VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

POROVNÁNÍ VARIANT GENETICKÉHO PROGRAMOVÁNÍ V ÚLOZE SYMBOLICKÉ REGRESE

COMPARISON OF GENETIC PROGRAMMING VARIANTS IN THE SYMBOLIC REGRESSION TASK

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **PETR DOLEŽAL**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MICHAELA DRAHOŠOVÁ, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Doležal Petr**
Program: Informační technologie
Název: **Porovnání variant genetického programování v úloze symbolické regrese**
Comparison of Genetic Programming Variants in the Symbolic Regression Task
Kategorie: Umělá inteligence

Zadání:

1. Seznamte se s principy evolučních algoritmů, genetického programování a symbolické regrese.
2. Navrhněte program umožňující provádět symbolickou regresi pomocí různých variant genetického programování.
3. Program z bodu 2 implementujte.
4. Na zadaných úlohách ověřte funkčnost programu a statisticky porovnejte jednotlivé varianty genetického programování.
5. Zhodnoťte dosažené výsledky a diskutujte přínos práce.

Literatura:

- Vanneschi, L. a Poli, R. Genetic Programming - Introduction, Applications, Theory and Open Issues. In: *Handbook of Natural Computing*. Springer, 2012, s. 709-739.
- Miller, J. F. *Cartesian Genetic Programming*. Springer-Verlag, 2011. Natural Computing Series.
- Schmidt, M., a Lipson, H. Distilling free-form natural laws from experimental data. *Science* 324.5923 (2009): s. 81-85.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Drahošová Michaela, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 29. října 2021

Abstrakt

Tato práce se zabývá porovnáváním různých variant genetického programování v úloze symbolické regrese. Na zadaných úlohách zkoumá rychlost konvergence a kvalitu nalezeného řešení. Klade si za cíl porovnat kartézské genetické programování, stromové genetické programování a jejich modifikace pomocí koevoluce. Byla použita vlastní implementace (bez využití knihoven), kde jednotlivé varianty spolu sdílí převážnou část kódu. Součástí práce je i ověření použitelnosti implementovaných přístupů při analýze reálných dat. Na základě experimentů bylo zjištěno, že všechny zkoumané přístupy jsou použitelné pro provádění symbolické regrese. Nejlepších výsledků ve zkoumaných oblastech (rychlost konvergence, kvalita nalezeného řešení) dosahovalo kartézské genetické programování s koevolucí.

Abstract

This thesis deals with comparison of genetic programming variants it the task of symbolic regression. Time to converge and quality of evolved solutions are evaluated on nine chosen benchmarks. In particular, tree-based genetic programming, cartesian genetic programming and their modifications using coevolutionary algorithm are investigated. An own implementation of employed methods (without a specific library use) allows to share as much code as possible. Moreover, an analysis of implemented methods efficiency on real world data is provided. Experimental results show that all of the investigated approaches are capable of finding solutions using symbolic regression. Cartesian genetic programming enhanced with coevolution seems to be the most suitable of the investigated approaches in terms of evolved solution quality and time to converge.

Klíčová slova

Symbolická regrese, genetické programování, kartézské genetické programování, koevoluce

Keywords

Symbolic regression, genetic programming, cartesian genetic programming, coevolution.

Citace

DOLEŽAL, Petr. *Porovnání variant genetického programování v úloze symbolické regrese.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michaela Drahošová, Ph.D.

Porovnání variant genetického programování v úloze symbolické regrese

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením paní doktorky Michaely Drahošové. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Petr Doležal
30. dubna 2022

Poděkování

Děkuji vedoucí Ing. Michaele Drahošové, Ph.D. za velmi aktivní a obětavý přístup při vedení mé práce.

Obsah

Kapitola 1

Úvod

Již v minulosti se vědci snažili přijít na různé přírodní zákonitosti a chtěli je co nejpřesněji definovat. K tomu využívali regresi, tedy metodu využívanou k získání matematického popisu nějakého jevu z experimentálně naměřených dat. [9]

Ta se dříve řešila analytickým způsobem. Například pomocí metody nejmenších čtverců se dala poměrně snadno aproximovat lineární funkce. Se stoupající komplikovaností požadované funkce, jejíž předpis hledáme, se však takováto regrese stává velmi obtížná (ne–li nereálná). Mnohdy nevíme, jakou regresní metodu zvolit. Stává se, že konkrétní problém nemusí být známými metodami pro analytickou regresi vůbec řešitelný.

Poslední dobou se stále více využívá *Natural computing*. Jedná se o algoritmy inspirované v přírodě [6]. Tyto algoritmy spadají pod metody umělé inteligence. Například viry mutují, aby se přizpůsobili prostředí a maximalizovaly tak svoji šanci na přežití. Slabí jedinci umírají, ti silnější přežijí. Tento princip můžeme využít i při již zmíněné regresi, kde můžeme „mutovat“ rovnice, odstraňovat méně vhodné a naopak nechat přežít ty, které více odpovídají našemu hledanému řešení.

Tento způsob můžeme aplikovat i v případě, že nemáme vůbec ponětí, co znamenají naměřená data, nerozumíme analytické regresi a někdy i dokonce nalezenému řešení. Oproti regresní analýze je symbolická regrese mnohem více obecná, avšak ne natolik, aby byla bez jakékoli modifikace vždy lepší nebo stejně dobrá jako jiné postupy. Pro jednoduchou rovnici pravděpodobně nebude vhodný algoritmus, který špičkově hledá řešení dlouhých a komplikovaných rovnicích a naopak. [11, 9]

Pokud bychom chtěli reprezentovat matematický vztah pomocí stromové struktury (například binárního stromu), záleží na maximální povolené hloubce. Pokud hledáme předpis rovnice, která obsahuje pár elementárních matematických operací, je vhodné, aby maximální hloubka byla malá. V opačném případě by řešení také mohlo být nalezeno, ale bylo by komplikovanější, některé uzly stromu by se prakticky vzájemně vyrušily a výpočet by trval dlouho. Kdybychom však stejný algoritmus se stejnou maximální hloubkou chtěli použít tentokrát na hledání řešení velmi komplikované rovnice, malá povolená hloubka by na vyjádření předpisu vůbec nemusela stačit. Na tomto příkladu jde vidět, jak moc důležitá je role parametrů symbolické regrese.

Cílem této práce je porovnání jednotlivých variant genetického programování, co se týče rychlosti konvergence k řešení požadované kvality a kvality nalezeného řešení po předem určené době běhu symbolické regrese.

Kapitoly 2 a 3 popisují teoretický úvod této práce. V kapitole 4 se nachází návrh programu schopného provádět symbolickou regresi různými způsoby a porovnávat tyto přístupy. Kapitola 5 obsahuje implementační detaily tohoto nástroje. V Kapitole 6 je porovnáváno

kartézské genetické programování, genetické programování a jejich modifikace koevolucí z hlediska rychlosti konvergence k řešení požadované kvality a přesnosti nalezených řešení. V kapitole 7 se nachází diskuze ohledně vlivu volby řídicích parametrů na výše zmíněné (rychlost konvergence, kvalita nalezených řešení) a roli takových parametrů při porovnání zkoumaných přístupů symbolické regrese. Závěrečné shrnutí práce popisuje kapitola 8.

Kapitola 2

Evoluční algoritmy

Historie evolučních algoritmů sahá do 50. let minulého století, kdy počítače začaly být více dostupné pro vědeckou komunitu. V dnešní době jsou velmi rozšířené a jsou používány k řešení velmi různorodých problémů, což značí jejich velkou využitelnost v praxi. [4]

Pojem *evoluce* obecně značí proces inkrementální změny, například ve stylu „evoluce“ lékařských procedur [4]. Když zmíníme pojem *evoluční algoritmus*, myslíme tím něco více konkrétního, a sice proces, při kterém dochází k přežití jen těch nejlepších jedinců, kteří se poté reprodukují, a současně k zániku slabých jedinců, kteří nejsou schopni přežít v daném prostředí a šířit tak svoji genetickou informaci. Účelem jejich zavedení však nebylo napodobit co nejlépe skutečnou evoluci, ale vytvořit nástroj, co by byl pokud možno co nejlépe schopen řešit složité výpočetní problémy. Tyto algoritmy pracují následujícím způsobem:

- Pracují s populací jedinců, což je množina reprezentují kandidátních řešení daného problému
- Jedincům je pomocí *fitness funkce* přiřazována hodnota *fitness*, která určuje jejich kvalitu a podle které se rozhodne o jejich přežití/vybrání k reprodukci
- Jedinci jsou vybíráni jako rodiče, kteří společně vytvoří potomky. Ti jsou svým rodičům podobní, avšak nejsou totožní.
- Vybraní členové populace jsou odstraňováni

Pokud tento evoluční algoritmus poběží po dobu několika cyklů (generací), přirozený výběr na základě fitness funkce způsobí, že postupně dojde ke konvergenci jedinců k řešení požadované kvality. Obecné schéma takového algoritmu je:

Algoritmus 1: Obecný evoluční algoritmus

```
1: Vytvoř náhodnou populaci o velikost M
2: while Není splněna ukončovací podmínka do
3:     Zvol rodiče, kteří společně vytvoří potomky
4:     Vyber jedince, kteří umřou
5: end
6: return Jedinec s nejvyšší fitness
```

Tento obecný algoritmus však na řešení problémů nestačí. Musíme specifikovat:

- Jaká je reprezentace jedinců v populaci

- Jak veliká je populace
- Na základě jakého kritéria jsou voleni rodiče
- Jakým způsobem jsou vytvářeni potomci z předků
- Kolik potomků má být v další generaci
- Kritérium pro určení přeživších

K nalezení požadovaného výsledku můžeme výše uvedené zvolit vícero způsoby. [4] Představení některých používaných způsobů se nachází v této kapitole.

2.1 Interní reprezentace jedinců

Interní reprezentace jedinců může být implementována dvěma odlišnými způsoby inspirovanými biologii: pomocí *genotypu*, nebo *fenotypu*. Genotyp více odpovídá biologickému genetickému kódu, protože se jedná o reprezentaci pomocí řetězce (například řetězec jednociferných dekadických čísel). Příkladem může být třeba řetězec `0130493249839043`. Takový genotyp je pak určitým způsobem interpretován (například znak „0“ značí přičtení vstupu, znak „1“ odečtení jedničky atp. Oproti tomu fenotyp neobsahuje žádné zakódování. Například binární strom, jehož uzly obsahují nějaký prvek z množiny {+,-} a listy z množiny {x,1} je příkladem fenotypu. [4]

2.2 Velikost populace

Velikost populace (tedy počet jedinců) zůstává za běhu programu konstantní. Neprokázalo se, že by variabilní velikost populace měla zásadní pozitivní vliv na dobu výpočtu. Kromě velikosti počáteční populace M je třeba zvolit i počet potomků K, které tato populace vyprodukuje. Počet jedinců v další generaci by ale se započítáním nově vytvořených potomků přesáhl hodnotu M, proto musí uživatel specifikovat, jakým způsobem se tato populace zpětně redukuje na požadovanou hodnotu. [4]

2.3 Volba selekce

Jak již bylo zmíněno dříve, programátor musí rozhodnout, jakým způsobem budou voleni rodiče a také na základě kterého kritéria budou „umírat“ nevhodní jedinci. Volení nejlepších jedinců (jedinců s nejvyšší fitness) pro tvorbu potomstva nemusí být vždy nejvhodnější, protože poté může dojít k *degeneraci*, což může vyústit v uváznutí v lokálním maximu. Naopak, když budou nejlepší jedinci neúměrně ignorováni, hledání řešení může trvat zbytečně dlouho, nebo nemusí být nalezeno vůbec. [4]

Existuje spoustu variant selekčních mechanismů. Nejjednodušším je tzv. deterministická selekce, kdy je pro reprodukci zvoleno K jedinců, kteří mají nejvyšší fitness. Dalším typem může být například turnajová selekce, kdy je náhodně vybráno S jedinců. Jedinec s nejvyšší hodnotou fitness je vybrán pro reprodukci. Tento mechanismus lze opakovat několikrát, dokud není vybráno požadované množství jedinců. [11]

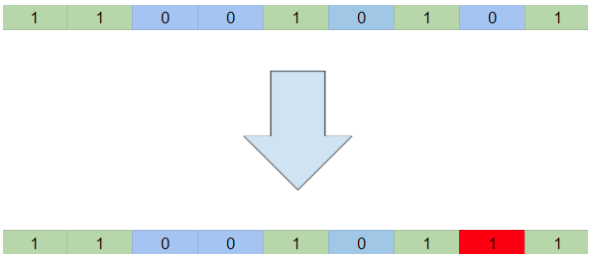
2.4 Tvorba potomků

Při tvorbě potomků je potřeba zohlednit již dosažené částečné řešení. Rozlišujeme dva odlišné typy reprodukce: sexuální a asexuální. Při asexuální je vytvořen klon předka, který je mírně pozměněn. Takováto operace se nazývá *mutace* (viz obrázek 2.1). Oproti tomu *křížení* spadá pod sexuální reprodukci, což značí kombinaci 2 nebo více rodičů do potomka, který nese znaky svých rodičů. Některé od prvního, jiné od druhého atd. Takto vzniklý potomek může sám být poté také ovlivněn mutací. V tradičních genetických algoritmech se využívají pro kombinaci 2 předci.

Z hlediska hledání řešení je mutace vhodná pro prohledávání lokálního prostoru, zatímco křížení má spíše globální charakter. [4]

2.5 Genetické algoritmy

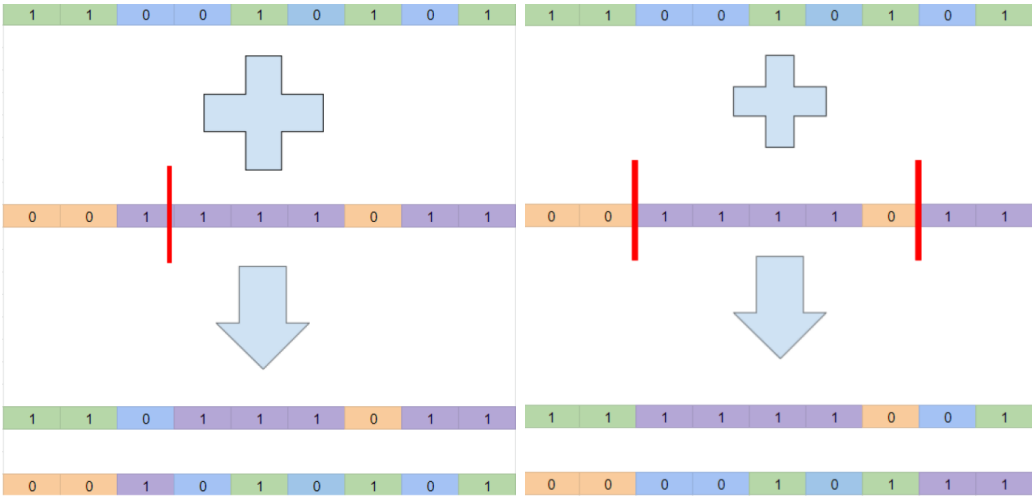
Genetické algoritmy jsou jednou z více forem evolučních algoritmů. Jsou specifické tím, že kladou velký důraz na selekci, křížení, mutaci a na vzájemné vazby mezi těmito operacemi. Na začátku bylo upřednostňováno křížení před mutací, což je strategie která stále převládá u některých forem genetických algoritmů, avšak v současné době je hojně používáno kombinování křížení a mutací. Využívají genotyp, který je dekódován a vyhodnocen. Původně se jednalo zpravidla o řetězce bitů [16], jak ukazuje obrázek 2.2.



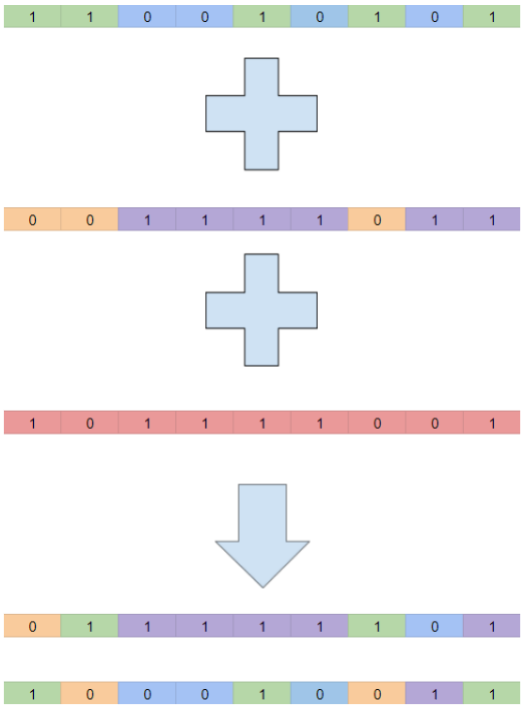
Obrázek 2.1: Operátor mutace

Obrázek 2.2: Ukázka genotypu

Existuje vícero druhů křížení, například tři základní: jednobodové, vícebodové a uniformní. U jednobodového se náhodně vygeneruje bod křížení, kde dojde k prohození genů. Vícebodové a uniformní křížení poskytuje větší prostor pro důkladnější promíchání genetických informací mezi rodiči. U vícebodového křížení dojde k vybrání minimálně 2 bodů, které určují, jakým způsobem bude probíhat předávání genetické informace. Pokud by došlo k rozdělení genotypu dvěma body na tři části, zdědil by potomek první část od prvního rodiče, druhou část genotypu od druhého rodiče a poslední úsek opět od prvního z rodičů. V případě uniformního křížení je pro každý gen potomka určeno, od koho se (tento gen) bude dědit (pomocí tzv. *masky*). Tyto formy kombinace znázorňují obrázky 2.4 a 2.3. [11]



Obrázek 2.3: Jednobodové křížení (vlevo) a vícebodové křížení (vpravo)



Obrázek 2.4: Uniformní křížení

2.6 Evoluční strategie

Evoluční strategie pracuje s vektorem reálných čísel jakožto genotypem. Základní varianta evoluční strategie využívá pouze operátoru *mutace*. Při mutaci se využívá Gaussovo rozložení s nulovou střední hodnotou a rozptylem σ. Hodnoty získané z tohoto rozložení jsou přičteny k hodnotám parametrů rodiče. Rovněž parametr σ je měněn za běhu, aby přibližně jedna pětina mutací vedla ke zlepšení potomka oproti svému předkovi. Vliv mutace je tedy zvyšován/snižován v závislosti na úspěšnosti potomků.

Nejjednodušší varianta evoluční strategie se označuje ES(1+1), což znamená, že z jednoho předka je pomocí mutace vytvořen jeho potomek. K nahrazení rodiče dojde v případě, že má jeho následovník vyšší hodnotu fitness. Pokud se tak nestane, je rodičem další generace úplně stejný jedinec, který vytvářel potomka v této generaci.

V současné době se pracuje s generacemi obsahujícími vícero jedinců. Nejčastější jsou strategie označované jako ES$(\mu + \lambda)$ a ES(μ, λ). Parametr μ značí počet rodičů a parametr λ počet potomků. Varianta ES$(\mu + \lambda)$ vybere nejlepších μ jedinců napříč populací, zatímco ES(μ, λ) vybírá pouze z množiny potomků.

Zjevným rozdílem oproti jiným evolučním algoritmům je, že dochází k pozměňování parametrů evoluce za běhu programu. V jedné chvíli dochází k evoluci jak jedinců, tak k úpravě řídicích parametrů. Dalším rozdílem je poměrně častá absence křížení, na kterém spousta jiných algoritmů stojí. [11]

2.7 Genetické programování

Reprezentace genotypu pomocí řetězce fixní délky (jak tomu bývá u genetických algoritmů) je velmi omezující podmínkou při hledání požadovaného řešení. Když hledáme řešení problémů z reálného světa, tak obvykle předem neznáme výslednou velikost genotypu, kterým by se dalo popsat. Tradiční genetické algoritmy neumožňují měnit interní reprezentaci za běhu, což je jejich nevýhoda. Tento problém byl vyřešen pomocí rozšíření genetických algoritmů, které se nazývá *Genetické programování* (dále „GP“).

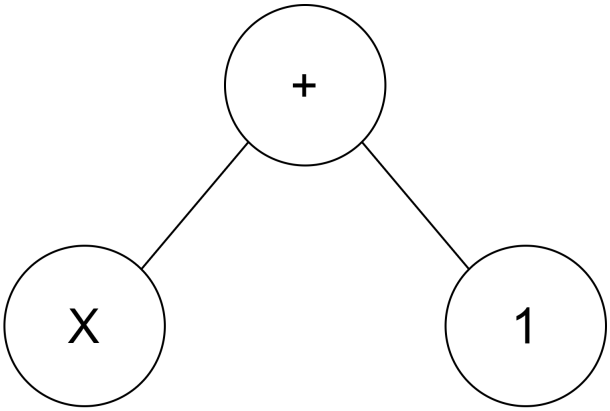
Namísto řetězce genů s předem danou maximální délkou GP využívá stromové struktury. Za běhu programu je možné měnit nejen velikost a tvar stromu, ale také obsah jednotlivých uzlů nebo listů. V současné době se často používá reprezentace takovéto stromové struktury pomocí pole, což vede k vyššímu výkonu. Algoritmus 2 popisuje fungování GP. [14]

Algoritmus 2: Genetické programování

```
1: Vytvoř náhodnou populaci jedinců
2: while Není splněna ukončovací podmínka do
3:     Přiřaď každému jedinci hodnotu fitness podle toho, jak dobře řeší problém
4:     Vyber rodiče na základě pravděpodobnostního modelu a hodnot fitness jedinců
5:     Přenes nějaké vybrané jedince bez jakýchkoli změn do nové generace
6:     Vytvoř potomky pomocí křížení dvou vybraných rodičů
7:     Vybrané jedince mírně náhodně pozměň
8: end
9: return Jedinec s nejvyšší fitness
```

2.7.1 Interní reprezentace

V jednotlivých uzlech se nachází vždy jedna funkce z množiny funkcí $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ a naopak v listech se nachází některý symbol z množiny terminálů $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$. Příkladem množiny funkcí může být například množina $\mathcal{F} = \{+, -, *, sin, exp\}$, u terminálů může být tvořena například následujícími proměnnými a konstantami: $\mathcal{T} = \{x, y, 1, 2, 3\}$. Výsledný strom může mít tedy například podobu binárního stromu o hloubce 1, kde v kořeni stromu se nachází operace `+`, v levém listu proměnná `x` a v pravém listu konstanta `1`. Tento strom tedy reprezentuje rovnici $y = x + 1$. Jeho znázornění je vidět na obrázku 2.5.



Obrázek 2.5: Jedinec v GP

V množině funkcí se nemusí nutně nacházet pouze matematické operace. Můžeme se setkat například s logickými funkcemi, jako například AND, OR, XOR ...

U některých funkcí je potřeba předem zjistit, jestli nepracují s nevalidními parametry. Typickým příkladem je dělení nulou. Problémy můžou nastat i v případě na první pohled bezpečných funkcí. Například u $y = e^x$ z matematického hlediska není problém, avšak u počítačů může velmi snadno dojít k přetečení.

