UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DIPLOMOVÁ PRÁCE

2024 Bc. Václav Hrubý

Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Dvouvrstvé genetické programování na vícerozměrných datech

Bc. Václav Hrubý

Diplomová práce
2024

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Václav Hrubý**
Osobní číslo:	**I22158**
Studijní program:	**N0613A140007 Informační technologie**
Téma práce:	**Dvouvrstvé genetické programování na vícerozměrných datech**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování

Cílem diplomové práce je otestování chování dvouvrstvého genetického programování na vícerozměrných datech. V teoretické části bude popsána problematika evolučních metod, genetických algoritmů, genetického programování, dvouvrstvého genetického programování a benchmark problémy používané v komunitě výzkumníků v oblasti genetického programování. V praktické části bude implementováno dvouvrstvé genetické programování ve vybraném programovacím jazyce (například v jazyce Python) a navrhne a provede experimenty (zejména problémy pro symbolickou regresi) na vybraných datových sadách. Výsledky experimentů přehledně zobrazí pomocí vhodně zvolených tabulek nebo grafů.

Rozsah pracovní zprávy: **cca 60 stran**
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

HYNEK, Josef. Genetické algoritmy a genetické programování. Průvodce (Grada). Praha: Grada, 2008. ISBN 978-80-247-2695-3.
GOLDBERG, David E. Genetic algorithms in search, optimization, and machine learning. Reading, Mass.: Addison-Wesley Pub. Co., c1989. ISBN 0201157675.
POLI, Riccardo; LANGDON, W. B.; MCPHEE, Nicholas F. a KOZA, John R. A field guide to genetic programming. [S.l.: Lulu Press], 2008. ISBN 1409200736.
LANGDON, W. B. a POLI, Riccardo. Foundations of genetic programming. New York: Springer, c 2002. ISBN 3540424512.
MERTA, Jan a BRANDEJSKÝ, Tomáš. Two-layer genetic programming. Online. Neural Network World. 2022, roč. 32, č. 4, s. 215-231. ISSN 23364335. Dostupné z: https://doi.org/10.14311/NNW.2022.32.013. [cit. 2023-10-11].

Vedoucí diplomové práce: **Ing. Jan Merta, Ph.D.**
Katedra softwarových technologií

Datum zadání diplomové práce: **8. listopadu 2023**
Termín odevzdání diplomové práce: **17. května 2024**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

prof. Ing. Antonín Kavička, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 30. listopadu 2023

Prohlašuji:

Práci s názvem Dvouvrstvé genetické programování na vícerozměrných datech jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 12. 05. 2024

Bc. Václav Hrubý

PODĚKOVÁNÍ

Chtěl bych poděkovat Ph.D. Janu Mertovi za vedení mé diplomové práce, cenné rady a podporu při tvorbě této práce. Dále bych chtěl také poděkovat svým rodičům, kolegům a přátelům za veškerou formu podpory při průběhu studia.

ANOTACE

Cílem diplomové práce je otestování chování dvouvrstvého genetického programování na vícerozměrných datech. V teoretické části student popíše problematiku evolučních metod, genetických algoritmů, genetického programování, dvouvrstvé genetické programování a benchmark problémy používané v komunitě výzkumníků v oblasti genetického programování. V praktické části student implementuje dvouvrstvé genetické programování ve vybraném programovacím jazyce (například v jazyce Python) a navrhne a provede experimenty (zejména problémy pro symbolickou regresi) na vybraných datových sadách. Výsledky experimentů přehledně zobrazí pomocí vhodně zvolených tabulek nebo grafů.

KLÍČOVÁ SLOVA

Genetické programování, genetický algoritmus, symbolická regrese

TITLE

Two-layer genetic programming on multi-dimensional data

ANNOTATION

The aim of this thesis is to assess the behaviour of two-layer genetic programming on multidimensional data. In the theoretical part, the student will describe the problem of evolutionary methods, genetic algorithms, genetic programming, two-layer genetic programming and benchmark problems used in the genetic programming research community. In the practical part, the student will implement two-layer genetic programming in a selected programming language (e.g. Python) and design and perform experiments (especially symbolic regression problems) on selected datasets. The results of the experiments will be clearly displayed using appropriately chosen tables or graphs.

KEYWORDS

Genetic programming, genetic algorithm, symbolic regression

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM ZKRATEK

GP – Genetické programování
LISP – List processing
GA – Genetický algoritmus
SR – Symbolická regrese
ML – Machine learning
API – Application programming interface
DEAP – Distributed Evolutionary Algorithms in Python

ÚVOD

Genetické programování je poměrně mladá technika evoluce programů, u které stále neexistují jasné odpovědi na některé základní otázky týkající se tohoto odvětví. To také souvisí s tím, že je tento evoluční algoritmus možno aplikovat na širokou škálu problémů, které mají výrazně odlišné vlastnosti a je tedy poměrně složité vytvořit a obhájit výroky, které by platily na veškerou množinu problémů, na které je možné genetické programování aplikovat.

V teoretické části bude zevrubně popsán jednoduchý běh evolučního algoritmu a veškeré genetické operace, které jsou v jeho základní podobě součástí. Poté bude popsáno samotné genetické programování, kde bude vysvětlena reprezentace jedinců v tomto evolučním algoritmu a jak se liší oproti ostatním implementacím genetických algoritmů. Pozornost bude také věnována různým typům křížení a mutací a nejčastější implementace těchto operátorů, které se používají v praxi a byly součástí několika již provedených výzkumů, budou vysvětleny na příkladech.

V praktické části bude pomocí vývojových diagramů znázorněn návrh dvouvrstvého genetického programování a vytvořena implementace tohoto přístupu. Pro tuto implementaci bude využit programovací jazyk Python, z důvodu nespočetného množství knihoven, které umožňují statistickou analýzu získaných výsledků a jejich zpracování do jednoduchých grafů pomocí kterých lze výkonosti jednotlivých konfigurací snadno porovnat. Ke zpracování bude také využit framework DEAP, který usnadňuje tvorbu a běh evolučních algoritmů a zároveň také poskytuje intuitivní způsob, jak definovat svoje vlastní metody, které budou použity pro dané genetické operace. Framework také poskytuje sadu již připravených implementací genetických operací, které lze při tvorbě genetického programování také použít.

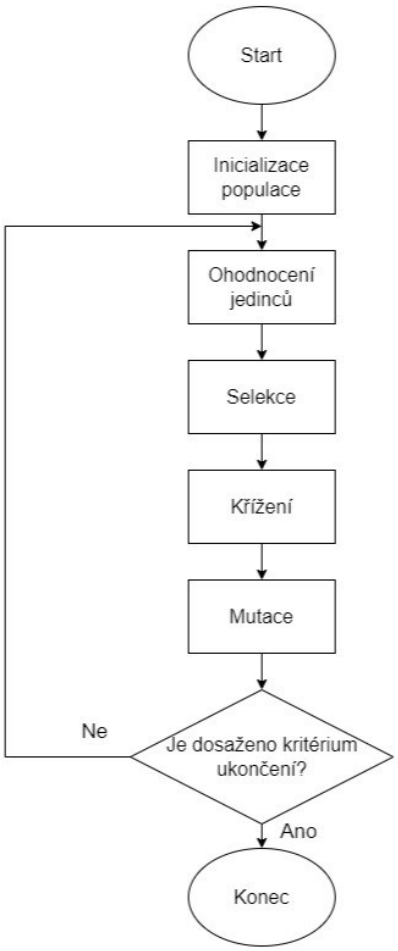
Cílem této diplomové práce je prozkoumat možný přínos a výhody, respektive nevýhody dvouvrstvého přístupu v oblasti genetického programování. Cílem je také porovnat výsledky získané použitím této metody s různými konfiguracemi genetických parametrů obou vrstev s obyčejným jednovrstvým genetickým programováním.

1 GENETICKÝ ALGORITMUS

Genetické algoritmy jsou vyhledávací algoritmy založené na mechanismech přírodního výběru a přírodní genetiky. Kombinují přežití nejsilnějších mezi řetězcovými strukturami se strukturovanou, byť náhodnou výměnou informací s cílem zkombinovat vyhledávací algoritmus s technikami přirozeného lidského vyhledávání, které nespočívá pouze v náhodném prohledávání prostoru, ale využívá jisté lidské intuice. V každé generaci je vytvořena nová populace umělých jedinců, kteří jsou tvořeni náhodnými částmi nejlepších jedinců předešlé generace a případnou částí zbrusu nového materiálu (nazývanou mutací). I přesto, že velká část genetického algoritmu je založena na náhodě, podobně jako v evoluci je tento typ algoritmu mnohem silnější a efektivnější než náhodný průchod, a to z toho důvodu že nepotřebuje žádné další informace o daném problému. Je totiž schopen efektivně využít historické informace z dřívějších generací k tomu, aby byl schopen naleznout nové vyhledávací body s předpokládaným lepším výkonem. Díky možnosti pracovat s prakticky žádnými vstupními daty, je schopný nalézt řešení k problémům, u kterých nejsou žádné jiné optimalizační metody schopny nalézt řešení z důvodu nedostatku návaznosti, linearity nebo jiných vlastností. [1, 2]

1.1 Průběh genetického algoritmu

Samotný průběh genetického algoritmu lze rozdělit na několik základních fází. Tyto fáze je možno vidět na obrázku níže a postupně si všechny tyto fáze popíšeme.



Obrázek 1 – schéma genetického algoritmu, inspirace z: [3]

1.1.1 Inicializace populace

Jako první krok, ještě před samotnou inicializací populace, je nutné definovat kódování, které se má použít pro reprezentaci jedince. Genetické algoritmy kladou důraz na použití „genotypu“, který je dekódován a vyhodnocován. Tyto genotypy jsou často jednoduché datové struktury. Nejčastější formou reprezentace jedinců (chromozomů, řešení) v populacích genetických algoritmů je reprezentace pomocí binárních řetězců. Binární řetězec je jednoduchá sekvence nul a jedniček u které lze jednoduše provést operace mutace a křížení, které budou vysvětleny v dalších kapitolách. Samozřejmě ale nelze vyjádřit řešení všech problémů pouze jednoduchými binárními řetězci a existuje nespočetné množství způsobů, jak daného jedince reprezentovat. [4, 5, 6, 7]

Pokud již máme rozmyšlenou formu reprezentace jedince, je možné vygenerovat počáteční populaci jedinců. Tato generace je většinou náhodná, je ale také možné poskytnout počáteční populaci iniciální data, která mohou být pro celkový běh algoritmu přínosná. Tato data je možné

například převzít z předchozích běhů genetického algoritmu, kde byl potomek nadmíru úspěšný, s cílem „nasměrovat“ evoluci správným směrem. [4]

