FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

# KIV/PC Semestrální práce

(Optimalizace funkce genetickým algoritmem)

Pavel Třeštík
(A17B0380P)

8. ledna 2019

# Obsah

# 1 Zadání

## 1.1 Zjednodušené zadání

Cílem je napsat přenositelnou konzolovou aplikaci, která bude hledat extrém funkce pomocí genetického alrgoritmu.
Základní princip genetickoho algoritmu:

1. Náhodně vytvoříme několik různých řešení zadané úlohy – jedinců.
2. Provedeme křížení jedinců.
3. Náhodně zmutujeme malý počet jedinců.
4. Ověříme, jak dobře každý jedinec řeší náš problém tak, že pro každého získáme hodnotu fitness funkce.
5. Vytvoříme novou generaci.
6. Opakujeme kroky 2 až 5.

## 1.2 Parametry programu

Aplikace bude přijímat z příkazové řádky 2 povinné argumenty:

- jméno souboru, ve kterém jsou metadata o testované funkci,
- počet generací, které mají být provedeny.

Program bude také přijímat jeden nepovinný argument:

- **-m** *procento mutací (0-100).*

Pokud nebudou na příkazové řádce uvedeny alespoň dva argumenty, vypište chybové hlášení a stručný návod k použití programu (v angličtině).

## 1.3 Popis programu

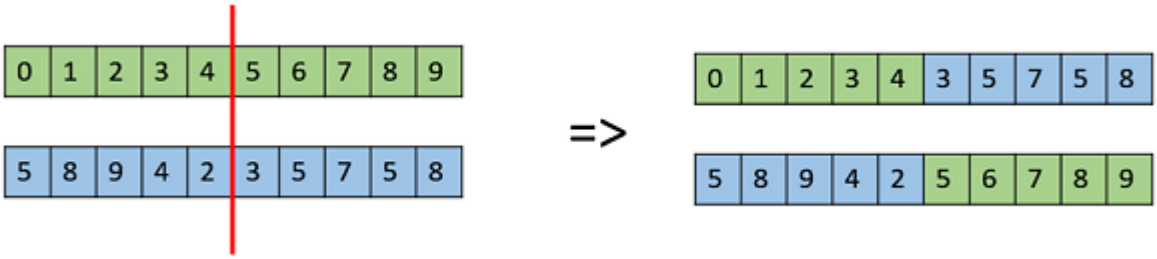
### 1.3.1 Mapování genu

Pro mapování genu se doporučuje použít dále uvedený způsob:

- Pro celá čísla použijte jejich binární zápis.
- Pro reálná čísla použijte konkrétní hodnoty daných proměnných.

**Křížení** Křížení dvou potomků se realizuje pro každou skupinu genů takto:

- Pro binární zápis celého čísla kombinujte části genů rodičů (viz Obrázek 1). Hranici bloků vyrábějte vždy náhodně, ale tak, aby sapdla mezi bity kódující interval uvedený u proměnné.
- Pro hodnotu reálného čísla vypočítejte pro potomka novou hodnotu jako aritmetický průměr hodnot rodičů.




Obrázek 1: Ukázka tvorby nových potomků zeleného a modrého rodiče
Zdroj: www.tutorialspoint.com

**Mutace** Mutaci genotypu provádějte následujícím způsobem. Podle parametru z příkazové řádky **-m** vyberte určitou část z populace, kterým náhodně upravíte část genů. Část binární reprezentace celého čísla náhodně přepíšete nulami a jedničkami. U reálného čísla, reprezentovaného hodnoutou vygenerujete náhodně nové číslo z uvedeného intervalu. U mutovaného potomka však vždy upravte jen jednu skupinu genů.

Dejte však pozor ať mutací neponičíte své nejlepší jedince.

### 1.4 Výstup

Výsledkem programu budou dva textové soubory (gen.txt a val.txt), ve kterých bude souhrn průběhu hledání optima. Protože celý proces může být velice časově náročný, tyto soubrory tvořte průběžně.

### 1.5 Originální zadání

https://www.kiv.zcu.cz/studies/predmety/pc/doc/work/sw2018-01.pdf

## 2 Analýza úlohy

V první řadě program bude muset načíst intervaly ze souboru s metadaty. Program bude číst soubor po řádkách a pokud načte řádku, která začíná na "#_(", bude vědět, že se jedná o řádku, která udává interval. Z této řádky následně získá potřebné údaje intervalu, tj. jeho spodní a horní hranici a typ.