Programátor se také musí zabývat otázkou, jestli ním specifikované terminály a funkce stačí na vyjádření přijatelného řešení. Pokud by jeho množinou funkcí byla množina $\{+, -\}$, nebyl by schopen kvalitně aproximovat složitější funkce než lineární. Velmi často však předem nevíme, jestli námi specifikované množiny budou stačit, či nikoli. Volba odpovídajících parametrů do značné míry závisí na znalosti daného problému. [14]

2.7.2 Inicializace počáteční populace

Existují 3 hojně využívané metody pro inicializaci jedinců: metoda *grow*, *full* a metoda *ramped half-and-half*. Všechny závisí na parametru d, který značí maximální hloubku stromové struktury. [5, 14] Jejich princip popisují algoritmy 3, 4 a 5.

Algoritmus 3: Metoda grow

```
1: Do kořene vlož náhodnou funkci z množiny funkcí
2: Do následujících uzlů vlož prvek z množiny F ∪ T
3: foreach Následující funkční uzel do
4:     if Hloubka is not d-1 then
5:         Rekurzivně volej algoritmus s přeskočením prvního kroku
6:     else
7:         Rekurzivně volej algoritmus, přeskoč první krok, vybírej jen z terminálů
8:     end
9: end
```

U takového algoritmu záleží, s jakou pravděpodobností bude do daného uzlu zvolen prvek z množiny terminálů a s jakou pravděpodobností bude z množiny funkcí. Při velkém důrazu na volbu terminálů bude stromová struktura dosahovat velmi malých hloubek, zatímco při preferování funkcí budou stromy velmi rozsáhlé.

Algoritmus 4: Metoda full

1: Do kořene vlož náhodnou funkci z množiny $\mathcal{F}$
2: Do následujících uzlů vlož prvek z množiny $\mathcal{F}$
3: **if** *Hloubka is not d-1* **then**
4: Rekurzivně volej algoritmus, přeskoč první krok
5: **else**
6: Rekurzivně volej algoritmus, přeskoč první krok, vybírej jen z terminálů
7: **end**

Oproti metodě *grow*, metoda *full* vybírá prvky pouze z množiny funkcí, dokud nedojde k dosažení maximální hloubky. [14]

Algoritmus 5: Metoda ramped half-and-half

1: $i = 0$
2: **while** *Populace není naplněná* **do**
3: Vytvoř 1/2d populace s maximální hloubkou i, metoda *grow*
4: Vytvoř 1/2d populace s maximální hloubkou i, metoda *full*
5: $i = i + 1$
6: **end**

Metody *grow* a *full* neposkytují dostatečnou diverzitu jedinců v populaci, což může ohrozit úspěšnost výpočtu. Z tohoto důvodu byl zaveden algoritmus *ramped half-and-half* (viz algoritmus 5), který vytváří dostatečně odlišné jedince. [5, 14]

2.7.3 Vyhodnocení fitness

Nejčastějším způsobem, jak vyhodnotit fitness daného jedince, je přes množinu trénovacích vektorů. Tyto vektory nesou informace o tom, jaký je očekávaný výstup při daném vstupu. Kdybychom například chtěli zjistit závislost, která by byla ve skutečnosti definována předpisem $f(x) = x^2 + 2$, měly by tyto vektory (v ideálním případě, kdy by nedocházelo k chybám měření apod.) formu například $\{(1, 3), (2, 6), (3, 11) \dots\}$. První vektor byl odvozen následujícím způsobem: $(1, 3)$ protože $f(1) = 3$ $(1^2 + 2 = 3)$. Obdobně $(2, 6)$, protože $f(2) = 6$ $(2^2 + 2 = 6)$. Stejným způsobem by se postupovalo i při vytváření dalších trénovacích vektorů. V rámci symbolické regrese by však mohl být zjištěn předpis i úplně jiné funkce, která by pro stejný vstup poskytovala totožný výstup (v rámci zadaných trénovacích vektorů).

Výsledná hodnota fitness daného jedince se spočítá například jako suma absolutních odchylek mezi očekávaným řešením a tím získaným pomocí jedince. Formálně by se tento výpočet dal specifikovat následujícím způsobem:

$$f_R(i) = \sum_{j=1}^{N} |S(i, j) - C(j)|$$

kde $S(i, j)$ je hodnota získaná z jedince i po předání vstupu j, $C(j)$ označuje očekávaný výstup. [14]

2.7.4 Selekce

Pro každého jedince v populaci platí, že jej potká jeden z následujících scénářů:

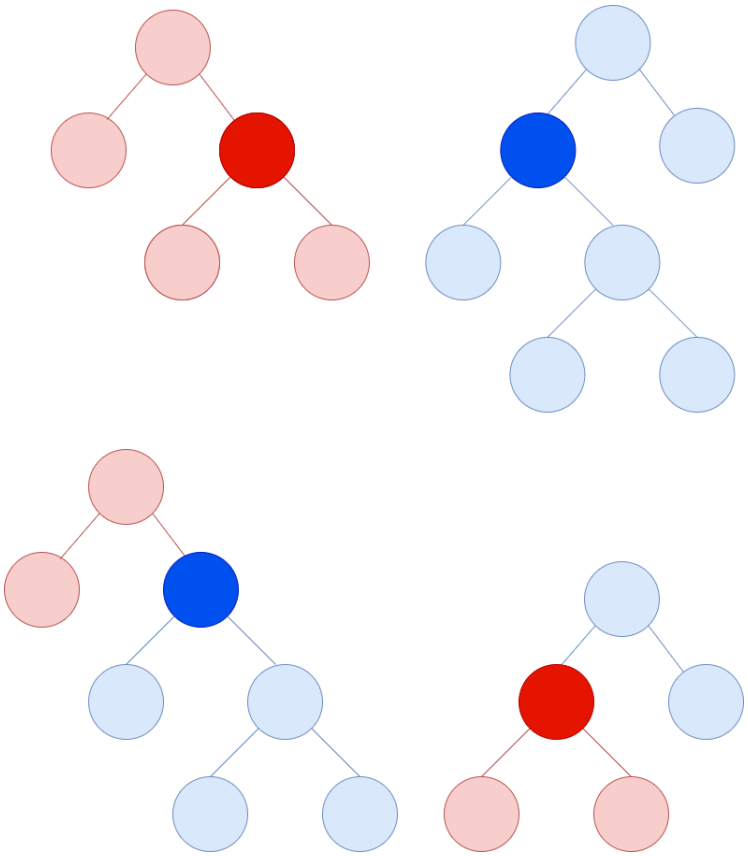
- bude převeden do následující generace beze změn
- bude upraven pomocí genetických operátorů
- bude odstraněn

Selekce, která určuje šanci na přežití daným jedincům, je jeden z nejdůležitějších mechanismů v oblasti GP. Stejné je to i u evolučních algoritmů obecně. Rychlost a úspěšnost evoluce závisí z veliké části právě na ní. [14] Některé druhy selekčních algoritmů již byly popsány v sekci 2.3, přičemž v případě GP je jejich použití obdobné.

2.7.5 Křížení

Při křížení nejprve dojde ke zvolení tzv. bodu křížení, angl. *crossover point*. Tento bod je nějaký uzel, případně list ve stromu. Kombinace proběhne mezi 2 jedinci, pro každý z nich je vygenerován daný bod. Běžně bývá pravděpodobnost zvolení uzlu (obsahující funkci) několikanásobně větší, než výběr listu (s terminálem).

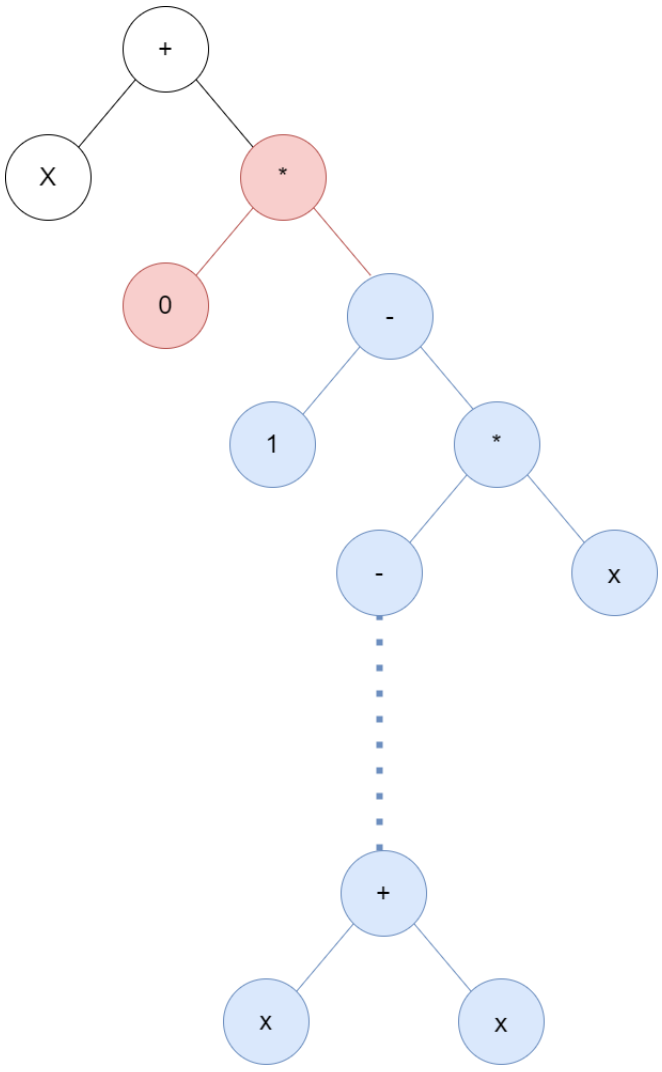
Samotná kombinace probíhá tak, že se prohodí jednotlivé podstromy obou rodičů. Většinou bývá používána tato strategie k symetrickému vytvoření dvou potomků. Obrázek 2.6 znázorňuje průběh křížení.



Obrázek 2.6: Křížení u GP

Při zvolení listu jako bodu křížení u jednoho rodiče a současném vybrání uzlu s nízkou hloubkou druhého rodiče může dojít k extrémnímu navýšení hloubky. To může být žádoucí na začátku evoluce, avšak později může dojít k nadbytečnému narůstání hloubky stromu, aniž by to mělo vliv na ohodnocení. Například u listu nesoucího terminál `0` a jemu nadřazenému uzlu, ve kterém je uložena funkce `*`, vůbec nezáleží na druhém podstromu tohoto uzlu, protože výsledek bude vždy 0. Kdyby tento podstrom byl extrémně veliký, rostla by pravděpodobnost, že bude vybrán jako bod křížení uzel právě z něj. Tímto způsobem by narůstala velikost stromu, aniž by to mělo vliv na výsledek. Tento problém se značí anglickým názvem *bloat* (viz obrázek 2.7).

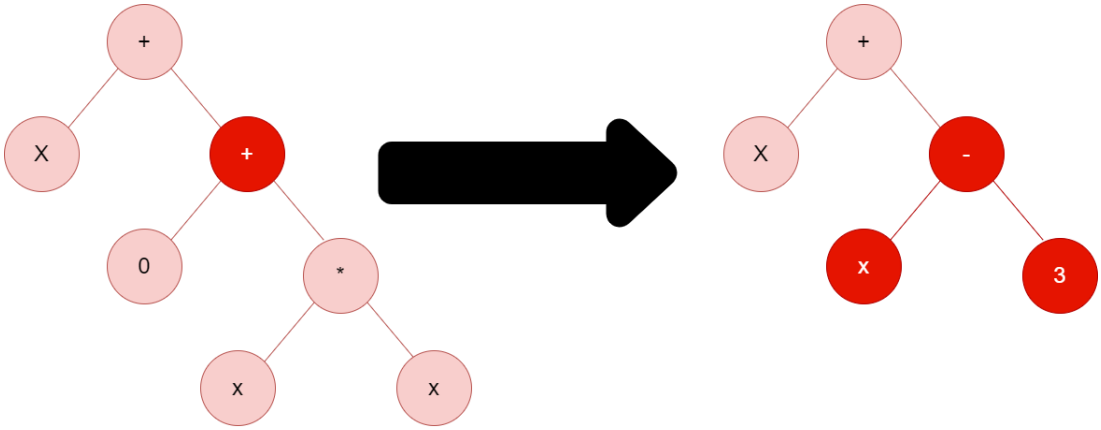
Tato problematika lze řešit vícero způsoby. Například s rostoucí hloubkou stromu by se mohla snižovat jeho fitness, nebo by mohli být jedinci, kteří přesáhnou povolenou hloubku, automaticky odstraněni z populace. Existují však i další způsoby, jako například snížení pravděpodobnosti výběru bodu křížení u uzlů blízko kořene a listům. [14]



Obrázek 2.7: Bloat

2.7.6 Mutace

Stejně jako u křížení, i u mutace nejprve musí dojít k vyhledání nějakého bodu, zde k bodu mutace. Ten je vybrán obdobným způsobem, jako u kombinace zmíněné v 2.7.5. Vybraný podstrom je odstraněn, poté dojde k jeho nahrazení náhodně vygenerovaným podstromem. Mutaci znázorňuje obrázek 2.8.



Obrázek 2.8: Mutace v GP

Při provádění mutace je potřeba zohlednit maximální povolenou hloubku d, v opačném případě by mohlo dojít k vytvoření většího jedince, než bylo povoleno. [14]

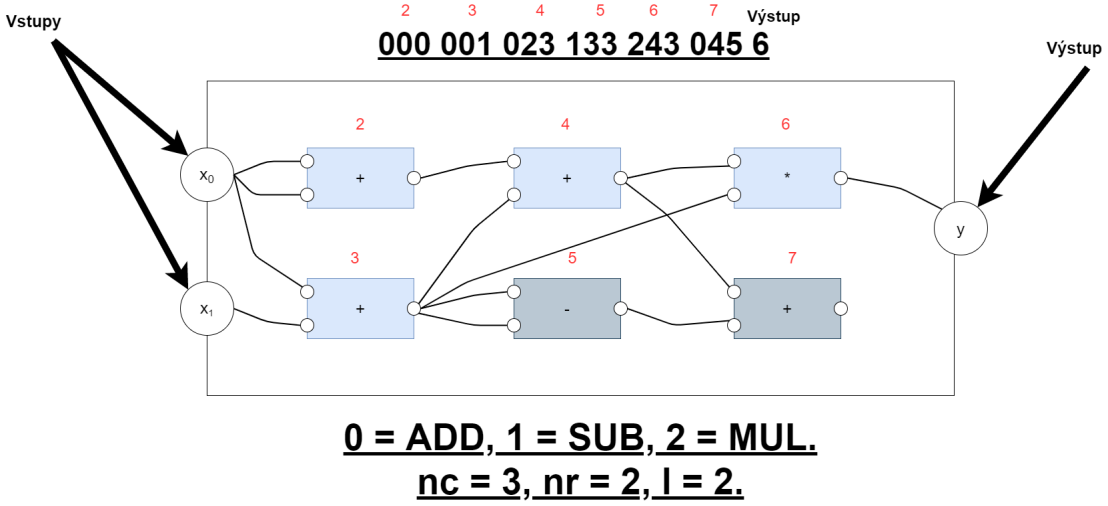
2.8 Kartézské genetické programování

Tato forma GP nese označení „kartézské“, protože kandidátní řešení jsou reprezentována formou dvou-dimenzionální mřížky uzlů. Geny mají formu celých čísel. Tato čísla označují jaká funkce je obsažena v daném uzlu a také z jakých uzlů pochází vstupy. Oproti GP genotyp u kartézského genetického programování (dále „CGP“) má předem pevně danou velikost, která se za běhu nijak nemění. Při dekódování genotypu na fenotyp se však může stát, že některé uzly zůstanou nevyužity. Množinu funkcí používanou pro řešení problémů touto formou definuje programátor. Jeden gen v uzlu reprezentuje funkci, zbytek vstupy. Tyto vstupy mohou být vstupy programu, ale i výstupy předchozích uzlů (uzlů v předešlých sloupcích, z čehož plyne, že cykly nejsou povolovány). Počet vstupů a výstupů omezen není.

Programátor také musí definovat 3 řídicí parametry: počet sloupců nc, počet řádků nr a také parametr l, který značí, jaké uzly mohou sloužit pro vstup (tento parametr nijak neovlivňuje možnost brát vstup přímo z primárního vstupu). Pokud je tento parametr nastaven na $l = 1$, vstupy uzlů mohou být pouze buď primární vstup, nebo uzly, které se nacházejí v předchozím sloupci. [7] Princip CGP znázorňuje obrázek. 2.9

2.8.1 Mutace v CGP

Tento operátor spočívá v nahrazení náhodně zvoleného genu v genotypu validní hodnotou. Pokud dojde k pozměnění genu udávajícího funkci, vybere se nějaká hodnota, která rovněž odpovídá nějaké funkci specifikované programátorem. Pokud dojde k nahrazení genu zodpovědného za vstup, vybere se hodnota primárního vstupu, nebo identifikátor uzlu s ohledem na parametr l. Příkladem nevalidní mutace za situace popsané obrázkem 2.9 by mohla být



Obrázek 2.9: Kartézské genetické programování

změna úplně prvního genu v genotypu na hodnotu 3. Za jistých okolností by některé geny mohly nést tuto informaci, avšak pro tento konkrétní gen je změna nevalidní, protože nebyla specifikována funkce s identifikátorem 3. Mutace mohou mít velký vliv na výsledný fenotyp. Pokud by došlo ke změně genu popisujícího zdroj výstupu, mohlo by být potenciální řešení nesené jedincem výrazně jinou formu, než tomu bylo u jeho rodiče. Tvorba potomků je obvykle implementována pomocí evoluční strategie $(\mu + \lambda)$, viz 2.6. Nejčastěji bývají tyto parametry voleny jako $\mu = 1$ a $\lambda = 4$. [7]

Pokud má nějaký potomek stejnou fitness jako jeho rodič, je tento potomek zvolen rodičem příští generace (namísto současného rodiče). V opačném případě by mohlo dojít k uváznutí v lokálním optimu řešeného problému. [15]

2.9 Koevoluční algoritmy

Koevoluční algoritmy, stejně jako ty evoluční, jsou inspirovány evoluční teorii o přežití těch nejsilnějších. U evolučních algoritmů dochází k ohodnocování pomocí již zmíněné fitness funkce (viz kapitola 2). Oproti tomu koevoluční algoritmy mimo to ohodnocují jedince i na základě jejich interakce s jinými jedinci ve stejné nebo jiné populaci. [8]

2.9.1 Predikce fitness funkce

Tato technika se používá pro nahrazení výpočetně náročné fitness funkce její predikcí (typicky pomocí malé podmnožiny trénovacích vektorů), která se optimalizuje souběžně s populací jedinců. Zmíněná predikce však nikdy nebude tak přesná, jako její originální verze. Pro takovýto odhad fitness se používá anglický pojem *fitness prediction*. [10]

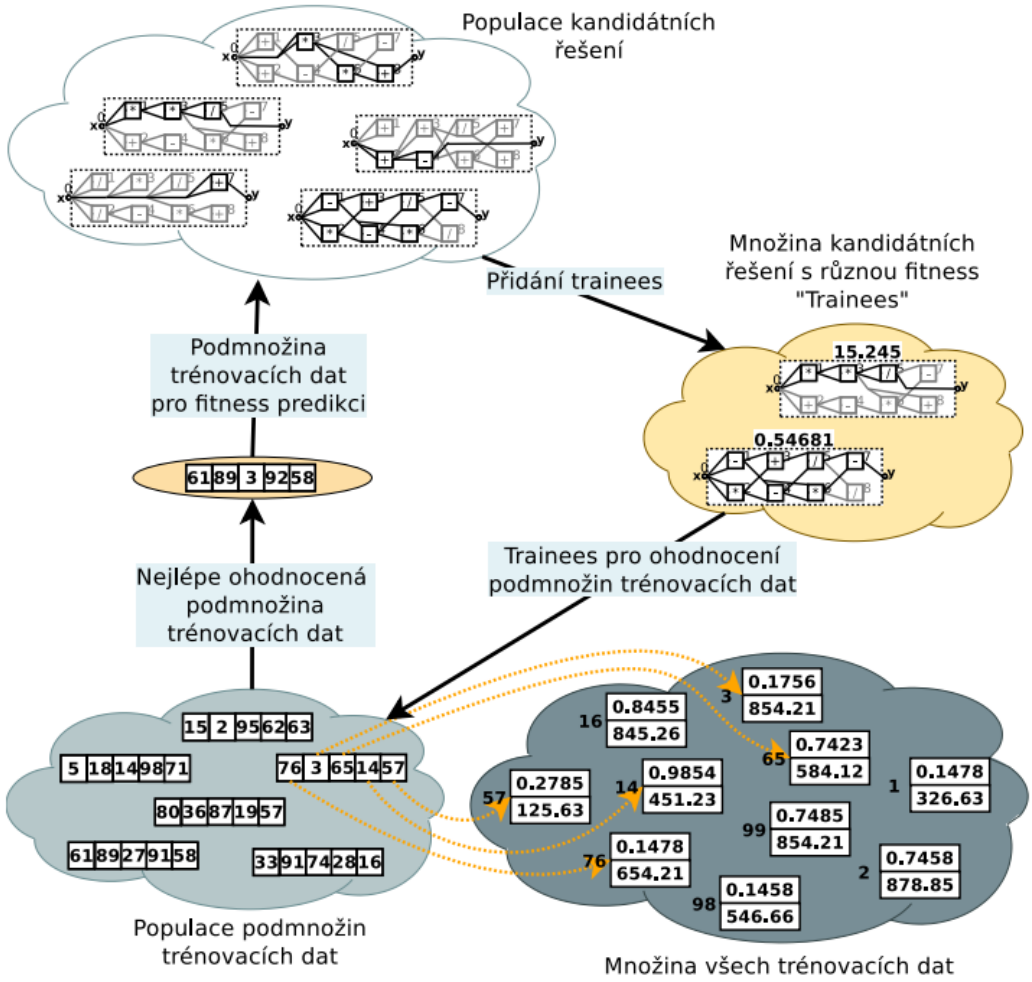
2.9.2 Populace v koevolučních algoritmech

Běžně se pracuje s následujícími populacemi:

- populace fitness prediktorů
- populace jedinců k řešení zadaného problému, ohodnocovaných fitness prediktory

- archiv trenérů (kandidátní řešení), jejichž přesná fitness se používá k trénování prediktorů

Populace jedinců je vyvíjena takovým způsobem, aby dosahovala co nejvyšší fitness podle nejlepšího fitness prediktoru z populace prediktorů. Prediktory jsou trénovány pomocí archivu trenérů za použití jednoduchého genetického algoritmu tak, aby jejich odchylka vůči skutečné fitness funkci byla co nejmenší. Trenéři jsou voleni takovým způsobem, aby bylo možné odhalit nedostatky u prediktorů (rozdílnost oproti originální fitness). Populace jedinců i prediktorů jsou zpočátku vytvořeny zcela náhodně. Archiv trenérů je po počáteční inicializaci složena z náhodně vybraných jedinců reprezentujících řešení (přímo z populace jedinců). Trenéři jsou obměňováni periodicky. Nejlépe ohodnocený jedinec (podle momentálně používaného fitness prediktoru) nahradí nejstaršího jedince vloženého do archivu trenérů. [10] Tento princip znázorňuje obrázek 2.10 převzatý z [17].