1.1.2 Ohodnocení jedinců

Po vygenerování iniciální populace jedinců musí být všichni jedinci obsaženi v populaci ohodnoceni. Toto ohodnocení se provádí výpočtem hodnoty vyhodnocovací funkce. Vyhodnocovací funkce, nebo také v oblasti evolučních algoritmů častěji nazývaná **fitness funkce** je funkce, jejíž úlohou je reprezentovat požadavky, na které by se měla populace zaměřit, aby je splnila. Vytváří základ pro selekci a zajišťuje postupné zlepšování. Přesněji řečeno definuje, co zlepšování znamená. Pokud bychom měli načrtnout příklad, jak by mohla být fitness funkce reprezentována, představme si příklad, u kterého se má najít neznámé číslo x, které maximalizuje hodnotu funkce x^2. Hodnota fitness pro každého jedince by byla v tomto případě pouze druhá mocnina čísla, kterým je jedinec reprezentován. Pokud by jedinec byl tedy reprezentován číslem 7, tak by jeho hodnota fitness byla rovna 7^2, tedy 49. Problémy řešené genetickým algoritmem nemusí být vždy takového charakteru, u kterého chceme hodnotu fitness maximalizovat. Často je problém řešený genetickým algoritmem optimalizační problém, při kterém se tuto hodnotu naopak snažíme minimalizovat, abychom dosáhli co možná nejmenší chyby, ceny, trasy nebo jiné veličiny. [8]

1.1.3 Selekce

Poté, co jsou všichni jedinci populace ohodnoceni, se může přejít k selekci. Tato operace se také někdy nazývá jako selekce rodičů nebo výběr partnerů. Jedná se totiž o operaci, při které se porovnávají jedinci na základě jejich kvality a vyberou se ti jedinci, kterým se dovolí být rodiči další populace. Jedinec se stává rodičem, pokud byl zvolen, aby prošel křížením za účelem vytvoření nového potomka. Díky této vlastnosti je tato operace zodpovědná za prosazování zlepšování kvality. Jedinci s vyšší hodnotou fitness mají totiž vyšší šanci, aby byli zvoleni, a tudíž potomci s nižší hodnotou často nedostanou příležitost křížení a jejich geny z populace postupně vymizí. Nicméně i tito potomci mají malou, ale nenulovou šanci na výběr, jinak by se celé vyhledávání mohlo stát až příliš chamtivé a celá populace by se s pozdějšími generacemi mohla zaseknout v lokálním optimu. [8]

K implementaci selekce je možno využít několik algoritmů, ty nejpoužívanější si následně představíme:

Výběr ruletového kola

Tento typ výběru bývá také někdy nazýván jako proporcionální výběr. Jedná se o první typ výběru, který byl zaveden, když byly genetické algoritmy poprvé vyvinuty. Byl pojmenován díky podobnosti výběru, který má s fyzickou ruletou, kterou můžeme znát z tradičního kasina. Funguje tak, že pro množinu N jedinců, kde má každý jedinec specifickou hodnotu fitness funkce, najde součet všech těchto hodnot a každému jedinci přiřadí určitou šanci k tomu být vybrán s hodnotou rovnou fitness jedince vydělenou celkovému součtu všech hodnot fitness. [9]

Využitím této metody získáme proporcionální šanci, že jakýkoliv jedinec bude vybrán. Jedna nevýhoda tohoto algoritmu je ale jeho časová komplexnost, která je $O(n^2)$, což by mohlo vést k časově náročnějšímu provádění algoritmu, zejména u populací s vysokým počtem jedinců. [9]

Turnajový výběr

Funguje na principu, že se z populace vybere *n* jedinců, kteří jsou proti sobě postaveni v turnajovém pavoukovi. Jedincům, kteří proti sobě momentálně stojí v zápase, se porovnávají hodnoty funkce fitness. Jedinec, který má hodnotu fitness větší, než jeho soupeř postupuje do dalšího kola, dokud se nenajde vítěz, který je zařazen do seznamu jedinců vybraných ke křížení. [12]

Výběr turnaje je užitečný a robustní výběr běžně používaný genetickými algoritmy (GA). Výběrová jistota turnajové selekce se přímo mění s velikostí turnaje – čím více soutěžících, tím vyšší je výsledná selekce. Je velmi jednoduchý na implementaci a je účinný jak pro neparalelní, tak pro paralelní architektury. Turnajový výběr může také jednoduše upravit výběrový tlak a přizpůsobit jej různým problémovým doménám. Výběrový tlak se zvyšuje (snižuje) jednoduchým zvýšením (snížením) velikosti turnaje. Všechny tyto faktory přispěly k rozsáhlému využívání turnajového výběru jako výběrového mechanismu pro GA. [11]

Selekce lineárním výběrem

Jedná se o variantu turnajového výběru, která se snaží překonat nevýhodu předčasné konvergence GA k lokálnímu optimu. Tato metoda je založena na pořadí jednotlivců místo jejich hodnoty fitness. Pořadí *n* (které představuje počet jedinců v populaci) je přiřazeno jedinci s nejlepší hodnotou fitness v populaci, zatímco nejhorší jedinec obdrží pořadí 1. Šance na zvolení jedince se tedy vypočítá podle následující rovnice (1).

$$p(j) = \frac{rank(j)}{n * (n-1)} \quad (1)$$

Kde, *j* představuje daného jedince a *n* počet jedinců v populaci. Poté probíhá volba jedince stejně jak u ruletového výběru. [10]

Chamtivý nadměrný výběr

Jedná se o způsob výběru, který je vhodný použít pro komplexní problémy, které vyžadují velký počet potomků v generaci, aby byl přínos této metody viditelný (bylo dokázáno, že tento výběr je výhodný pro populace s počtem jedinců vyšším než 1000). Hlavní motivací je zvýšení efektivity tím, že se zvýší šance výběru jedinců s vyšší hodnotou fitness.

Algoritmus funguje tak, že se populace jedinců se rozdělí na 2 skupiny:

- Skupina 1 bude obsahovat jedince, kteří dohromady vytváří x% celkového součtu všech hodnot fitness celé populace.
- Skupina 2 bude obsahovat všechny zbývající jedince. [13]

Rozhodnutí, z jaké skupiny se bude výběr jedinců provádět záleží na náhodné šanci, přičemž v 80 % případech bude jedinec volen z první skupiny a ve zbývajících 20 % případů bude jedinec volen z druhé „horší“ skupiny. [13]

Hodnotu *x*, zmíněnou u procentuálního zastoupení první skupiny jedinců, je možno stanovit dynamicky například podle toho, jak různorodá je aktuální generace, ale často se tato hodnota stanovuje staticky, podle celkové velikosti populace. [13]

Pro velikost populace rovnou 1000, 2000, 4000 a 8000 se často příslušné hodnoty *x* nastavují na 32 %, 16 %, 8 % a 4 %. [13]

1.1.4 Křížení a mutace

Na generaci vygenerovanou v předchozím kroku se dále použije sada genetických operátorů, které by v nejlepším případě měly vést k vytvoření potomků s lepší hodnotou fitness v rámci jejich opakovaných aplikací. Tyto genetické operátory se také někdy označují jako variační operátory. Ty se dělí na dva typy na základě jejich arity (počtu jedinců, kteří v dané operaci figurují). [14]

Křížení

Křížení je proces, při kterém se náhodně párují rodiče, kteří byli zvoleni ve fázi selekce. Vznikne tak několik potomků, kteří kombinují prvky z obou rodičů s cílem vytvořit potomky, kteří budou mít příznivější hodnotu fitness než jejich předci. Křížení je stochastický operátor to znamená že volba, které části rodičů budou zkombinovány a jakým způsobem, je náhodný proces. Účel této operace se v různých aplikacích liší. Většinou se ale jedná o hlavní vyhledávací operátor, který slouží k prohledávání stavového prostoru daného problému. Křížení více než dvou potomků najednou je matematicky možné a triviální na implementaci, neobsahuje ale žádný biologický ekvivalent, což je nejspíše důvod, proč se tento způsob křížení v praxi téměř nevyužívá i přestože několik studií prokázalo jeho pozitivní účinek na evoluci. [14]

Motivace za zavedením křížení je jasná. Tím že spojíme dva jedince, kteří mají rozdílné, ale žádoucí vlastnosti bychom měli být schopni vytvořit nového jedince, který bude disponovat kombinací těchto pro nás přínosných vlastností z obou předků. Tento princip má silnou oporu v křížení květin a chovu hospodářských zvířat, kde se tímto způsobem snažíme vylepšit požadované vlastnosti daných jedinců, a naopak odstranit, nebo alespoň potlačit vlastnosti nežádoucí. Genetické algoritmy vytváří velké množství potomků náhodným křížením, a i přesto že většina potomků nejspíše nepřinese žádnou výraznou změnu, někteří z nich budou mít lepší charakteristiky, které budou mít pozitivní dopad na evoluci. [14]

Mutace

Jako unární variační operátor se běžně nazývá mutace. Je aplikovaná na jednoho jedince a vytváří mírně modifikovaného „mutanta“. Operátor mutace je podobně jako operátor křížení stochastický. Výsledný potomek, který z této operace vznikne vždy závisí na sérii náhodných rozhodnutí. Ne každý unární operátor by se ale měl nazývat mutací. Například pokud bychom vytvořili operátor, který by v jedincích cíleně hledal jejich slabé stránky a pokusil se je vylepšit, tak už by se dle definice nejednalo o mutaci. V obecné mutaci by se měly provádět pouze náhodné nezaujaté změny. Popsaný algoritmus by bylo tedy více příslušné pojmenovat jako „heuristická unární mutace“, protože se nejedná o čistě náhodnou operaci. [4]

Variační operátory tvoří evoluční implementaci elementárních vyhledávacích kroků. Vytvoření potomka představuje jakési objevení nového bodu v tomto vyhledávacím prostoru. Existuje teorie, která říká, že pokud je evolučnímu algoritmu poskytnuto dostatek času, tak bude algoritmus schopen najít globální optimum k jakémukoli zadanému problému. Nejjednodušší

způsob, jak tuto podmínku umožnit, je povolit operátoru mutace skočit do jakéhokoliv stavu, tím pádem vytvořit nenulovou pravděpodobnost přechodu jednoho stavu do jakéhokoliv dalšího stavu. Často se ale tímto způsobem mutace neimplementuje, protože se z praxe ukázalo, že to konvergenci spíše uškodí, než že by jí to pomohlo a konvergence do globálního optima tak poté trvá déle času. Je také důležité mít na paměti, že všechny variační operátory jsou závislé na typu reprezentace jedince. Ne všechny metody, které je možno použít při reprezentaci jednoho typu, je možné použít u jiných reprezentací bez úpravy daných metod. [4, 14]

1.1.5 Ukončovací kritérium

Posledním krokem jednoduchého genetického algoritmu je kontrola splnění ukončovacího kritéria. Existují dvě základní formy vhodného ukončovacího kritéria. [14]

První způsob je možné uplatnit, pokud má problém nějakou známou optimální hodnotu fitness. Ideální čas ukončit vykonávání algoritmu by poté právě byl, pokud by některý z jedinců úspěšně dosáhl této hodnoty. Příkladem typu problému, na který by bylo možné tuto formu ukončovacího kritéria aplikovat, může být například aproximace hodnot funkce, u které se snažíme minimalizovat chybu mezi funkcí vytvořenou genetickým algoritmem a aproximovanou funkcí. Pro tento typ problému bude ideální hodnota fitness 0 (aproximovaná funkce by poté pro všechny zkoumané body měla stejnou hodnotu jako funkce vytvořena genetickým algoritmem). Pokud ale například víme, že náš model obsahuje nutná zjednodušení nebo může být ovlivněn nějakým datovým šumem, který by znemožňoval nalézt skutečné optimální řešení, je možné stanovit jistou přesnost, respektive nepřesnost, kterou jsme schopni tolerovat a ve chvíli, kdy se v populaci objeví jedinec, který bude spadat do této tolerance, algoritmus ukončit. Genetické algoritmy jsou ale stochastické a často nemáme žádnou garanci, že takového optima dosáhneme, proto nemusí být toto ukončovací kritérium nikdy naplněno a algoritmus by pak nikdy neskončil. Toto ukončovací kritérium se tedy často kombinuje ještě s druhým způsobem, který určuje, kdy má algoritmus přestat běžet. [14]

Tato druhá forma ukončení je deterministická a je tedy zaručeno, že se po určité době vykonávání algoritmus ukončí. Tuto formu je možné reprezentovat mnoha způsoby, mezi které například patří:

1. Nastavení maximální doby, po kterou může algoritmus běžet.
2. Nastavení maximálního počtu generací, který může algoritmus vyprodukovat.

