráno tak, že směry hledání jsou sdružené ($p_i * A * p_j = 0$). Je více různých postupů, jak tento koeficient počítat, podle kterých jsou poté pojmenovány i samotné algoritmy. Například sdružená gradientní metoda Polak-Ribiér ji definovala takto:

$$\beta_n^{PR} = \frac{\Delta x_n^T * (\Delta x_n - \Delta x_{n-1})}{\Delta x_{n-1}^T * x_{n-1}}$$

α je vybírána Line Search metodou.

Line Search

Pro získání délky kroku se tradičně používá metoda Line Search. Jedná se o prohledávání prostoru okolo bodu takovým způsobem, že se snažíme najít krok α_n, který ve směru s_n minimalizuje funkci $f(x)$.

Existují dva základní přístupy, jak provádět line search.

- Exaktní line search — pro svou relativní složitost méně používaný přístup spočívá v řešení optimalizačního problému $\arg\min_{\alpha} f(x_n + \alpha * s_n)$.
- Přibližný line search — tento způsob prohledávání se snaží najít takové α, které bude splňovat určené podmínky. Například backtrackingový line search, který začíná s momentální funkční hodnotou a vysokým odhadem α a postupně jej zkracuje, dokud hodnota nesplňuje konvergenční podmínky. Pro backtracking jsou typické Armijo-Goldstein podmínky [11], většina přibližných line search metod ovšem pracuje s podmínkami Armijo-Wolfe.

  První podmínkou každého přibližného line search je tzv. Armijova podmínka. Zajišťuje, že α minimalizuje hodnotící funkci „dostatečně“. Definována je následovně:

  $$f(x_k + \alpha_k * s_k) \leq f(x_k) + c_1 * \alpha_k * s_k^T * \nabla f(x_k)$$

  Druhou podmínkou je tzv. zakřivující Wolfova podmínka. Tato nerovnost zaručuje, že pokles sklonu funkce bude dostatečný. Druhá podmínka může být dvojího typu:

  - silná Wolfova podmínka - $|s_k^T * \nabla f(x_k + \alpha_k * s_k)| \leq c_2 * |s_k^T * \nabla f(x_k)|$ — která zajistí, že velikost α posune zkoumaný parametr co nejblíže stacionárnímu bodu nebo lokálnímu minimu[11].
  - slabá Wolfova podmínka — $s_k^T * \nabla f(x_k + \alpha_k * s_k) \geq c_2 * s_k^T * \nabla f(x_k)$. Tento druh podmínky je jednodušší ke splnění a tedy přijme i kroky, které silná podmínka ne.

  kde $0 < c_1 < c_2 < 1$. c_1 je běžně určeno jako malá hodnota, zatímco c_2 jako hodnota vysoká. Pro metodu sdružených gradientů navrhl Nocedal [11] koeficienty $c_1 = 10^{-6}, \quad c_2 = 0.1$.

Vliv výběru podmínky je velice závislý na druhu problému, pro některé problémy konvergenci urychlí, některým ji znemožní.

Směr sestupu

Směr sestupu je v metodě sdružených gradientů určen vztahem

$$d_0 = -\nabla f(x_0)$$
$$d_k = -\nabla f(x_k) + \beta_k * d_{k-1}$$

Podmínky konvergence

Algoritmus se běžně ukončuje splněním některé ze 3 podmínek [11] :

- Dosažení maximálního počtu iterací

- Rozdíl mezi dvěma po sobě jdoucími řešeními je menší než zadané ϵ
- Je nalezen takový bod, pro který již nebude možné splnit line search podmínky.

2.5 Algoritmy inspirované přírodou

Pokud je nějaká populace jedinců vystavena působení určitého selekčního tlaku, začíná se v následujících generacích na tento tlak adaptovat [10]. Jinak řečeno, zlepšuje se průměrná zdatnost jedinců v populaci vzhledem k tomuto tlaku. Je to dáno tím, že jedinci lépe adaptovaní na okolní podmínky mají větší šanci přežít a rozmnožit se, přičemž geny kódující tuto schopnost předávají dále na své potomky. V další generaci je pak zastoupeno větší procento takto přizpůsobených jedinců než v generaci předchozí.

2.5.1 Evoluční strategie

Aby byla evoluce funkční, jsou nezbytné tři věci. Za prvé je nezbytné umět ze dvou existujících řešení vytvořit nové, této operaci se říká křížení. Za druhé umět vytvořené řešení náhodně pozměnit, tato operace se nazývá mutace. A za třetí, pro vybraného jedince nalézt vhodného partnera ke křížení, tato operace je tzv. selekce.

V následujících kapitolách je detailněji popsán případ implementace, ve které je jedinec reprezentován sekvencí bitů.

Algoritmus

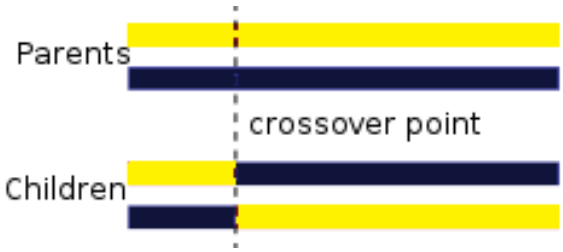
Nejjednodušší algoritmus evoluční strategie je popsán pseudokódem 5.

Algoritmus 5 Alogoritmus Evoluční strategie

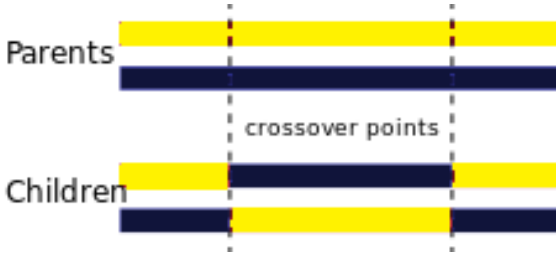
Input: $n_{max}, p_{max}, f(x), c(x)$ ▷ f(x) — hodnotící funkce, c(x) — funkce omezující optimalizovaný model
Output: x^*
  p_0 := náhodně vygenerovaná populace o velikosti p_{max}
  Ohodnoť každého jedince hodnotící funkcí $f(x)$
  while $n < n_{max}$ **do**
    $n := n + 1$;
    $p_n := Select(p_{n-1})$;
    $p_n := Crossover(p_n)$;
    $p_n := Mutate(p_n)$;
    Ohodnoť nově vzniklé jedince hodnotící funkcí
  end while

Selekce

Cílem operace selekce [10] je výběr vhodných jedinců pro křížení a vytvoření následující generace. Existuje více způsobů, jak tohoto docílit — každý by však měl zajistit, že jedinci s lepším ohodnocením mají větší šanci k páření. Několik vybraných způsobů je popsáno v literatuře [10] takto:



Obrázek 2.9: Jednobodové křížení.



Obrázek 2.10: Dvoubodové křížení.

- Turnajový výběr — pro každého nového jedince je uspořádán "turnaj", ve kterém soutěží n náhodných jedinců. Nejvhodnější se stává vítězem tohoto turnaje a bude, spolu s vítězi dalších turnajů, poslán dále do fáze křížení.
- Ruletový výběr — každý jedinec z populace má šanci k výběru ke křížení úměrnou své fitness hodnotě. Úskalím této varianty je fakt, že může a bude zásadně ovlivněna několika vysoce nadprůměrně hodnocenými jedinci, kteří tak vytvoří dominantní část nové populace. Pokud byly tito jedinci pouze lokálním řešením, může nastat uváznutí a metoda ztratí svůj globální charakter.
- Zkracovaný výběr — Jedinci jsou seřazeni podle svých fitness hodnot a pro reprodukci je vybráno n prvních jedinců. Jedná se o nejjednodušší ze způsobů selekce, i tak je ovšem velice účinným.

Křížení

Je kombinací rodičovských genů za cílem vytvoření jednoho, či více, potomků. Pro základní přehled jsou zde uvedeny běžné druhy křížení tak, jak je popsal Mitchell v knize [10].

- Jednobodové křížení — v chromozomech o délce n je náhodně zvolen index $i, 1 \leq i < n$. Prvních 0 až i genů chromozomu je poté odvozeno z jednoho rodiče, i až n z dalšího. Tímto způsobem mohou vzniknout až dva potomci. Znázorněno na obrázku 2.9.
- Vícebodové křížení — analogické jednobodovému křížení. Je vygenerováno k různých indexů $(i_0, ...i_k)$ pro které platí $(1 \leq i_1 < i_2 < ..., < i_k < n)$. Chromozom potomka vzniká postupným kopírováním genů chromozomu jednoho z rodičů. Začíná se náhodně zvoleným rodičem. Pokaždé, když je zkopírován gen s pořadovým číslem obsaženým ve vygenerovaných indexech, dojde ke změně rodičovského chromozomu za chromozom druhého rodiče. Příklad dvoubodového křížení je znázorněn na obrázku 2.10.
- Uniformní křížení — Pro každý gen je pravděpodobnostní funkcí rovnoměrného rozložení zvoleno, ze kterého z rodičovských chromozomů bude zkopírován.

Mutace

Operace mutace je obvykle zcela náhodná změna několika náhodně vybraných jedinců. Její účinek je běžně velice destruktivní, vzhledem ke zděděným vlastnostem. Smyslem mutace je

vytváření zcela nových jedinců, tedy zvyšování různorodosti populace a prevence uváznutí v lokálním extrému kriteriální funkce. Pravděpodobnost mutace se volí obvykle nízká; při neúměrně vysoké mutaci totiž dochází k narušování slibných řešení dříve, než mají čas se vyvinout. Pokud je naopak pravděpodobnost příliš nízká, hrozí nebezpečí, že populace bude zahlcena jedním typem řešení, které bude pouze lokálním optimem. Z praktického pohledu je problematické, že tyto dva stavy jsou obtížně rozeznatelné, jelikož se projevují podobně.

V binární reprezentaci chromozomu je operací mutace často myšlena změna 0 až n genů jedince na jejich opačnou hodnotu.

2.5.2 Optimalizace mravenčí kolonií

Mravenčí kolonie (ACO – Ant Colony Optimization) představují poměrně novou metodu diskrétní optimalizace. Inspirace pochází z chování skutečných mravenčích kolonií. Aplikací jednoduchých pravidel, jimiž se řídí jednotliví jedinci, vzniká komplexní chování celku, schopné řešit složité optimalizační úlohy [4]. Jedná se o algoritmus z rodiny rojové inteligence, stejně jako například optimalizace hejnem částic.