Obrázek 2.10: Koevoluce, převzato z [17]

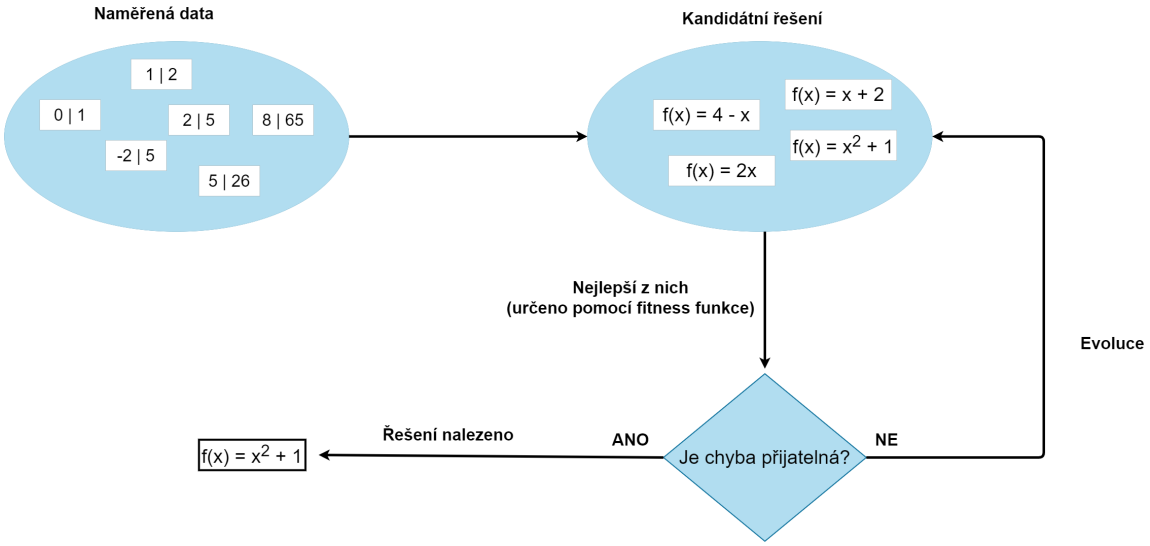
Kapitola 3

Symbolická regrese

Po staletí se vědci snažili analyzovat získaná data a objevit tak nové fyzikální zákonitosti. Když už lidstvo konečně mělo k dispozici velkou výpočetní sílu, stále naráželo na problém neexistence prostředků pro jejich automatickou analýzu. Klíčovým úkolem tehdy bylo jak zjistit matematické vztahy z dostupných informací. Právě matematickými vztahy lze vcelku snadno popsat leckteré (nejen) fyzikální zákony. [9]

Tuto problematiku řeší symbolická regrese. Ta je metodou založenou na evolučních algoritmech pro prohledávání prostoru ve smyslu zisku matematických závislostí naměřených dat s cílem o co nejmenší odchylku od reality. Oproti metodám pro regresní analýzu není nijak omezena tvarem, ve kterém lze řešení očekávat. Počáteční výrazy jsou vytvořeny náhodně za využití elementárních funkcí, konstant a proměnných. Další rovnice jsou předmětem evoluce, jak popisuje kapitola 2. Jakmile je nalezeno dostatečně přesné řešení, je evoluce u konce.

Výsledné zjištěné závislosti mohou mít pro jeden problém nespočet řešení. Tato řešení se od sebe mohou dramaticky lišit, dvě na první pohled úplně rozdílné rovnice mohou být ekvivalentní. Pokud nám při evoluci vyjde vícero řešení stejné kvality, zpravidla vybíráme to nejkratší z nich. [9] Průběh symbolické regrese znázorňuje obrázek 3.1.



Obrázek 3.1: Ilustrace průběhu symbolické regrese

Kapitola 4

Návrh nástroje pro porovnání variant symbolické regrese

Rozhodl jsem se pro implementaci v jazyce Java, verze 8. Pro co nejrovnější podmínky je potřeba, aby spolu jednotlivé varianty genetického programování sdílely pokud možno co nejvíce kódu. Rovněž bude nutné se pokusit o co nejvhodnější nastavení algoritmů. Zde nemám namysli pouze řídicí parametry, ale i nastavení typu vyhodnocování matematicky nevalidních výrazů (typicky dělení nulou). Existuje spoustu způsobů, jak se s tím vyrovnat a mým cílem bude se snažit o nalezení co nejvhodnějších metod. Z těchto důvodu jsem se rozhodl, že nebudu využívat žádných knihoven zabývajících se symbolickou regresí a napíši program podle zadané literatury (citované v kapitolách 2 a 3) úplně od začátku.

4.0.1 Jádro programu

Plánuji využít objektově orientovaného programování, tedy například dědění metod a atributů společných pro CGP i GP od nějakého jejich předka. Nejdůležitější částí programu bude třída, která bude obsahovat vše, co mají CGP i GP společné. Rovněž bude třeba naimplementovat koevoluci, kterou bude využívat tento obecný evoluční algoritmus nehledě na to, jestli zvolím symbolickou regresi pomocí CGP, nebo GP.

Program bude obsahovat hlavní smyčku, která bude vytvářet nové a nové generace na základě jejich kvality a po nalezení řešení požadované kvality se ukončí. Tento cyklus bude omezen jak časem, tak maximálním počtem generací. Plánuji experimentovat s vícero způsoby ohodnocování jedinců.

4.0.2 Koevoluce

Koevoluce bude probíhat paralelně. Zatímco vlákno vyvíjející kandidátní řešení bude běžet neustále (do ukončení z důvodu nalezení řešení, vypršení času aj.), druhé vlákno vyvíjející prediktory bude periodicky spouštěno a po dokončení své úlohy bude vypínáno. Také bude třeba naimplementovat přístup ke sdílené paměti těchto procesů (například populace trenérů). K tomu bude třeba využít synchronizačních prostředků.

4.0.3 Načítání dat

Data budou načítána ze souborů typu CSV. Není žádoucí, aby uživatel musel do tohoto souboru explicitně vypisovat kolik se zde nachází závislých proměnných, kolik nezávislých,

jaké je množství vstupních dat a podobně. Bude tedy třeba implementovat načítání, které toto automaticky určí.

4.0.4 Kandidátní řešení

Jedinci v populaci kandidátních řešení budou rovněž tvořeni třídami, nebudu tedy pracovat s reprezentací jedinců pomocí řetězců. V rámci těchto tříd by mohla být ukládána dodatečná data (například ve kterém sloupci se logický CGP uzel nachází, aby na základě parametru *lback* mohl být určen jeho vstup). Protože třída, která bude předkem tříd zabývajících se CGP a GP, může pracovat s vícero typy kandidátních řešení, bude třeba spoustu metod přepsat v rámci potomků.

Program nebude pracovat nutně jen s jednou populací, v případě vylepšení koevolucí přibudou i populace prediktorů a populace, respektive archiv, trenérů.

4.0.5 Návrh kartézského genetického programování

Pro tvorbu nové generace budu využívat evoluční strategie (kapitola 2.6). To vše se bude odehrávat v metodách třídy, která bude potomkem třídy popisující obecný evoluční algoritmus pro symbolickou regresi. Dále bude využívat logických uzlů, které budou rovněž reprezentovány pomocí třídy. Těchto členů však bude v rámci jednoho kandidátních řešení vícero, proto bude třeba použít ještě jednu třídu, která bude reprezentovat kandidátní řešení a obsahovat tyto komponenty.

4.0.6 Návrh genetického programování

Zde budu pracovat se stromovými strukturami. Strom bude obsahovat uzly, popisující matematické operátory, a listy, nesoucí proměnnou, nebo konstantu. Všechny zmíněné komponenty budou mít opět nějakého předka, skrze kterého půjde přistupovat jednotným způsobem ke všem článkům stromu. Například kopírování podstromu se bude volat nad tímto předkem, přičemž jak listové, tak uzlové struktury (třídy) tuto metodu přepíší. Bude potřeba stanovit, na základě čeho bude vybírán obsah generovaných stromů a podstromů (tím mám na mysli volbu mezi novým terminálem a funkčním uzlem).

Stromová struktura bude binární strom. V případě unární matematické operace (například $sin(x)$) bude druhý podstrom (ten pravý) zcela ignorován.

4.0.7 Testování

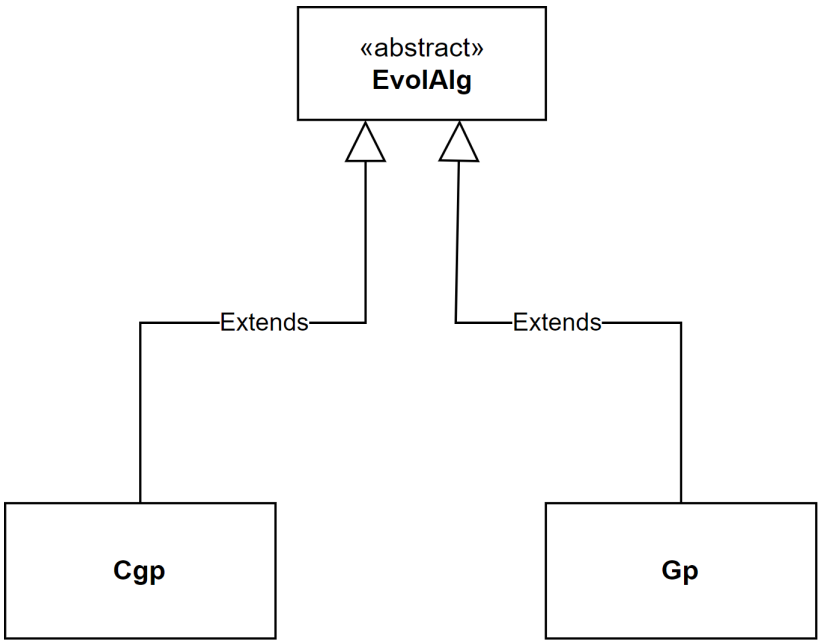
Experimenty budou probíhat opakovaným spouštěním programu a vyhodnocováním výsledků. Předmětem testování bude doba potřebná k nalezení řešení požadované kvality, průměrná fitness v čase a nejlepší dosažená fitness za pevně stanovenou dobu běhu.

Veškeré výsledky se budou vypisovat do textového souboru. Rovněž bude možné vykreslit regresní křivku souběžně s poskytnutými daty a ohodnotit kvalitu regrese na základě tohoto vykreslení.

Kapitola 5

Implementace nástroje pro porovnání variant symbolické regrese

Pro porovnávání variant genetického programování jsem použil vlastní implementaci podle zadané literatury, ze které jsem rovněž čerpal při psaní teoretického úvodu (viz kapitoly 2 a 3). Jednotlivé varianty spolu sdílí podstatnou část kódu a rovněž používají stejné atributy a metody pro koevoluci. Tímto jsem se snažil dosáhnout co nejrovnějších podmínek, aby bylo porovnání co nejpřesnější. Třída *EvolAlg* zastupuje evoluční algoritmus obecně, její 2 potomci *Gp* a *Cgp* poté zkoumané varianty implementují, viz diagram 5.1.



Obrázek 5.1: Nejdůležitější třídy programu

Algoritmus 6 popisuje hlavní smyčku programu. V tomto případě se jedná o variantu bez koevoluce. Jelikož program v takovémto případě pracuje pouze se jednou populací, nese tato metoda označení *singlePopEvolution*.

V případě spuštění symbolické regrese s koevolucí dojde ke spuštění jiné metody, a sice *coevolution* (viz algoritmus 7). Tato metoda pracuje se 2 populacemi a 2 archivy:

- Populace kandidátních řešení (Gp stromů nebo Cgp mřížek)
- Populace prediktorů fitness s pevnou velikostí těchto prediktorů
- Archiv trenérů
- Vybraný prediktor pro ohodnocování kandidátních řešení

Algoritmus 6: singlePopEvolution

```
1: Inicializuj trénovací vektory
2: Vytvoř počáteční populaci
3: while (nenalezeno řešení and nebyl překročen maximální povolený čas and nebyl
     překročen maximální počet generací) do
4:     Vytvoř novou generaci
5:     Zaznamenej statistiky
6:     Inkrementuj počítadlo generací
7: end
8: return Souhrnné statistiky
```

Algoritmus 7: coevolution

```
1: Inicializuj trénovací vektory
2: Vytvoř počáteční populaci
3: Inicializuj trenéry
4: Inicializuj prediktory
5: Získej nejpřesnější prediktor
6: while (nenalezeno řešení and nebyl překročen maximální povolený čas and nebyl
     překročen maximální počet generací) do
7:     Vyměň aktuálně používaný prediktor, pokud byl změněn
8:     Vytvoř novou generaci
9:     Každých N generací spusť evoluci prediktorů
10:    Zaznamenej statistiky
11:    Inkrementuj počítadlo generací
12: end
13: return Souhrnné statistiky
```

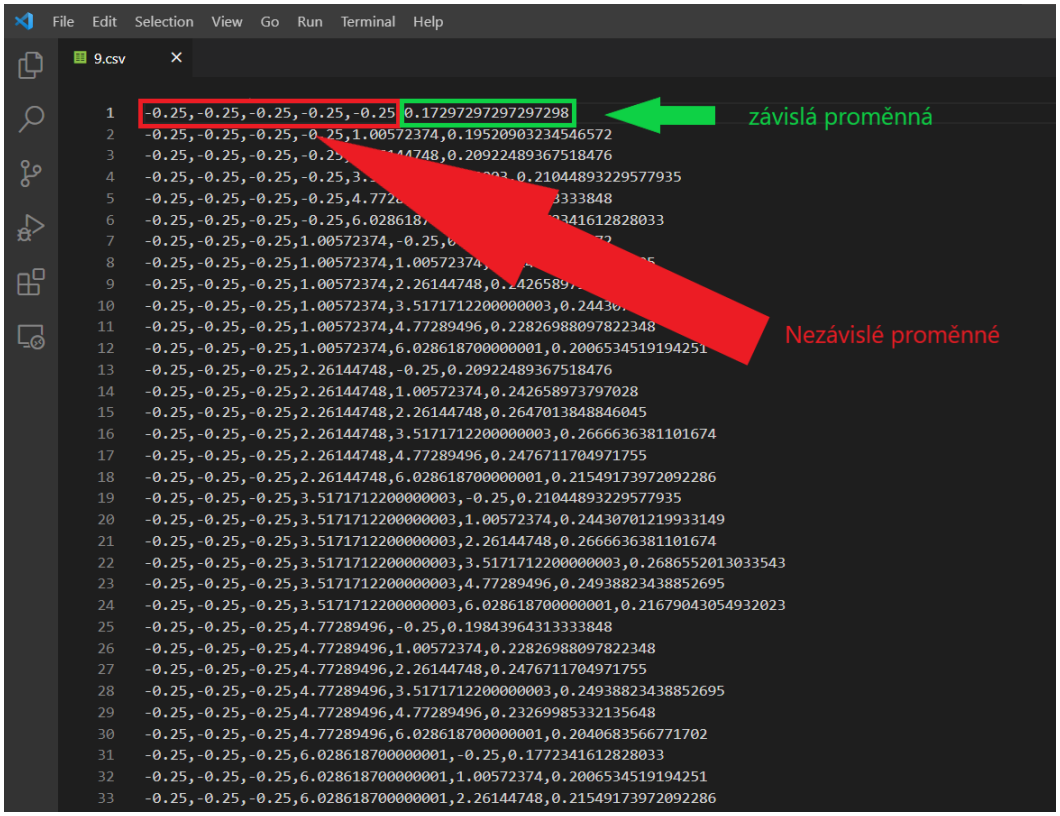
5.1 Symbolická regrese bez koevoluce

5.1.1 Trénovací vektory

Jako první dojde k inicializaci trénovacích vektorů. Pro ty byla vytvořená třída *TrainerVector*. Ta obsahuje 2 atributy: nezávislé proměnné a závislá proměnná. Reprezentace nezávislých proměnných je realizována jako seznam datového typu *double* (pracuji s možností více proměnných). Funkční hodnota je implementována jako jedno číslo téhož datového typu.

5.1.2 Načítání trénovacích vektorů

Pro účely poskytnutí vstupních dat, se kterými program pracuje, se používají CSV soubory. Poslední hodnota na řádku představuje funkční hodnotu, zatímco zbytek vstup (reprezentace jde vidět na obrázku 5.2). Počet trénovacích vektorů není programem omezen, je tedy teoreticky (když opomeneme paměť apod.) možné načíst libovolně veliká vstupní data.



Obrázek 5.2: Reprezentace trénovacích vektorů v CSV souboru

Při načítání vstupních dat algoritmus detekuje počet proměnných, které se v poskytnutých datech vyskytují. Danou hodnotu uloží do atributu *inputSize*.

5.1.3 Vytvoření počáteční populace

Dalším krokem je inicializace počáteční generace. Vzhledem k tomu, že abstraktní[1] třída *EvolAlg* neví, jestli jedinci budou kartézské mřížky, nebo stromové struktury, je tato metoda implementována potomky zmíněné třídy. Populace je reprezentována pomocí rovněž abstraktní struktury (třídy) *Jedinec*. Algoritmus pracuje s polem této třídy, velikost populace se za běhu programu měnit nemůže. Atribut *POP_SIZE* určuje, kolik jedinců se bude v populaci kandidátních řešení nacházet.

Jak již bylo zmíněno dříve, metodu inicializace populace je třeba doimplementovat pro každou variantu genetického programování. Například u GP dojde k zavolání metody *ramped half-and-half* (viz algoritmus 5), zatímco u CGP je nutné zvolit jinou formu vytvoření počáteční populace. Popis těchto implementací se nachází v sekcích 5.3 a 5.4.

[1]Poznámka: v prostředí Java abstraktní třída značí takovou třídu, která nemůže být instanciována (je potřeba využít nějakého neabstraktního potomka)

5.1.4 Hlavní smyčka

Hlavní smyčka tvořící jádro programu může být ukončeno 3 různými způsoby, viz algoritmy 6 a 7. Nalezením řešení se myslí zaznamenání takového jedince, který má požadovanou kvalitu. Může to být třeba tolerovaná průměrná odchylka, nebo třeba 100% přesné řešení. V druhém případě však z důvodu zaokrouhlovacích chyb (které vznikají kvůli omezené přesnosti datových typů) se za nulovou odchylku považuje takové řešení, které udává konstanta *approximatelyZero*. Ta je nastavena na 10^{-5}.

V případě symbolické regrese se může stát, že algoritmus uvázne v nějakém lokálním optimu a dále nedochází ke zlepšování jedinců v populaci kandidátních řešení. Z toho důvodu byl zaveden maximální povolený počet generací a tolerovaná doba běhu programu. Tyto 2 podmínky můžou způsobit, že je algoritmus předčasně ukončen, aniž by nalezl požadované řešení.

V rámci porovnávání variant genetického programování jsem se soustředil i na srovnávání nejlepších dosažených fitness při zvolené maximální délce běhu programu (kapitola 6). V tomto případě může být algoritmus ukončen po uplynutí dané doby standardním způsobem, vypršení poskytnutého času tedy nutně neimplikuje selhání.

Během hlavní smyčky dochází k vytváření nových generací (sekce 5.1.5). Rovněž je každých M sekund evidována nejlepší dosažená fitness pro účely statistik. Dále se zvyšuje počítadlo generací pro účely ukončení programu po daném počtu iterací.

5.1.5 Nové generace

Metoda *newGeneration* je rovněž abstraktní, je tedy potřeba ji přepsat. CGP i GP mají odlišný přístup ke tvoření nových generací, obecně tedy být popsány nemohou. Popis takovéto evoluce je obsažen v sekcích 5.3 a 5.4.

5.1.6 Generování konstant

Další společnou částí CGP a GP je generování konstant. V tomto případě jsem zvolil řešení v podobě tabulky (konstant), která se inicializuje ještě před spuštěním hlavní smyčky. Tato tabulka obsahuje desetinná čísla v rozmezí předem stanoveného intervalu s pevným krokem. Konkrétně obsahuje konstanty v intervalu $< -20, 20 >$, kde krok $i = 0.2$. K tomu jsou přidána ještě další čísla, jako například číslo π. Tuto tabulku konstant znázorňuje tabulka 5.1. Všechny konstanty jsou vybírány se stejnou pravděpodobností, nedochází tedy k upřednostňování např. kladných čísel, celých čísel apod. Existuje jedna výjimka. Konstanta $c = 1$ je využívána mnohem častěji než ostatní konstanty, protože byla zvolena jako návratová hodnota v případě vyhodnocení nějakého nevalidního výrazu. Pokud k něčemu takovému dojde (objeví se nevalidní výraz při vyhodnocování), je navrácena vždy právě tato konstanta a nikdy nedojde k použití nějaké jiné (viz sekce 5.3.7 a 5.4.3).

-20.00	-19.80	-19.60	-19.40	-19.20	-19.00	-18.80	-18.60	-18.40	-18.20
-18.00	-17.80	-17.60	-17.40	-17.20	-17.00	-16.80	-16.60	-16.40	-16.20
-16.00	-15.80	-15.60	-15.40	-15.20	-15.00	-14.80	-14.60	-14.40	-14.20
-14.00	-13.80	-13.60	-13.40	-13.20	-13.00	-12.80	-12.60	-12.40	-12.20
-12.00	-11.80	-11.60	-11.40	-11.20	-11.00	-10.80	-10.60	-10.40	-10.20
-10.00	-9.80	-9.60	-9.40	-9.20	-9.00	-8.80	-8.60	-8.40	-8.20
-8.00	-7.80	-7.60	-7.40	-7.20	-7.00	-6.80	-6.60	-6.40	-6.20
-6.00	-5.80	-5.60	-5.40	-5.20	-5.00	-4.80	-4.60	-4.40	-4.20
-4.00	-3.80	-3.60	-3.40	-3.20	-3.00	-2.80	-2.60	-2.40	-2.20
-2.00	-1.80	-1.60	-1.40	-1.20	-1.00	-0.80	-0.60	-0.40	-0.20
-0.00	0.20	0.40	0.60	0.80	1.00	1.20	1.40	1.60	1.80
2.00	2.20	2.40	2.60	2.80	3.00	3.20	3.40	3.60	3.80
4.00	4.20	4.40	4.60	4.80	5.00	5.20	5.40	5.60	5.80
6.00	6.20	6.40	6.60	6.80	7.00	7.20	7.40	7.60	7.80
8.00	8.20	8.40	8.60	8.80	9.00	9.20	9.40	9.60	9.80
10.00	10.20	10.40	10.60	10.80	11.00	11.20	11.40	11.60	11.80
12.00	12.20	12.40	12.60	12.80	13.00	13.20	13.40	13.60	13.80
14.00	14.20	14.40	14.60	14.80	15.00	15.20	15.40	15.60	15.80
16.00	16.20	16.40	16.60	16.80	17.00	17.20	17.40	17.60	17.80
18.00	18.20	18.40	18.60	18.80	19.00	19.20	19.40	19.60	19.80
20.00	π	e							

Tabulka 5.1: Tabulka konstant

5.1.7 Fitness kandidátních řešení

Vzhledem k tomu, že program zajišťuje (symbolickou) regresi, rozhodl jsem se pracovat s fitness hodnotami následujícím způsobem: Pracuji s odchylkami od očekávaných řešení, **menší fitness** (představující menší odchylku) tedy **značí lepšího jedince**.