3. Průběžná kontrola, zda dochází ke zlepšování hodnoty fitness za danou periodu času (například pokud se za určitý počet generací hodnota fitness procentuálně zlepší o stanovenou hodnotu).
4. Rozmanitost populace klesne pod danou hodnotu.

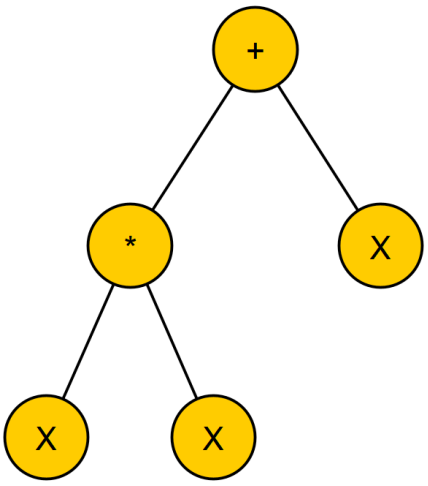
V případě, že se používají obě formy kritérií ukončení, tak jsou nastaveny disjunktně, tedy pokud je jedno z kritérií naplněno (dosaženo optimum nebo je nějaké dané kritérium splněno), tak se algoritmus ukončí. [14]

2 GENETICKÉ PROGRAMOVÁNÍ

Genetické programování je evoluční výpočetní technika, která řeší problémy bez nutnosti specifikace výsledku, nebo jeho struktury uživatelem, před započetím výpočtu. Jedná se o speciální typ evolučního algoritmu, který je podmnožinou strojového učení. [15]

Většinu počítačových programů lze chápat jako provádění sekvencí funkcí s argumenty. Velká část jazykových kompilátorů nejprve přeloží program do tzv. derivačního stromu a poté vygenerují sekvenci strojových instrukcí, které mohou být spuštěny na počítači. Derivační stromy jsou proto přirozenou volbou reprezentace počítačových programů. [16]

Jedinci jsou v populaci typického genetického programování reprezentováni hierarchickou kompozicí primitivních funkcí a terminálů vhodných pro danou problémovou oblast. Množina použitých primitivních funkcí typicky zahrnuje aritmetické operace, matematické funkce, podmíněné logické operace, nebo doménově specifické funkce. Sada používaných terminálů obvykle zahrnuje vstupy odpovídající problémové oblasti a různé číselné konstanty. Složení primitivních funkcí a terminálů přímo odpovídá počítačovým programům vytvářených v programovacích jazycích jako je LISP (kde se nazývají jako symbolické výrazy nebo zkráceně S-výrazy). Tyto výrazy mohou být reprezentovány bodově označeným stromem s uspořádanými větvemi, ve kterém jsou všechny vnitřní body a kořen stromu označeny jako funkce a vnější body (listy) stromu jsou označeny jako terminály. [17]



Obrázek 2 – Příklad rovnice $x^2 + x$ zapsané v S-výrazovém stromě, zdroj: vlastní

Abychom byli schopni na specifický problém aplikovat metodu genetického programování, je nejprve třeba definovat problém na vysoké úrovni. Tuto definici je možné shrnout do následujících kroků:

1. Specifikace množiny terminálů.
2. Definice počátečních „primitivních“ funkcí pro každou větev genetického programu.
3. Volba fitness funkce – Hodnota této funkce pro daného potomka reprezentuje jeho pravděpodobnost, že bude zvolen ke křížení.
4. Příprava jakýchkoli speciálních výkonnostních parametrů pro řízení běhu.
5. Volba kritéria ukončení a metod pro dosažení cílů běhu. [18]

Po těchto krocích může program běžet samovolně. Podobně jak u běžných evolučních algoritmů pro běh nejsou požadována žádná další data. Je tedy schopen nalézt (pokud bude mít k dispozici dostatek času) optimální řešení bez závislosti na externích datech. [18]

2.1 Symbolická regrese

Symbolická regrese (SR) je regresní metoda založená na strojovém učení, která vychází z principů genetického programování, která integruje techniky a postupy z různorodých vědních oborů a je schopna poskytnout analytické rovnice čistě z dat. Tato pozoruhodná vlastnost snižuje potřebu zahrnout jakékoliv znalosti o zkoumaném systému. SR dokáže odhalit hluboké a objasnit nejednoznačné vztahy. Tuto vlastnost lze aplikovat v mnoha různých oblastech od vědy, technologie, ekonomiky až po sociální vědy. [19]

Symbolická regrese je typ regresní analýzy, při které se se odvozuje matematická funkce, která popisuje daný soubor dat. Zatímco konvenční regresní metody (např lineární, kvadratická atd.) mají předem určenou nezávislou proměnnou (proměnné) a snaží se upravit řadu číselných koeficientů, aby dosáhly dokonalého přizpůsobení, SR se pokouší nalézt parametry a rovnice současně. SR se obvykle implementuje pomocí evolučních algoritmů. Současně jsou nejrozšířenějšími koncepty pro konstrukci SR převzaty právě z genetického programování. [19]

2.2 Terminály

Terminály jsou v genetickém programování reprezentovány buď jako proměnné (reprezentující například hodnoty různých senzorů, detektorů, nebo jiných stavových hodnot v nějakém systému) nebo konstanty, které mohou nabírat různých typů hodnot od jednoduchých celočíselných čísel až po booleovské konstanty. [17]

Množina terminálů se tedy může skládat z:

- Externích vstupů programu – Typicky se značí jako pojmenované proměnné (např. x, y).

- Konstant – Mohou být předem definované, náhodně generované při vytváření stromu nebo vytvořené při mutaci.
- Bezparametrických funkcí – Jedná se o funkce, které při jejich volání nepřijímají žádné vstupní parametry. Může se jednat o funkce, které vrací odlišnou hodnotu pokaždé když jsou volány, jako například funkce *rand()*, která vrací náhodná čísla nebo funkce *dist_to_wall()*, která by například vracela vzdálenost robota, který je ovládán pomocí genetického programování. Dalším důvodem, proč mezi tuto kategorii danou funkci řadit je z důvodu, že má vedlejší účinky. Funkce s vedlejšími účinky dělají více než jen, že vrací jednu hodnotu. Funkce spadající do této kategorie mohou měnit globální data struktur, vykreslovat data na obrazovku, ovládat motory robota atd. [12]

2.3 Funkce

Seznam funkcí používaný v genetickém programování se obvykle řídí typem problému. V jednoduchých číselných problémech si v seznamu funkcí vystačíme s pouhými aritmetickými funkcemi (+, -, *, /). Každá funkce vyžaduje specifický počet argumentů, tento počet argumentů se nazývá arita. [12]

Pokud bychom chtěli vypsat množinu některých možností funkcí, jež je možné pro genetické programování využít, může se jednat například o následující typy funkcí:

- aritmetické funkce (+, -, *, atd.),
- matematické funkce (*sin*, *cos*, *log*, *exp*),
- booleovské operace (*AND*, *OR*, *NOT*),
- podmínkové operátory (*If*, *Else If*, *Else*),
- funkce způsobující iteraci (cykly *do-while*, *for*, *while*),
- funkce způsobující rekurzi,
- jakékoli další funkce přizpůsobené specifickému problému. [20]

Samozřejmě se nejedná o vyčerpanou množinu operátorů a je ji možné obohacovat dle libosti. Příkladem problému, pro který je nutné navrhnout funkce s cílem tento specifický problém vyřešit, může být například problém, ve kterém je naším cílem dovést robota na konec bludiště. Pro tento problém bychom mohli definovat množinu funkcí, která by mohla obsahovat operace jako *move* nebo *turn*, kterými bychom mohli robotem hýbat a dovést ho tak ke stanovenému

cíli. V tomto typu problému nám například obyčejné aritmetické funkce nebudou příliš užitečné. [12]

V genetickém programování by měla být množina terminálů a množina funkcí zvolena tak, aby bylo možno naplnit požadavky uzavřenosti a dostatečnosti. [20]

2.3.1 Uzavřenost

Tento požadavek požaduje to, aby byla každá funkce z množiny funkcí schopna přijmout jakoukoliv hodnotu a datový typ, který je schopna vrátit jakákoli jiná funkce z množiny funkcí. Mimo jiné je také nutné zajistit, aby to stejné platilo pro množinu terminálů, respektive aby bylo možné jakýkoli terminál vložit jako argument do jakékoli funkce. Funkce by tedy měly být správně definované a uzavřené pro libovolnou kombinaci argumentů, která by mohla nastat. [20]

Z takto stanovené podmínky vznikají dva hlavní problémy, které je nutné vyřešit.