Systémy rojové inteligence [3] se typicky skládají z populace jednoduchých agentů interagujících lokálně mezi sebou a s okolním prostředím. Agenti spolu mohou komunikovat přímo nebo nepřímo působením na lokální prostředí. Ačkoli tyto systémy nemají žádnou centrální kontrolu chování agentů, lokální interakce mezi agenty a jednoduché vzory chování agentů často vedou k emergenci globálního chování.

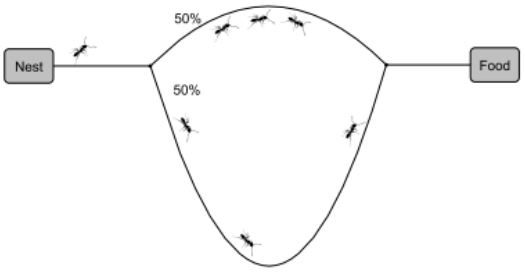
Získávání potravy

Mravenci jsou považováni za sociální hmyz – jejich chování směřuje k zachování kolonie. Nejdříve se mravenci pátrající po potravě nahodile pohybují v okolí mraveniště. Jakmile naleznou potravu, vrací se stejnou cestou, jakou k potravě dospěli a cestu značí feromonem. Ostatní mravenčí pátrači, kteří narazí na feromonovou stopu, se po ní vydají spíše než aby pokračovali v průzkumu okolí mraveniště. Čím více mravenců se pohybuje mezi zdrojem potravy a mraveništěm, tím silnější je feromonová stopa a tím větší je pravděpodobnost, že přitáhne další mravence (tím atraktivnější je pro ostatní mravence).

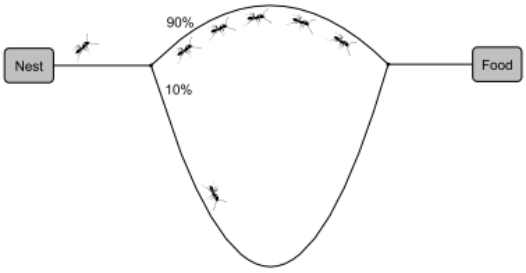
Popis tohoto chování, vychází z laboratorního pokusu známého jako experiment s dvojitým mostem (Double Bridge Experiment) [4]. Během pokusu s dvojitým mostem bylo hnízdo mravenců druhu *Limepithema humile* propojeno se zdrojem potravy dvěma trasami. Mravenci se časem naučili používat krátkou trasu a když jim byla nabídnuta nová, ještě kratší cesta, nedokázali ji využít. Mravenčí stroj uvázl v lokálním extrému řešeného problému. Feromonová stopa značící původně nejkratší trasu byla tak výrazná, že se novou cestou vydávalo příliš málo průzkumníků na to, aby ji označili jako nejvýhodnější. Experiment je znázorněn na obrázcích 2.11 a 2.12.

Algoritmus

Mravenčí optimalizace [4] probíhá v několika po sobě jdoucích iterací. V každé iteraci provede m mravenců své úkoly a vyhodnotí se cesty, které objevili. Proces optimalizace končí splněním ukončujících podmínek, ke kterým může patřit nalezení optimálního řešení, určitý počet iterací bez zlepšení nebo jednoduše dosažení určitého počtu iterací. Tento algoritmus je popsán pseudokódem 6.



Obrázek 2.11: Počátek mravenčího algoritmu. Mravenci si cestu vybírají se stejnou pravděpodobností.



Obrázek 2.12: Stav při konci algoritmu. Feromonová stopa na kratší cestě zvyšuje pravděpodobnost výběru právě té trasy.

Algoritmus 6 Algoritmus mravenčí kolonie

Input: $n_{max}, k_{all}, f(x), d(x)$ ▷ f(x) — hodnotící funkce, d(x) — funkce vzdálenosti dvou bodů

```
Vygeneruj počáteční feromonovou matici P aby odpovídala topologii dané úlohy;
n := 0;
while nesplněny ukončující podmínky do
    Rozmísti mravence náhodně na vrcholy grafu
    for all k in k_all do
        Jdi vpřed n kroků v grafu. Vybírej směr podle pravděpodobnostního pravidla;
        Vyhodnoť kvalitu řešení
        Ulož nové feromony na svou cestu;
    end for
    Aplikuj vypařování feromonů;
    n := n + 1;
end while
```

Umělí mravenci

Pro účely implementace optimalizačních algoritmů je vytvářen umělý mravenec, jehož chování je inspirováno chováním toho skutečného. Ovšem oproti skutečným mravencům byly některé vlastnosti umělých mravenců posíleny tak, aby došlo ke zlepšení výsledků algoritmů řešících konkrétní problémy [4].

- Umělý mravenec pracuje v diskrétním světě.
- Není zcela slepý a je možné při výběru cesty aplikovat heuristických funkcí pro vylepšení hledání, případně vyváznutí z lokálního minima.
- Množství feromonu může být ovlivněno cenou řešení, feromon také nemusí být vypouštěn za každé situace. (například existují implementace, ve kterých feromon vypouští pouze mravenec s nejlepším řešením).
- Mravenec má paměť s vlastními vnitřními stavy.

Heuristická funkce pro pravděpodobnost přesunu

Umělý mravenec k nacházející se ve vrcholu grafu $G = (N, A)$ se přesune do sousedního vrcholu j s pravděpodobností p_{ij}^k [4]

$$p_{ij}^k = \frac{\tau_{ij}^\alpha * \eta_{ij}^\beta}{\sum_{l \in N_i^k} (\tau_{il}^\alpha * \eta_{il}^\beta)}$$

kde N_i^k představuje množinu všech vrcholů dostupných mravenci k z vrcholu i, τ_{ij} odpovídá množství feromonů na hraně spojující vrchol i a j. η_{ij} vyjadřuje informaci vhodnosti přechodu dané hrany (běžně určenou jako "cena hrany").

Ukládání feromonů na trasu

Konkrétní trasa k-tého mravence T^k, délka trasy C^k a množství potravy nalezené mravencem L^k poté vyjádří kvalitu nalezeného řešení. Tato kvalita je následně použita pro výpočet nové feromonové trasy podle vztahu z literatury [4]:

$$\Delta\tau_{ij}^k = \frac{1}{C^k}$$

pro každou hranu a_{ij} náležící do T^k. Následná aktualizace je poté vyjádřena jako:

$$\tau_{ij} = \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k$$

Mimo ukládání feromonů dochází také k vypařování [4]. V základní variantě optimalizace mravenčí kolonií se po každé iteraci zmenší množství feromonu uloženého na každé hraně podle vztahu:

$$\tau_{ij} = (1 - \rho) * \tau_{ij}$$

Koeficienty α, β a ρ jsou obecnými parametry algoritmu určenými uživatelem.

Kapitola 3

Návrh a implementace

Následující kapitola nejprve popisuje veřejné rozhraní modulu pro knihovnu SIMLIB/C++ a následně seznamuje čtenáře s implementací jednotlivých optimalizačních algoritmů a jejich rozhraním. V závěrečné sekci budou prezentovány výsledky jednotlivých metod na vybraných testovacích modelech.

Navržené řešení bylo implementováno v jazyce *C++* na operačním systému *Linux Mint*. Zdrojové kódy byly také otestovány i na jiných operačních systémech, například *Windows 10*, kde jsou plně funkční.

Deklarace tříd použitých pro veřejné rozhraní se nachází v hlavičkových souborech `Parameters.h`, `Bridge.h`, `CostFunction.h`, `Optimization.h`.

3.1 Návrh rozhraní

Před samotnou implementací algoritmů a jejich testováním bylo třeba navrhnout vhodné rozhraní. Vzhledem k informacím získaným při průzkumu ostatních optimalizačních nástrojů, knihovny SIMLIB/C++, pro kterou má být modul stvořen, a spolu s ohledem na požadavek jednoduché rozšiřitelnosti byl zvolen strukturální návrhový vzor Bridge. Svými vlastnostmi umožňuje jednoduchou rozšiřitelnost s využitím polymorfismu a zapouzdření.

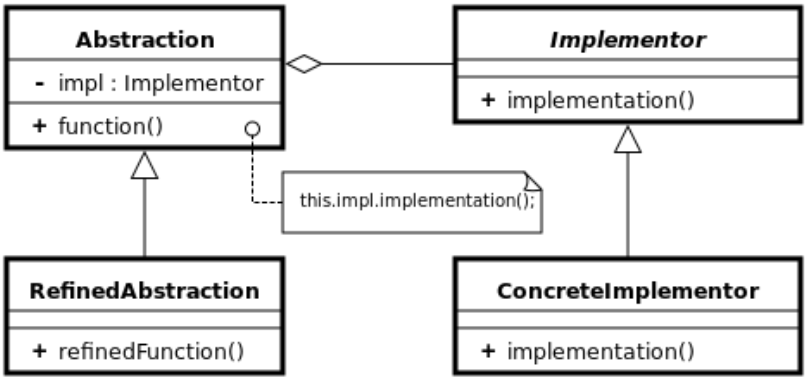
3.1.1 Využití návrhového vzoru Bridge

Bridge je návrhový vzor pro strukturu objektů. Používá se, když chceme oddělit abstrakci od její implementace tak, aby se obě mohly měnit nezávisle. Klient posléze využije některou z implementací nepřímo prostřednictvím abstrakce. [5]. Jeho výhody jsou :

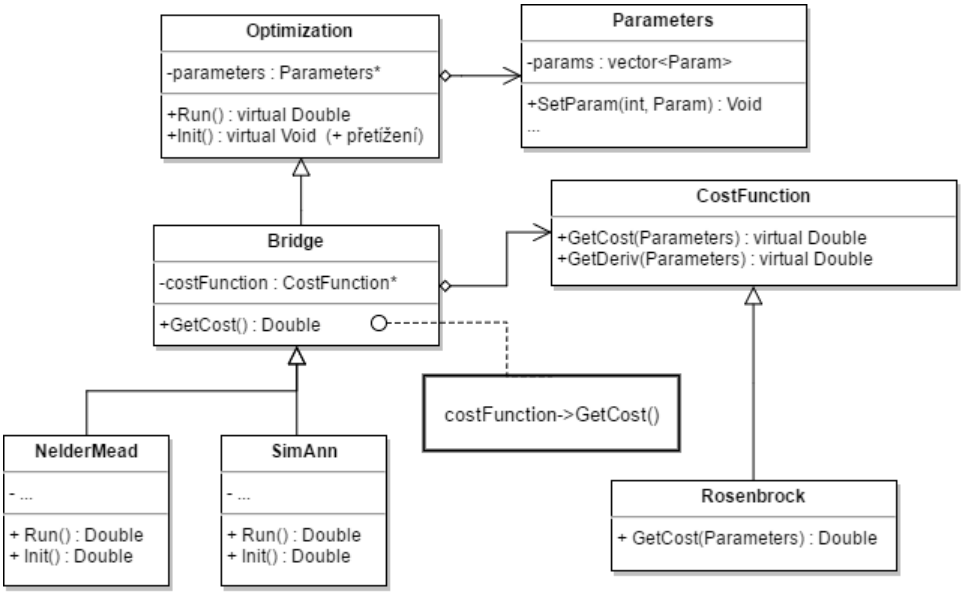
- zabraňuje pevnému propojení mezi implementací a abstrakcí
- abstrakce i implementace jsou nezávisle rozšiřitelné
- schovává implementaci před klientem

Jednoduchý diagram [5] na obrázku 3.1.