5.1.8 Fitness funkce

Program pracuje s vícero způsoby pro ohodnocení jedinců. Použity byly následující fitness funkce [2]:

MAE

Funkce MAE (mean absolute error) značí průměrnou absolutní odchylku, tedy zprůměrovaný součet absolutních hodnot všech odchylek, kde odchylka značí rozdíl mezi předpokládanou a výslednou hodnotou, viz následující rovnice 5.1:

$$MAE = \frac{1}{n}\sum_{i=1}^{N}|Y_i - \hat{Y}_i| \tag{5.1}$$

kde $\hat{Y}_i$ je odhadovaná hodnota (získaná za použití kandidátního řešení a vstupu trénovacího vektoru) a Y_i je skutečná hodnota (funkční hodnota v trénovacím vektoru).

MSE

Druhá funkce, MSE (mean squared error), je té první do značné míry podobná. Odchylka se pouze umocní (viz vztah 5.2).

$$MSE = \frac{1}{n}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2 \tag{5.2}$$

RMSE

RMSE značí root-mean-square error. Jak již název napovídá, vychází z MSE. Předpis popisuje rovnice 5.3:

$$RMSE = \sqrt{MSE} \tag{5.3}$$

MASE

Poslední funkce, MASE (mean absolute scaled error), naopak využívá MAE [3]. Viz rovnice 5.4.

$$MASE = \frac{MAE}{\frac{1}{n-1}\sum_{i=2}^{n}|Y_i - Y_{i-1}|} \tag{5.4}$$

5.1.9 Turnajový výběr

Metoda *tournament*, jak již název napovídá, má na starosti turnajový výběr mezi 2 jedinci. Náhodně vybere 2 jedince z populace kandidátních řešení a porovná jejich fitness. Jedinec, který má menší chybu (tzn. **menší** hodnotu fitness, což značí **kvalitnější řešení**) je navrácen jako výsledek.

5.1.10 Kontrola kvality a hledání řešení

Obě varianty genetického programování při tvorbě nové generace vyberou nejlepšího jedince z té současné. Tohoto jedince poté beze změny přesunou i do generace nové. Nalezení takovéhoto kandidátního řešení zajišťuje metoda *getFittest*. V rámci této metody se zavolá i metoda *checkForSolution*, které je předán tento jedinec jako parametr (i jeho fitness).

Tato metoda (*checkForSolution*) je důležitá proto, že kontroluje, jestli jedinec předaný jako parametr splňuje požadavky na kvalitu řešení, nebo ne. Pokud ano, nastaví příznak *solution_found* na *true* a evoluce je u konce.

5.1.11 Použité matematické operátory

Symbolická regrese pracuje s následujícími operátory: $a + b$, $a - b$, $a * b$, a/b, $\sin(a)$, $\cos(a)$, e^a a $\log(a)$, jak bylo nastaveno v [18]. CGP navíc obsahuje operátor $CONST$, značící konstantu.[2]

[2] V GP jsou konstanty zpracovávány v rámci terminálů, viz 2.7

5.1.12 Závěrečné statistiky

Jakmile je evoluce u konce (z jakéhokoli důvodu), dojde k navrácení hodnoty odpovídající třídě *Results*. Ta eviduje:

- Dobu běhu programu
- Nejlepší nalezenou fitness
- Nejlepší nalezené řešení
- Množinu trénovacích vektorů
- Seznam nejlepších dosažených fitness v čase (zaznamenáváno každých M sekund, zvolil jsem $M = 0.25$)

5.2 Symbolická regrese s koevolucí

Koevoluce představuje nadstavbu nad genetickým programováním bez koevoluce. Využívá výše zmíněné metody (popsané v kapitole 5.1), přidává nové a některé existující mírně upravuje.

Inicializace trenérů probíhá následujícím způsobem: jsou postupně kopírováni jedinci z populace kandidátních řešení a v případě, že je trenérů větší počet než v populaci řešení dochází k vytváření náhodných jedinců až do zaplnění archivu.

Vytvoření počáteční generace prediktorů probíhá pouze za pomocí náhodného vytváření jedinců této populace. Každý prediktor tak odpovídá náhodně vytvořené podmnožině množiny trénovacích vektorů (jednotlivé trénovací vektory se v rámci jednoho prediktoru neopakují).

5.2.1 Vlákno vyvíjející prediktory

V mé implementaci jsou využívána 2 vlákna: hlavní vlákno, které se stará o evoluci kandidátních řešení za pomocí jednoho zvoleného prediktoru a sekundární vlákno, které prediktory vyvíjí. V případě druhého vlákna dochází k spuštění a vypínání periodicky. Jakmile počet generací dosáhne určité hodnoty, je spuštěno vlákno pro vývin prediktorů. To najde nejkvalitnější prediktor a přiřadí jej hlavnímu vláknu. Po dokončení této úlohy dojde i k ukončení tohoto vlákna. To je později znovu spuštěno, opět po proběhnutí určitého počtu generací.

5.2.2 Vývin kandidátních řešení

V případě koevoluce jsou kandidátní řešení ohodnocována podle jednoho (nejlepšího) prediktoru fitness. Na začátku je tedy třeba jej stanovit. Hlavní vlákno spustí vývoj prediktorů (sekundární vlákno) a čeká, dokud mu toto druhé vlákno nějaký prediktor nepřiřadí. Jakmile získá prediktor, vstupuje do hlavní smyčky.

5.2.3 Hlavní smyčka v koevoluci

Podmínky pro ukončení běhu programu (tedy konec hlavního cyklu) jsou stejné, jako při řešení bez koevoluce (viz sekce 5.1.4).

V rámci každé nové generace hlavní vlákno nejprve ověří, jestli nedošlo k aktualizaci prediktoru. Toto zajišťuje metoda *accessCurrentPredictor*, kde dochází k předání nového

prediktoru přes další pomocný atribut. Tato metoda je *synchronizována*, což v prostředí Java znamená, že do ní může vstoupit pouze jedno vlákno v čase (tzv. monitor).

5.2.4 Nové generace prediktorů

Konstanta *generationsToAdapt* udává, po kolika generacích (kandidátních řešení) dojde k evoluci prediktorů. Při dosažení tohoto počtu je spuštěno sekundární vlákno, které na základě archivu trenérů aktualizuje prediktor fitness používaný hlavním vláknem.

Třída *PredictorEvolver* se stará o tento vývin. Jedná se o potomka vestavěné třídy *Thread*, pracuje tedy jako vlákno. Na začátku dojde k výlučnému[3] přístupu k trenérům a je vytvořena jejich lokální kopie v rámci tohoto vlákna.

Při tvorbě nové generace prediktorů jsem nejprve používal genetický algoritmus (GA). Nicméně jsem se ve výjimečných případech (cca 2 % běhů) potýkal s problémem zacyklení, který souvisel s nemožností nalezení lepšího prediktoru, než je ten stávající. Po provedení několika experimentů mi jako nejlepší řešení této problematiky přišla následující strategie (pro vytvoření příští generace): Do populace je přidán jeden jedinec, který je úplně nový a zcela náhodně vytvořený. Zbytek jsou jedinci zvolení turnajovým výběrem, na které byl aplikován operátor mutace. Turnajový výběr vytváří selekční tlak a upřednostňuje tak lepší jedince před méně kvalitními. Pomocí mutace lze pak tyto jedince mírně měnit, a tím i potenciálně vylepšovat, čímž se přibližují k hledanému řešení. Mutace spočívá v nahrazení náhodně vybraných genů nějakým jiným, validním genem[4].

Pro vyhledání nejvhodnějšího prediktoru fitness je využíván archiv trenérů. Pro každého trenéra se vypočítá jeho objektivní fitness, jeho subjektivní fitness (ta se určuje podle zkoumaného prediktoru) a tento rozdíl se přičte do sumy, která vypovídá o rozdílnosti subjektivní a objektivní fitness. Tato suma je nakonec vydělena počtem trenérů, jedná se tedy o MAE, viz vztah 5.1. Opět platí, že čím menší odchylka, tím lepší řešení.

5.2.5 Aktualizace trenérů

Archiv trenérů je rozdělen na dvě poloviny. První polovina je kruhově obměňována s každou změnou nejlepší nalezené fitness (při změně prediktoru je tento práh posunut na fitness současně nejlepšího jedince). Nejstaršího jedince nahradí ten nejlepší ze současné populace kandidátních řešení. Druhá polovina je měněna periodicky s každou novou generací prediktorů. Vyzkoušel jsem i jiné způsoby, které se však v mém případě ukázaly méně efektivní (periodická obměna obou polovin, změna 1 jedince z každé poloviny při každém zlepšení fitness).

5.2.6 Požadované subjektivní řešení

Pokud při koevoluci dojde k nalezení řešení požadované kvality, dojde nejprve ke kontrole pomocí objektivní fitness funkce. Algoritmus nastaví příznak *solution_found* na *true*, pokud je daný jedinec i podle objektivní fitness dostatečně kvalitní. Tímto dojde k ukončení evoluce.

Může se stát, že nějaké kandidátní řešení bude splňovat požadovanou kvalitu podle prediktoru fitness (tzv. subjektivní fitness), ale nebude tato kritéria splňovat podle fitness objektivní. V takovém případě dojde k vynucenému vývoji prediktorů a aplikování nového

[3] Opět se jedná o monitor, stejně jako u metody *accessCurrentPredictor*, popsané v sekci 5.2.3

[4] Validním se myslí takový gen, aby prediktor neobsahoval nějaký trénovací vektor 2×

prediktoru fitness, přičemž hlavní vlákno je po dobu evoluce v tomto případě dočasně pozastaveno.

5.3 Implementace kartézského genetického programování

Třída *CGP* se stará o inicializaci jedinců v populaci a o tvorbu nových generací. Třída *Grid* reprezentuje kartézskou mřížku. Třídy *Box* a *InputBox* zase jednotlivé logické bloky. Tyto bloky nesou atributy jako příslušný řádek a sloupec, vstupy, operaci, konstantu aj. CGP tedy není implementováno pomocí jednoho řetězce, který se poté interpretuje, ale pomocí několika vzájemně závislých tříd.

5.3.1 Inicializace populace

Pro vytvoření počáteční populace se využívá pouze náhodného generování jedinců. Není využívána nějaká optimalizační metoda, jako například u GP (kapitola 5.4).

5.3.2 Vytvoření nové generace

CGP využívá *evoluční strategie* (popsané v sekci 2.6). Nejlepší jedinec ze staré generace je přesunut do nové, zbylí jedinci jsou mutací prvního. Jedinou výjimkou je poslední kandidátní řešení v populaci, které je vytvořeno náhodně.

5.3.3 Jedinci kartézského genetického programování

Třída *Grid* obsahuje následující atributy:

- parametr lback
- parametr nr
- parametr nc
- outputSource, značící logický blok, jehož výstup je zpracováván
- inputSize, označující počet proměnných, se kterými CGP pracuje
- seznam logických bloků (tam jsou započítávány i vstupní bloky, viz 5.3.6)
- maxMutationCount, určující maximální počet mutací na genotypu jedince [18]
- operations - seznam povolených matematických operací (dělení, násobení...)

5.3.4 Konstruktor

V konstruktoru dojde k nastavení řídicích parametrů. Další důležitou součástí je vytvoření množiny logických bloků. Mimo $nr * nc$ bloků jsou do této množiny přidány i pseudobloky zodpovědné za vstup. Nakonec je náhodně vybraný blok vybrán jako výstup, přičemž příslušný atribut *outputSource* má podobu indexu (tedy celého čísla).

5.3.5 Mutace

Na začátku mutace jedince se určí počet provedených mutací. Ten je stanoven jako náhodné číslo v intervalu $< 1, maxMutationCount >$. [18] Pro každou mutaci se vybere náhodně logický box (náhodný index pro výběr ze seznamu boxů). Ve vybrané instanci bude pozměněn jeden z následujících atributů:

- Operace
- První vstup
- Druhý vstup
- Konstanta (využívána v boxech s operací $CONST$)

Může se stát, že náhodně vybraný box bude právě ten, který má na starosti vstup. V takovém případě dojde k mutaci na výstupu (změní se index *outputSource*).

5.3.6 Zisk funkční hodnoty

Třída *Grid* přiřadí každému ze vstupních boxů odpovídající vstupní proměnnou. Poté zavolá metodu *getValue* instance třídy *Box*, určené atributem *outputSource*. Poté dochází k rekurzivnímu volání této metody pro všechny aktivní boxy, přičemž na konci tohoto řetězce se vždy nachází speciální vstupní box (potomek třídy *Box* s názvem *InputBox*), který navrátí vstupní proměnnou, nebo klasický box s konstantou.

Vyhodnocování v mé implementaci probíhá čistě rekurzivně. Nedochází k žádnému určování aktivních uzlů apod. Z toho plyne, že některý box může vyhodnocovat svoji hodnotu vícekrát. Na základě mých experimentů (jejichž výsledky byly porovnávány s [18]) se však neprokázalo, že by tento postup měl zásadní vliv na zpomalení symbolické regrese. Naopak, v případě jednoduchých funkcí docházelo k rapidní akceleraci, která zásadně převyšovala zpomalení u těch komplexnějších.

5.3.7 Zpracovávání nevalidních výrazů

Pokud při vyhodnocení funkční hodnoty dojde k chybě (způsobené např. dělením nulou), dojde k navrácení konstanty 1. Chyba při vyhodnocování však může nastat i v případě matematicky nezávadných výrazech, jako například e^{2000}. Teoreticky vzato zde nedochází k problému spjatého s definičním oborem funkce e^x, nicméně prakticky dojde k přetečení, přičemž prostředí Java na tuto skutečnost zareaguje vložením příznaku *Infinity* do dané proměnné typu *double*. To poté vede na velké množství výsledků typu *NaN*. Zmíněná konstanta 1 je tedy navrácena jako výsledek i za takovýchto okolností.

5.4 Implementace genetického programování

Oproti CGP popsaném v sekce 5.3 nedochází v případě GP k zaplnění počáteční populace bez jakékoli heuristiky, ale využívá se algoritmus ramped half-and-half (viz algoritmus 5). GP obsahuje následující atributy, které reprezentují řídicí parametry:

- functionRate, což je pravděpodobnost vložení funkce do nově vytvořeného uzlu (doplněk značí pravděpodobnost vytvoření listu obsahujícího terminál)
- maximální hloubku stromu d

- varProb, tedy pravděpodobnost uložení proměnné do listu stromu (doplněk je pravděpodobnost konstanty)
- mutationRate, tedy pravděpodobnost mutace nového jedince v populaci (vzniklého po křížení)
- velikost populace

5.4.1 Genetické operátory

Křížení a mutace provádějí metody *GPcrossover* a *GPmutation*. V případě první zmíněné dojde k vytvoření 2 hlubokých kopií rodičů, vybere se náhodný uzel, nebo list (nedojde k výběru kořene) a poté dojde k nahrazení vybrané části prvního stromu vybraným podstromem toho druhého. Jako výsledek je navrácen 1 potomek.[5]

V případě mutace dojde rovněž k náhodnému výběru určitého podstromu, přičemž je tento prvek nahrazen zbrusu novým stromem.

Po zavolání libovolné výše zmíněné metody je třeba spustit metodu *truncate*, která zredukuje velikost stromu, pokud došlo k překročení maximální povolené hloubky.

5.4.2 Jedinci genetického programování

Zde jsou kandidátní řešení reprezentovány pomocí abstraktní třídy *Node* a jejích potomků *TerminalNode* a *FunctionNode*.

Atributy *left* a *right* jsou ukazatele na podstromy (můžou být v případě terminálů *null*). Atribut *operation* zaznamenává matematickou operaci příslušící danému uzlu (v případě listu nevyužito a neurčeno) index proměnné (používaný listy) a hodnotu konstanty (opět využívanou pouze listy).

Metoda *getCrossPoint* slouží k výběru náhodného podstromu pro genetické operátory. Postupně prochází stromovou strukturu a s pravděpodobností určenou jako $1/(d-1)$ zvolí uzel jako výsledný podstrom. V případě, že dojde k listu, dále nepokračuje a volí tento list.

Metoda *truncate* sloužící ke zkracování příliš dlouhých stromů funguje na principu nahrazování příliš hluboko umístěných funkčních bloků terminály.

5.4.3 Vyhodnocování výrazů

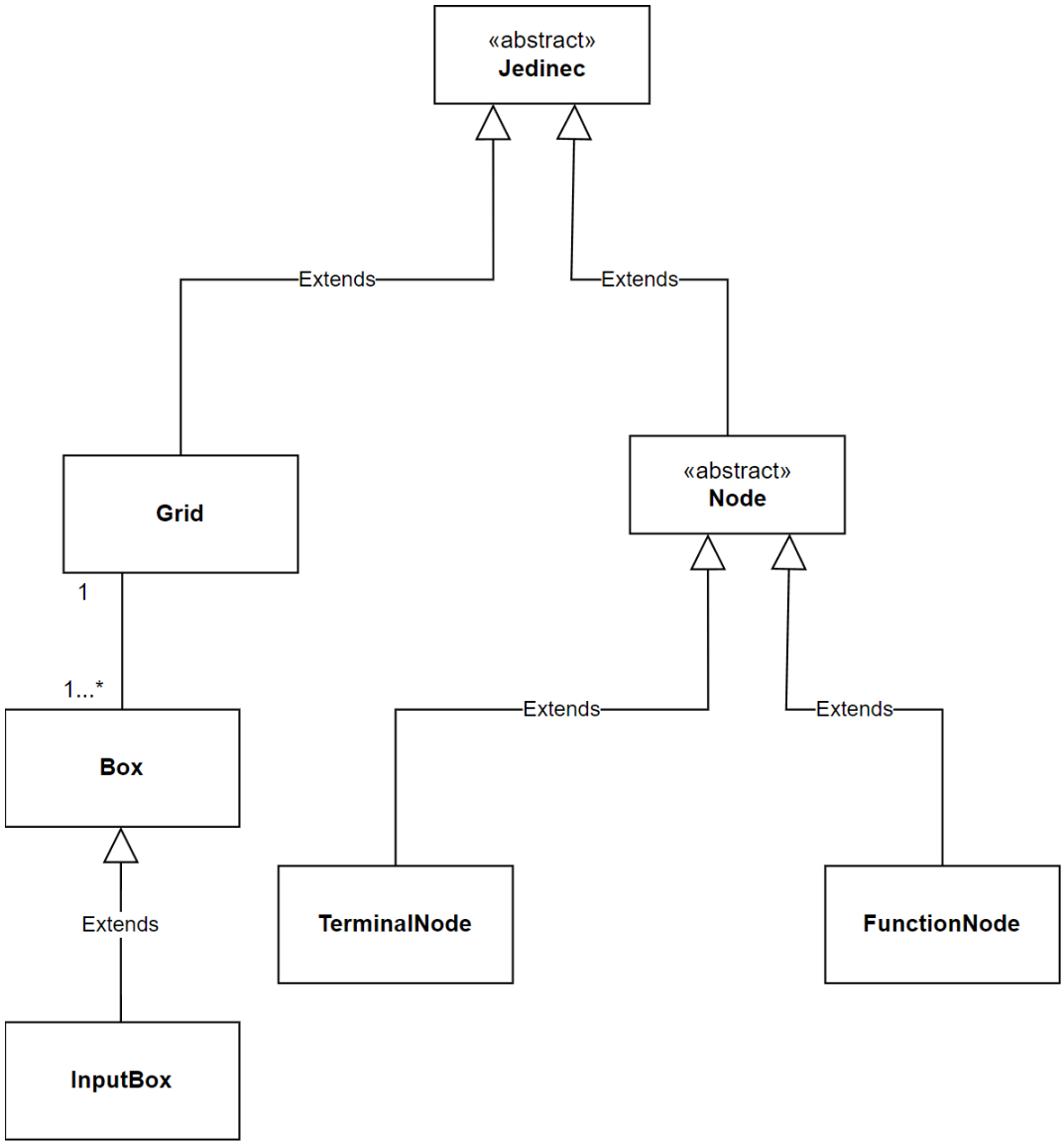
Zjištění funkční hodnoty funguje na stejném principu, jako u CGP (viz sekce 5.3.6). Opět plně rekurzivně, opět při nevalidní operaci dojde k navrácení konstanty 1. Na začátku se zavolá metoda *getValue* nad kořenem, poté dochází k postupnému vyhodnocování od tohoto kořene až po terminály s konstantami, nebo proměnnými.

5.5 Shrnutí abstrakce jedinců

V této kapitole bylo popsáno, že obecný abstraktní evoluční algoritmus pracuje s kandidátními řešeními třídy *Jedinec*. Tato třída má v případě CGP potomky typu *Grid*, kteří obsahují N logických členů (pro vstup, nebo pro operaci). Dále se, tentokrát v případě GP, používá pro reprezentaci jedinců abstraktní třída *Node*, s potomky *FunctionNode*, tedy

[5] Vytvoření jen jednoho potomka není v tomto případě obvyklé, většinou se používá přístup, kdy po křížení dojde k vytvoření 2 potomků, viz 2.7.5

uzlem stromu a *TerminalNode*, listem. Následující diagram (obrázek 5.3) tyto vztahy znázorňuje.



Obrázek 5.3: Reprezetnace kandidátních řešení

5.6 Třídy pro statistické a vizualizační účely

Program umožňuje spouštět následující experimenty:

- Porovnání rychlosti nalezení přijatelného řešení
- Porovnání nejlepší dosažené fitness za stanovený čas
- Srovnání nejlepších nalezených fitness v čase
- Vykreslení regresní křivky

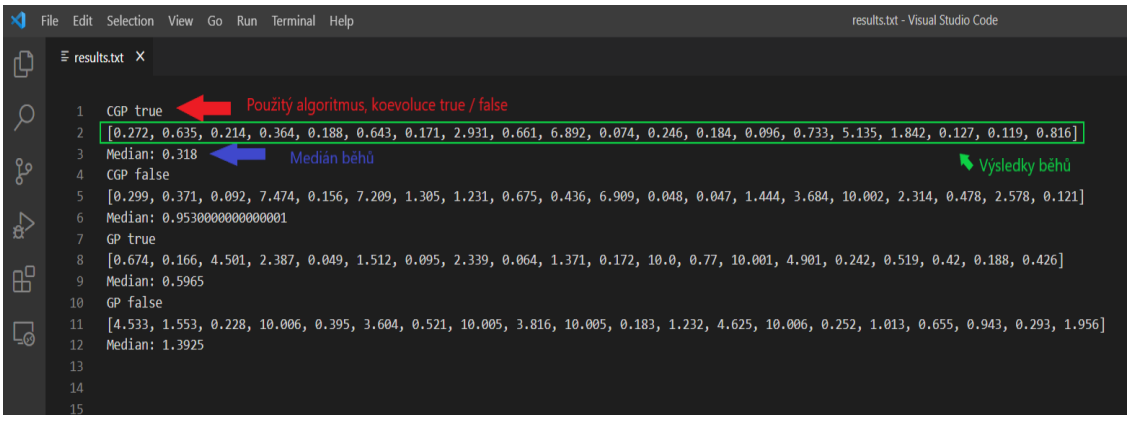
Třída *Tester* je hlavním bodem testování, ve které dochází k instanciaci a spouštění jednotlivých tříd. Experimenty probíhají následujícím způsobem: Nastaví se maximální povolený čas běhu, předmět porovnávání (rychlost aj.), fitness funkce (MAE, $MSE\ldots$), počet běhů a vstupní data.