V další části už program začne pracovat na geneticém algoritmu. Jelikož už má intervaly, může vytvořit první generaci, jejíž jedinci mají kompletně náhodné koeficienty z daných intervalů.

Aby program mohl vytvořit další generace, bude potřebovat zjistit fitness hodnotu každého jedince. Protože se jedná o hledání extrému funkce, tak hodnotu fitness funkce bude udávat extrém funkce, daného jedince. K počítání extrému je nám poskytnut program, který extrém vypočte a proto se fitness bude počítat úpravou souboru s metadaty a voláním poskytnutého programu.

Před vytvářením další generace, program seřadí generaci podle fitness hodnoty.

V této fázi program může vytvářet další generaci. Další generace se vytvoří tak, že jedinci z lepší části generace se kříží mezi sebou a vytváří tím nové jedince. Jedinci se kříží způsobem uvedeným v zadání. To je:

- z reálných koeficientů se udělá aritmetický průměr,
- z celých koeficientů se prohodí část jejich binárního zápisu.

U prvního případu není problém, protože aritemetický průměr dvou čísel z intervalu bude také ležet v tomto intervalu. Problém nastává u celých čísel, kdy při prohození pouze určitého počtu bitů může nastat situace, kde nový koeficient bude mimo interval. Jelikož ale neznám žádný algoritmus, který by zajišťoval, že po prohození určitého bloku binárního zápisu čísel, výsledek bude také v intervalu původních čísel, tak se pravděpodobně budu muset spolehnout na brute-force přístup. To znamená, že algoritmus bude muset testovat prohození každé možné délky bloku čísla a také by měl vzít v potaz, aby vůbec nastala nějáká změna v hodnotách (aby neprohodil pouze blok o takové velikosti, kde jsou bity shodné, tudíž by nenastala žádná změna v koeficientu).

Ačkoliv reálné koeficienty při křížení davají pouze jeden výsledek, prohazování bloků celých čísel dá výsledky dva, čímž při křížení dvou jedinců získáme dva nové jedince. Z tohoto důvodu se horší část generace bude přepisovat novými jedinci, získanými křížením lepší poloviny generace.

V předposledním kroku se provede mutace generace. Mutace jedince se provádí pouze na jedné skupině genů. Na základě typu genu, který se mutuje, se mutace provádí, jak je popsáno v zadání. Tedy reálnému koeficientu se vygeneruje nové náhodné číslo z daného intervalu a celému koeficientu se prohodí náhodný počet bitů jedničkami za nuly a opačně. Podobně jako u křížení ale u mutace celého koeficientu vzniká problém, zda nové číslo bude patřit do náležitého intervalu. Narozdíl od křížení se zde neprohazují pouze jednotné bloky čísla, ale jednotlivé bity nezávisle na pozici. Proto algortimus, který bude zajišťovat, aby nové číslo patřilo do intervalu, nemusí být vyloženě brute-force. Pokud nové číslo přeshuje horní hranici intervalu, tak vzhledem k tomu, že můžeme změnit

libovolný bit, jednoduše zmutovanému číslu změníme nejbližší bit s výšší hodnotou, než je rozdíl nového čísla a horní hranice intervalu. Podobně bude fungovat i dolní hranice intervalu, ale místo aby se nejbližší vyšší bit přepisoval na 0, tak se bude nejbližší bit s vyšší hodnotou, než rozdíl dolní hranice intervalu a nového čísla, přepisovat na 1.

V poslední řadě zbývá akorát udělat výstup do souborů. Do gen.txt se po každé nové generaci zapíše její pořadí a její nejlepší jedinec, resp. jeho extrém (fitness) a koeficienty. Do val.txt se po každé generaci zapíší hodnoty extrémů všech jedinců generace a jejich koeficienty.

# 3 Popis implementace

## 3.1 Obecný popis

Program se skládá ze dvou souborů s příponou .h (specimen.h a generation.h) a tří souborů s příponou .c (specimen.c, generation.c a main.c).

## 3.2 Podrobné popisy

### 3.2.1 specimen.h

**specimen.h** obsahuje dvě globální proměnné, *calc_func, která uchovává řetězec na spuštění programu pro spočítání extrému a *meta_nace, která uchovává pouze název souboru s metadaty (bez cesty).

Dále obsahuje tři struktury (viz Obrázek 2).

Struktura **coef** ukládá všechny koeficienty jedince.