V implementovaném module je Bridge využit pro přemostění funkčního objektu reprezentujícího hodnotící funkci a samotného optimalizačního algoritmu 3.2.



Obrázek 3.1: Myšlenka návrhového vzoru most.



Obrázek 3.2: Konkrétní využití mostu v optimalizačním modulu.

3.1.2 Výsledné veřejné rozhraní

S ohledem k požadavkům na rozšířitelnost, jednoduchou správu kódu a možného použití externích knihoven bylo za použití návrhového vzoru Bridge vytvořeno následující rozhraní.

Hodnotící funkce

Návrh hodnotící funkce je záležitost uživatele. Bylo tedy třeba připravit takovou reprezentaci, kterou může uživatel jednoduše rozšířit jeho vlastním modelem a předat jako parametr optimalizační metodě.

Proces vytvoření nového modelu k optimalizaci je následující — Uživatel si vytvoří vlastní třídu, která je potomkem třídy *CostFunction*, implementuje virtuální metodu `GetCost(Parameters*)`, do které umístí model, který si přeje optimalizovat. Objekt, reprezentující vstupní parametr modelu získá z ukazatele *Parameters* metodou `Param GetParam(const char* name)` podle jména parametru (například "x") nebo `Param GetParam(int index)` podle pořadí, v jakém byl přidán. K získání samotné hodnoty je poté třeba zavolat ještě metodu `double GetValue()`.

Tento příklad je znázorněn na diagramu 3.2 třídou *Rosenbrock*, která reprezentuje Rosenbrockovu hodnotící funkci.

Všechny metody, které má uživatel k dispozici k virtuálnímu přepsání jsou :

- `double GetCost(Parameters*)` pro hodnotící funkci
- `bool IsInConstraints(Parameters*)` pro funkci omezení
- `vector<double> PartialDerivations(Parameters*)` pro funkci parciálních derivací
- `double Distance(unsigned x, unsigned y)` pro případ, kdy je optimalizovaným modelem graf. Tato metoda poté slouží k získání vzdálenosti mezi bodem x a y v grafu.

Optimalizační metody

Požadavkem na rozhraní optimalizačních metod je rozšiřitelnost a jednoduchost použití. Každá optimalizační metoda dědí přímo z objektu *Bridge* a nepřímo z *Optimization*. Instanciace objektu optimalizační metody vyžaduje instanci třídy parametrů *Parameters* a instanci *CostFunction* reprezentující cenovou funkci. Dále, pokud si uživatel přeje zaznamenávat průběh vyhodnocování řešení, předá jako třetí parametr ukazatel na instanci třídy *Logger*. Této třídě se bude věnovat podsekce 3.1.2. Pro představu, jak pracovat s třídou *Parameters* je přidán obrázek 3.3.

Pro případ, kdy uživatel nechce reprezentovat hodnotící funkci objektem, existuje přetížení konstruktoru, který namísto objektu přijme ukazatel na funkci. Stejným způsobem je možné předat i funkci omezující hodnocení a funkci diferenciálních počtů. Je vhodné poznamenat, že takto předané funkce musí implementovat stejnou typovou signaturu, kterou by implementovaly jako metody objektu popsány v předchozí podsekci.

Ukazatele na funkci ceny, omezující funkci a funkci derivací jsou deklarovány v souboru *OptDefs.h* jako *CostFuncPtr*, *ConstraintsFuncPtr* a *PartDerivFuncPtr*.

Param
name : string patamType : enumeration value,max,min : double
Param<T>(string, T, T) GetValue() : double GetName() : string

Parameters
params : vector<Param>
AddParameter<T>(string, T, T) : void operator[] (int) : Param GetParam(int) : Param GetParam(string) : Param

Obrázek 3.3: Konkrétní implementace třídy Parameters a Param ve stylu třídního diagramu.

Parametry modelu

Hledané parametry minima jsou reprezentovány třídou *Parameters*. Vzhledem k různorodosti problémů, především různému druhu a počtu parametrů modelu, je instanciace této třídy a její naplnění prvotními daty zodpovědností uživatele. Po skončení optimalizačního algoritmu bude právě tento objekt obsahovat parametry, minimalizující zadanou hodnotící funkci. Instanciace a naplnění této třídy vypadá následovně:

1. Uživatel vytvoří ukazatel na instanci třídy *Parameters*.
2. Pro přidání parametru využije metodu `void AddParameter<datovyTyp>(...)`, které předá jméno reprezentující novou proměnnou (například "x") , minimální a maximální hodnotu, jaké může tato proměnná dosáhnout. Šablonový parametr *datovTyp* - *double* nebo *integer* - rozhoduje o typu parametru. *Double* je běžná volba. V případě *int* bude třída a optimalizační systém předpokládat, že se jedná o celočíselný vstupní parametr, a bude s ním takto i pracovat. Pokud proměnná není omezena, existuje přetížení přijímající jméno a počáteční hodnotu parametru.

   Více podrobností o třídě znázorněno na obrázku 3.3.

Třída *Logger*

Pro zaznamenávání postupu hledání minima do souboru je připravena jednoduchá třída *Logger*. Při instanciaci vyžaduje cestu k souboru, do kterého budou následně zaznamenávány výsledky. Pokud si uživatel přeje spouštět optimalizaci vícekrát, třída umožňuje vytvářet histogram. V takovém případě je třeba jej inicializovat metodou `void HistogramInit(double firstTreshold, double secondTreshold, double increaseFactor, unsigned initialAmount)`. Význam jednotlivých parametrů je následující :

- `firstTreshold, secondTreshold` — hranice prvního rozmezí histogramu
- `increaseFactor` — činitel, o který se budou postupně zvyšovat hranice rozmezí histogramu.
- `initialAmount` — počáteční množství sloupců histogramu, zvyšovaných postupně o *increaseFactor*. V případě záznamu, který by byl mimo rozmezí budou automaticky vytvořeny další sloupce.

Po ukončení optimalizace je následně třeba zavolat metody `void PrintBasicStatistics()` — pro vytištění zmíněného histogramu a nejlepšího výsledku, a volitelně `Close()` pro uzavření souboru.

3.1.3 Spuštění optimalizace

Po vytvoření vlastního potomka třídy *CostFunction*, vytvoření *Parameters* a přidání potřebných parametrů, již k optimalizaci stačí pouze vytvořit instanci vybraného optimalizačního algoritmu, inicializovat a spustit. Inicializace probíhá metodou `void Init_ZkratkaMetody (parametrymetody)`, kterou jsou předány všechny potřebné parametry pro optimalizaci (běžně různé koeficienty). Ke spuštění stačí poté zavolat metodu `Run()`.

3.2 Implementace metod

Následující sekce si klade za cíl povrchně popsat implementaci jednotlivých metod. Popis každé metody bude rozdělen do několika částí — v první části popíše části algoritmu, jejichž implementace není pevně daná, případně vyzdvihne prvky rozdílné od běžného způsobu implementace. V druhé části bude popsáno veřejné rozhraní, poslední část se věnuje řídícím parametrům algoritmů.

3.2.1 Simulované žíhání

Metoda je realizována třídou *SimAnn*, deklarovanou v hlavičkovém souboru `SimulatedAnnealing.h` a implmentovanou v `SimulatedAnnealing.cc`. Algoritmus používá dvou cyklů, jeden pro snižování teploty a druhý v rámci algoritmu Metropolis. Tento princip byl již popsán v sekci 2.3.1 a je zde jen připomenut. Kritickými částmi algoritmu jsou způsob generování sousedního stavu, tedy *Perturbance*, popsán v 2.3.1 a chladící rozvrh. U chlazení systému je možné vybrat si mezi několika implementovanými způsoby podle řídícího parametru *eCoolingSchedule*.

Chladící rozvrh

Je realizován třídou *Cooling*. Kromě klasického exponencionálního chlazení, které je popsáno v 2.3.1, umožňuje chladit i logaritmicky, trigonometricky nebo lineárně (v lineárním případě je třeba vybrat jestli aditivně nebo multiplikativně). Výběr z těchto druhů chlazení se provádí nastavením parametru *cooling* na jednu z hodnot výčtového typu *eCoolingSchedule*. Dostupné hodnoty a rozvrh, kterým chladí systém, je znázorněn v tabulce 3.1. Implementováno na základě článku [2].

Počet chladících iterací je vypočítán jako $i = \frac{log(finalTemp/initTemp)}{log(0.95)}$

Generování sousedního stavu

Tento jev již je popsán v podsekci 2.3.1 jako Perturberance. Zkráceně se jedná o simulaci pohybu molekul při chlazení oceli, což využijeme pro generování nového sousedního vztahu. Vztah :

$$x_{t+1} = x_t + |x_{high} - x_{low}| * (Random() - 0.5) * 2 * \frac{T_n}{T_{max}}$$

implementovaný metodou *Perturberance()*, představuje právě tento šum a nestabilitu stavu.