Pro porovnání například rychlosti nalezení dostatečně kvalitního řešení[6] postupuje program následujícím způsobem: Vytvoří soubor results.txt, do kterého bude ukládat výsledky. Pro každou variantu (CGP a GP, s koevolucí i bez ní) spustí několikrát (dáno parametrem *runs*) symbolickou regresi. Jakmile je nalezeno přijatelné řešení, program přidá dobu výpočtu do seznamu. V případě, že nedojde k získání řešení požadované kvality včas, uloží se rovněž doba běhu, v tomto případě se bude jednat o maximální povolený čas. Nakonec se zapíší po dokončení všech běhů do souboru všechny časy a jejich mediány, a to pro každou variantu, viz obrázek 5.4.

Obdobným způsobem lze porovnávat nejlepší nalezené fitness, kdy program (nějaká ze zkoumaných variant) běží po dobu stanovenou maximálním povoleným časem a poté eviduje nikoli svoji dobu běhu, ale nejlepší dosaženou fitness. V případě koevoluce se jedná o nejlepší nalezenou objektivní fitness, viz sekce 5.2.6. Pokud je nalezeno přesné řešení bez jakékoli odchylky (s ohledem na zaokrouhlovací chyby, jak je popsáno v sekci 5.1.4), je zaznamenána 0, jakožto nulová odchylka (tedy maximálně přesné řešení).

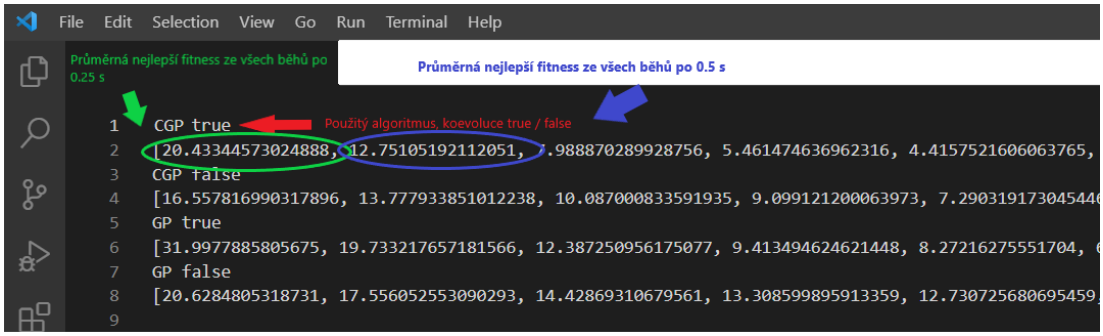
V případě porovnávání fitness v čase dochází k průměrování. Evoluční algoritmus zaznamenává každých 0.25 sekund nejlepší nalezenou fitness a na konci běhu tato data přičte do proměnné (do seznamu evidující fitness každých 0.25 sekund). Tato proměnná se nakonec (po dokončení všech běhů) zprůměruje. Teprve poté dojde k zapsání do souboru s výsledky, přičemž tyto výsledky mají mírně odlišný formát, viz obrázek 5.5.

Vykreslování regresní křivky funguje na jiném principu. Spustí se jeden běh s jednou uživatelem zvolenou strategii, přičemž uživatel rovněž musí upřesnit maximální dobu běhu programu, fitness funkci aj. Po ukončení výpočtu dojde volitelně k vykreslení (vykreslí se nejlepší nalezené kandidátní řešení), které má na starosti třída *Plot*. Nakonec je do souboru zapsán předpis této regresní křivky (společně s dobou běhu a nejlepší nalezenou fitness).



Obrázek 5.4: Soubor results.txt obsahující porovnávání rychlosti

[6] Obvykle se jedná o nějakou povolenou odchylku, například $e = 0.03$



Obrázek 5.5: Soubor results.txt obsahující porovnávání fitness v čase

Kapitola 6

Experimenty a vyhodnocení

V kapitole 5.6 bylo popsáno, na jakém principu fungují experimenty a co vše lze měřit. Pro takovéto výpočty bude potřeba:

- Získat vstupní data
  - Benchmarkové úlohy, ve kterých trénovací vektory přesně odpovídající nějaké funkci
  - Dataset z reálného světa
- Určit řídicí parametry

Na problému z reálného světa se bude hlouběji ověřovat funkčnost programu.

6.1 Použité funkce

První část umělých datasetů byla vytvořena na základě [18]. Jedná se o soubor 5 funkcí, které jsou vhodné pro testování symbolické regrese. Jejich předpisy (kde i značí krok v intervalu, přičemž je stejný pro každou proměnnou, pokud není uvedeno jinak) jsou:

$$F1 : f(x) = x^2 - x^3, x \in <-10, 10>, i = 0.1 \tag{6.1}$$
$$F2 : f(x) = e^{|x|}\sin(x), x \in <-10, 10>, i = 0.1 \tag{6.2}$$
$$F3 : f(x) = x^2 e^{\sin(x)} + x + \sin(\frac{\pi}{x^3}), x \in <-10, 10>, i = 0.1 \tag{6.3}$$
$$F4 : f(x) = e^{-x}x^3\sin(x)\cos(x)(\sin^2(x)\cos(x) - 1), x \in <-0, 10>, i = 0.05 \tag{6.4}$$
$$F5 : f(x) = \frac{10}{(x-3)^2 + 5}, x \in <-2, 8>, i = 0.05 \tag{6.5}$$

Následující funkce převzaté z [19] obsahují mnohem větší množství datových bodů:

$$F6 : f(x) = x^4 + x^3 + x^2 + x, x \in <-1, 1>, i = 0.02 \tag{6.6}$$
$$F7 : f(x_1, x_2, x_3, x_4, x_5) = 6.87 + 11\cos(7.23x_1^3), x_n \in <-50, 10>, i = 9.5 \tag{6.7}$$
$$F8 : f(x_1, x_2) = (x_2 - 5)F4(x_1), x_n \in <-0.5, 10.5>, i = (0.05, 0.5) \tag{6.8}$$
$$F9 : f(x_1, x_2, x_3, x_4, x_5) = \frac{10}{5 + \sum_{i=1}^{5}(x_i - 3)^2}, x_n \in <-0.25, 6.35>, i = 1.255 \tag{6.9}$$

6.2 Hledání vhodných řídicích parametrů

Velmi důležitou součástí symbolické regrese je nalezení co nejkvalitnějších řídicích parametrů. Typickým příkladem je třeba maximální hloubka stromu d u GP. Pokud se budeme pokoušet najít jednoduchou funkci F1 (viz 6.1), je vhodné, aby maximální hloubka stromu byla poměrně malá. Parametr d nastavený na nepřiměřeně velkou hodnotu by mohl vést ke výraznému zpomalení výpočtů, spoustu neaktivních uzlů (viz problém bloat v sekci 2.7.5). Malá povolená hloubka by tímto problémem příliš netrpěla a řešení by bylo nalezeno rychleji.

Na druhou stranu, pokud bychom chtěli symbolickou regresí najít předpis funkce F4 na základě trénovacích vektorů, nemusela by tato malá povolená hloubka vůbec stačit na nalezení dostatečně přesného řešení. Z toho plyne, že dokonalé řídicí parametry pro jednu funkci můžou být naprosto nedostatečné pro nějakou jinou. Nelze najít parametry, které by byly perfektní pro všechny datasety. Z tohoto důvodu jsem se snažil o kompromis, který by poskytoval přijatelné výsledky pro všechny funkce.

Parametry jsem nastavoval na základě experimentů s prvními 5 funkcemi (F1-F5). Při jejich volbě u CGP (nr, $nc\ldots$) jsem se do značné míry inspiroval v práci [18]. Aplikoval jsem řídicí parametry na základě této práce a poté jsem je mírně poupravil, opět na základě experimentů.

U GP jsem nevyužil žádný zdroj, co se volby parametrů pro tyto funkce týče. Zkoušel jsem různě měnit parametry a porovnával jsem výsledky. GP pro F1-3 pracovalo vždy poměrně přijatelně, ale u F4-5 byly výsledky mnohem horší. Zatímco v případě F5 nedocházelo k rapidnímu zrychlení symbolické regrese na základě změny parametrů, u F4 byly rozdíly obrovské. Snažil jsem se tedy nastavit parametry tak, aby pro F4 byla symbolická regrese co nejrychlejší nehledě na ostatní funkce (F1-3, F5). U prvních tří funkcí byl dopad řídicích parametrů úplně opačný, jak u té poslední. Pokud nedošlo ke zvolení na první pohled nefunkčního nastavení, probíhala regrese rychle, přičemž vliv parametrů byl malý (podstatně menší než u F4).

Při hledání co nejvhodnějších řídicích parametrů jsem používal pouze funkci MAE. Následující sekce popisují finální podobu nastavení. Význam jednotlivých parametrů popisují sekce 5.3 a 5.4.

6.2.1 Řídicí parametry CGP

Parametr POP_SIZE byl nastaven na hodnotu 12. Další parametry: $nr = 1$, $nc = 32$, $lback = nc$, $maxMutationCount = 8$

6.2.2 Řídicí parametry GP

Zde bylo nastavení určeno následujícím způsobem: $functionRate = 65\%$, $POP\_SIZE = 35$, $d = 7$, dále $varProb = 75\%$ a $mutationRate = 75\%$.

6.2.3 Řídicí parametry koevoluce

I zde jsem částečně čerpal z [18]. Počet trenérů byl nastavena na 8, velikost prediktoru na 15 % počtu trénovacích vektorů (max. 150), počet prediktorů v populaci prediktorů na 100, parametr $generationsToAdapt$ na 200. Pravděpodobnost mutace 1 genu prediktoru byla určena jako 10%.

6.3 Porovnávání

Při implementaci, ověřování korektnosti a testování programu jsem velmi často pracoval s [18]. První testování tedy bude směřovat na úlohy podobného typu. Autoři citované práce využili k testování zařízení s jiným výpočetním výkonem než já, a je tedy evidentní, že doby běhů programu budou dost rozdílné. Jiný byl i přístup k ohodnocování jedinců. Ve zmíněné práci byl použit přístup počítání skóre za každý trénovací vektor, který se bude nacházet v rámci tolerované chyby. Tento postup popisuje následující rovnice:

$$\begin{aligned} f(s) &= \sum_{j=1}^{n} g(y(j)), \text{ přičemž} \\ g(y(j)) &= \begin{cases} 0 & \text{pokud } |y(j) - t(j)| \geq \epsilon \\ 1 & \text{pokud } |y(j) - t(j)| < \epsilon \end{cases} \end{aligned} \tag{6.10}$$

kde ϵ je povolená odchylka stanovená uživatelem. [18]

Odlišné způsoby ohodnocování, rozdíly ve výkonosti a jiné rozdíly (rekurzivní vyhodnocování) vedou k tomu, že nemůžu porovnávat moje výsledky a jejich (co se týče kvality nalezených řešení či doby běhu programu). Nicméně, na základě výsledků této práce bych měl očekávat 2 × až 5.5× rychlejší nalezení řešení s koevolucí oproti obdobné strategii bez ní.

6.4 Testování

6.4.1 Testování rychlosti algoritmů

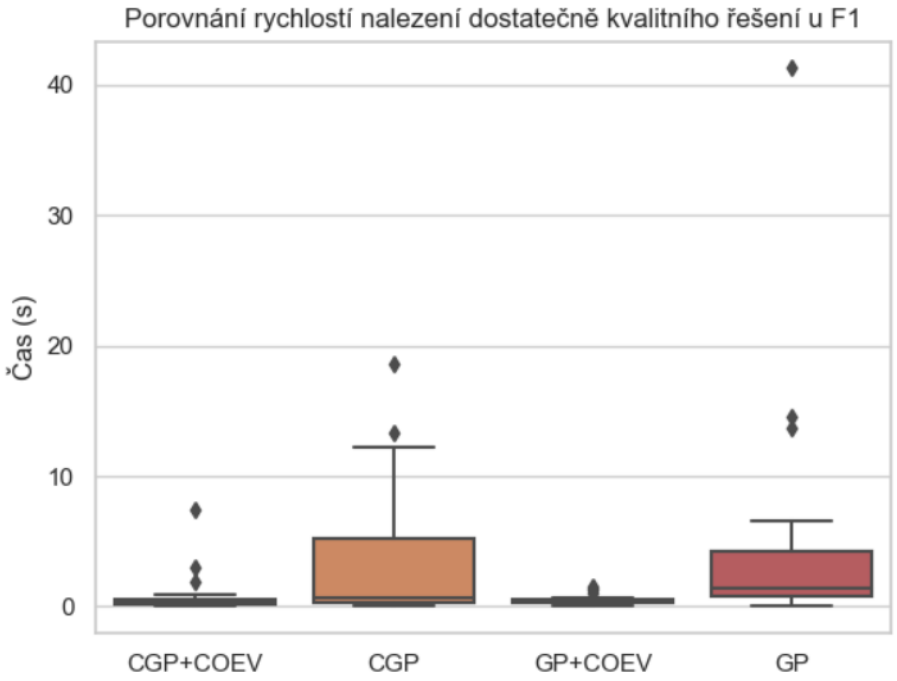
Testování rychlosti spočívá v tom, že nechám běžet algoritmus tak dlouho, dokud nedojde k nalezení řešení požadované kvality (například $e = 0.3$). Spustím vícero běhů (konkrétně 30, pokud není uvedeno jinak) a vytvořím graf, který prezentuje výsledky. Jednotlivé běhy mají vlastní časový limit, po jehož překročení běh skončí.

Jako první jsem otestoval rychlost zkoumaných variant genetického programování na úloze F1. Na obrázku 6.1 jde vidět, že CGP a GP s koevolucí dosahují srovnatelných výsledků. Tyto algoritmy bez koevoluce však nejsou v obou případech tak rychlé. Časový limit byl nastaven na 60 s, všechny běhy tak byly dokončeny včas nalezením řešení požadované kvality.

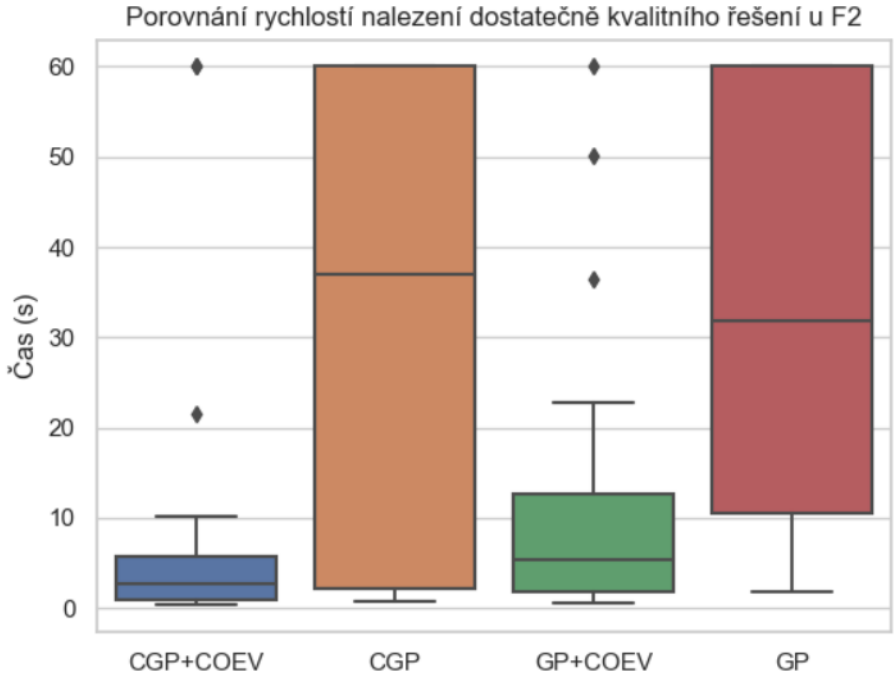
U takto jednoduché funkce dochází k nalezení řešení velmi rychle. Další grafy poukazují výsledky na mírně složitějších funkcích, které však také nejsou extra komplikované. Obrázek 6.2 poukazuje na výsledky stejného měření, tentokrát ale na úloze F2.

I zde byl časový limit nastaven na 60 sekund. Jak je vidět na obrázku 6.2, v tomto případě již docházelo k ukončení z časových důvodů, což lze odvodit z přítomnosti běhů, které běžely právě 60 s. Na tomto grafu je rozdíl mezi variantou s koevolucí a bez ní velmi výrazný. Když porovnáme CGP a GP jako takové, zjistíme, že si CGP vedlo o něco lépe. Je třeba ale podotknout, že rozdíly mezi CGP a GP (nehledě na koevoluci) jsou do značné míry ovlivněny volbou řídicích parametrů, viz sekce 6.2.1 a 7. Rozdíly mezi nějakou variantou bez koevoluce a tím samým algoritmem s koevolucí má však velkou vypovídající hodnotu, jelikož je nastavení společné pro obě varianty.

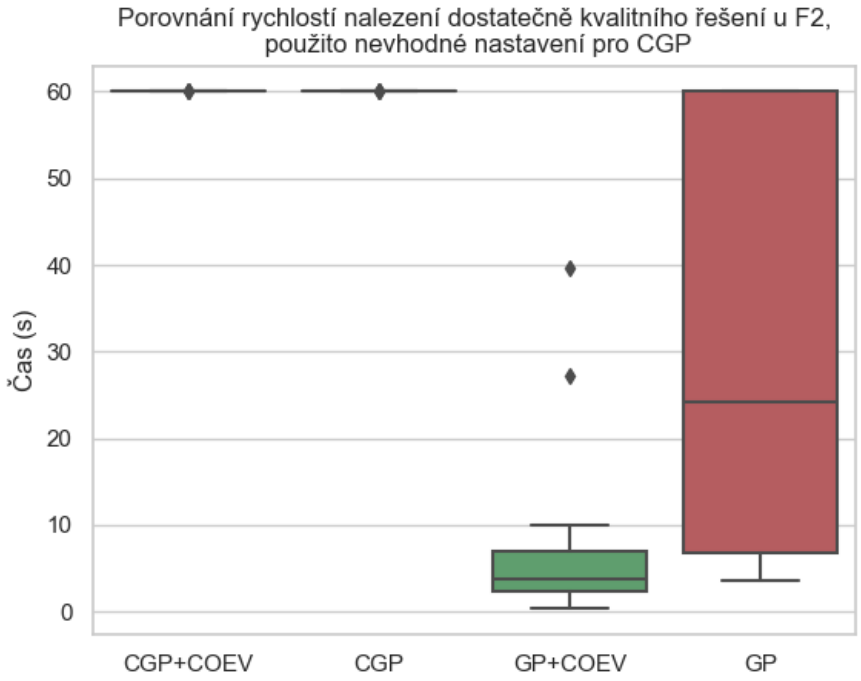
Abych demonstroval vliv řídicích parametrů, spustím celou simulaci znovu, tentokrát ale zcela záměrně s nevhodným řídicím parametrem $lback = 1$ v případě CGP. Graf 6.3 ukazuje velmi špatné výsledky u CGP za použití na první pohled validní změny nastavení.



Obrázek 6.1: Porovnání rychlosti u F1, fitness funkce: MAE (viz rovnice 5.1)



Obrázek 6.2: Porovnání rychlosti u F2, fitness funkce: MAE

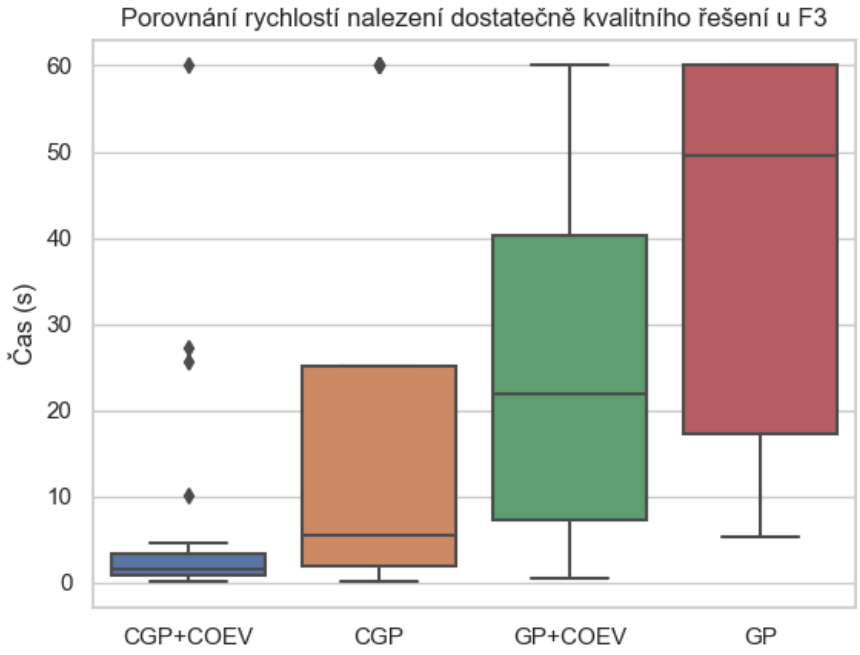


Obrázek 6.3: Porovnání rychlosti u F2, nevalidní CGP nastavení, funkce: MAE

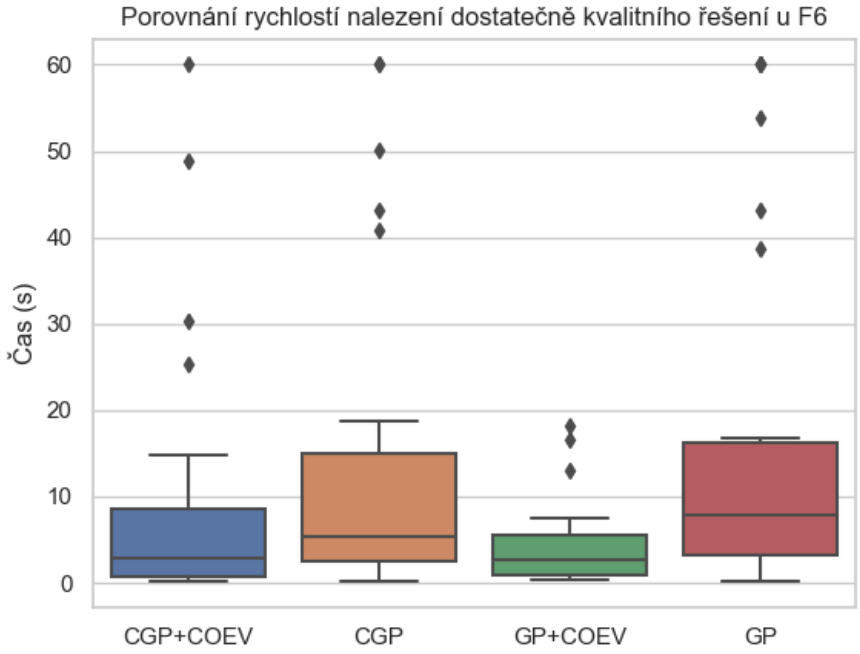
Změny parametrů jsem poté vrátil zpět. Při testování rychlosti jsem u funkce F3 dostal výsledky znázorněné obrázkem 6.4. U této úlohy se projevilo CGP jako mnohem lepší než GP. Nicméně pro každou variantu minimálně 1 běh nedokázal nalézt řešení včas.