Struktura **specimen** (struktura jedince) obsahuje jedincovo id, ext (extrém = hodnota fitness) a *coef - pointer na pole koeficientů jedince.

Struktura **interval** s proměnými: typ - typ intervalu, bot - spodní hranice intervalu, top - horní hranice intervalu.

```
typedef struct{
  char type;
  double num;
} coef;

typedef struct{
  int id;
  double ext;
  coef *coefs;
} specimen;

typedef struct{
  char type;
  float bot;
  float top;
} interval;
```

Obrázek 2: Struktury z hlavičkového souboru specimen.h

V poslední řadě specimen.h obsahuje prototypy (viz Obrázek 3). Prototyp create_specimen slouží k vytvoření jednice. Get_extreme používaný k získání extrému jedince - neboli jeho fitness hodnoty. Crossbreed kříží dva jedince. Mutate mutuje předaného jedince.

```
void create_specimen(specimen **spec, coef coefs[], int coef_cnt);
double get_extreme(specimen *spec/*, char orig_source[]*/);
void crossbreed(specimen *par1, specimen *par2, specimen **spec1,
   specimen **spec2, int coef_cnt, interval is[]);
void print_spec(specimen *spec);
void mutate(specimen **spec, interval is[], int coef_cnt);
```

Obrázek 3: Prototypy z hlavičkového souboru specimen.h

### 3.2.2 specimen.c

Souboru specimen.h náleží **specimen.c**, jehož funkce následují.

Funkce **create_specimen**, vzhledem k tomu, že důležité věci pro jedince jsou předány parametry, tak funkce pouze alokuje na tyto struktury paměť.

Funkce **get_extreme** zjišťuje extrém předaného jedince. Funkce nepočítá nic sama o sobě, pouze upravuje zdrojový soubor a volá poskytnutou funkci, která extrém spočítá.

Funkce **crossbreed** zkříží skupiny genů rodičů a vytvoří tak 2 nové jedince. Pokud je křížený gen reálné číslo funkce pouze udělá jeho aritmetický průměr a výsledek přiřadí oběma potomkům. Pokud se ale jedná o celé číslo, funkce ho nejdříve převede na binární zápis, protože ve stuktuře "coef"je celé číslo reprezentováno hodnoutou a né binárním zápisem, jak je doporučeno v zadání. Během tohoto převodu funkce navíc získá pozici nejvýznamnějšího bitu převáděného čísla. Následně zkouší prohazovat bloky o zvětšující se velikosti od nejvýznamnějšího získaného bitu do nejméně významného bitu. Bohužel, jak předpokládáno v analýze, funkce toto dělá pro každou velikost bloků, kdy nastane změna (tzn. bity na konečné pozici bloků se neshodují) tudíž se jedná a použití burte-force. Pokud není možné prohodit žádné bity, funkce pouze prohodí původní hodnoty celých čísel. Pokud je více než 1 možnost funkce náhodně jednu zvolí.

Funkce **mutate** zmutuje pouze jednu skupinu genů předaného jedince. Funkce tedy náhoně vybere jeden koeficient. Pokud je vybraný koeficient realné číslo, funkce vygeneruje nové náhodné číslo z náležitého intervalu. Pokud je koeficient celé číslo, funkce převede koeficient na binární zápis přičemž, stejně jako u crossbreed uloží pozici nejvýznamnějšího bitu koeficientu. Program následně vybere náhodný počet pozic, ovšem maximálně polovinu genu, které zneguje. Pokud zmutovaný koeficient přesahuje hranici intervalu, funkce zneguje bit s nejbližší vyšší hodnout než je rozdíl horní hranice intervalu a výsledku. Pokud je číslo menší než spodní hranice intervalu, tak program neguje bity od nejméně významné bitu, dokud se číslo v intervalu nenachází.

### 3.2.3 generation.h

Soubor **genegeneration.h**. Obsahuje jednu globální proměnnou - mutation_rate - procento mutací.

Tento soubor má jednu strukturu a tou je generation (viz. Obrázek 4). Generation obsahuje gen_num udávající o kolikátou generaci se jedná a species - pole jedinců v generaci.

```
typedef struct{
  int gen_num;
  specimen species[GENERATION_SIZE];
} generation;
```

Obrázek 4: Struktura z hlavičkového souboru generation.h

Prototypy generation.h (viz Obrázek 5).

```
generation* create_first_generation(interval is[], int inter_cnt);
void next_generation();
void calc_generation_extremes();
void print_generation();
void free_gen();
int print_gen_to_file();
int print_val_to_file();
int print_files();
```

Obrázek 5: Prototypy z hlavičkového souboru generation.h

### 3.2.4 generation.c

Souboru generation.h náleží **generation.c** s následujícími funkcemi.