Rozhraní

Dostupné konstruktory :

E_Default	$T_{n+1} = T_n - 1$
E_ExponentialMultiplicative	$T_{n+1} = T_0 * n^{\alpha}$
E_LogarithmicalMultiplicative	$T_{n+1} = \frac{T_0}{1+\alpha * log(1+n)}$
E_LinearMultiplicative	$T_{n+1} = \frac{T_0}{1+\alpha * n}$
E_LinearAdditive	$T_{n+1} = T_n + (T_0 - T_e) * \frac{i-n}{i}$
E_TrigonometricAdditive	$T_{n+1} = T_n + \frac{1}{2} * (T_0 - T_e) * (1 + cos(n * \frac{\pi}{i}))$

Tabulka 3.1: Tabulka výčtových typů a jejich chladícího rozvrhu. i je celkový počet chladících iterací, T_e je finální teplota a α je koeficient speciální pro jednotlivé rozvrhy, n je momentální chladící iterace

- `SimAnn(Parameters* par, CostFunction* cF, Logger* l)` — základní konstruktor. Vytvoření objektu třídy *Parameters*, *CostFunction* a *Logger* bylo již popsáno v předešlé sekci. Parametr l je nepovinným.
- `SimAnn(Parameters* par, CostFuncPtr costPtr, ConstraintsFuncPtr constrPtr, Logger* l)` — tento konstruktor je spíše experimentální. Umožňuje obejít princip návrhového vzoru Bridge a předat metodě ukazatel na uživatelskou hodnotící funkci a omezující funkci mimo rozhraní *CostFunction*.

Řídící parametry

Řídící parametery pro algoritmus Simulovaného žíhání jsou předány metodou `Init_Sa(double initTemp, unsigned stateTransitions, double finalTemp, double sigma, eCoolingSchedule cooling)`, deklarovanou v třídě *Optimization* a implementovanou v *SimAnn*.

- `initTemp` — počáteční teplota
- `stateTransitions` — počet cyklů algoritmu Metropolis pro nalezení optimálnějšího sousedního stavu
- `finalTemp` — konečná teplota
- `sigma` — povolená odchylka od konečné teploty.
- `cooling` — výčtový typ určující způsob chlazení. Různé chladící rozvrhy a jejich ekvivalentní výčtový typ jsou uvedeny v tabulce 3.1.

3.2.2 Nelder-Mead simplexová metoda

Realizována třídou *NelderMead*, která je deklarována v modulu `NelderMead.h`. Implementaci lze nalézt v `NelderMead.cc`.

Třída *Vertices* a *SimplexPoint*

Pro účely práce se Simplexem byly vytvořena třídy *Vertices* a *SimplexPoint*. *Vertices* reprezentuje celý Simplex a *SimplexPoint* zase jeden bod v simplexu. V implementaci je využito variadických šablon C++11, které, spolu například s rekurzí, umožňují inicializovat simplexový bod libovolným počtem parametrů a tedy pro jakoukoliv dimenzi.

Simplexový bod je přidáván do třídy *Vertices* metodou `AddSimplexPoint(...)`.

Rozhraní

Dostupné konstruktory jsou v podstatě stejné, jako u simulovaného žíhání:

- `NelderMead(Parameters* par, CostFunction* cF, Logger* l)`
- `NelderMead(Parameters* par, CostFuncPtr costPtr, ConstraintsFuncPtr constrPtr, Logger *l)`

Řídící parametry

Analogicky s metodou simulovaného žíhání se řídící parametry nastavují metodou *void Init_NM(Vertices& startSimplex, double epsilon, unsigned maxIter, double alpha, double gamma, double rho, double sigma).*

- `startSimplex` — počáteční Simplex vytvořený uživatelem
- `epsilon` — nejmenší dovolený rozdíl mezi dvěma výsledky
- `maxIter` — maximální počet iterací před ukončením algoritmu.
- `alpha`, `gamma`, `rho`, `sigma` — koeficienty reflexe, expanze, kontrakce a zmenšení. Jedná se o nepovinné parametry, které jsou ve výchozím stavu nastaveny na hodnoty 1, 2, 0.5, 0.5. Více jsou tyto parametry popsány v podsekci 2.3.2.

3.2.3 Sdružené gradienty

Je reprezentována třídou *ConjugateGradient*. Deklarace se nachází v souboru `ConjugateGradient.h` a definice ve stejně pojmenovaném `.cc` souboru. Metoda využívá gradientu k nalezení minima. Ten, spolu s dalšími koeficienty, určuje směr, kterým se pohnout. Více informací o algoritmu je uvedeno v podsekci 2.4.1. Metoda pracuje s maticemi, pro práci s nimi bylo tedy třeba vytvořit objektovou reprezentaci. Tento jednoduchý podsystém se nachází v modulu `Matrix.h`.

Matice

Pro potřeby gradientních (a případně i jiných) metod byla vytvořena jednoduchá třída reprezentující matici *Matrix*. Parametry konstruktoru třídy je počet sloupců a řádků požadované matice. Prvky matice jsou poté přístupné přes operátory [] stejně, jako například dvourozměrné pole.

Třída implementuje statické metody pro operace s maticemi, které vypadají následovně:

- `Matrix MatrixMul(Matrix A, Matrix B)`
- `Matrix MatrixAdd(Matrix A, Matrix B)`
- `Matrix MatrixTranspone(Matrix A)`
- `Matrix MatrixMulScalar(Matrix A, double scalar)`

Line-Search

Metoda používá backtrackingový line search se zmenšovacím faktorem 0.9 a silnými Wolfovými podmínkami, tak jak jsou popsány v podsekci 2.4.1.

Výpočet koeficientu β

Koeficient β určuje směr, kterým se metoda vydá při hledání minima, jedná se tedy o nejkritičtější část metody. Více informací je v podsekci 2.4.1. Implementovaný vztah pro výpočet je založen na navrženému výpočtu z článku [1], kde je také podán důkaz konvergence pro tento způsob. Zmíněny výpočet vypadá následovně:

$$\beta_k^{HZ*} = \frac{||g_k||^2 - (||g_k||/||g_{k-1}||) * |g_k^T g_{k-1}|}{-g_{k-1}^T * d_{k-1} + \Theta * |g_k^T * d_{k-1}|}$$

Rozhraní

Dostupné konstruktory jsou v podstatě stejné jako u simulovaného žíhání:

- `ConjugateGradient(Parameters* par, CostFunction* cF, Logger* l)`
- `ConjugateGradient(Parameters* par, CostFuncPtr costPtr, PartDerivFuncPtr derivPtr, Logger *l)` — jiná varianta speciálního konstruktoru. Namísto ukazatele na funkci omezení je zde ukazatel na funkci parciálních derivací.

Řídící parametry

Vzhledem k charakteru gradientních metod a jejich neschopnosti překonat lokální minimum jsou, na rozdíl od ostatních metod, připraveny dvě inicializační metody pro předání parametrů.

- `void Init_CG(double convergenceEpsilon, Parameters initPosition, int iterations)` — pro případ jednoho spuštění, uživatel musí v takovémto případě poskytnout počáteční bod.
- `void Init_CG(double convergenceEpsilon, unsigned tries, int iterationsPerTry)` — pro inicializace opakovaného spuštění. Restartování není řízeno žádnou heuristickou či pravděpodobnostní funkcí.
- `convergenceEpsilon` — hraniční rozdíl mezi hodnotou dvou řešení před ukončením metody.
- `initPosition` — úvodní bod, ze kterého bude metoda vycházet.
- `iterations, iterationsPerTry` — maximální počet iterací vylepšování řešení.
- `tries` — počet restartování algoritmu s náhodným počátečním bodem.

3.2.4 Evoluční strategie

Implementována třídou *EvolutionStrategy*. Deklarace se nachází v souboru `EvolutionStrategy.h`. Definice v `EvolutionStrategy.cc`.

Metoda hledá nejlepšího jedince v populaci, kde každý jedinec reprezentuje řešení. Každý jedinec je chromozomem, který se zase na oplátku skládá z genů, nad kterými jsou prováděny operace za pomoci genetických operátorů. Více o metodě evoluční strategie lze nalézt v podsekci 2.5.1.

Reprezentace chromozomů a genů

Každý gen chromozomu představuje jeden parametr optimalizovaného modelu. Tento vztah je v implementaci vyjádřen třídou *Chromozome* a strukturou *Gen*, která obsahuje datový typ *union Solution*. *Solution* je tvořen speciálním datovým typem *myint32* a klasickým typem *float*. Binární délka těchto typů je stejná, a tedy při přístupu k opačnému typu (z celočíselného na desetinný a obráceně) k binární, namísto implicitní typové konverze. Nevýhodou tohoto přístupu je fakt, že není možné využít zvýšené přesnosti typu *double*, výhodou je jednoduchost práce a převodu typů.

Genetické operátory

Genetické operátory byly již popsány v podsekci 2.5.1. V modulu jsou tyto operátory implementovány jako metody, které za pomoci binárních operací upravují celočíselný prvek genu *Solution*.

- *Selekce* je prováděna turnajovým výběrem vždy ze dvou náhodných prvků.
- *Křížení* je implementováno dvoubodově tak, jak je popsáno v podsekci 2.5.1. Rodiče jsou vybíráni náhodně z množiny jedinců, vytvořených v předchozím kroku.
- *Mutace* je jednobodová, přesněji jedno-bitová. Který z jedinců, a který z bitů, je vybráno náhodně na základě zadaného zadané šance k mutaci. Je vhodné poznamenat, že pro vybraného jedince bude zmutován jeden bit každého z genů.

Rozhraní

Dostupné konstruktory pro tuto metodu se neliší od ostatních, jsou tedy následující:

- `EvolutionStrategy(CostFunction *backend, Parameters* parameters, Logger* l)` — jako u ostatních konstruktorů je parametr *l* nepovinný.
- `EvolutionStrategy(Parameters* par, CostFuncPtr functionPtr, ConstraintsFuncPtr constraintPtr, Logger* l)` — stejně jako u ostatních metod je tento konstruktor použit k předání hodnotící a omezující funkce ukazatelem, namísto zapouzdření do třídy *CostFunction*.

Řídící parametry

Jsou předání touto metodou `void Init_ES(int populationSize, int numberOfGenerations, int tournametSize, float crosoverRate, float mutationRate)`.