Graf 6.5 popisuje výsledky experimentování s F6, která spadá pod druhou sadu funkcí. V tomto případě se projevilo GP jako rychlejší než CGP, v případě použití koevoluce. Bez ní je sice nepatrně horší, nicméně se zde ukazuje, že na některé funkce může být použití GP vhodnější. Je možné, že k lepším výsledkům GP došlo v tomto případě z důvodu větší pravděpodobnosti použití proměnné oproti CGP (u GP jsou z důvodu nastavení častěji využívány operace se vstupem než u CGP).

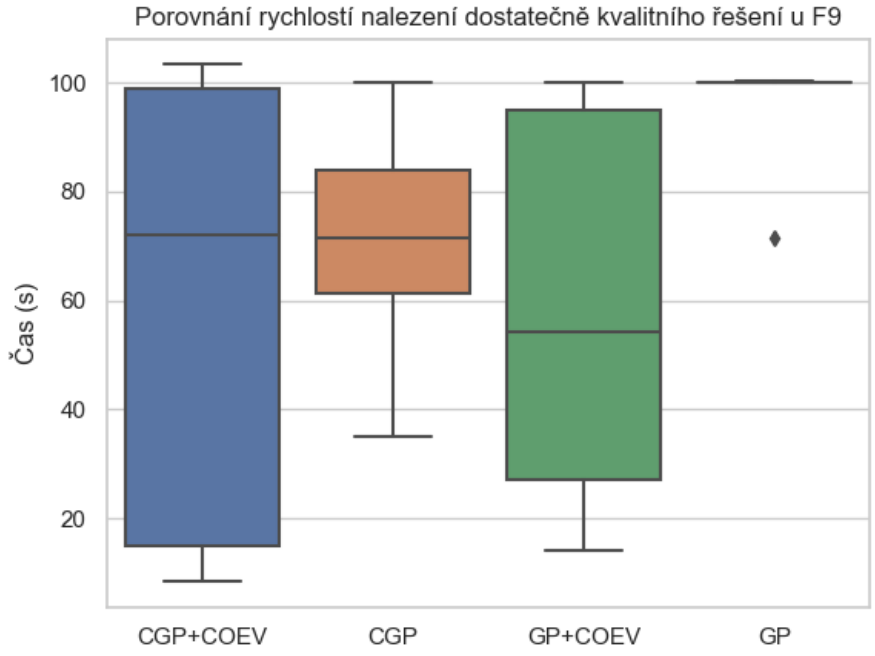
Nakonec této sekce uvádím srovnání variant u velmi komplikované funkce F9. Jak ukazuje graf 6.6, čas potřebný pro nalezení řešení je v tomto případě výrazně vyšší, než u jednodušších funkcí. Opět se koevoluce ukázala jako výrazný akcelerátor. Hlavně v případě genetického programování, kde algoritmus bez koevoluce našel řešení včas jen jednou. Tolerovaná chyba byla v tomto případě nastavena na $e = 0.105$. Tuto hodnotu jsem zvolil sám, na základě experimentů. Při nepatrně vyšší toleranci docházelo k nalezení konstanty během pár sekund, zatímco při o něco málo nižší povolené odchylce trvala iterace příliš dlouho.



Obrázek 6.4: Porovnání rychlosti u F3, funkce: MAE



Obrázek 6.5: Porovnání rychlosti u F6, funkce: MAE



Obrázek 6.6: Porovnání rychlosti u F9, funkce: MAE

6.4.2 Porovnávání nejlepší dosažené fitness

V této sekci je porovnávána nejlepší dosažená objektivní[1] fitness kandidátních řešení za předem stanovený čas běhu programu.

Jako první jsem testoval program na úloze F4. Evoluce byla spuštěna vždy po dobu 20 s, přičemž výsledkem byla nejlepší dosažená fitness. Pokud program našel řešení s nulovou odchylkou (respektive menší než 10^{-5}, z důvodů zmíněných v kapitole 5.1.4), byla zaznamenána hodnota 0 a běh byl ukončen předčasně.

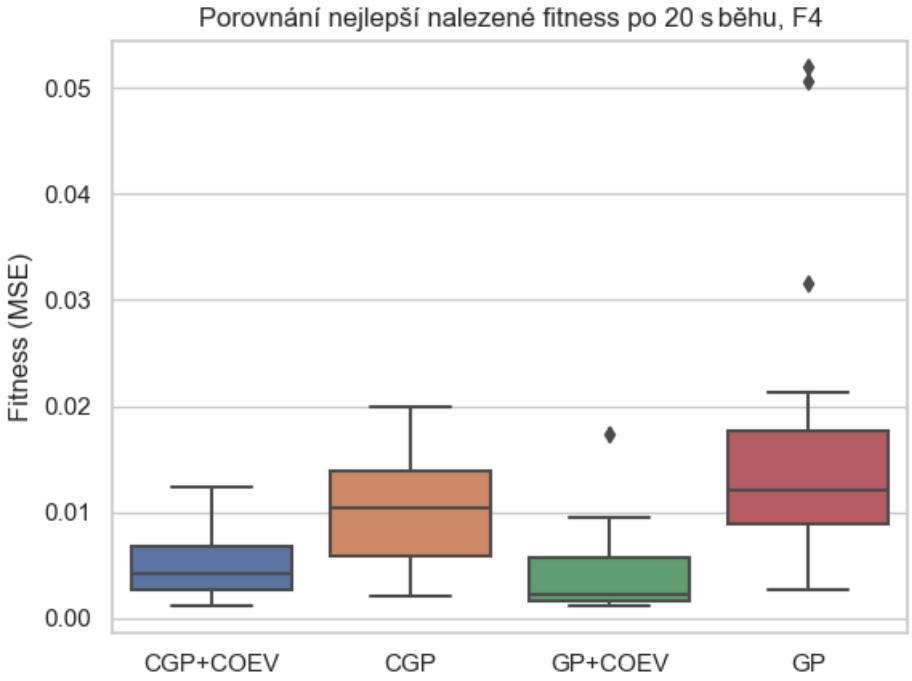
Pro co nejpřesnější výsledky bylo spuštěno několik výpočtů se stejnou strategii. Jednalo se o 30 běhů pro každou variantu. Oproti předchozí kapitole jsem zde zkusil použít srovnání fitness podle MSE, přičemž na konci kapitoly se nachází zhodnocení variant a vlivu koevoluce pro všechny fitness funkce.

Výsledky jsou vidět na obrázku 6.7. Je opět vidět, že vylepšení koevolucí vedlo k nalezení jedinců s vyšší fitness v kratším čase. GP s koevolucí zde mělo lepší výsledky než CGP s koevolucí, předpokládám že se projevil vliv řídicích parametrů testovaných v případě GP hlavně na F4 (kapitola 6.2).

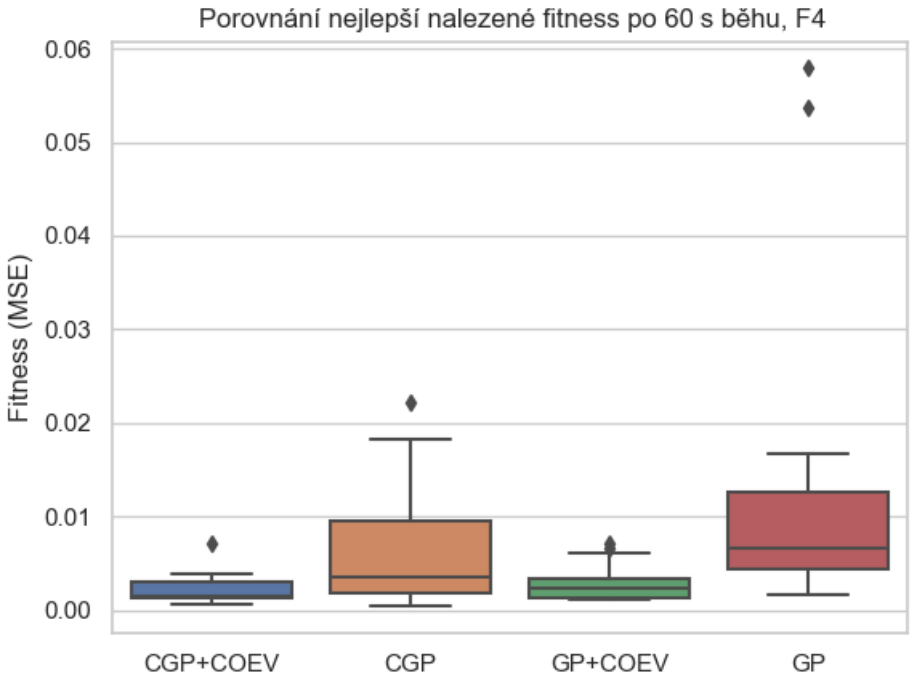
Pokud bych algoritmům poskytl větší čas k nalezení co nejkvalitnějšího řešení, dá se předpokládat, že výsledky všech variant budou lepší. Proto jsem spustil stejnou simulaci znovu, avšak s jednou výjimkou, a sice prodloužení doby běhu. Graf 6.8 ukazuje, k jakému zlepšení došlo, když byl poskytnutý čas ztrojnásoben na 60 s.

Je zjevné, že ke zlepšení došlo. A to v případě všech variant. Pokud bych čas ještě prodloužil, výsledky by byly ještě lepší. Může se stát, že algoritmus přestane konvergovat a zůstane uvázlý na nějakém lokálním extrému. Například najde konstantu, která má pouze

[1] V případě použití koevoluce se nejlepší dosažené subjektivní fitness nezaznamenává



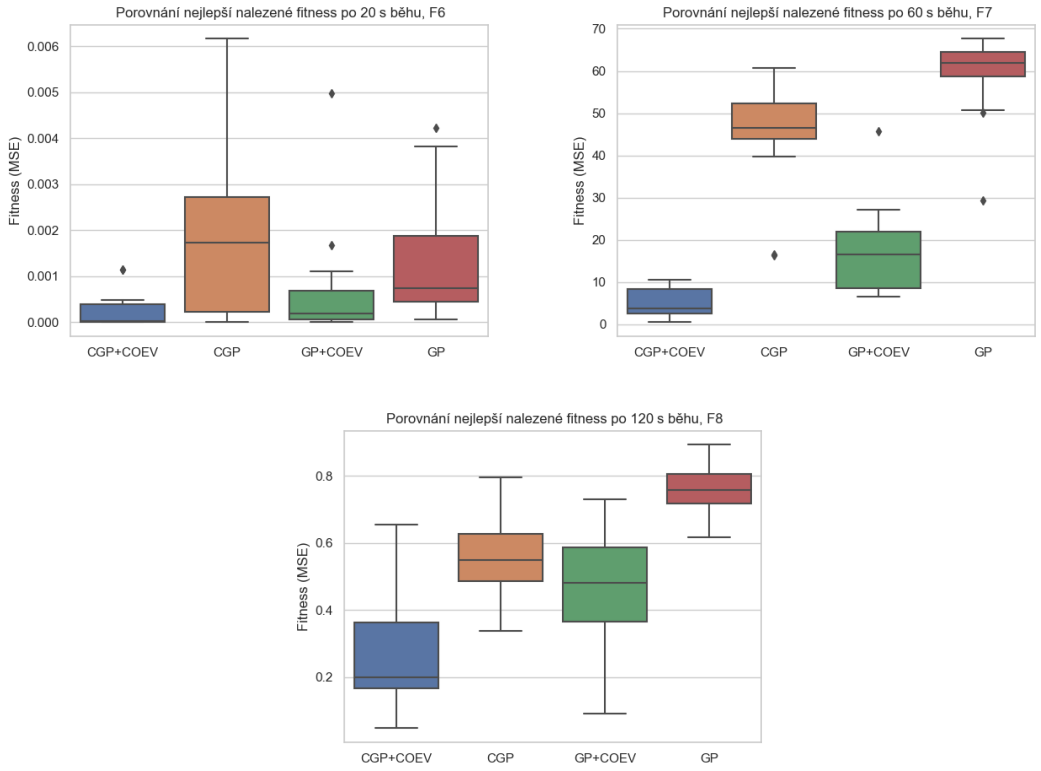
Obrázek 6.7: Porovnání nejlepších dosažených fitness po 20 s běhu



Obrázek 6.8: Porovnání nejlepších dosažených fitness po 60 s běhu

malou odchylku od hledané funkce, avšak ani po tisícovkách nových generací během evoluce nedojde k nalezení lepšího řešení.

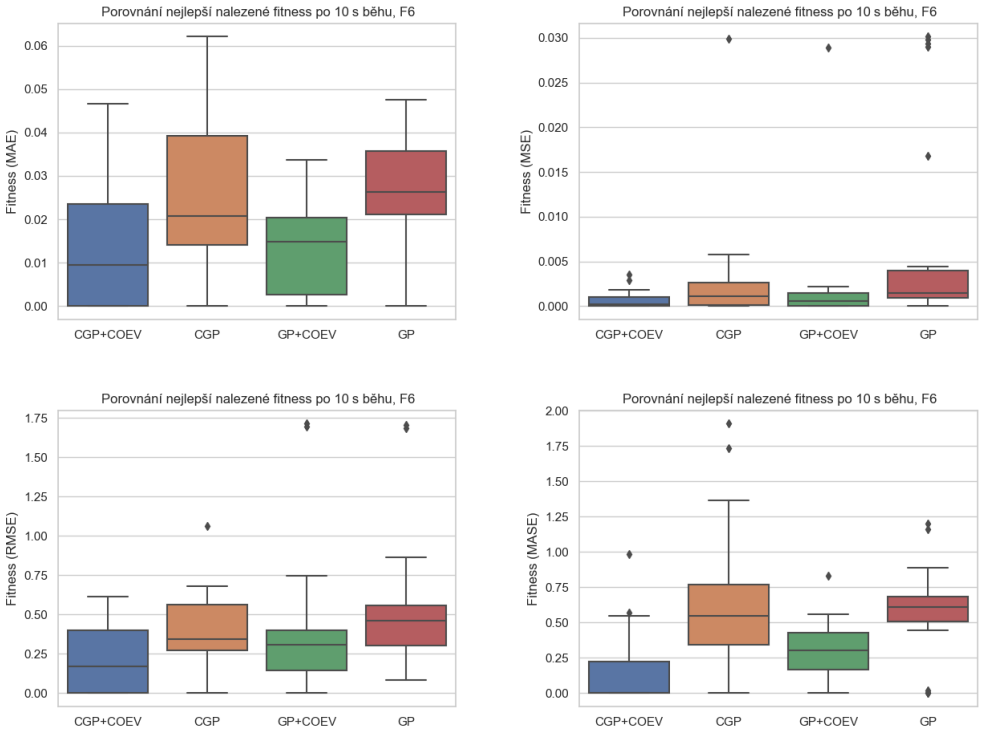
Následující grafy (obrázek 6.9) zobrazují výsledky zkoumání téhož faktoru, a to u funkcí F6, F7 a F9. Úloha F6 příliš nevyniká ve složitosti ani počtu datových vzorků. Oproti tomu funkce F7 je sice jednoduchá, avšak obsahuje obrovské množství trénovacích vektorů. Což vede k ještě rapidnější akceleraci výpočtu koevolucí. Úloha F9 opět obsahuje velké množství dat, v tomto případě se ale o jednoduchou funkci rozhodně nejedná. Vliv koevoluce je výraznější než v předchozích případech, avšak menší než u F7.



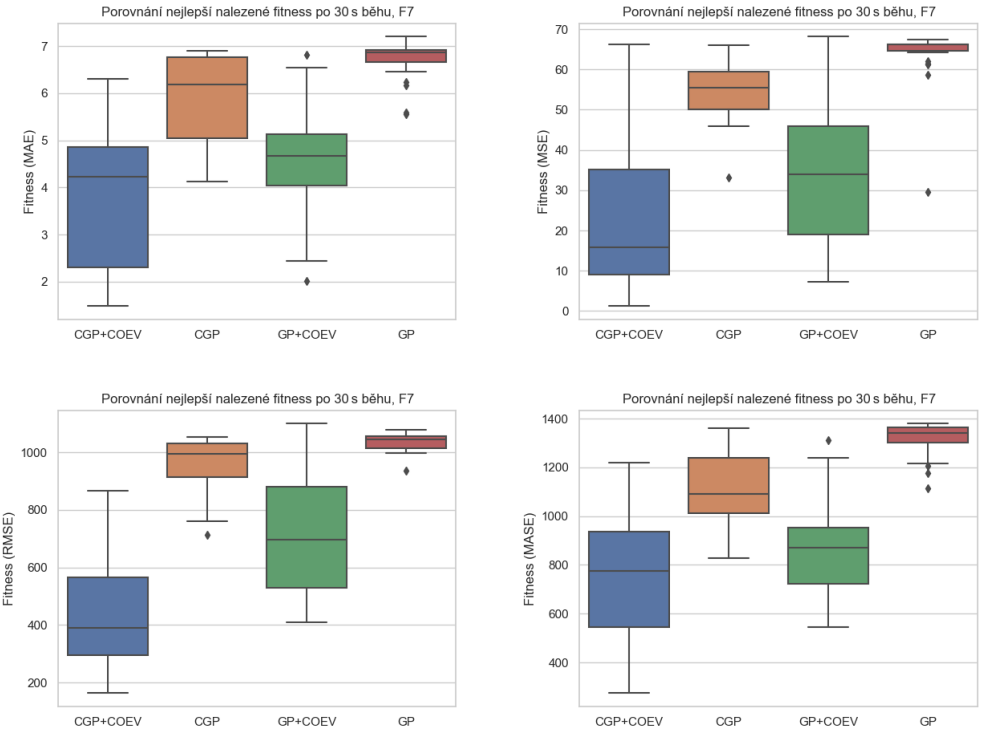
Obrázek 6.9: Porovnání nejlepších dosažených fitness za stanovený čas

Následující grafy popisují srovnání u všech zkoumaných fitness funkcí. Tato simulace byla spuštěna pro zjištění vlivu koevoluce skrze všechny zkoumané fitness funkce, rovněž byl porovnáván rozdíl mezi CGP a GP, pokud jde o různé zmíněné fitness funkce. Obrázek 6.10 znázorňuje stejnou simulaci (nejlepší nalezená fitness pro F6 po 10 s běhu), tentokrát pro všechny zkoumané fitness. Ve všech případech se CGP s koevolucí jevilo jako nejefektivnější. I GP s koevolucí bylo, vždy, na stejném místě, konkrétně druhém. CGP bez vylepšení koevolucí bylo buď mírně lepší, nebo srovnatelné jak GP bez koeovoluce.

U MSE se zdá, že je zde vliv koevoluce výrazně menší, než u ostatních fitness funkcí. To může ale být způsobeno tím, že je tato fitness pro danou úlohu mnohem vhodnější a potřebuje méně času pro nalezení vhodného řešení. Z toho důvodu jsem spustil obdobnou simulaci, tentokrát ale na funkci F7 (viz obrázek 6.11). Zde se však u MSE vliv koevoluce výrazně projevil. Soudě dle obou grafů (6.10, 6.11), vliv koevoluce je obecně velmi podobný pro všechny funkce fitness. U $RMSE$ byl vliv pro funkci F6 menší než například u $MASE$, nicméně v případě F7 to bylo právě naopak.



Obrázek 6.10: Nejlepší nalezená fitness za 10 s



Obrázek 6.11: Nejlepší nalezená fitness za 30 s

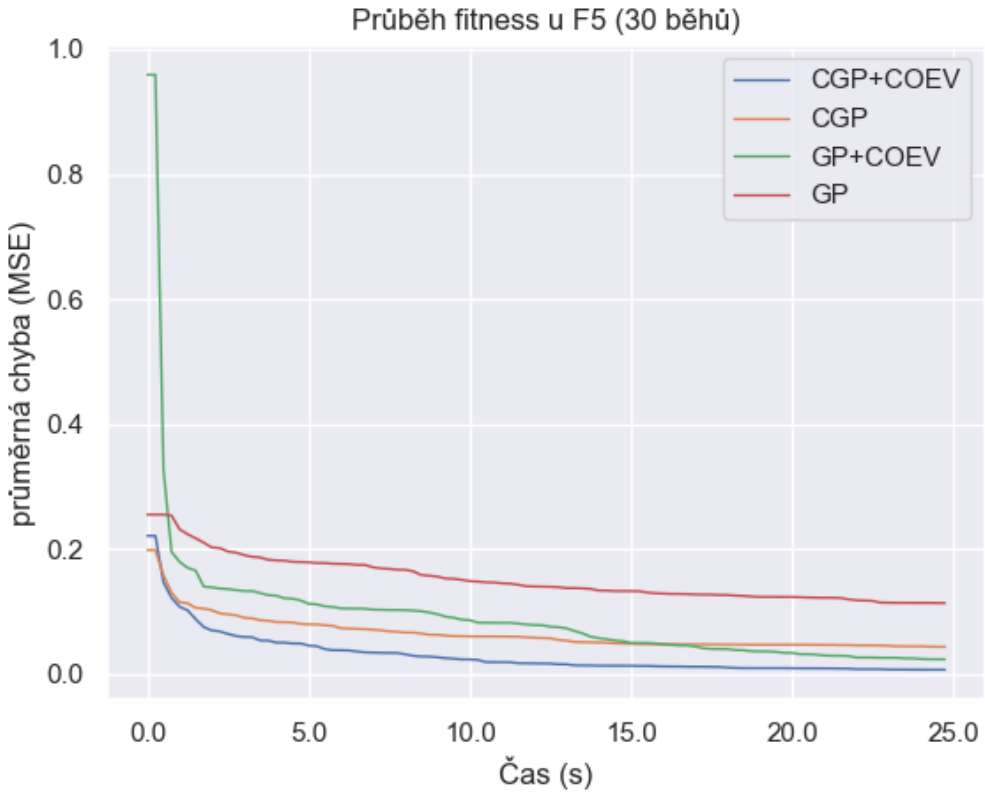
6.4.3 Porovnávání fitness v čase

V předchozí části 6.4.2 byla porovnávána nejlepší dosažená fitness za předem stanovený čas. S rostoucím množstvím poskytnutého čase se podle očekávání zlepšovaly i výsledky. Tato sekce poskytuje komplexnější informace o průběhu fitness v čase.