Konzistence typu

Všechny datové typy u všech funkcí musí být stejné (všechny funkce musí přijímat argumenty jednoho typu a také vracet jako návratovou hodnotu výsledky toho samého datového typu). Tato podmínka je důležitá, protože musíme být schopni použít náhodnou funkci kdekoliv v libovolném stromě a bez splnění této podmínky by to nebylo možné. [12]

Pokud se nad tímto problémem zamyslíme, zjistíme, že abychom docílili konzistence typu napříč všemi funkcemi a terminály, je nutné nemalou část funkcí řešit jiným způsobem, než jsme byli obvykle zvyklí. V obyčejných počítačových programech jsou často součástí číselné proměnné a funkce operující s aritmetickými operátory. Tento fakt sám o sobě problém nezpůsobuje, ale pokud bychom například zahrnuli v množině funkcí booleovské funkce, jako *AND*, *OR* atd. tak bude nutné samotné fungování daných funkcí mírně pozměnit. Není totiž možné, abychom z funkcí vraceli booleovskou hodnotu, protože by s ní jiné funkce (které nejsou booleovské) neuměly pracovat. Musíme tedy nějakým způsobem zajistit, aby byly tyto operace proveditelné. Možný způsob, jak toto zapříčinit, je místo booleovských návratových typů vracet hodnotu 1, pokud je výsledek roven pravdě, nebo hodnotu 0, pokud je výsledek nepravda. [20]

Bezpečnost vyhodnocení

Další problém, který je nutné vyřešit je bezpečnost vyhodnocení. Ta je vyžadována, protože u velké části využívaných funkcí může při jejich provádění nastat výjimka (dělení nulou, nebo

neproveditelná operace jako například *moveForward()*, když robot čelí zdi). Pro splnění této podmínky je nutné tyto stavy nějakým způsobem ošetřit, aby mohlo vykonávání algoritmu pokračovat. Tato podmínka se obvykle splňuje tím, že se vytváří bezpečné verze daných funkcí, které by obvykle v chybových stavech vracely výjimku jako například operace dělení, logaritmus, exponent atd. Tyto „bezpečné“ verze nejprve kontrolují, zda by mohlo dojít při provádění funkce k výjimce a pokud se zjistí, že by funkce při provedení dané operace vrátila výjimku, tak se místo tradiční návratové hodnoty funkce vrátí nějaká výchozí hodnota. Často se jako tato výchozí hodnota vrací číslo 1. [12, 20]

Alternativní možnost, kterou je možné použít, abychom tuto podmínku splnili, je taková, že pokud by měla nastat neproveditelná operace, tak se výjimka zachytí a danému jedinci se silně sníží jeho hodnota fitness. Tento přístup ale může být nebezpečný, pokud je šance na výskyt chybového stavu poměrně vysoká. Mohl by totiž nastat stav, kdy by příliš mnoho jedinců v populaci mělo téměř stejnou (velice špatnou) hodnotu fitness. Populace by tak ztratila svoji diverzitu a pro algoritmus by poté bylo velice obtížné tuto diverzitu obnovit a najít kvalitní řešení. [12, 20]

2.3.2 Dostatečnost

Další podmínku, kterou je nutné splnit, je podmínka, která stanovuje, že s použitím kombinací různých terminálů a funkcí bude mít algoritmus schopnost vytvořit řešení k zadanému problému. Programátor genetického programování by měl vědět, nebo se alespoň domnívat, že zvolená množina funkcí a terminálů je daný problém schopna vyřešit. V některých problémových oblastech je identifikace těchto proměnných triviální a naprosto zřejmá. Existují ale problémy, kde tato podmínka může být bohužel splněna pouze pomocí teoretických výpočtů nebo zkušeností. V některých typech problémů je však zajištění této podmínky prakticky nemožné (například předpovídání úrokových sazeb nebo výsledků voleb) a v těchto případech je jediná možnost metoda pokus-omyl. [12, 20]

2.4 Fitness

Jeden velký rozdíl oproti většině ostatních evolučních algoritmů, pokud se zaměříme na funkci fitness, je v kontextu genetického programování její vyhodnocování. Jelikož jsou struktury generované pomocí genetického programování počítačové programy, je nutné pro jejich ohodnocení spouštět všechny programy v populaci, a to většinou i vícekrát, abychom eliminovali prvek náhody. Z tohoto důvodu se v praxi používají interprety, které vykonávají

uzly stromu v takovém pořadí, aby nebyl uzel vykonáván, dokud nejsou známy hodnoty všech jeho argumentů. [12]

2.5 Generování počáteční populace

Předtím, než si vysvětlíme metody, které se používají pro generování počáteční populace u genetického algoritmu, musíme si vysvětlit pojem hloubka stromu a hloubka uzlu, které budeme potřebovat k pochopení následujících metod.

Hloubka uzlu je minimální počet uzlů, které je třeba projít, abychom se dostali od kořenového uzlu stromu k vybranému uzlu. [21]

Hloubka stromu je největší hloubka uzlu, která se nachází v daném stromě. Jedná se tedy o hloubku nejníže položeného listu. Tento pojem se často používá v kontextu nastavení maximální hloubky stromu z důvodu omezení růstu velikosti stromů do nekonečna. [21]

Nyní již přejdeme k popisu samotných metod generování počáteční populace.

2.5.1 „Full“ metoda

Tato metoda, jak už z názvu vyplývá, generuje celé stromy, tzn. že všechny listy v takto vygenerovaném stromě mají stejnou hloubku. Uzly se tedy vybírají náhodně ze seznamu funkcí, dokud není dosažena maximální stanovená hloubka stromu. Jakmile je tato hloubka dosažena, mohou být jako další uzly vybrány pouze terminály, aby již strom nemohl růst do větší hloubky. Přestože tato metoda generuje stromy s listy ve stejné hloubce, nemusí to nutně znamenat, že všechny vygenerované stromy budou mít stejnou velikost (počet uzlů). Aby tento případ nastal, je nutné, aby měly všechny funkce v primitivním setu stejnou aritu. Nicméně i s odlišnou aritou může být generování dostatečně diverzních stromů problematické. [12]

2.5.2 „Grow“ metoda

Tento přístup funguje zpočátku obdobně jako předchozí metoda, s tím rozdílem, že se nevybírá pouze ze seznamu funkcí, ale algoritmus může volit z celého primitivního setu. Je tedy možné zvolit jak funkce, tak i terminály a po dosažení maximální hloubky, se stejně jako u předchozí metody bude volit pouze ze seznamu terminálů. Pokud se nad tímto přístupem zamyslíme, tak pokud se do větve stromu dostane náhodným generováním terminál, tak je tato větev ukončena i přesto, že nebyla dosažena maximální stanovená hloubka. Díky tomuto pravidlu budou vznikat stromy různých velikostí a tvarů. Při použití této metody ale velmi často vygenerujeme stromy, které nedosáhnou maximální stanovené hloubky. Z tohoto důvodu vznikla následující metoda, která se v praxi používá nejčastěji. [21]

2.5.3 Metoda ramped half-and-half

Jedná se o kombinaci předchozích dvou metod, s kterou přišel v roce 1992 John R. Koza. Tato metoda byla vyvinuta, protože žádná z předchozích metod nebyla schopna vytvořit dostatečně pestrou kolekci stromů odlišných tvarů a velikostí. Pestrost je totiž v populacích GP hodně vzácná a žádaná. Princip této metody je velice jednoduchý: Jedna polovina populace je vytvořena metodou full a druhá polovina je vytvořena metodou grow. Pokud by tedy například byla maximální hloubka stromu nastavena na hodnotu 6, všichni potomci z populace by byly rovnoměrně rozděleni do skupin, kde bude jejich hloubka inicializována na 2, 3, 4, 5 a 6. Pro každou tuto skupinu bude polovina stromů inicializována pomocí metody full a polovina pomocí metody grow. Tímto způsobem dostaneme velice pestrou kolekci stromů, které budou užitečné pro další evoluci. [12, 21]

2.6 Selekce

Selekce je v rámci genetického programování stejná jak u ostatních evolučních algoritmů. Stejně jako u většiny evolučních algoritmů i zde jsou jedinci voleni na základě hodnoty fitness, to znamená, že jedinci s nejlepší hodnotou fitness mají vyšší šanci na zvolení pro reprodukci než jedinci s nižší. Strategie pro volbu daného jedince se také od ostatních evolučních algoritmů žádným způsobem neliší. [12]

2.7 Křížení

Oproti selekci se genetický algoritmus v této genetické operaci vůči ostatním evolučním algoritmům významně liší v implementaci této operace, a to zejména z důvodu reprezentace jedinců v populaci. Jedinci jsou totiž reprezentování ve formě syntaktických stromů, takže musí být způsob provedení křížení podobně jako u mutace alterován. [12]

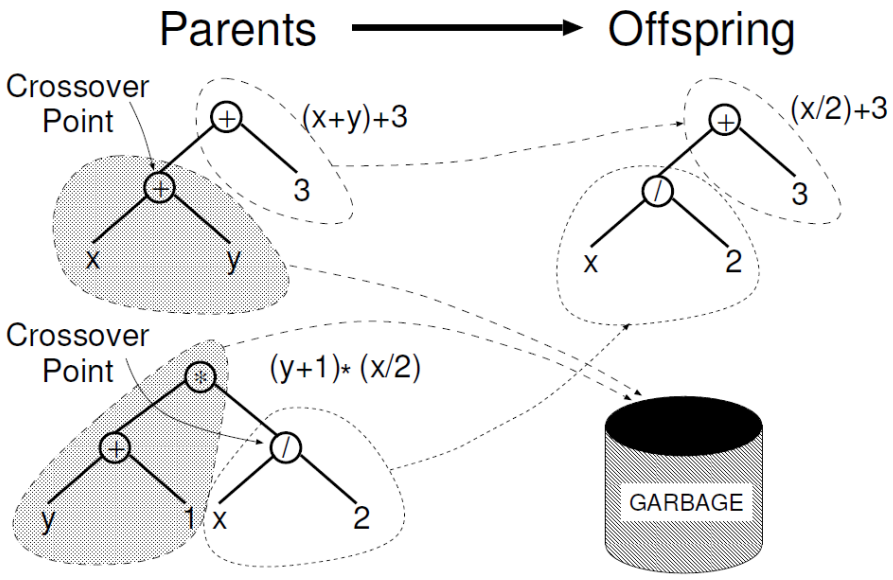
Níže budou představeny některé z nejpoužívanějších typů křížení, které jsou prakticky používány dodnes a byly součástí nespočtu experimentů v poli genetického programování.

2.7.1 Křížení podstromů

Jedná se o nejčastěji používanou formu křížení. U obou rodičů se náhodně a nezávisle na sobě zvolí bod přechodu, který představuje libovolný uzel. Výsledný potomek vzniká nahrazením podstromu s kořenem v bodě přechodu prvního rodiče podstromem s kořenem v bodě přechodu druhého rodiče. Samotní jedinci, kteří byli pro křížení využiti, nejsou žádným způsobem upravováni a mohou být tak součástí dalšího křížení ve stejném stavu. Při křížení jsou totiž vytvořeny kopie, které jsou nezávislé na původních jedincích. [12]

U tohoto typu křížení lze využít dvě základní varianty, a to variantu při níž budou výsledkem křížení dva potomci, nebo je možnost pouze vybrat jednoho z těchto dvou potomků a jako výstup z křížení respektovat jej. V praxi se používají obě varianty, s tím že varianta s jedním potomkem je mírně populárnější. [20]

Níže je možno vidět obrázek, který zobrazuje variantu tohoto typu křížení v praxi.