Funkce **create_first_generation** vytvoří novou generaci s kompletně náhodnými koeficienty jedinců. Nejdříve alokuje paměť pro koeficenty všech jedinců. Následně generuje náhodné hodnoty koeficientů pro každého jedince. Spočítá extrémy všech jedinců a seřadí je.

Funkce **next_generation** vytváří další generaci. Nejdříve vygeneruje dvě pole uchovávající indexy lepší poloviny generace, které budou tvořit páry, které se budou mezi sebou křížit. Následně se provádí křížení těchto párů. Po křížení se generace mutuje. Na rozdíl od zadání je ale možné mutovat pouze 5 až 45 procent generace. To je z důvodu, aby v každé generaci nastala alespoň nějáká změna (proto minimálně 5 procent mutace pro každou generaci) a 45 procent je maximum proto, aby mutace nepoškodily 5% nejlepší jedinců a ani nové jedince vytvořené křížením, kteří tvoří polovinu generace. Nakonec funkce spočte extrémy a seřadí generaci.

Funkce **calc_generation_extremes** pouze volá funkci get_extreme pro každého jedince generace.

Funkce **print_generation** vypisuje do konzole nejlepšího a nejhoršího jedince proběhlé generace.

Funkce **free_gen** uvolňuje pamět coeficientů generace.

Funkce **print_gen_to_file** vypíše do souboru gen.txt číslo generace a jejího nejlepšího jedince.

Funkce **print_val_to_file** vypšíe do souboru val.txt extrémy všech jedinců a jejich koeficienty.

Funkce **print_files** volá print_gen_to_file a print_val_to_file, aby nebylo třeba je volat zvlášť (ovšem aby to bylo možné).

Mimo tyto funkce generation.c ještě obsahuje funkci **compare** sloužící k seřazení jedinců v generaci.

#### 3.2.5 main.c

Soubor **main.c** obstarává obsluhu programu. Obsahuje následující funkce.

Funkce **prepare_func_file** připravý soubor s metadaty a vytvoří jeho kopii v pracovním adresáři programu, s kterou poté bude pracovat.

Funkce **init_intervals** inicializuje intervaly ze souboru, způsobem popasným v analýze.

Funkce **help** vypíše do konzole nápovědu k použití programu.

Funkce **init** obstarává korektnost argumentů.

Funkce **run** provozuje celý program.

Funkce **shutdown** po doběhnutí programu uvolňuje paměť.

# 4 Uživatelská příručka

## 4.1 Přeložení

Program se přeloží zavoláním příkazu **make** na linuxu. Na Windows uživatel bude potřebovat prostředí umožňující volat make. K tomuto uživatel muže využít například Visual Studio nebo MinGW apod. (pro každé prostředí make funguje jinak!! Pro použití prostředí si přečtěte jeho dokumentaci!). **Předpokládané použití** pokud uživatel používá Visual Studio, musí přejmenovat poskytnutý makefile.win na pouze makefile a z příkazové řádky Visual Studia zavolat příkaz **nmake**. Pro MinGW uživatel musí stejným způsobem přejmenovat makefile.win a z příkazové řádky na pozici pracovního adresáře zavolat "disk:\cesta\k\mingw32-make.exe"

## 4.2 Spuštění a obsluha

Po přeložení se vytvoří spustitelný soubor **gms** (resp. **gms.exe** pro Windows), který lze spustit způsobem **./gms** [**meta_source_file**] [**number_of_generations**] **-m** [**percent_of_mutations**]. První dva parametry jsou povinné. "-m"je nepovinný parametr, udávající jaké procento generace se zmutuje, při vytváření nové generace. Pokud je použit přepínač "-m"musí povinně být zadán čtvrtý parametr - procento generace, jaké bude zmutováno, v rozmezí 5 až 45. Pokud nejsou zadány alespoň 2 parametry nebo je použit nesprávný přepínač a nebo pokud je použit správný přepínač, ale nezadá se procento mutace (popřípadě je zadáno chybné procento mutace), program vypíše chybovou hlášku a nápovědu k použití (viz Obrázek 6). Uživatel také může zobrazit nápovědu z Obrázku 6 bez chybové hlášky použitím jediného parametru **-h** (příklad volání: **./gms -h**).

```
[redacted]./gms res/func01_meta.txt 50 -m xxx
Invalid mutation percent!
Usage:
   gms [meta_file] [generation_count] -optional
      meta_file - file with function data
      generation_count - number of generations to be done
   Optional parameter: -m [number]
      number <5, 45> - percent of generation to be mutated
If option -m isn't used, default mutation percent is 5
```