Následující grafy byly vytvořeny tímto způsobem: Spustil jsem několik běhů a každých 0.25s zaznamenal nejlepší dosaženou fitness. Závěrečné výsledky jsem zprůměroval. Zatímco u předešlých statistik byl velmi vypovídající medián, v tomto případě je přesnější průměr. Pokud bych testoval 2 varianty s tím, že bych pro každou spustil 20 běhů, mohlo by v případě nalezení řešení u nadpoloviční většiny běhů dojít k následující situaci: Srovnáváme 2 varianty. 11 běhů z 20 má nejlepší dosaženou fitness 0, takže medián celku je 0. V jednom případě dosahuje zbylých 9 běhů výrazně lepších výsledků než ve druhém. Oba mají však medián od určitého času na 0, což je zavádějící. Z tohoto důvodů jsem využil průměrování.

Jelikož inicializace počáteční populace zabere poměrně hodně času v případě, že daná úloha obsahuje velké množství trénovacích vektorů, byly hodnoty fitness před spuštěním evoluce nastaveny na stejnou hodnotu, jaká byla zaznamenána v rámci prvního zaevidování fitness hodnoty.

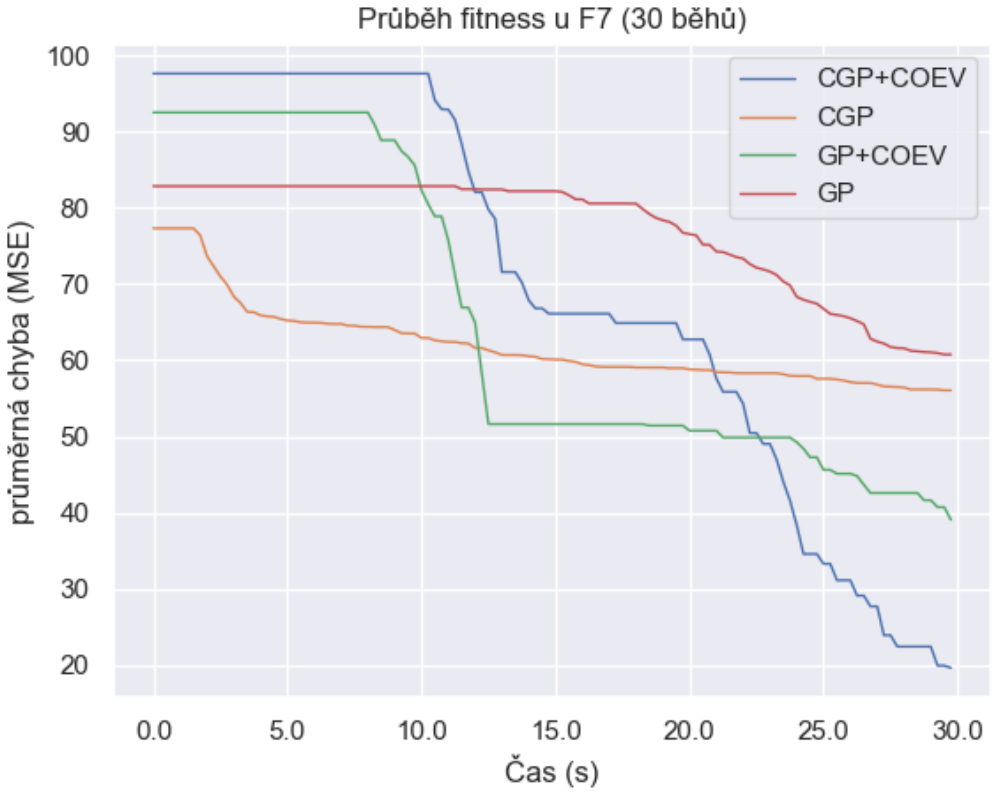
První graf (6.12) v této sekci poukazuje na průběh fitness pro jednotlivé varianty genetického programování v úloze F5.



Obrázek 6.12: Průběh nejlepší nalezené fitness v čase

Jak ukazuje obrázek 6.12, v případě F5 se jako nejúčinnější ukázalo CGP s koevolucí. Následuje GP s koevolucí, které ale na začátku měření neposkytovalo tak dobré výsledky, jako CGP bez koevoluce. Nejhorší výsledky opět poskytuje GP bez koevoluce.

V minulé sekci bylo prokázáno, že s narůstajícím počtem trénovacích vektorů roste i vliv koevoluce na kvalitu řešení nalezeného za stanovení časového limitu. Graf 6.13 ukazuje průběh fitness u jednoduché funkce se spoustou trénovacích vektorů.

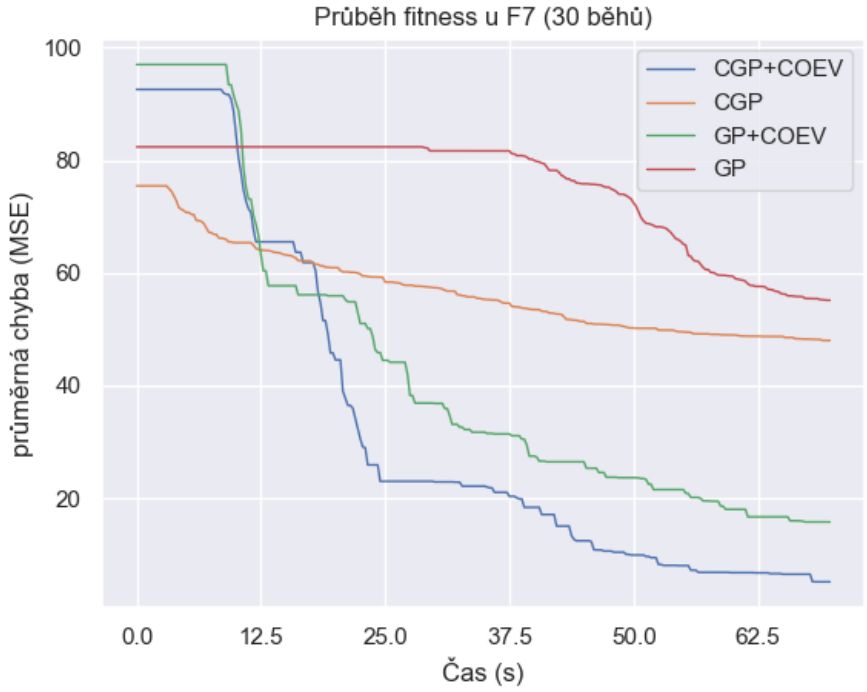


Obrázek 6.13: Průběh nejlepší nalezené fitness v čase

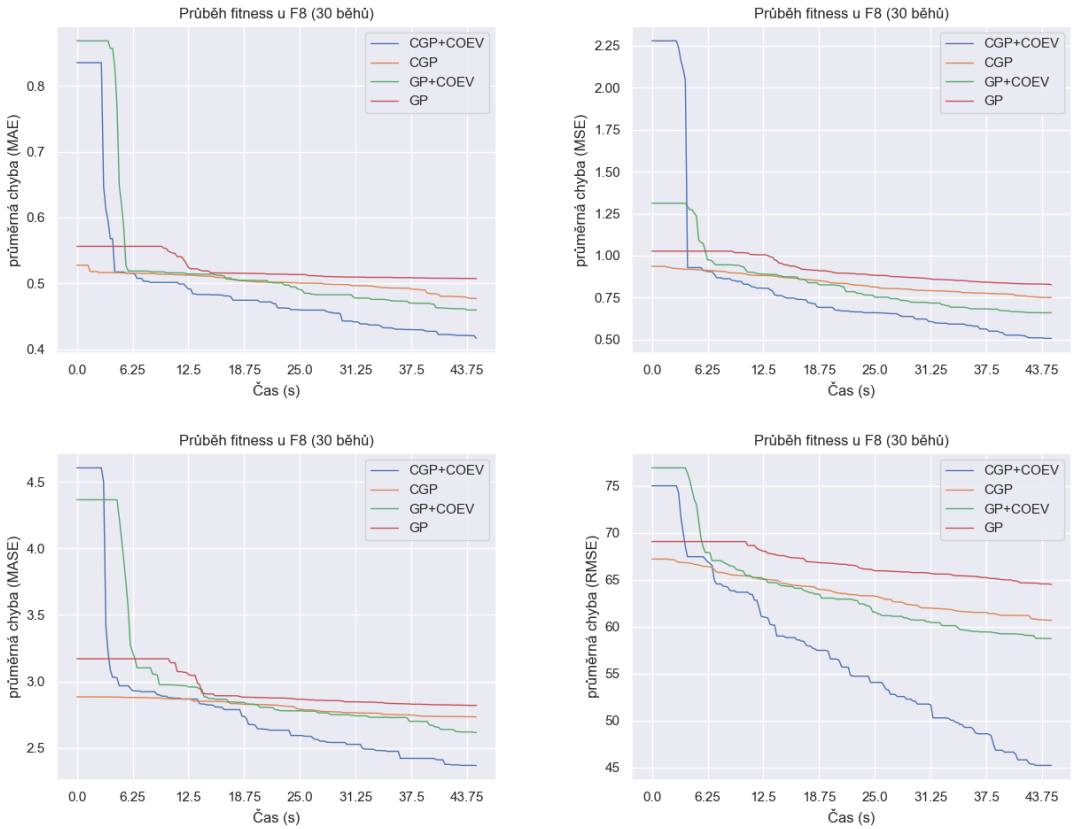
Zde nejprve došlo k počátečnímu zdržení z důvodu inicializace. Jakmile došlo k zahájení iterace, varianty s koevolucí rychle předehnaly ty bez koevoluce.

Abych ověřil vypovídající hodnotu 30 běhů, spustil jsem stejnou simulaci znovu, v tomto případě jsem navíc i prodloužil dobu běhu programu (obrázek 6.14). I v tomto případě je vidět obdobný charakter fitness v čase pro různé fitness funkce.

Další graf (6.15), znázorňující průběh fitness u komplikované F8, také obsahuje již použité porovnání všech fitness funkcí. Oproti očekávání zde byl u $RMSE$ velký rozdíl mezi CGP s koevolucí a zbytkem, ostatní grafy vypadají velmi podobně a v souladu s dosavadním měřením.



Obrázek 6.14: Průběh nejlepší nalezené fitness v čase

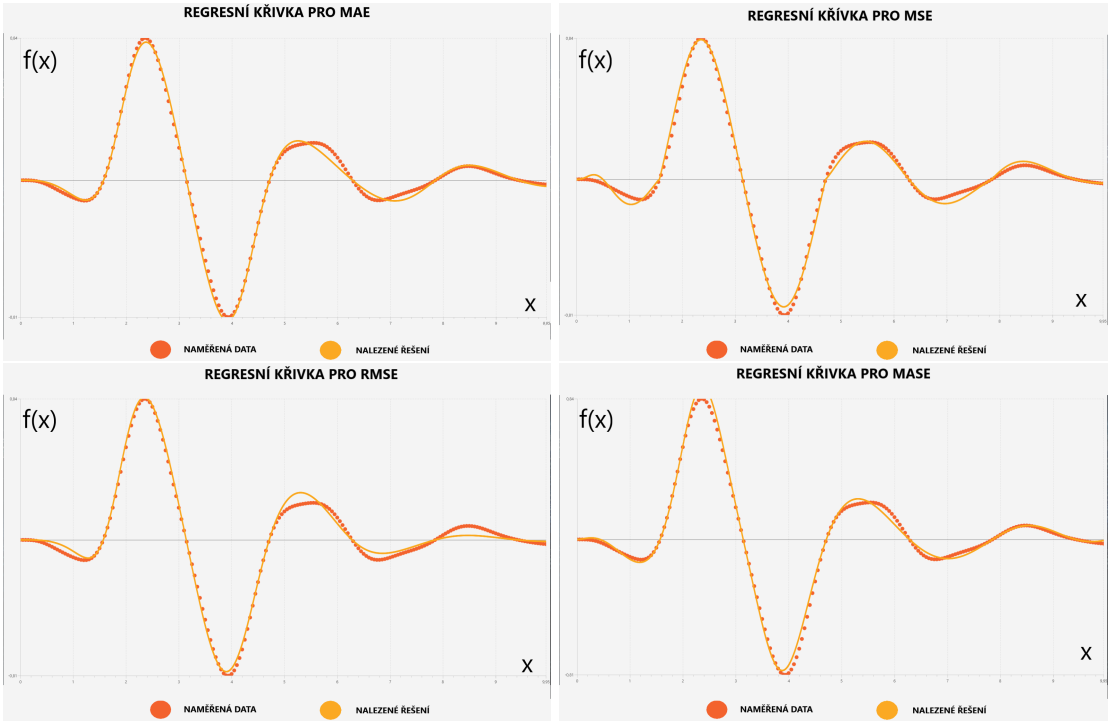


Obrázek 6.15: Průběh nejlepší nalezené fitness v čase

6.4.4 Porovnávání fitness funkcí

Dále je třeba ověřit, jestli dané fitness funkce jsou schopné dosáhnout skutečně přesného řešení. Kvalita řešení na základě hodnoty fitness může být těžko představitelná. Doposud jsem v této kapitole pracoval s poměrně abstraktní hodnotou fitness, nyní je třeba ověřit pomocí vizualizace, jak si symbolická regrese ve skutečnosti vede.

Pro úlohu F4 jsem spustil 30 běhů, každý měl k dispozici 30 sekund. Poté jsem nechal vykreslit trénovací vektory a k tomu regresní křivku za pomocí nejlepšího jedince ze všech běhů. Používal jsem CGP s koevolucí.



Obrázek 6.16: Porovnání fitness funkcí u F4

Jak ukazuje obrázek 6.16, všechny použité fitness funkce jsou schopné posloužit k nalezení poměrně přesného řešení. A to i při poměrně malém poskytnutém čase.

6.5 Úloha z reálného světa

Pro účely otestování symbolické regrese nad naměřenými neumělými daty jsem využil *Energy efficiency Data Set* [1, 13, 12]. Tato datová sada byla rovněž použita v [19]. Data vypovídají o požadavcích na vytápění budov v závislosti na parametrech budovy. Na rozdíl od [19] jsem nevyužíval pro regresi $RMSE$, ale MAE, které mi poskytovalo lepší výsledky.

Spustil jsem CGP s koevolucí (doposud se jevilo jako nejlepší ze zkoumaných variant) po dobu 1h 20m a obdržel jsem řešení s $MAE = 0.665697252480577$. Nejlepší nalezený jedinec byl velmi komplikovaný matematický vztah:

$$
\begin{aligned}
y = & ((((cos((cos(x2) * x1)) + ((x6 + x4) + (x6 + x4))) + x6) - log(x3)) \\
& - ((log(((cos((cos(x2) * (log(x0) - ((x1 + x3) * log(x0))))) + log(x0))/((cos((cos(x2) \\
& * x1)) + ((x6 + x4) + (x6 + x4))) * (cos((cos(x2) * x1)) + ((x6 + x4) + (x6 + x4)))))) \\
& - ((cos((cos(x2) * x1)) + ((x6 + x4) + (x6 + x4))) + x6)) + (((((cos((cos(x2) \\
& * (log(x0) - ((x1 + x3) * log(x0))))) + log(x0))/((cos((cos(x2) * x1)) + ((x6 + x4) \\
& + (x6 + x4))) * (cos((cos(x2) * x1)) + ((x6 + x4) + (x6 + x4)))))/x6) \\
& - (x6 * ((x6 + x4) + (x6 + x4)))) + ((((cos((cos(x2) * (log(x0) \\
& - ((x1 + x3) * log(x0))))) + log(x0))/((cos((cos(x2) * x1)) \\
& + ((x6 + x4) + (x6 + x4))) * (cos((cos(x2) * x1)) + ((x6 + x4) \\
& + (x6 + x4)))))/x6) + (((cos((cos(x2) * (log(x0) - ((x1 + x3) \\
& * log(x0))))) + log(x0))/((cos((cos(x2) * x1)) + ((x6 + x4) + (x6 + x4))) \\
& * (cos((cos(x2) * x1)) + ((x6 + x4) + (x6 + x4)))))/x6)))))
\end{aligned}
\tag{6.11}
$$

Poznámka: je potřeba brát v potaz, že dělení nulou, přetečení proměnné typu *double* apod. vede k navrácení konstanty $c = 1$ (viz 5.3.7, 5.4.3). Tento výraz tedy nesmí být brán doslovně ($y = log(-x)$ neznamená $y = log(-x)$, ale $y = 1$). Pro intuitivní zhodnocení kvality jedince zde uvedu několik náhodně vybraných funkčních a jim odpovídajících predikovaných hodnot:

Index vektoru	$\boldsymbol{Y}$	$\hat{\boldsymbol{Y}}$
76	10.71	10.827917963128485
114	36.97	35.40823231312716
152	25.41	25.28861726765092
190	12.71	12.425557268539016
228	11.44	11.187816950848458
418	12.31	11.904804212139567
494	24.6	25.538381977823597
684	28.93	28.72034078568364
760	17.69	19.46229774160144

U rovnice 6.5 mě překvapilo, že se ve výsledném výrazu nenachází žádná konstanta. Ověřil jsem tedy CGP na vlastních úlohách zaměřených na konstanty. Zde byly tyto konstanty hojně využívány. V mé implementaci využívá CGP konstant mnohem méně než GP (nejspíš vlivem nastavení). Pro zajímavost zde vypíši nalezené řešení (v případě stejné úlohy) za použití GP s koevolucí a s fitness $MAE = 0.7875746282792604$:

$$
\begin{aligned}
y = & ((((x6 * sin(log((x3 - 4.4)))) * -16.0)/((cos(sin(x1)) + (((x4 + x3) \\
& - cos(x3))/((x2/x6)/sin(14))))) + (log(cos(cos(x2)))/cos(x4)))) + (((sin(x2) \\
& + (sin((x2 + 17.6)) + 14.2)) + ((((x4 + x6) - (-2.0/x4)) - 1.8) - (cos((x1 + x1)) \\
& - sin((x4/ - 12.8))))) - (sin((((x1/x4) + x0) * ((x3 * x5)/(x5 + x5)))) \\
& + (exp(((x7 - 9.2)/(x6 * x4))) + ((log(-16.0)/log(x6)) + (-4.8/cos(x4)))))))
\end{aligned}
\tag{6.12}
$$

Kapitola 7

Diskuze

Jak již bylo krátce zmíněno v 6.4.1 a 6.2, velmi důležitou součástí symbolické regrese a evolučních algoritmů obecně jsou řídicí parametry. Nastavení těchto parametrů může mít za důsledek, že nějaká varianta genetického programování bude velmi vhodná pro řešení např. F1-F3, zatímco bude kompletně nevhodná pro F4 a průměrně dobrá pro F5. Zcela jiné nastavení též varianty může mít za následek pravý opak. Je velmi obtížné mezi sebou srovnávat CGP a GP, právě kvůli volbě řídicích parametrů. Snažil jsem se, jak již bylo zmíněno, o co nejrovnější podmínky, volil jsem tedy takové parametry, aby byly co nejlepší jak pro CGP, tak pro GP.

Trochu jsem nastínil vliv nastavení již v sekci 6.4.1, kde při zvolení na první pohled vhodného řídicího parametru došlo k tak rapidnímu zhoršení dané varianty, že byla pro zadanou úlohu prakticky nepoužitelná.

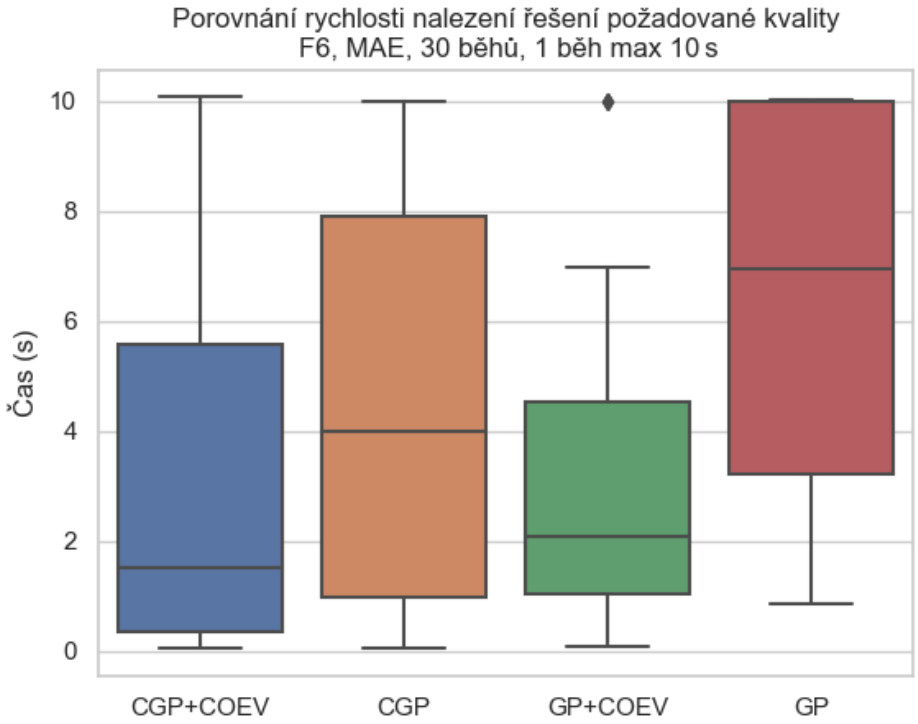
V této kapitole se budu důkladněji věnovat řídicím parametrům a jejich vlivu na průběh symbolické regrese. Dále budu řešit důležitost optimální volby takového nastavení, když jde o srovnávání jednotlivých variant genetického programování. Pro testování a porovnávání jsem zvolil funkci F6, konkrétně testování rychlosti nalezení dostatečně kvalitního řešení. Každá varianta bude mít k dispozici 30 běhů, každý běh bude trvat maximálně 10 sekund. Jako fitness funkci jsem zvolil MAE, za řešení požadované kvality považuji takové kandidátní řešení, které má chybu menší než $e = 0.03$.

Úplně první graf (obrázek 7.1), poukazuje na výsledky simulace za použití běžných řídicích parametrů. Nedošlo tak k žádné úpravě vzhledem ke kapitole 6. Opět, soudě dle mediánů, bylo nejvhodnější CGP s koevolucí, následované GP s koevolucí, poté CGP a nakonec GP.