Obrázek 3 – Příklad křížení podstromů [12]

Z obrázku je možno vidět, že za body křížení se v prvním rodiči zvolil uzel s funkcí + a v druhém rodiči uzel s funkcí /. Následně se vytvořily dva podstromy, kde se v prvním rodiči zvolí všechny uzly nad zvoleným bodem křížení a v druhém rodiči se zvolí všechny uzly pod zvoleným bodem křížení včetně vybraného uzlu. Je možno vidět, že výsledkem operace je pouze jeden potomek, pokud bychom chtěli, aby byli výsledkem dva potomci, pouze bychom volbu podstromů obrátili a tím bychom získali druhého potomka. [12]

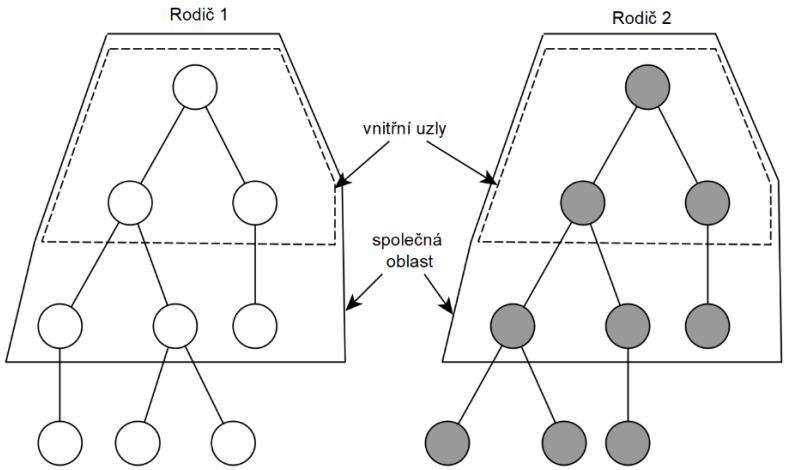
Často nejsou body přechodu vybírány s rovnoměrným rozdělením pro všechny uzly. To z toho důvodu, že by velice často mohla nastat situace, kde si dva rodiče mezi sebou vymění pouze dva listy. Proto John R. Koza navrhnul přístup, ve kterém se stanoví šance pro výběr funkcí na hodnotu 90 % a šance na výběr listů na hodnotu 10 %. Tímto způsobem máme zaručeno, že ve většině případů nebude docházet pouze k velmi malé výměně genetického materiálu, ale stále zde bude existovat šance na výměnu pouze jednotlivých listů. [17]

2.7.2 Rovnoměrné křížení

Tento typ křížení vznikl iniciální inspirací od stejnojmenného operátoru využívaného při aplikacích genetických algoritmů. Hlavní princip tohoto typu křížení je takový, že pro každou alelu je dána 50 % šance, že se vybere z prvního rodiče a 50 % šance, že se vybere z druhého. Je tedy stejně pravděpodobné, že informace v každém místě genu pochází od jednoho nebo druhého rodiče a v průměru každý rodič věnuje 50 % svého genetického materiálu. Celá operace samozřejmě staví na faktu, že jsou všechny chromozomy v populaci stejně strukturované a mají stejnou délku. Tato podmínka ale nejde u genetického programování zaručit, protože rodičovské stromy v genetickém programování budou mít téměř vždy odlišný počet uzlů a mohou být strukturálně jiné. [22]

Pokud bychom ale provedli pozorování různých syntaktických stromů, zjistili bychom, že mnoho z těchto stromů si jsou alespoň částečně strukturálně podobné. Pokud začneme procházet stromy od kořene dolů a pokračujeme hlouběji, často trvá nějakou dobu, než nalezneme uzly, jež se nachází na stejném místě, ale mají jinou aritu. Všechny uzly, nad těmi s různou aritou je možné mezi stromy vyměnit bez porušení původní struktury stromu. Rozdělíme tedy uzly do dvou oblastí: uzly, které se nachází na stejném místě v obou stromech budeme považovat za tzv. *společnou oblast.* Páry uzlů, které se nachází v této oblasti a mají stejnou aritu budeme považovat za *vnitřní uzly.* Toto znamená, že společná oblast nutně zahrnuje vnitřní uzly. [22]

Níže je možno vidět příklad dvou stromů a vyznačení příslušných oblastí.

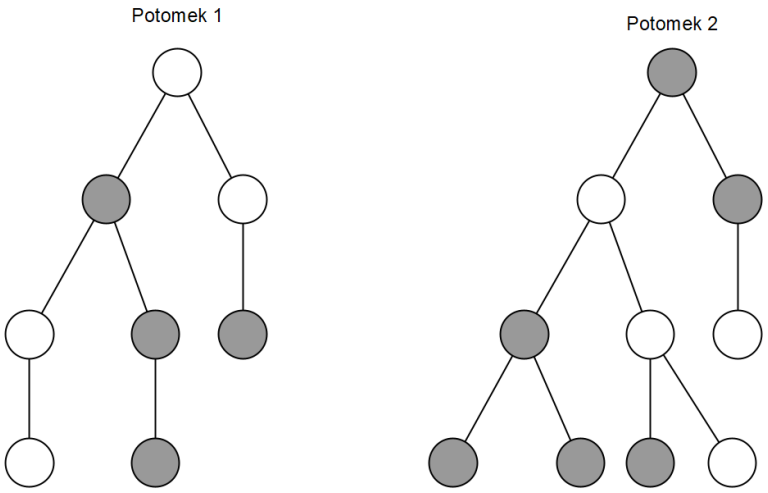


Obrázek 4 – Zvýraznění oblastí používaných pro rovnoměrné křížení [22]

Potom, co jsme identifikovali všechny oblasti, samotné křížení probíhá následovně. Stejně jako u většiny ostatních křížení jsou nejprve vytvořeny kopie obou stromů, aby mohli být původní

rodiče v jejich původním stavu součástí dalšího potencionálního křížení. Dále se vyberou vnitřní uzly podle stanovené pravděpodobnosti (často 50 %, z důvodu, aby byla dodržena původní myšlenka rovnoměrného křížení) a tyto uzly se mezi stromy vymění. Uzly, které nebyly zvoleny k výměně, zůstávají nezměněné a uzly, které se nenachází ve společné oblasti, vůbec nejsou brány v úvahu. Je také důležité podotknout, že je také možná výměna neinterních uzlů, které se nachází ve společné oblasti, ale v takovém případě se musí vyměnit celé podstromy, protože vyměněné uzly mohou a často mají jinou aritu, jelikož jsou struktury podstromů jiné. [22]

Níže je možno vidět možný výsledek daného křížení na stromech z obrázku výše, kde jsou bílou barvou zobrazeny uzly z prvního rodiče a šedou uzly z druhého rodiče.



Obrázek 5 – Výsledek možného rovnoměrného dělení jedinců z předchozího obrázku [22]

2.7.3 Jednobodové křížení

V tomto typu křížení se z obou stromů rodičů vybere „společný“ bod pro křížení, a následně se vymění příslušné podstromy vytvořené z tohoto bodu. Tato metoda umožňuje rodičům mít různé tvary tím, že prochází všechny uzly od kořene a vybírá bod křížení nacházející se v tzv. společné oblasti, což je oblast, ve které mají oba rodiče stejný tvar (uzly v této oblasti nemusí být u obou rodičů identické, ale musí mít stejnou aritu). Jedná se o stejnou společnou oblast, která byla definována v implementaci rovnoměrného křížení v genetickém programování, která již byla popsána v předchozí podkapitole. [12]

2.8 Mutace

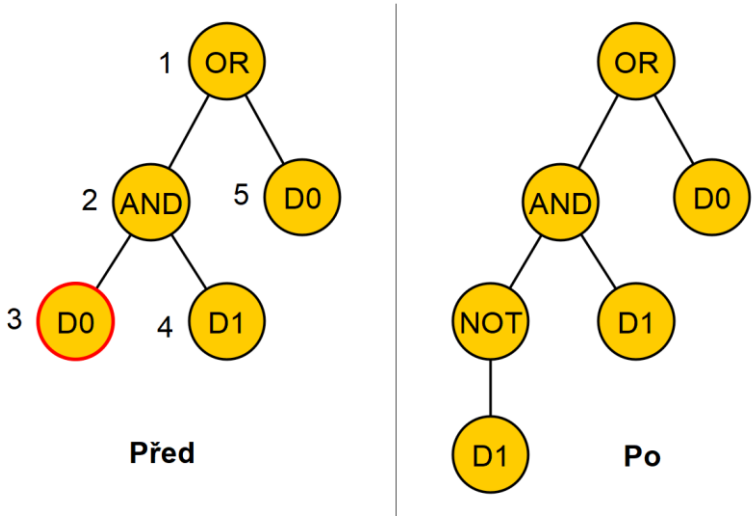
Implementace mutace se stejně jako implementace křížení od běžných evolučních algoritmů liší. V této kapitole budou popsány ty, které jsou v praxi používány nejčastěji.

2.8.1 Mutace podstromů

Jedná se o formu mutace, která funguje na stejném principu jako křížení podstromů popsané v předchozí kapitole. Nejprve se náhodně zvolí bod mutace, který není žádným způsobem limitován (může být zvolena jak funkce, tak terminál). Mutační operace poté odebere zvolený uzel a celý jeho podstrom a místo něj vloží náhodně vygenerovaný podstrom.[1] [20]

Často se limituje velikost tohoto nově vygenerovaného podstromu, aby se předešlo velkým stromům, které by mohly být počátkem vznik bloatu. [12]

Níže je možno vidět příklad tohoto typu mutace.



Obrázek 6 – Příklad mutace podstromů [20]

Na příkladu byl zvolen uzel číslo 3 (*D0*) jako bod mutace. Tento uzel byl ze stromu odstraněn a místo něj byl vygenerován nový podstrom. V tomto případě se jedná o podstrom s kořenem *NOT* a jediným potomkem *D1*. Výsledek mutace je možno vidět na pravé straně obrázku. [20]

2.8.2 Bodová mutace

Další častou formou mutace je bodová mutace, která je hrubým ekvivalentem bit-flip mutace používanou v genetických algoritmech. Funguje tak, že se zvolí náhodný uzel a primitivní typ

[1] Stejný přístup se také někdy používá při křížení, v tomto případě se operace nazývá „headless chicken crossover“ („bezhlavé kuře“). Pro více informací viz. [23]

uložený v tomto uzlu je nahrazen jiným primitivním typem z primitivního setu s jedinou podmínkou a to, aby měly oba stejnou aritu. Pokud se v setu nenachází žádný jiný typ s odlišnou aritou, tak se s daným uzlem žádná operace neprovede a zůstává nezměněný. Tímto způsobem se projdou všechny uzly stromu a každý uzel má tedy šanci zmutovat. Jedná se o velice často používanou a efektivní variantu mutace v genetickém programování, je ale nutné mít na paměti, že je nutné nastavit menší procentuální šanci na výměnu primitivních typů, aby výsledek operace nebyl naprosto odlišný strom. V praxi se často používají hodnoty od jednoho do pěti procent. [12]

2.9 Parametry GP

U genetického programování, podobně jako u ostatních evolučních algoritmů, nelze pevně specifikovat obecné parametry, které by fungovaly dobře pro všechny problémové domény. Problémy, které se dají řešit pomocí genetického programování jsou na toto zobecnění až příliš rozmanité. Je ale možné nalézt nejpoužívanější kombinace, které se v oboru používají a které dříve pomohly dosáhnout úspěšných výsledků a tyto parametry použít jako počáteční bod pro naše experimenty. [12]