Obrázek 6: Příklad nápovědy a chybového hlášení při spuštění programu, kdy je zadáno chybné procento mutace

Příklad správného spuštění programu a průběžného výstupu generace do konzole - Obrázek 7.

```
./gms res/func01_meta.txt 50 -m 10

GENERATION NUMBER: 1
TOP SPEC
Specimen 8
Extreme: 12.094950
---------
WORST SPEC
Specimen 36
Extreme: -7.918574
**********
```

Obrázek 7: Příklad správného spuštění a průběžného výpisu

### 4.3 Výstupy

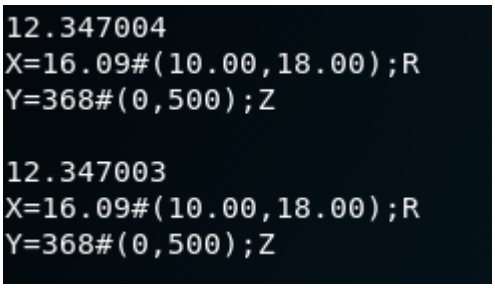
Výstupy programu jsou dva textové soubory **gen.txt** a **val.txt**, které se tvoří průběžně během běhu programu. Mimo tyto soubory program také vypisuje informace o provedené generaci do konzole (viz Obrázek 7).

Soubory mají formát popsaný v originálním zadání. Příklad záznamu ze souboru gen.txt - Obrázek 8 a ze souboru val.txt - Obrázek 9.

```
--- GENERATION 50 ---
12.347004
X=16.09#(10.00,18.00);R
Y=368#(0,500);Z

--- GENERATION 49 ---
12.347004
X=16.09#(10.00,18.00);R
Y=368#(0,500);Z
```

Obrázek 8: Příklad posledních dvou záznamů gen.txt získaných voláním příkladového příkazu z Obrázku 7




Obrázek 9: Příklad posledních dvou záznamů val.txt získaných voláním příkladového příkazu z Obrázku 7

# 5 Závěr

Zadání program víceméně splňuje, ačkoliv má pár změn. Jedna změna je v ukládání celých čísel, kdy zadání doporučuje ukládat čísla v binární podobě, ale můj program je má uložené hodnotou. Druhá změna je rozsah procent mutací, kdy v zadání je naznačeno že procenta můžou být v rozsahu 0-100, ovšem zadání také říká, abychom si dali pozor, aby mutace neponičila nejlepší jedince. Vzhledem k tomu, že program provádí mutaci až po křížení jedinců, což znamená, že půlka nové generace jsou kříženci předchozí, tak jsem zvolil rozsah procent mutací 5-45. Pět procent z důvodu, aby se v každé generaci nějáký jednotlivec změnil, protože pokud by bylo procento mutací 0, po určitém počtu generací by se jednotlivci přestali zlepšovat a všichni by byli stejní. Čtyřicetpět je horní hranice proto, že nová generace je z poloviny tvořena kříženci a -5% předpokládaných, aby se nemutovali nejlepší jedinci.

Na linuxu program probíhá rychle, tak rychle, že téměř není znát náročnost některých výpočtů a funkcí. Na Windows ovšem kompletně stejný program běží i více než deset krát pomaleji, což naznačuje, že některé části programu budou mít velkou výpočetní složitost nebo jsou z jiného důvodu velmi časově náročné na systému Windows.

Bylo by vhodnější prozkoumat, zda neexistují lepší algoritmy například na křížení celých čísel dvou jedinců, protože stávající algoritmus je ve své podstatě brute-force, tedy velmi výpočetně složitý. Také by chtělo vzít v potaz uložení některých proměnných (konkrétně celého čísla), protože pokud by bylo uloženo jako binární číslo, jak doporučuje zadání, ušetřilo by se značné množšství převodů celého čísla do binární podoby. V poslední řadě pravděpodobně poměrně náročný proces je zapisování dat do souborů, kdy proto aby soubory měly vyžadovaný formát, či aby se soubor změnil, jak je požadováno tak se musí přesouvat přes pomocný soubor. Tento proces je ale omezením jazyka a není k němu žádné výrazné vylepšení.