- `populationSize` — velikost populace

- `numberOfGenerations` — počet vytvořených generací, než skončí algoritmus. Prakticky se jedná o počet iterací.
- `tournamentSize` — velikost turnaje, tedy velikost množiny jedinců vybraných pro křížení
- `crossoverRate` — pravděpodobnost jedince ke křížení, v rozmezí $[0, 1]$
- `mutationRate` — pravděpodobnost jedince ke zmutování jeho genů v rozmezí $[0, 1]$.

3.2.5 Optimalizace mravenčí kolonií

Pro metodu optimalizace mravenčí kolonií nevznikly žádné speciální struktury. Bylo ovšem třeba přidat další metodu do třídy *CostFunction*, která poskytuje vzdálenost mezi dvěma přijatými body.

Třída představující optimalizaci mravenčí kolonií se jmenuje *AntColony*, je deklarována v hlavičce `AntColony.h` a definována v souboru `AntColony.cc`.

Úprava hodnotící funkce

Do třídy hodnotící funkce *CostFunction* přibyla nová metoda `double Distance(unsigned x, unsigned y)`. Jak již název napovídá, metoda vrací vzdálenost mezi dvěma sousedními body x a y v grafu. Pokud tento přechod neexistuje, metoda se zavazuje k vrácení speciální hodnoty označující tuto skutečnost. Tato zástupná hodnota je také předána metodě při inicializaci. Vhodným kandidátem je například znak `NaN` nebo `INF`.

Graf

Graf, který má být optimalizování, je typicky reprezentován dvourozměrným polem. Parametry metody `double Distance(unsigned x, unsigned y)` poté označují index řádku a sloupce, a vzdálenost těchto bodů se nachází ve zmíněném poli právě na těchto indexech.

Rozhraní

Rozhraní se neliší od ostatních optimalizačních metod. Opět je připravena inicializační metoda `void Init_ACO(ACO_Options acoOptions, double ethaCoeff, double tauCoeff, double pheromonesQCoeff, double pheromonesEvaporationCoeff, double maxStartingPheromoneValue )`

kterou jsou předány parametry optimalizačního algoritmu.

Řídící parametry

Vzhledem k velkému množství vstupních parametrů metody byla vytvořena struktura *ACO_Options*, která je zapouzdřuju. Prvky této struktury — a tedy i řídící parametry algoritmu — jsou následující:

- `int numberOfAnts` — počet mravenců, kteří budou hledat cestu
- `int numberOfNodes` — počet uzlů v optimalizovaném grafu
- `int startingIndex` — pořadové číslo (index) uzlu, na kterém mravenci začínají

- `int endingIndex` —- pořadové číslo (index) uzlu, ke kterému je třeba najít cestu
- `int iterations` — počet iterací hledání cesty mravenci
- `double missingEdgeValue` — hodnota, která v maticové reprezentaci grafu vyjadřuje neproveditelný přechod
- `bool traverseAll` — příznak, určující
- `int triesBeforeTimeout` — nepovinný parametr. Určuje kolikrát se má algoritmus pokoušet najít validní cestu, pokud se mu to nepodaří napoprvé. Výchozí hodnotou tohoto parametru je 250.

Další argumenty funkce `Init_ACO()` mají následující význam:

- `ethaCoeff` — určuje váhu, kterou má při výběru cesty pro mravence cena. Výchozí hodnota je 0.5.
- `tauCoeff` — určuje váhu, kterou mají při výběru cesty pro mravence feromony. Výchozí hodnota je 0.8.
- `pheromonesQCoeff` — určuje podíl feromonu vůči cenně řešení, který mravenec zanechá na přechodech. Výchozí hodnota je 80.
- `pheromonesEvaporationCoeff` — rychlost vypařování feromonů. Výchozí hodnota je 0.2.
- `maxStartingPheromoneValue` — maximální hodnota počátečních náhodných feromonů. Výchozí hodnota je 2.0.

Optimalizací mravenčí kolonií je tedy možné podle kombinace parametrů *traverseAll*, *startingIndex* a *endingIndex* hledat několik různých cest :

- Nejkratší cestu mezi bodem A a B
- Takovou Hammingovu kružnici, která protne všechny body grafu (tzv. problém obchodního cestujícího)
- „Nejrychlejší“ okruh
- Cestu mezi body A a B, která po cestě protne všechny ostatní body grafu.

3.3 Testovací příklady

Implementované metody pro spojitou optimalizaci byly podrobeny třem optimalizačním úlohám. Vzhledem k jednoduché grafické reprezentaci se jedná o běžné matematické funkce a nikoliv o simulační modely, funkčnost optimalizačních úloh není tímto faktem poznamenána. Jedná se o funkci Rosenbrockovu 3.13, Bealeho 3.11 a Rastriginovu 3.12. Tyto funkce byly převzaty ze stránky dostupné z odkazu [16]. Dále byl, pro testování optimalizace grafů, vytvořen jednoduchý graf o osmi uzlech s náhodně vybranou cenou přechodů.

Pro každou testovací úlohu bylo vykonáno tisíc běhů každé z optimalizačních metod. Algoritmy byly vykonány vždy se stejnými parametry. Zaznamenáváno bylo počet prováděných experimentů jednotlivými metodami, výsledky a přesnost nejlepšího výsledku k reálnému optimu.

3.3.1 Popis testovacích funkcí

Následující sekce popíše matematické funkce, nad kterými byly prováděny testy. Funkce byly vybrány tak, aby měli odlišné vlastnosti a byl na nich znázorněn charakter implementovaných metod. Testovací modely se nacházejí v souborech `Rosenbrock.h`, `Beale.h`, `Rastering.h` a `TravelingSalesman.h`.

Rosenbrockova funkce

Rosenbrockova funkce 3.13 [16], označována také za "banánovou"funkci, je jedna z nejpoužívanějších pro testování optimalizačních algoritmů.

Funkce je unimodální, její globální minimum leží v úzkém, parabolickém údolí. Toto údolí je většinou jednoduché najít, funkce testuje spíše schopnost metod konvergovat k minimu.

Je vyjádřena vztahem

$$f(\mathbf{x}) = \sum_{i=1}^{d-1} [100 * (x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$$

Optimum se nachází v bodě $\mathbf{x^*} = (1, ..., 1)$ a $f(\mathbf{x^*}) = 0$. Funkce je v tomto případě omezena na interval $[-1.5, 1.5]$ pro x a $[-0.5, 1.5]$ pro y. Dalším omezením jsou funkce $(x-1)^3 - y + 1 < 0$ a $x + y - 2 < 0$.

Výsledky hledání optima této metody je znázorněno na obrázku **??**. Počet provedených experimentů jednotlivými metodami zase na 3.6.

Bealeho funkce

Baeleho funkce 3.11 [16] je multimodální, na krajích má ostré vrcholy a ve svém středu je téměř rovná.

Je vyjádřena vztahem

$$f(\mathbf{x}) = (1.5 - x_1 + x_1 * x_2)^2 + (2.25 - x_1 + x_1 * x_2^2)^2 + (2.625 - x_1 + x_1 * x_2^3)^2$$

Optimum se nachází v bodě $\mathbf{x^*} = (3, 0.5)$ a $f(\mathbf{x^*}) = 0$.

Funkce je v běžně omezována na interval $[-4.5, 4.5]$ pro x i y.

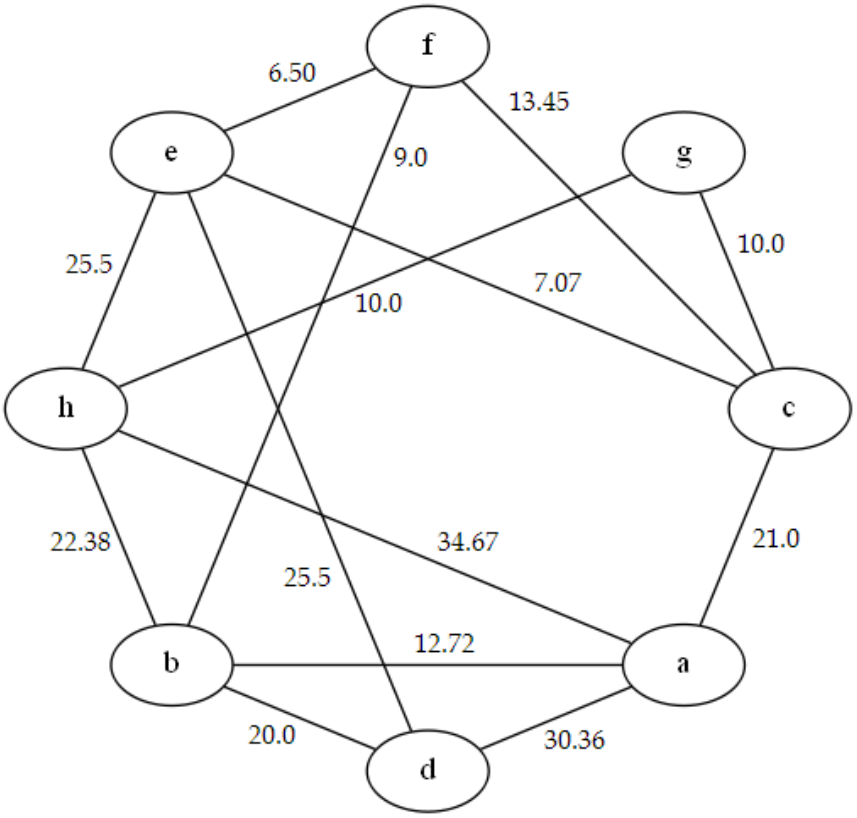
Výsledky hledání optima této metody je znázorněno na obrázku 3.10. Počet provedených experimentů jednotlivými metodami zase na 3.7.

Rastriginova funkce

Rastriginova funkce 3.12 [16] má několik lokálních minim. Je vysoce multimodální, ovšem minima jsou rozdělena pravidelně po povrchu. Toto téměř znemožňuje použití gradientních metod.

Je vyjádřena vztahem

$$f(\mathbf{x}) = A * d + \sum_{i=1}^{d} [x_i^2 - 10 * cos(\omega * x_i)]$$



Obrázek 3.4: Vytvořený graf pro testování optimalizace mravenčí kolonií.

A udává strmost lokálních extrémů a ω je frekvencí jejich výskytu V testovacím modelu jsou nastaveny na $A = 10$, $\omega = 2\pi$.