Generování počáteční populace se nejčastěji provádí pomocí metody half-and-half s hloubkou stanovenou od 2 do 6. Použití této metody zajistí dostatečnou tvarovou rozmanitost počáteční populace. Koza ve svých experimentech doporučuje 90 % potomků vytvářet pomocí křížení a v ostatních 10 % případů potomky rovnou překopírovat do další generace. Nejčastěji používaná forma křížení je křížení podstromů. Na téma mutace jsou v komunitě poměrně smíšené názory. Koza ve svých experimentech mutaci vůbec nepoužíval, protože tím chtěl dokázat, že genetické programování neprovádí pouze náhodné vyhledávání množiny stavů. Nicméně později vzniklo několik výzkumů, u kterých se při použití mutace výkon GP zlepšil. Použití směsi křížení a různých mutací v poměru 50:50 také poskytlo dobré výsledky. [12, 17]

Pro velikost populace a počet generací je velice obtížné stanovit konkrétní hodnotu. Často záleží na schopnostech systému, na kterém genetické programování běží. Obtížnější problémy, u kterých existují velké stavové prostory budou vyžadovat větší počet jedinců a generací, aby byly schopny řešení nalézt. Dobrý počáteční bod pro velikost populace, kterou je vhodné zvolit pro první experimenty se uvádí okolo hodnoty 500. S tím že počet generací záleží na komplexnosti problému. [12]

2.10 Bloat

Již od počátku vzniku genetického programování je bloat (česky řečeno bobtnání) jedním z nejčastěji studovaných jevů v tomto oboru. Tento jev je obvykle definován jako nárůst průměrné velikosti stromu jedince v populaci bez výrazného zlepšení jeho hodnoty fitness. Tento nárust je většinou exponenciální a velice rychlý, stromy tak dosáhnou enormních velikostí, což způsobuje mimo jiné zvýšení výpočetní náročnosti genetických operací (s čímž souvisí také delší doba vykonávání algoritmu). V odborné literatuře o GP bylo navrženo několik teorií, které vysvětlují proč k tomuto fenoménu dochází. Jedna z teorií například uvádí, že příčina bloat je taková, že rozložení velikostí programů během evoluce je zkreslené způsobem, který umožní jednoduššího výskytu bloat tím, že malí jedinci (s menším počtem uzlů) se z populace vytrácejí a při selekci se upřednostňují potomci větší (s větším počtem uzlů). Toto rozložení se nazývá Lagrangeovo a jedná se o rozložení, kde malé programy mají vyšší frekvenci než velké. Například tedy křížení vytvoří spoustu jednouzlových jedinců, kteří mají téměř nulovou šanci vyřešení daného problému. Tím pádem programy, které mají vyšší, než průměrnou velikost mají selektivní výhodu a tím pádem se průměrná velikost programu zvýší. [12, 24]

2.10.1 Praktické řešení bloat

Pro praktické řešení tohoto problému bylo vyvinuto několik metod, které se snaží explicitně kontrolovat a korigovat rozložení velikosti programů v rámci vyvíjející se populace a snažit se tomuto jevu zamezit. V této podkapitole si představíme několik těchto řešení včetně jejich výhod a nevýhod. [12]

Limitování velikosti a hloubky stromů

Existují tři hlavní přístupy zavedení této metody.

1. Když potomek překročí stanovené limity, vrátí se kopie rodiče. Toto ovšem vede k tomu, že se v populaci bude udržovat velké množství stromů, které jsou velice blízko k tomu, aby překročily stanovenou hranici, což není optimální.
2. Při překročení stanovených limitů bude potomek ohodnocen hodnotou fitness rovnou 0, takže při selekci bude z populace s největší pravděpodobností odstraněn. Tento přístup ale může způsobit to, že jedinec, který měl velice dobrou hodnotu fitness bohužel náhodou překročí při genetické operaci maximální stanovený limit hloubky nebo velikosti a je i přes jeho původní žádoucí řešení odstraněn z populace.

3. Pokud potomek překročí stanovené limity, tak se provede genetická operace znovu. Tato metoda může být výpočetně náročná, pokud je velká část populace blízko danému limitu, za to ale poskytuje nejmenší zásah do konvergence genetického programování.

Každý z těchto přístupů má své výhody i nevýhody a volba jakou z nich použít záleží na tom, jaké nevýhody jsme schopni při provádění algoritmu akceptovat. [12]

Anti-bloat genetické operandy

Jedná se například o přidání kontroly velikostí podstromů při křížení, aby výsledný strom nebyl tak velký. Toto lze aplikovat i při mutaci, kdy se kontroluje velikost možného stromu a limituje se tak výběr uzlů, které mohou být zvoleny jako mutační body. [12]

Anti-bloat selekce

Existuje spousta způsobů, jak tuto metodu implementovat (Tarpeian, Parsimony pressure, Minimum description length, Covarient parsimony pressure). Funguje tak, že při selekci se kontroluje velikost stromu a podle této velikosti může být penalizována hodnota fitness jedince. [12]

3 BENCHMARKOVÉ PROBLÉMY PRO GENETICKÉ PROGRAMOVÁNÍ

Benchmarking je postup, při kterém se vezme soubor problémů (často z různých domén) a na této množině problémů se demonstruje výkonnost daného algoritmu, přičemž se tato výkonnost často porovnává s výkonností jiných algoritmů. Typický přístup akademických výzkumníků je navrhnout nový algoritmus a poté jej porovnat s ostatními nejmodernějšími algoritmy ve výzkumné literatuře tím, že se porovná jejich výkonnost na referenčních testech. Ústředním předpokladem strojového učení je to, že testovací data jsou vybrána ze stejného rozdělení jako trénovací data. Benchmarky by měly být vybrány ze stejného rozdělení jako instance reálného problému. To znamená, že testovací data, která se v benchmarku použijí, by měla být zvolena tak, aby co nejlépe odrážela reálné situace, s kterými se algoritmy v praxi setkají. Jestliže trénovací data (na kterých se algoritmy učí) a testovací data mají podobné rozdělení, je poté větší pravděpodobnost, že budou dané algoritmy dobře fungovat i v praxi. V této kapitole budou představeny nejpoužívanější benchmarky vyvinuté pro genetické programování. [25]

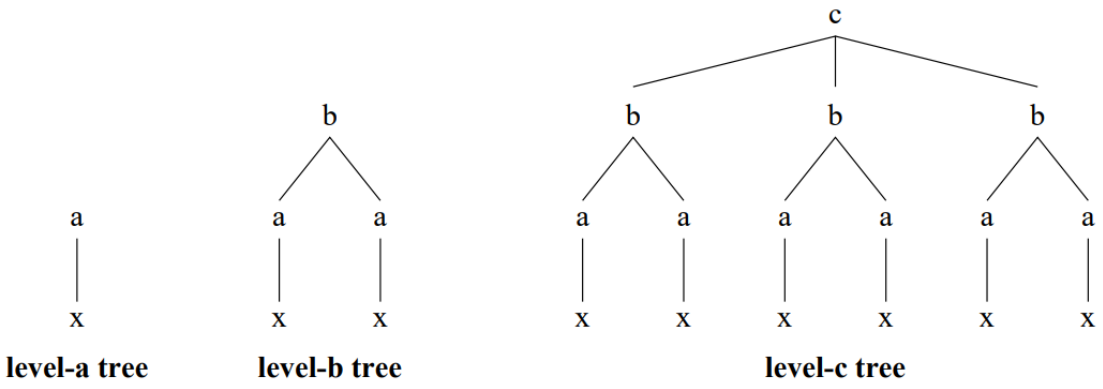
3.1 Problém královského stromu

Vzhledem k množství programů genetického programování, které by mohly poskytnout správné řešení na daný problém je obtížné posoudit efektivitu různých architektonických řešení nebo nastavení parametrů na výkonnost systému genetického programování. Pro obyčejné genetické algoritmy již existoval jeden specifický algoritmus, s kterým přišel John Rolland, který se nazýval „The Royal Road“ (Královská cesta). [26]

Při vývoji benchmarkových metod pro genetické algoritmy se vycházelo z hypotézy o stavebních blocích, která zní: „Genetický algoritmus hledá optimální výkonnost pomocí kombinace krátkých, nízkých a vysoce výkonných schémat, tzv. stavebních bloků.“ Z této hypotézy vyplývá, že důležitou složkou výkonnosti GA by měl být fakt, do jaké míry je rozložení hodnot fitness jedinců hierarchické. Díky této hypotéze vznikl nápad vytvořit populaci tak, aby obsahovala jasné rozvržení stavebních bloků, které bude výhodné pro aplikaci genetického algoritmu. Sestavení těchto bloků by mělo vést k jednoduché konvergenci (Královské cestě). [27]

Z této metody královské cesty aplikované na genetických algoritmech se inspiroval Erik Goodman, který přišel se svou vlastní metodou královského stromu. Vhodný benchmark by měl mít pouze jediný strom, který by byl brán jako odpověď, což je kontrast oproti většině problémů, které GP řeší. Způsobů, jak se ale k tomuto řešení dostat je více. [26]

Královský strom se skládá z jedné základní funkce, která je specializována na tolik funkcí, kolik je vyžadováno. Je tedy možné pomocí nastaveného počtu těchto funkcí řídit složitost daného problému. Tyto funkce jsou specifické tím, že mají rostoucí aritu. Pokud bychom tedy specifikovali příklad, kde bychom měli tři funkce *a*, *b*, *c*, tak by funkce *a* měla aritu rovnou 1, funkce *b* aritu rovnou 2 a tak dále. Společně s množinou funkcí specifikujeme také množinu terminálů *x*, *y*, *z*. Pro jakoukoliv hloubku definujeme takzvaný „perfektní“ strom. Strom úrovně *a* bude mít pouze jeden prvek *a* s jediným potomkem *x*. Strom úrovně *b* bude mít dva stromy úrovně *a* jako svoje potomky. Strom úrovně *c* bude mít tři stromy úrovně *b* jako svoje potomky a tento vzor se opakuje pro stromy dalších úrovní. [26]



Obrázek 7 – "Perfektní" stromy různých úrovní [26]

Počet uzlů hlubších stromů se prudce navyšuje. Strom úrovně *f* se například již skládá z 1927 uzlů. Princip pro vyhodnocování hodnoty fitness potomků zůstává podobný jako u původní implementace královské cesty. Funguje na principu, že potomek je ohodnocen velice dobře, pokud dosáhne perfektního řešení. Tímto se dosáhne „schodišťové“ evoluce, která byla u této metody využívána. Pokud máme za cíl najít hlubší stromy, jsou i menší úrovňové stromy, které byly v průběhu evoluce nalezeny ohodnoceny velmi pozitivně, jelikož se výsledné řešení z těchto stromů bude skládat. [26]