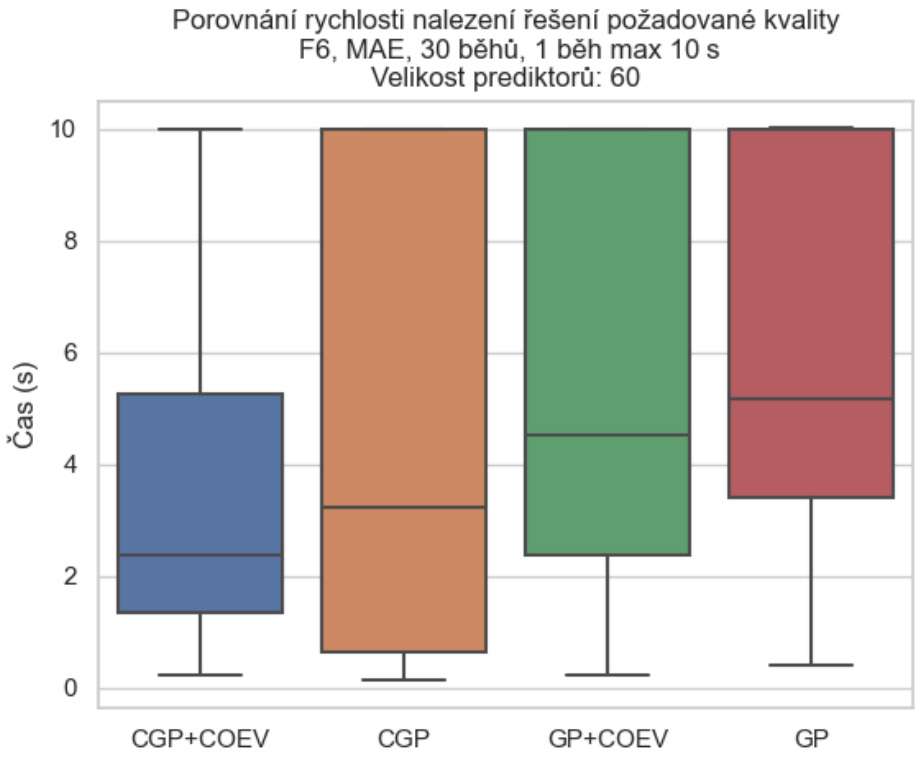
Ještě než začnu upravovat řídicí parametry jednotlivých variant genetického programování, zkusím otestovat změnu parametrů koevoluce. Nastavil jsem tedy velikost prediktorů na 60, přičemž úloha F6 má 100 trénovacích vektorů. Normálně by byla jejich velikost daná jako 15 % z počtu trénovacích vektorů, v této simulaci byl však tento parametr dočasně upraven.

Výsledky takového měření zobrazuje obrázek 7.2 (který byl porovnáván s obrázkem 7.1). Je vidět, že větší velikost prediktorů, kdy se subjektivní fitness přibližuje té objektivní a zmenšuje se tak vliv koevoluce, měla předpokládaný dopad na zpomalení variant vylepšených koevolucí. V obou případech došlo ke zhoršení (CGP s koevolucí oproti CGP bez koevoluce, GP s vylepšením koevolucí oproti GP bez tohoto vylepšení), mediánové hodnoty se téměř vyrovnaly.

Tento obrázek 7.2 poukazuje na fakt, že i vliv koevoluce samotné je velmi ovlivněn vhodným zvolením řídicích parametrů. Při neúměrně velkých prediktorech se pozitivní do-



Obrázek 7.1: Srovnání rychlosti za použití běžných parametrů spuštění

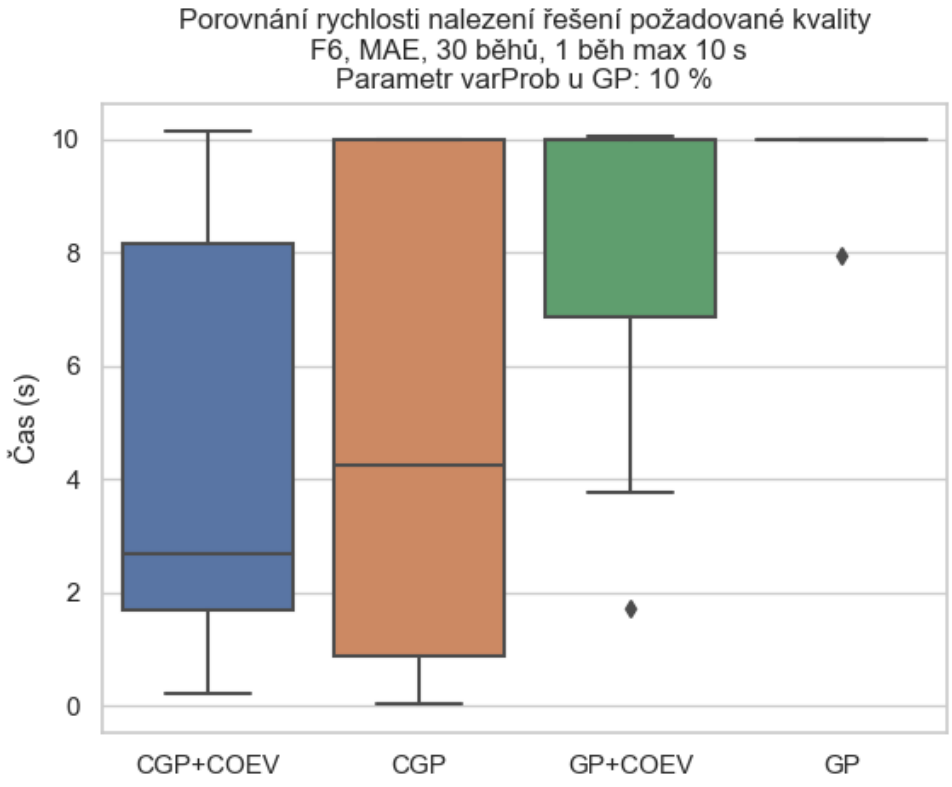


Obrázek 7.2: Srovnání rychlosti se zvýšenou velikostí prediktorů

pad této akcelerace výrazně snížil. Kdybych tyto prediktory naopak nepřiměřeně snížil, nemusely by pro vyjádření daného funkčního předpisu vůbec stačit.

Z obrázků 7.1 a 7.2 je vidět, že při změně velikosti prediktorů byly mírně ovlivněny i varianty bez koevoluce, ačkoli by být neměly. Vzhledem k nedeterministickému chování evolučních algoritmů jsou takovéto odchylky mezi stejnými simulacemi běžné. Nicméně je evidentní, že varianty s koevolucí (minimálně v případě GP) byly ovlivněny výrazně více než ty bez koevoluce.

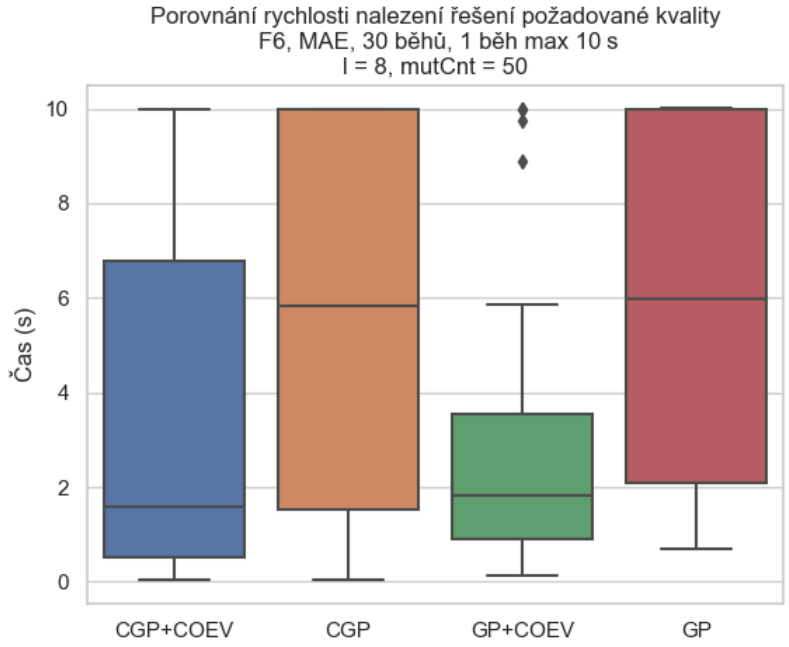
Nyní budu testovat (předchozí změny byly vráceny zpět) rozdíly mezi CGP a GP v případě přímé změny nastavení v rámci jednotlivých algoritmů. Parametr *varProb*, určující pravděpodobnost umístění proměnné do listu (připomínám, že doplněk je pravděpodobnost konstanty) je obvykle nastaven na 75 %. Ve funkci F6 se nachází pouze proměnné, jakožto terminály. Žádné konstanty. Tento atribut jsem snížil na 10 %. Podle očekávání měla tato změna devastující účinek na rychlost konvergence u GP (viz obrázek 7.3). Kdyby však funkce F6 obsahovala obrovské množství konstant, tato změna by naopak vedla ke zrychlení, protože by terminály byly často obsazovány konstantami, nikoli proměnnými.



Obrázek 7.3: Srovnání rychlosti se sníženou pravděpodobností použití proměnné u GP

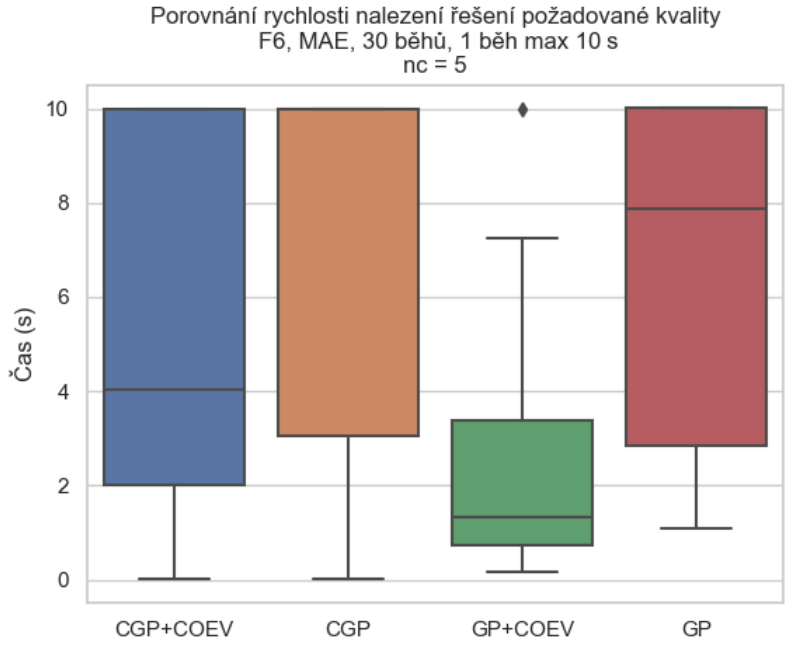
V následujícím porovnání (graf 7.4) jsem naopak pozměnil parametr u CGP, zatímco změny u GP byly vráceny zpět. Parametr $maxMutationCount$, udávající maximální množství mutací při mutování jednoho jedince, jsem zvýšil z 8 na 50. Dále také byl změněn parametr l z 32 na 8. CGP a GP se v tomto případě více vyrovnalo, rozdíly však nebyly na první pohled příliš patrné.

V rámci posledního experimentu v této sekci jsem opět vrátil všechno nastavení na původní hodnoty. Následně jsem změnil parametr nc na hodnotu 5 (z původních 32). Tato



Obrázek 7.4: Srovnání rychlosti s pozměněnými CGP parametry

hodnota je dostatečná pro nalezení řešení (je tedy možné nalézt předpis F6 s tímto počtem sloupců za použití jednoho řádku). Prostor pro nalezení funkce mnoha ekvivalentními předpisy se výrazně snížil. A s tím se i snížila rychlost CGP, jak ukazuje obrázek 7.5.



Obrázek 7.5: Srovnání rychlosti s pozměněnými CGP parametry

V této kapitole byl demonstrován vliv řídicích parametrů. Některé změny vedou ke zhoršení průběhu symbolické regrese, avšak kdyby probíhalo testování na jiné funkci, mohlo by naopak dojít ke zlepšení. CGP a GP se vzájemně porovnává obtížně, vždy je možné najít řídicí parametry, se kterými se libovolný algoritmus bude chovat lépe, než ten druhý. A na jiných funkcích se to může projevit právě obráceně. Potvrdilo se (jak bylo mnohokrát opakováno dříve), že hledání pokud možno co nejvhodnějšího nastavení je velmi důležitou součástí jak symbolické regrese samotné, tak i porovnávání jejích variant. Tím byla dokázána i nutnost hledání co nejlepších řídicích parametrů pro všechny varianty ještě před samotným porovnáváním.

S tím souvisí tzv. „No-free-lunch“ teorém. Nastavení, které by bylo maximálně vhodné pro všechny úlohy, najít nelze. [4]

Vypovídající hodnota algoritmu bez koevoluce a toho samého algoritmu s ní je však vysoká. Používají se stejné řídicí parametry, vliv koevoluce tedy je bezesporu velmi podstatný. Takovéto vylepšení bez debat vede ke zrychlení výpočtu a mnohdy i k nalezení kvalitnějšího řešení (když varianty bez koevoluce například uváznou v lokálním optimu, ze kterého se, na rozdíl od variant s koevolucí, nemusí dostat).

Kapitola 8

Závěr

Všechny zkoumané varianty genetického programování jsou schopné řešit úlohy symbolické regrese. Koevoluce vede ke zrychlení výpočtu jak u CGP, tak u GP. Její vliv na rychlost nalezení řešení a kvalitu nalezeného řešení se mění množstvím trénovacích vektorů a složitostí funkce. Vylepšení koevolucí má obdobný dopad u CGP i GP, přičemž u některých funkcí je více ovlivněno CGP, u jiných GP. Nedá se tedy říct, že by koevoluce ovlivňovala genetická programování rovnoměrně. Obecně vzato nicméně nedochází k výrazným rozdílům.

U symbolické regrese prováděné pomocí CGP a GP velmi záleží na počátečním nastavení, které výrazným způsobem ovlivňuje rychlost regrese a kvalitu nalezeného řešení. Je obtížné srovnávat mezi sebou CGP a GP, právě z důvodu zmíněných parametrů spuštění. Řídicí parametry můžou způsobit, že se GP bude jevit jako efektivnější v případě hledání předpisů jednoduchých funkcí a CGP zase v případě těch složitých. Přitom by to mohlo být naopak, kdyby byly řídicí parametry zvoleny jinak.

Snažil jsem se proto najít parametry, které budou na testovaných úlohách poskytovat pokud možno co nejlepší výsledky. Při takto zvoleném nastavení jsem došel k závěru, že v mojí implementaci je nejúčinnější CGP s koevolucí. Následuje GP s koevolucí. Poté CGP bez koevoluce a nakonec GP bez koevoluce, viz tabulka 8.1.

Pořadí	**Název**
1	CGP s koevolucí
2	GP s koevolucí
3	CGP bez koevoluce
4	GP bez koevoluce

Tabulka 8.1: Pořadí variant genetického programování

Vyzkoušel jsem i vícero fitness funkcí pro ohodnocování jedinců v symbolické regresi, všechny byly pro testované úlohy použitelné a ve většině případů se příliš nelišily (viz například sekce 6.4.4). Při řešení některých úloh nicméně docházelo k rozdílům.

Kdybych měl řešit neznámý problém pomocí symbolické regrese, zvolil bych na základě dosažených výsledků variantu CGP s koevolucí. Pokud by daná úloha obsahovala spoustu proměnných a nalezené řešení by nebylo dostatečně přesné, popřemýšlel bych o navýšení parametru *nc*. Pokud by mi program i přesto neposkytoval přijatelné výsledky, experimentoval bych s různými fitness funkcemi.

V budoucnu se nabízí příležitost zkoumat i další faktory, například v této práci nevyužité porovnání pomocí maximálního počtu generací. Zajímavé by bylo i srovnání strategii pro

vytváření nových generací (pro CGP, GP i prediktory fitness). I zpracovávání nevalidních výrazů a různé přístupy k této tématice by mohly být předmětem zkoumání.

Při psaní této práce a programování symbolické regrese jsem se seznámil s problematikou řešení regrese pomocí strojového učení. Naučil jsem se řešit regresi pomocí automatizovaných nástrojů a jsem schopen vyřešit takovéto problémy, aniž bych jim sám rozuměl (například k vztahům, které popisující závislosti v úloze z reálného světa, jsem se dopracoval dříve, než jsem si přečetl, co daná data vlastně znamenají). V budoucnu bych se mohl zabývat i porovnáním jiných variant GP (jiné než stromové reprezentace). Rovněž bych se rád dostal k implementování určité formy evoluční inteligence za využití koevoluce.

Literatura

[1] *Machine Learning Repository* [online]. [cit. 2022-04-26]. Dostupné z: http://archive.ics.uci.edu/ml/index.php.

[2] Chugh, A. *MAE, MSE, RMSE, Coefficient of Determination, Adjusted R Squared — Which Metric is Better?* [online]. [cit. 2022-03-21]. Dostupné z: https://medium.com/analytics-vidhya/mae-mse-rmse-coefficient-of-determination-adjusted-r-squared-which-metric-is-better-cd0326a5697e.

[3] Hyndman, R. Another Look at Forecast Accuracy Metrics for Intermittent Demand. *Foresight: The International Journal of Applied Forecasting*. International Institute of Forecasters. Leden 2006, sv. 4, s. 43–46. ISSN 1555-9068.

[4] Kenneth De Jong. Generalized Evolutionary Algorithms. In: Grzegorz Rozenberg, J. N. K., ed. *Handbook of Natural Computing*. Springer-Verlag Berlin Heidelberg, 2012, s. 626–634. ISBN 978-3-540-92910-9. Dostupné z: https://doi.org/10.1007/978-3-540-92910-9_20.

[5] Koza, J. A genetic approach to the truck backer upper problem and the inter-twined spiral problem. In: *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*. Baltimore, MD, USA: IEEE, 1992, sv. 4, s. 310–318. ISBN 0-7803-0559-0. Dostupné z: https://doi.org/10.1109/IJCNN.1992.227324.

[6] Leandro Nunes de Castro. *Fundamentals of Natural Computing*. New York: Chapman and Hall/CRC, 2006. 696 s. ISBN 9781584886433.

[7] Miller, J. F. Cartesian Genetic Programming. In: Miller, J. F., ed. *Cartesian Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, s. 17–34. DOI: 10.1007/978-3-642-17310-3_2. ISBN 978-3-642-17310-3. Dostupné z: https://doi.org/10.1007/978-3-642-17310-3_2.

[8] Popovici, E., Bucci, A., Wiegand, R. P. a De Jong, E. D. Coevolutionary Principles. In: Rozenberg, G., Bäck, T. a Kok, J. N., ed. *Handbook of Natural Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 987–1033. DOI: 10.1007/978-3-540-92910-9_31. ISBN 978-3-540-92910-9. Dostupné z: https://doi.org/10.1007/978-3-540-92910-9_31.

[9] Schmidt, M. a Lipson, H. Distilling Free-Form Natural Laws from Experimental Data. *Science (New York, N.Y.)*. 2009, sv. 324, č. 5923, s. 81–85. DOI: 10.1126/science.1165893. ISSN 0036-8075. Dostupné z: https://www.science.org/doi/abs/10.1126/science.1165893.

[10] Schmidt, M. D. a Lipson, H. Coevolution of Fitness Predictors. *IEEE Transactions on Evolutionary Computation*. 2008, sv. 12, č. 6, s. 736–749. DOI: 10.1109/TEVC.2008.919006. ISSN 1089-778X.

[11] Sekanina, L., Vašíček, Z., Růžička, R., Bidlo, M., Jaroš, J. et al. *Evoluční hardware: Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Praha, CZ: Academia, 2009. 328 s. Edice Gerstner. ISBN 978-80-200-1729-1. Dostupné z: https://www.fit.vut.cz/research/publication/9123.

[12] Tsanas, A. *Accurate telemonitoring of Parkinson's disease symptom severity using nonlinear speech signal processing and statistical machine learning*. 2016. Disertační práce. Oxford University, UK.

[13] Tsanas, A. a Xifara, A. Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools. *Energy and Buildings*. 2012, sv. 49, s. 560–567. DOI: https://doi.org/10.1016/j.enbuild.2012.03.003. ISSN 0378-7788. Dostupné z: https://www.sciencedirect.com/science/article/pii/S037877881200151X.

[14] Vanneschi, L. a Poli, R. Genetic Programming — Introduction, Applications, Theory and Open Issues. In: Rozenberg, G., Bäck, T. a Kok, J. N., ed. *Handbook of Natural Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 710–733. DOI: 10.1007/978-3-540-92910-9_24. ISBN 978-3-540-92910-9. Dostupné z: https://doi.org/10.1007/978-3-540-92910-9_24.

[15] Walker, J. A., Miller, J. F., Kaufmann, P. a Platzner, M. Problem Decomposition in Cartesian Genetic Programming. In: Miller, J. F., ed. *Cartesian Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, s. 35–99. DOI: 10.1007/978-3-642-17310-3_3. ISBN 978-3-642-17310-3. Dostupné z: https://doi.org/10.1007/978-3-642-17310-3_3.

[16] Whitley, D. a Sutton, A. M. Genetic Algorithms — A Survey of Models and Methods. In: Rozenberg, G., Bäck, T. a Kok, J. N., ed. *Handbook of Natural Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 638–669. DOI: 10.1007/978-3-540-92910-9_21. ISBN 978-3-540-92910-9. Dostupné z: https://doi.org/10.1007/978-3-540-92910-9_21.

[17] Šikulová, M. *Symbolická regrese a koevoluce*. Brno, 2011. Diplomová práce. FIT VUT v Brně. Dostupné z: http://hdl.handle.net/11012/54137.

[18] Šikulová, M. a Sekanina, L. Coevolution in Cartesian Genetic Programming. In: *Proc. of the 15th European Conference on Genetic Programming*. Springer Verlag, 2012, sv. 7244, s. 182–193. Lecture Notes in Computer Science. DOI: 10.1007/978-3-642-29139-5_16. ISBN 978-3-642-29138-8. Dostupné z: https://www.fit.vut.cz/research/publication/9832.

[19] Žegklitz, J. *Enhancements of Genetic Programming-based Symbolic Regression Algorithms*. Prague, 2021. Disertační práce. Czech technical university in Prague. Faculty of Electrical Engineering. Department of Cybernetics. Dostupné z: https://dspace.cvut.cz/handle/10467/96767.

Příloha A

Obsah DVD

Readme soubory

`README.txt` popisuje obecné informace a rozdíl mezi 2 variantami programu, které se nacházejí ve 2 různých složkách. Soubory `*/README.txt` popisují kompilaci, spuštění a dodatečné informace pro dané verze.

Plná verze programu

Plná verze programu se nachází ve složce `SymbRegComp`. Místy pracuje s GUI, tudíž nelze spustit na zařízeních bez grafického rozhraní (merlin, wsl...).

Verze bez GUI

Program nevyužívající GUI, který je možné spustit na serveru merlin, wsl apod. se nachází ve složce `SymbRegCompNoGui`.

Technická zpráva

Technická zpráva k bakalářské práci se nachází v souboru `BP.pdf`. Soubory k vytvoření tohoto pdf (overleaf projekt) se nacházejí v archivu `BP.zip`.

Zdrojové soubory

Zdrojové soubory se nachází ve 2 složkách: `*/src`. Tyto soubory jsou pro obě verze velmi podobné (liší se v možnosti vykreslovat regresní křivku).

Dokumentace

Dokumentaci lze najít ve složce `SymbRegComp/doc`, startovní bod je `index.html`.

Makefile

Pro překlad je využíván Makefile, viz `*/Makefile` (opět jeden Makefile pro každou verzi). Překlad: `make`. Výpis nápovědy: `make help`.

Datové soubory

Soubory `*/*.csv` obsahují trénovací vektory pro zkoumané úlohy (9 základních + problém z reálného světa).

Spustitelné soubory

Viz soubory `*/SymbRegComp.jar`. K vytvoření došlo pomocí Makefile.