Optimum se nachází v bodě $\mathbf{x^*} = (0, ..., 0)$ a $f(\mathbf{x^*}) = 0$.

Funkce je v běžně omezována na interval $[-5.12, 5.12]$ pro x i y.

Výsledky hledání optima této metody je znázorněno na obrázku 3.8. Počet provedených experimentů jednotlivými metodami zase na 3.5.

Průchod grafem

Pro potřebu testování průchodu grafem byl vytvořen diskrétní model o osmi uzlech a několika přechodech mezi nimi. Tyto body byly náhodně umístěny do grafu a jejich vektorová vzdálenost udává cenu přechodu. Dále bylo pro ztížení optimalizace odstraněno několik přechodů. Vytvořený graf je na obrázku 3.4.

Referenční parametry

U nezmíněných parametrů se předpokládá jejich výchozí hodnota.

Simulované žíhání bylo spouštěno s parametry
initTemp = 300000 *stateTransitions* = 200 *finalTemp* = 0 *sigma* = 0.01 *cooling* = E_ExponentialMultiplicative

Nelder Mead byl spouštěn s:
startSimplex = náhodně vygenerovaný simplex v rozmezí parametrů funkce *epsilon* = 0.000001 *maxIter* = 1000

Evoluční Strategie měla jako své parametry:
populationSize = 64 *numberOfGenerations* = 16 *tournamentSize* = 32 *crossoverRate* = 0.8 *mutationRate* = 0.2

Sdruženým gradientům bylo předáno:
convergenceEpsilon = 0.00001 *tries* = 1000 *iterationsPerTry* = 100

Mravenčí kolonie.Vzhledem k odlišnosti této metody od ostatních byla mravenčí kolonie testována na stejném případu, ovšem vždy s jiným cílem. Tedy:

- **Problém obchodního cestujícího** měl parametry
  numberOfAnts = 5 *numberOfNodes* = 8 *startingIndex* = 0 (A) *endingIndex* = 0 (A) *iterations* = 10 *missingEdgeValue* = NEG_INF *traverseAll* = true

- **Hledání nejrychlejší cesty mezi body *A* a *F***
  numberOfAnts = 5 *numberOfNodes* = 8 *startingIndex* = 0 (A) *endingIndex* = 5 (F) *iterations* = 10 *missingEdgeValue* = NEG_INF *traverseAll* = false

- **Hledání nejrychlejší cesty mezi body *A* a *F* s protnutím všech ostatních**
  numberOfAnts = 5 *numberOfNodes* = 8 *startingIndex* = 0 (A) *endingIndex* = 5 (F) *iterations* = 10 *missingEdgeValue* = NEG_INF *traverseAll* = true

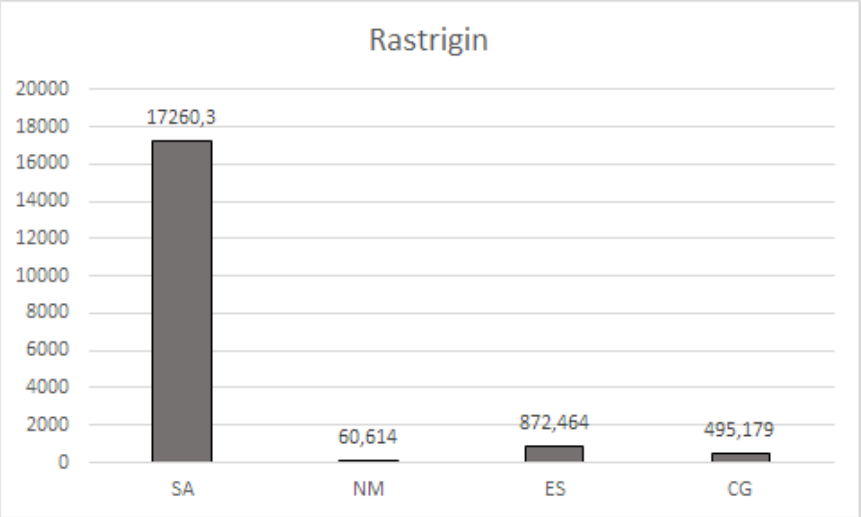
3.3.2 Zhodnocení výsledků

Rosenbrock

Z grafů 3.13, 3.6 a výsledku je patrné, že největší úspěch měla metoda *Nelder Mead*. Správný výsledek nalezla ve 48% případů, její výsledek 3.2 velice přesný a vyžadovala i nejmenší počet experimentů. Nejhůře dopadlo simulované žíhání. Zajímavý je fakt, že přestože model neobsahuje lokální minima, gradientní metoda nebyla schopna vždy konvergovat ke globálnímu minimu (Za tuto skutečnost je pravděpodobně zodpovědná dříve zmíněná silná Wolfova podmínka pro line search [1].)

Beale

Co se týče přesnosti výsledku a konvergenci k němu, je zde jasným vítězem simulované žíhání 3.10, které optimum nalezlo v 99,8% případů. Bohužel k tomu potřebovalo o mnohem více experimentů 3.7, než ostatní metody. Vzhledem k tomu, že přesnost simulovaného žíhání pro tento problém také nebyla tou nejlepší, stojí za úvahu, jestli by lepším zvolením parametrů nedošlo ke stejným výsledkům za méně experimentů.



Obrázek 3.5: Graf znázorňující průměrný počet experimentů provedených metodou k dosažení výsledku optimalizace Rastriginovy funkce 3.12. Na ose x jsou vidět anglické zkratky pro jednotlivé metody (SA - simulované žíhání, NM - Nelder Meadova metoda, ES - Evoluční strategie, CG - sdružené gradienty), osa y znázorňuje provedený počet experimentů. Průměr je vypočten z tisíce běhu optimalizace.

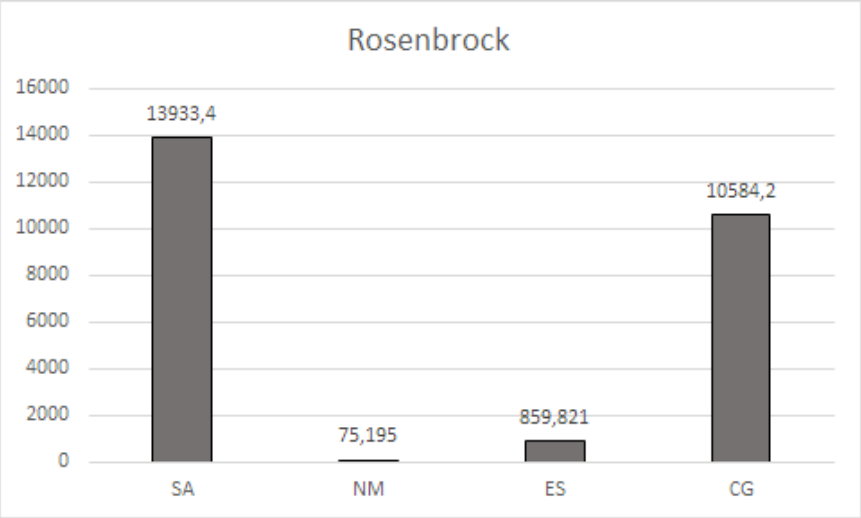
Rastrigin

Pro tento model pro změnu vyhrála evoluční strategie. Do globálního optima se trefila v přibližně 93% všech běhů 3.8. Žádná jiná metoda zde příliš nezapůsobila. Za zmínku stojí skutečnost, že přestože metoda simulovaného žíhání nenašla "skutečné optimum", přibližně 40% všech výsledků stále bylo v rozmezí $[0.01 - 0.1]$, což je v údolí globálního minima. Opět ovšem vyžadovala zdaleka nejvíce experimentů 3.5. Pro tento typ problému se zdá být evoluční strategie, optimum našla přesně! (viz tabulka 3.2)

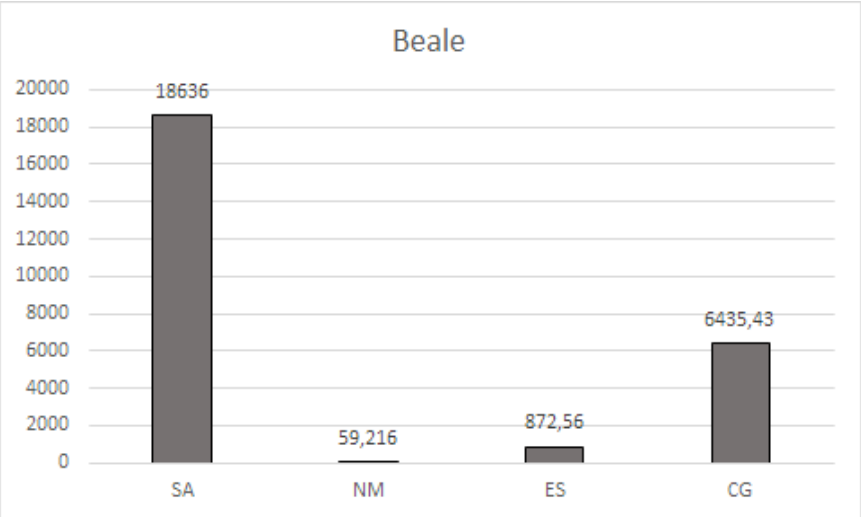
Průchody grafem

Výsledky:

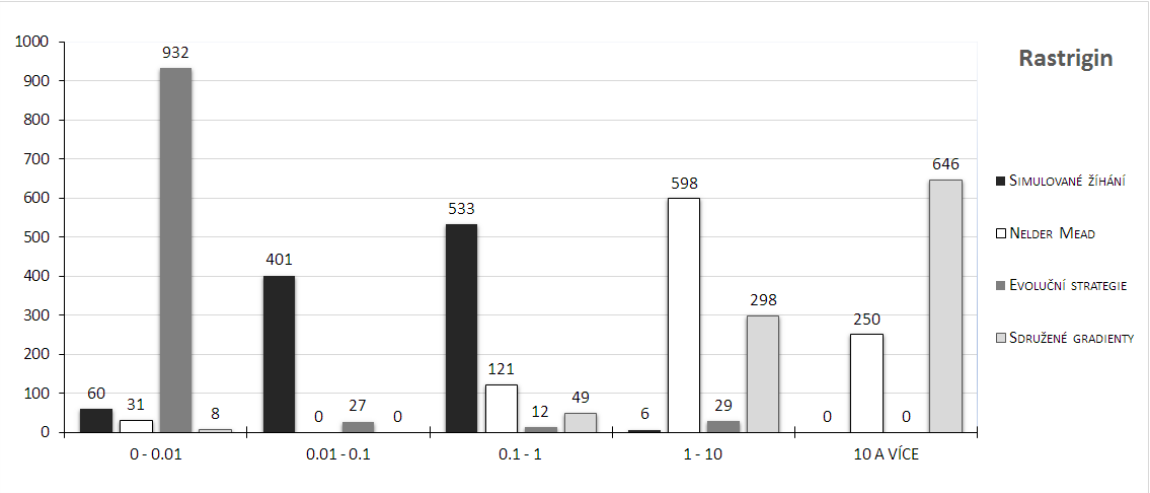
- *TSP* — pro graf 3.4 z bodu A do A byla nalezena cesta $A \to H \to G \to C \to E \to F \to B \to D \to A$ s cenou 127.50. Tato cesta je v grafu pro splnění zadaných podmínek skutečně nejkratší.
- A do F — byla nalezena cesta $A \to B \to F$. Jedná se sice o triviální příklad, potvrzuje ovšem, že algoritmus je schopen z více validních cest vybrat tu nejkratší.
- A do F s protnutím všech ostatních bodů. — nalezena cesta $A \to D \to B \to H \to G \to C \to E \to F$. Toto řešení skutečně splňuje všechny požadavky na něj kladené; protíná všechny uzly, dostává se do konečného bodu a je zároveň i nejlevnějším řešením z pohledu hodnotící funkce.



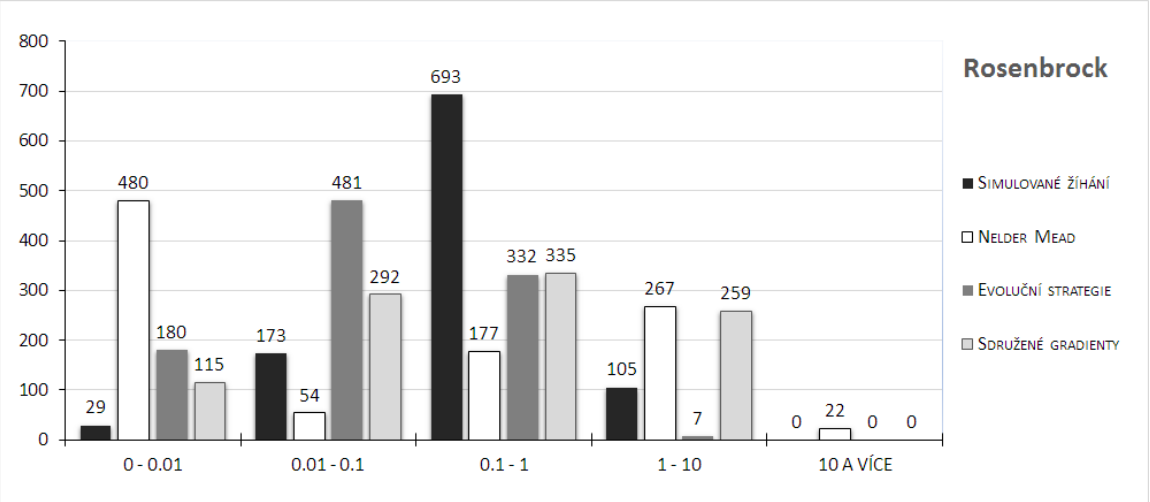
Obrázek 3.6: Graf znázorňující průměrný počet experimentů provedených metodou k dosažení výsledku optimalizace Rosenbrockovy funkce 3.13. Na ose x jsou vidět jsou vidět anglické zkratky pro jednotlivé metody (SA - simulované žíhání, NM - Nelder Meadova metoda, ES - Evoluční strategie, CG - sdružené gradienty), osa y znázorňuje provedený počet experimentů.. Průměr je vypočten z tisíce běhu optimalizace.



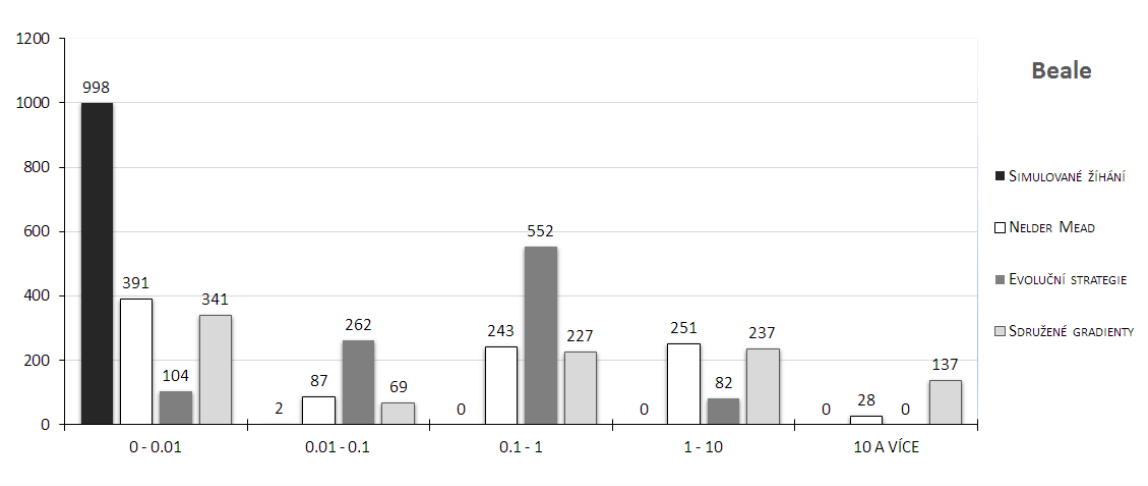
Obrázek 3.7: Graf znázorňující průměrný počet experimentů provedených metodou k dosažení výsledku optimalizace Bealeho funkce 3.11. Na ose x jsou vidět anglické zkratky pro jednotlivé metody (SA - simulované žíhání, NM - Nelder Meadova metoda, ES - Evoluční strategie, CG - sdružené gradienty), osa y znázorňuje provedený počet experimentů. Průměr je vypočten z tisíce běhu optimalizace.



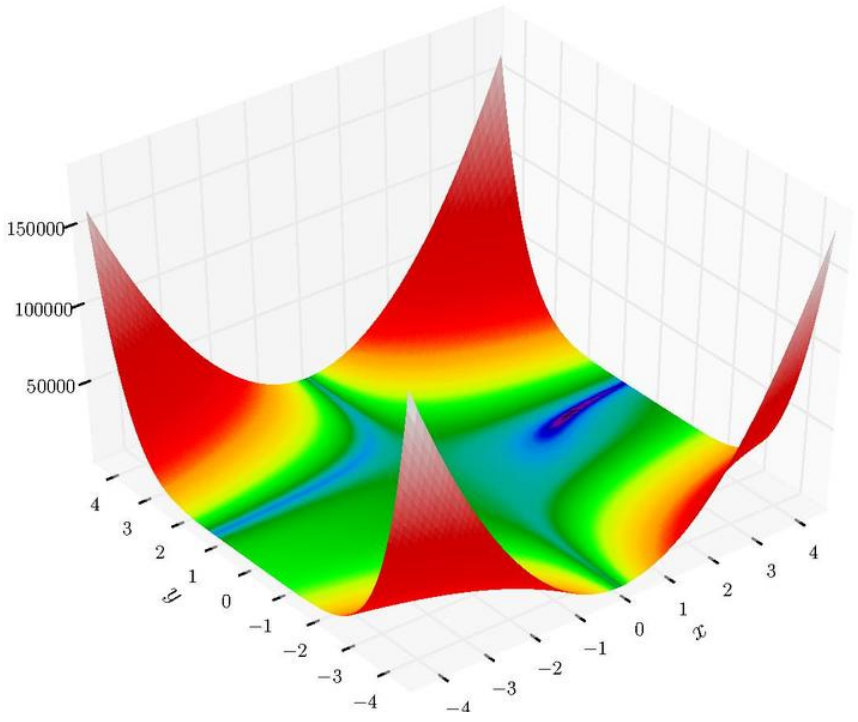
Obrázek 3.8: Histogram tisíce vzorků výsledků 4 optimalizačních algoritmů nad Rastriginovou funkcí 3.12 dvou proměnných. Optimum této funkce je hodnota 0 v bodě $[0, 0]$. Na ose x jsou znázorněny rozmezí funkčních hodnot jednotlivých prvků histogramu. Osa y znázorňuje četnost nalezení výsledku v daném rozmezí. Jak je vidět z grafu, nejlépe si počínala metoda evoluční strategie.



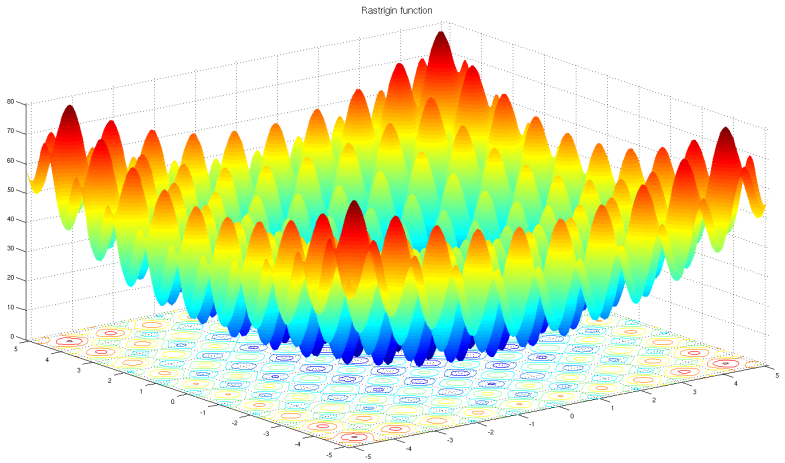
Obrázek 3.9: Histogram tisíce vzorků výsledků 4 optimalizačních algoritmů nad Rosenbrockovou funkcí 3.13 dvou proměnných. Optimum této funkce je hodnota 0 v bodě $x = 1, y = 1$. Na ose x jsou znázorněny rozmezí funkčních hodnot jednotlivých prvků histogramu. Osa y znázorňuje četnost nalezení výsledku v daném rozmezí. Jak je vidět z grafu, nejlépe si počínala Nelder Mead metoda.