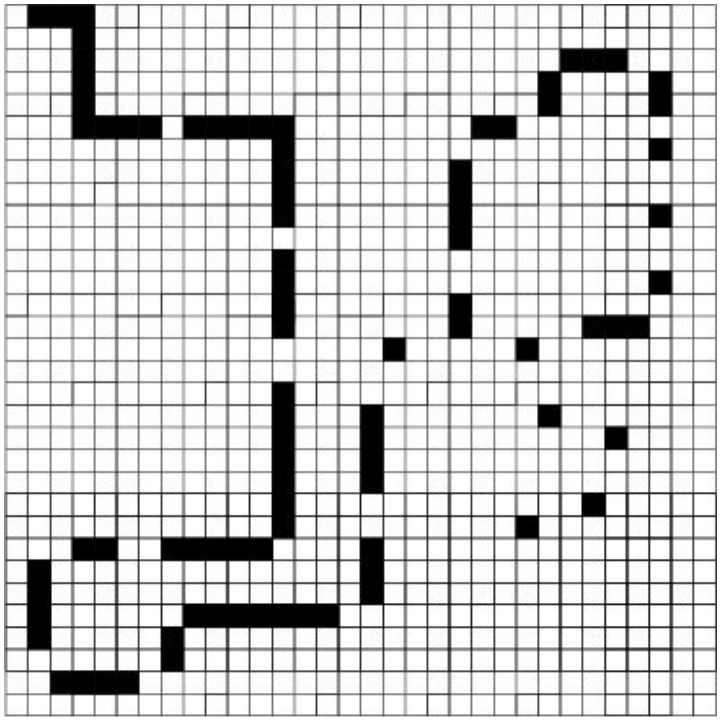
3.2 Problém Santa Fe stezky

Jedná se o stezku, kterou poprvé představil Christopher Langton. Spočívá v tom, že se umělý mravenec snaží najít všechnu potravu, která je rozmístěna na nepravidelné trase. Cílem je nalézt konečný stavový automat pro plnění této úlohy a docílit toho, aby mravenec zkonzumoval veškerou potravu za co nejmenší možný počet kroků. [20]

Umělý mravenec je postaven do pole velikosti 32 x 32, v původním zadání tohoto problému je zasazen do levého horního rohu a otočeným směrem na východ. Celá stezka obsahuje celkem

89 kusů potravy, které nejsou rozmístěny rovně a souvisle, ale nachází se mezi nimi různě velké mezery. Tyto mezery se rozlišují na jednomístné mezery, dvoumístné mezery, mezery v rozích, dvojité mezery v rozích (stejné jako krátký pohyb figurkou koně v šachu) a trojité mezery v rozích (dlouhý pohyb koně v šachu). Stezka Santa Fe je o poznání složitější než John Muirova stezka, která byla originálně použita k řešení tohoto problému. [20]

Níže je možno vidět přesnou podobu stezky Santa Fe. Potrava je reprezentována černými čtverci, počáteční umístění mravence se nachází vlevo nahoře. Mezery v trase (místa, kde se nenachází potrava) jsou reprezentovány bílým čtvercem. Jak je z obrázku vidět, trasa je zpočátku velmi jednoduchá a téměř souvislá s jednoduchými mezerami. Pokud se ale podíváme na druhou polovinu stezky, můžeme si všimnout, že se začínají objevovat čím dál složitější mezery mezi potravou. Zpočátku se jedná pouze o jednoduché mezery v rozích, ale ke konci stezky se už mezi potravou nachází dvojité a trojité mezery v rozích, které jsou pro algoritmus mnohem složitější na rozeznání. [20]



Obrázek 8 – Stezka Santa Fe [28]

Samotný mravenec má možnost provedení pouze následujících operací: otočení doprava o 90 stupňů, otočení doleva o 90 stupňů a pohyb vpřed ve směru, ve kterém je mravenec momentálně otočen. Pokud při pohybu vpřed narazí mravenec na pole s potravou, tuto potravu zkonzumuje. [20]

3.3 Genetické programování potřebuje lepší benchmarky

V roce 2012 vyšla publikace „Genetic Programming Needs Better Benchmarks“, což byla přímá reakce na nedostatečné množství kvalitních benchmarkových problémů, které byly jednoduše realizovatelné a reprodukovatelné.

V tuto dobu nebylo genetické programování obor, který by byl proslulý svými rigorózními benchmarky. Spousta benchmarků se v GP používala jenom z toho důvodu, že byly použity v historických evolučních algoritmech, kde byly úspěšné. V kontextu genetického programování byly tyto benchmarky terčem kritiky, za to že jsou moc jednoduché nebo dostatečně nereplikují reálné problémy, pro které by bylo využít GP výhodné. [29]

Existuje několik základních vlastností, kterými by měl dobrý benchmarkový problém disponovat. Dobrý benchmarkový problém by mělo být obtížné vyladit a měl by také být dostatečně rozmanitý, aby odlišné typy GP měly šanci ukázat své přednosti. Nemělo by se také jednat o problém, který by měl povahu takzvaného „toy“ problému, což je fenomén, kterému spousta benchmarkových algoritmů pro GP propadla. Jedná se o případ, kdy je problém příliš jednoduchý a dostatečně nereflektuje problémy na které by v praxi mohlo být GP výhodné. Samozřejmost pro dobrý benchmarkový problém je také poskytnutí dostatečné dokumentace, uvedení použitých nastavení parametrů a popsání využitých genetické operátory, aby bylo výsledky v rámci možností jednoduše replikovat. [29]

3.4 Moderní benchmarky pro genetické programování

Situace se ale za posledních 12 let zlepšila a vzniklo velké množství sad benchmarků, které se snažily nevýhody původních benchmarků napravit a poskytnout tak lepší zázemí pro budoucí výzkum. [30]

V této kapitole bude nastíněno několik benchmarků, které se snažily tuto „mezeru“ v oboru zaplnit.

3.4.1 PSB a PSB2

PSB1 a PSB2 (Program Synthesis Benchmark) jsou sady benchmarkových problémů, které byly použity v různých výzkumných publikacích. V roce 2015 byla vydána první z této dvojice sad – PSB1. PSB1 obsahovala 29 problémů, které mohly být využity k porovnávání syntetických systémů. Od doby jejího vydání vzniklo více než 80 výzkumných prací ve kterých byla použita k porovnávání různých systémů právě tato benchmarková sada. Většina z těchto prací byla zaměřena na genetické programování, nicméně v některých pracích byla tato sada

využita i pro algoritmy, které nebyly evoluční. Mezi tyto algoritmy patřil například gradientní algoritmus se zpožděním nebo vyhledávání stromu metodou Monte Carlo. Zpočátku většina obsažených problémů byla na tehdejší poměry pro GP složitá, ale v dnešní době je již velká část problémů obsažených v sadě vyřešena a procentuální úspěšnost běhů, která se na počátku ze vzorku sto běhů pohybovala v jednotkách, nebo maximálně desítkách úspěšných běhů, se nyní blíží stovce. Z tohoto důvodu již bylo nutné přijít se složitějšími úkoly, protože původní sada již byla téměř vyčerpaná. [31]

Začalo se tedy pracovat na druhé sadě, která nese jméno PSB2. Tato sada obsahuje 25 problémů, které se snaží navázat na úspěch původní sady. I přes úspěch první sady se ale autoři snažili problémy vylepšit a vzít v potaz zpětnou vazbu, kterou dostali od komunity. Jedním ze zlepšení je například poskytnutí datasetu, ve kterém jsou ke každému problému přiloženy pro daný vstup správné výstupy. Bylo také učiněno rozhodnutí odstoupit od problémů s binárními výstupy, protože by často mohly programy vrátit správný výsledek i přestože k výsledku došly špatnou cestou, z důvodu že existují pouze dva možné výstupy a šance, že se do daného výstupu trefí, je vysoká a tím pádem tedy nastává takzvaný overfitting (nadměrné přizpůsobení). [31]

Jeden z příkladů problémů v sadě PSB2 je následující:

„Při zadání řetězce představujícího tweet ověřte, zda tweet splňuje původní požadavky Twitteru na znaky. Pokud má tweet více než 140 znaků, vraťte řetězec "Příliš mnoho znaků". Pokud je tweet prázdný, vraťte řetězec "Nic jste nenapsali". V opačném případě vraťte řetězec "Váš tweet má *X* znaků", kde *X* je počet znaků v tweetu.“ [31]

3.4.2 SRBench

Jedná se o tzv. „žijící benchmark“, což znamená, že se jedná o jakýsi rozcestník, který je dynamický a s časem se mění. Lze zde najít výsledky předchozích experimentů včetně detailní dokumentace ohledně použitých parametrů a genetických operátorů. [30]

Tento benchmark se v současné době skládá ze 14 metod symbolické regrese, 7 dalších metod ML (strojového učení) a 252 datových sad z PMLB (Penn Machine Learning Benchmarks), včetně reálných a syntetických datových sad z procesů se základními modely i bez nich. [32]

Aby byla zajištěna reprodukovatelnost, je definováno společné prostředí (prostřednictvím programu Anaconda) s pevnými verzemi programů a jejich závislostmi. Na rozdíl od většiny studií symbolické regrese jsou veškeré instalační kódy, kódy experimentu, výsledky a analýzy k dispozici prostřednictvím úložiště pro použití v budoucích studiích. Aby byl SRBench co možná nejvíce rozšiřitelný, byl také automatizován proces začleňování nových metod a

výsledků analýzy. Uložiště přijímá průběžné příspěvky nových metod, které splňují minimální požadavky API. Pokud tedy dorazí příspěvek, který je kompatibilní s referenčním kódem, tak je tento nový dokument ihned přiložen do seznamu nových benchmarků. [33]

SRBench pořádal svou první soutěž na konferenci GECCO 2022 v Bostonu. Tato soutěž se snažila zlepšit praxi symbolické regrese tím, že hodnotí předložené metody symbolické regrese na dosud neznámých, reálných a syntetických souborech dat. Tyto datové sady pocházely především z oblastí fyziky, epidemiologie a bioinformatiky. Celkem se zúčastnilo 13 oficiálních soutěžících. [34]

3.4.3 DIGEN

DIverse and GENerative ML Benchmark (DIGEN) je sbírka syntetických (vygenerovaných dat na základě reálných dat) datasetů, které slouží pro reprodukovatelné a interpretovatelné benchmarky různých vývojových algoritmů pro klasifikaci binárních výsledků. Celkem se v datasetu nachází 40 matematických funkcí. Těchto 40 funkcí bylo nalezeno heuristickým algoritmem, který byl navržen, aby maximalizoval různorodost výkonu mezi populárními algoritmy strojového učení a díky tomu vytvořil užitečnou testovací sadu pro porovnávání nových metod. [35]

Celý dataset je velice podrobně zdokumentován a obsahuje Docker script, aby bylo možné vytvořit identické prostředí tomu, ve kterém byly výsledky získány. Součástí prostředí jsou také již připravené skripty, pomocí kterých lze jednoduše zobrazit a porovnat výsledky s těmi získaných z námi provedených experimentů. Toto lze realizovat pomocí Python knihoven, které se stáhnou společně s připraveným prostředím. [2] [36]

3.4.4 Používané problémy na symbolickou regresi

Častým typem problémů používaným pro genetické programování je právě symbolická regrese. Existuje velké množství kolekcí různých funkcí, které byly v minulosti použity pro aproximaci s použitím různých vývojových algoritmů. Jeden z článků, který vyšel jako reakce na nedostatek problémů v této oblasti a zejména špatné dokumentaci, která zabraňovala jednoduché reprodukci prezentovaných výsledků, je článek od vývojářů, kteří stojí za projektem HeuristicLab. [38]