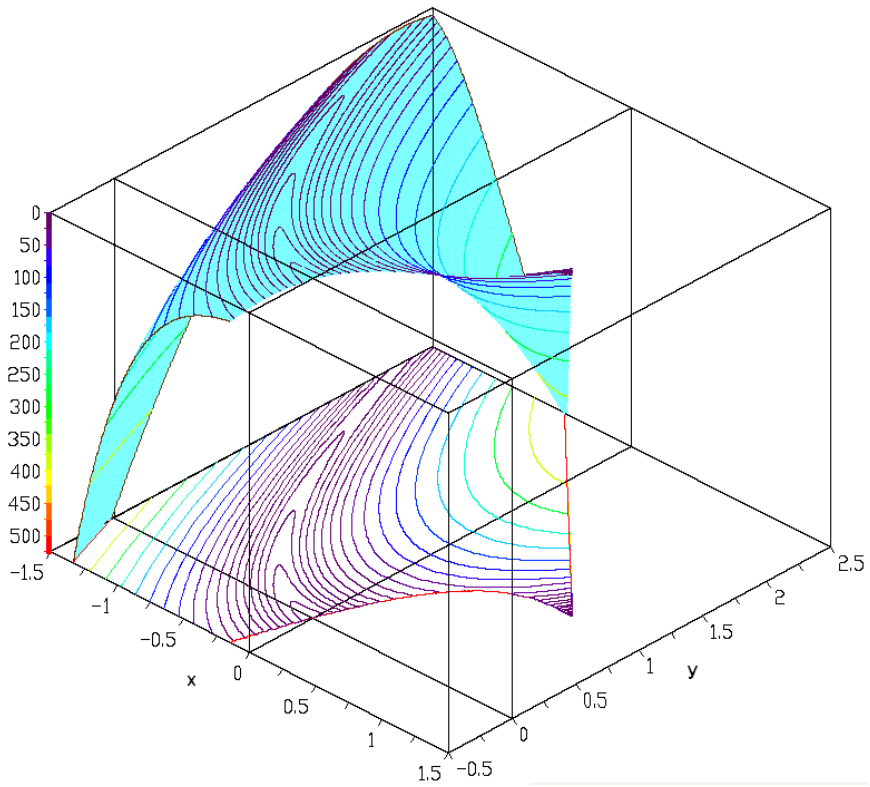
Obrázek 3.10: Histogram tisíce vzorků výsledků 4 optimalizačních algoritmů nad Bealeho funkcí 3.11 dvou proměnných. Optimum této funkce je hodnota 0 v bodě $[3, 0.5]$. Na ose x jsou znázorněny rozmezí funkčních hodnot jednotlivých prvků histogramu. Osa y znázorňuje četnost nalezení výsledku v daném rozmezí. Jak je vidět z grafu, nejlépe si počínala metoda simulovaného žíhání.



Obrázek 3.11: Graf Bealeho funkce, omezené na intervalu $[-4.5, 4.5]$ pro x i y. Jak je vidět funkce má pouze jedno globální minimum, specifická je ovšem v tom, že obsahuje dlouhé "rovné"plochy, které algoritmy hůře překonávají.



Obrázek 3.12: Graf Rastriginova funkce dvou proměnných, omezená na intervalu $[-5.12, 5.12]$ pro x i y. Funkce obsahuje mnoho lokálních a jedno globální minimum.



Obrázek 3.13: Graf omezené Rosenbrockovy funkce. Parametry jsou omezeny na $[-1.5, 1.5]$ pro x a $[-0.5, 1.5]$ pro y. Dále platí omezení $(x-1)^3 - y + 1 < 0$ a $x + y - 2 < 0$.

Optimalizovaný model	Optimalizační metoda	x	y	Výsledek
Rastrigin	Simulované žíhání	4.3663e-04	1.0812e-03	2.6969e-04
	Nelder Mead	-4.1754e-06	4.8035e-06	8.0364e-09
	Evoluční strategie	3.9238e-24	1.7666e-19	0
	Sdružené gradienty	4.5318e-07	3.0384e-06	1.8723e-09
Beale	Simulované žíhání	2.9973	0.4992	1.3326e-06
	Nelder Mead	2.9998	0.4999	5.7976e-09
	Evoluční strategie	2.9870	0.4964	2.9203e-05
	Sdružené gradienty	2.9977	0.4993	8.8853e-07
Rosenbrock	Simulované žíhání	0.9991	1.0001	3.6744e-04
	Nelder Mead	1.0000	1.0000	2.6071e-09
	Evoluční strategie	0.9972	0.9946	1.2344e-05
	Sdružené gradienty	0.9997	0.9994	7.9914e-08

Tabulka 3.2: Tabulka s nejlepšími nalezenými výsledky všech metody pro každý model. Každý uvedený výsledek je nejlepším z tisíce běhů a mělo by se na něj nahlížet spolu s grafy 3.8, 3.10, 3.9, 3.5, 3.7 a 3.6.

Kapitola 4

Závěr

Cílem práce bylo rozšíření knihovny SIMLIB/C++ o optimalizační modul. V knihovně byly doposud implementovány dvě optimalizační metody, jednoduchá podoba simulovaného žíhání a Hooke-Jeeves pattern search metoda. Dále byly připraveny třídy pro práci s řídícími parametry optimalizovaných modelů. Navržené rozšíření vylepšilo stávající metodu simulovaného žíhání, přidalo další metody — evoluční strategii, Nelder Mead metodu simplexů, metodu sdružených gradientů a optimalizaci mravenčí kolonií pro pohyb v grafech.

Navržené rozšíření využilo existujících tříd pro práci s modely a rozšířilo tuto myšlenku s použitím návrhového vzoru Bridge a vlastního rozhraní reprezentující optimalizovanou funkci nebo model. Použití návrhového vzoru Bridge dovoluje jednoduchou rozšiřitelnost o nové uživatelské metody, a jeho podobnost s návrhovým vzorem Adaptér umožňuje jednoduchou integraci externích optimalizačních knihoven do zmíněného rozhraní. To vše bylo následně implementováno v jazyce C++.

Přesnost a efektivita implementovaných metod byla otestována na třech různě náročných příkladech (Rosenbrockova funkce, Baeleho funkce, Rastriginov funkce) a jednom vlastním jednoduchém grafovém modelu. Z výsledků provedených testů lze implementované metody z hlediska funkčnosti označit za uspokojující. Různorodostí navržených metod bylo prokázáno, že volbou správné metody je uživatel schopen nalézt optima i v obtížných optimalizačních problémech. Metaheuristické metody jsou zároveň schopny optimalizovat i omezené problémy využitím bariérového přístupu.

Z hlediska budoucího vývoje by bylo vhodné přidat další metody, především některou z pattern search metod a aproximačních metod, jejichž vlastnosti tato práce nezmínila. Dále upravit metodu sdružených gradientů a nelder mead metodu k použití některé z technik řízeného restartování (pravděpodobnostní restart atd.). Vzhledem ke skutečnosti, že se jedná o stále se rozvíjející obor je vhodné sledovat i nově navržené postupy optimalizace a aktualizovat ty stávající právě o tyto poznatky.

Literatura

[1] Alhawarat, A.; Salleh, Z.: Modification of Nonlinear Conjugate Gradient Method with Weak Wolfe-Powell Line Search. *Abstract and Applied Analysis*, ročník 2017, č. 7238134, 3 2017.

[2] Anonymous: *A Comparison of Cooling Schedules for Simulated Annealing (Artificial Intelligence)*. What When How, [Online; navštíveno 15.03.2017].
URL `http://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/`

[3] Bonabeau, E.; Dorigo, M.; Theraulaz, G.: *Swarm Intelligence: From Natural to Artificial Systems*. Proceedings volume in the Santa Fe Institute studies in the sciences of complexity, OUP USA, 1999, ISBN 9780195131581.

[4] Dorigo, M.; Maniezzo, V.; Colorni, A.: Ant System: Optimization by a Colony of Cooperating Agents. *Trans. Sys. Man Cyber. Part B*, ročník 26, č. 1, Únor 1996: s. 29–41, ISSN 1083-4419, doi:10.1109/3477.484436.
URL `http://dx.doi.org/10.1109/3477.484436`

[5] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software (Adobe Reader)*. Pearson Education, 1994, ISBN 9780321700698.

[6] Grešovnik, I.; Owen, D.; Rodič, T.: *A general purpose computational shell for solving inverse and optimisation problems: applications to metalforming processes*. Department of Civil Engineering, University College of Swansea, 2000.

[7] Jaroš, J.; aj.: *Simulované žíhání*. FIT VUT v Brně, [Online; navštíveno 10.03.2017].
URL `http://www.fit.vutbr.cz/~jarosjir/groups/eva/sa.html.cz`

[8] Kirkpatrick, S.; Gelatt, C. D.; Vecchi, M. P.: Optimization by Simulated Annealing. *Science*, ročník 220, č. 4598, 5 1983: s. 671–680.

[9] Maňas, M.: *Optimalizační metody*. SNTL, 1979.

[10] Mitchell, M.: *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998, ISBN 0262631857.

[11] Nocedal, J.; Wright, S.: *Numerical Optimization*. Springer-Verlag New York, Inc., první vydání, 1999, ISBN 0-387-98793-2.

[12] Pearl, J.: *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. The Addison-Wesley Series in Artificial Intelligence, Addison-Wesley, 1984, ISBN 9780201055948.

[13] Rardin, R. R.: *Optimization in Operations Research*. Pearson, druhé vydání, 2017, ISBN 978-0134384559.

[14] Shewchuk, J. R.: An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technická zpráva, Pittsburgh, PA, USA, 1994.

[15] Singer, S.; Nelder, J.: *Nelder-Mead algorithm*. Scholarpedia, [Online; navštíveno 4.04.2017].
URL `http://www.scholarpedia.org/article/Nelder-Mead_algorithm`

[16] Surjanovic, S.; Bingham, D.: *Virtual Library of Simulation Experiments : Test Functions and Datasets*. Simon Fraser University, [Online; navštíveno 18.04.2017].
URL `https://www.sfu.ca/~ssurjano/optimization.html`