V tomto článku je možné vidět shrnutí sady problémů z různých děl, které se zabývaly právě symbolickou regresí. Autor článku na samotné testování aproximace použil algoritmus regrese

[2] Celý benchmark včetně postupu instalace je k dispozici na https://github.com/EpistasisLab/digen/

pomocí náhodných lesů, ale problémy tohoto typu nejsou na tuto metodu limitovány a lze na jejich řešení použít širokou škálu evolučních metod. Mezi obsaženými problémy se nachází funkce velké rozmanitosti od velice jednoduchých funkcí jako například odmocnina z x, přes funkce s goniometrickými funkcemi až po složité funkce, u kterých autory použitá regrese pomocí náhodných lesů naprosto selhala. Obsažené problémy jsou dílem mnoha autorů, mezi které patří například Ekaterina Vladislavleva nebo Su Nguyen jejichž funkce budou použity pro naše experimenty v praktické části. [3] [38]

[3] Seznam všech funkcí včetně výsledků autorova testování aproximace pomocí náhodné regrese lesa je k dispozici na: https://dev.heuristiclab.com/trac.fcgi/blog/gkronber/symbolic_regression_benchmark

4 PRAKTICKÁ ČÁST

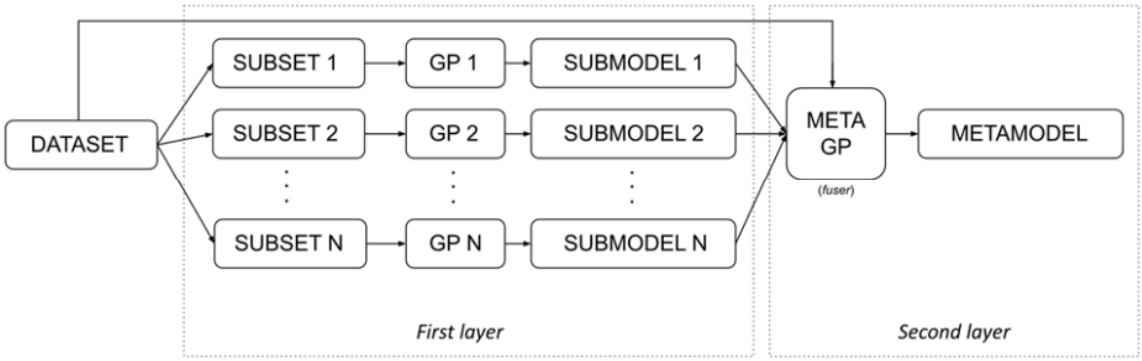
4.1 Dvouvrstvé genetické programování

Jedná se o návrh, který vznikl jako inspirace z úspěchu metod sborového učení (ensemble learning) z oblasti strojového učení. Princip spočívá v tom, že se genetické programování rozdělí na 2 vrstvy. Kombinace těchto vrstev by měla pomoci GP využít vyhledávací prostor a zvýšit tak výslednou přesnost. V první vrstvě se zároveň spustí více genetických programování, které mají za cíl vygenerovat submodely, které se poté použijí jako terminály ve druhé vrstvě. V druhé vrstvě se tyto submodely, získané z běhů genetických programování v první vrstvě použijí jako jakési stavební bloky, které by měly již představovat poměrně přesné řešení daného problému a kombinací těchto bloků bychom měli dosáhnout ještě přesnějšího řešení. [37]

Celé dvouvrstvé genetické programování lze shrnout do následujících kroků:

1. Vytvoření submodelů z nezávislých běhů jednoduchého GP.
2. Přidání vytvořených submodelů do sady terminálů druhé vrstvy dvouvrstvého GP.
3. Jeden běh druhé vrstvy s použitím jednoduchého GP včetně přidaných submodelů.
4. Získání výsledku z druhé vrstvy.

Obě vrstvy mají nezávislé parametry, je tedy pro tyto vrstvy možné (a často i výhodné pro dosažení lepších výsledků) nastavit tyto parametry odlišně. Mezi tyto parametry patří například počet generací, velikost populace, sada terminálů, sada funkcí a další. [37]



Obrázek 9 – Schéma dvouvrstvého genetického programování [37]

Hlavní motivací k vytvoření tohoto algoritmu byla snaha vyhnout se bobtnání a vytváření neustále se zvětšujících stromů bez výraznějšího zlepšení hodnoty fitness tím, že „restartujeme“ GP, ve kterém se vytvořené stromy z běhů GP v první vrstvě použijí jako stavební bloky, takže

se „starý“ kód využije novým konstruktivním způsobem. Tento přístup měl rovněž za cíl zvýšit vyjadřovací schopnost genetického programování tím, že umožní efektivnější využití již existujícího kódu a jeho variant. V neposlední řadě je také možnost paralelního zpracování nezávislých GP v první vrstvě a s tím související zrychlení běhu algoritmu. [37]

4.2 DEAP

DEAP je evoluční výpočetní framework psaný v Pythonu pro rychlé vytváření prototypů a testování jejich přesnosti. Jeho cílem je zprůhlednit algoritmy a datové struktury. Lze jednoduše kombinovat s paralelizačními mechanismy obsaženými v Pythonu, jako je například knihovna *multiprocessing*. Pro naše účely bude framework využit pouze pro genetické programování, nicméně framework podporuje i ostatní typy genetických algoritmů a poskytuje volnost ve formě reprezentací jedinců. [39]

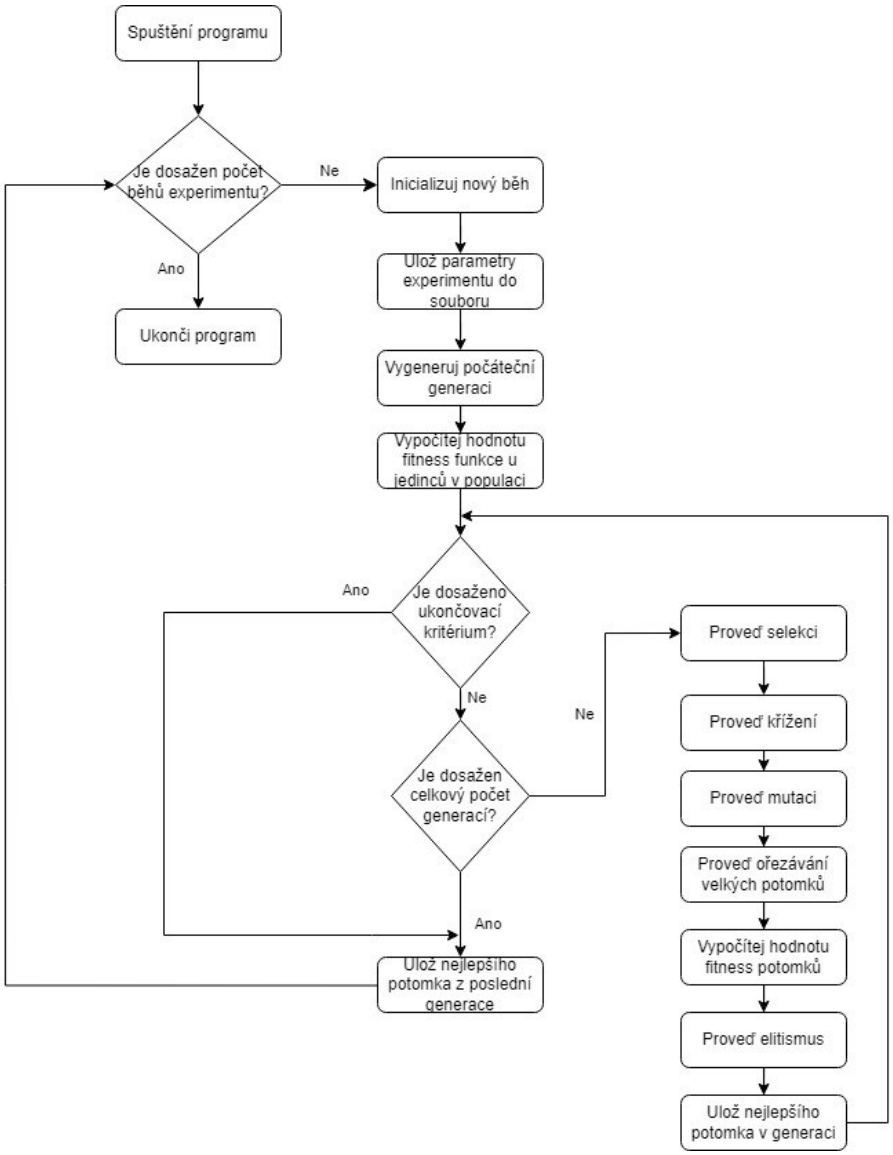
V tomto frameworku místo toho, aby byl programátor limitován předdefinovanými typy a genetickými operátory, je většina funkcí řešena pomocí takzvaného toolboxu. Toolbox je jakýsi kontejner nástrojů, které se dají libovolně nastavit a jednoduše upravovat pro různé experimenty. Je zde obsaženo také několik základních implementací genetických operací, nicméně je zde možno vytvořit vlastní funkce a poté tyto funkce zaregistrovat pomocí reference. Při provádění genetického programování se poté bude přímo pro danou operaci volat naše vlastní funkce. Jediná limitace je taková, že například pro křížení musí daná funkce přijímat dva argumenty, které specifikují dva rodiče a návratová hodnota musí obsahovat dvojici potomků. Není tedy například možné vytvořit křížení mezi více než dvěma rodiči apod. [39]

Pro potřeby genetického programování je k dispozici primitivní set, který obsahuje všechny vstupní argumenty, množinu terminálů a množinu funkcí společně s jejich aritou. [40]

Jedna z nevýhod tohoto frameworku je bohužel nemožnost provádění běhů na GPU, proto jsou běhy algoritmů poměrně časově náročné.

4.3 Vývojové diagramy vyvinutých algoritmů

V této podkapitole si představíme vývojové diagram vyvinutých algoritmů na genetické programování. Proces je u všech experimentů obdobný, proto budou zobrazeny pouze dva diagramy. Jeden diagram, který bude vyobrazovat jednovrstvý přístup (standardní GP) a druhý, který bude zobrazovat dvouvrstvý přístup.



Obrázek 10 – Vývojový diagram jednovrstvého GP, zdroj: vlastní

Nejprve tedy začneme jednovrstvým GP. Každý běh má stanoven určitý počet experimentů. To z toho důvodu, abychom mohli získat dostatek dat pro danou konfiguraci parametrů a tím pádem byli schopni pozorovat, jaké efekty na fitness jedinců dané změny parametrů způsobují. Pro většinu běhů bylo zvoleno číslo 500, které se z původních experimentů jevilo jako dostatečné, což bylo potvrzeno i metodou sledování ustálení kumulativního průměru.

V dalším kroku proběhne příprava nového běhu algoritmu, kde se vytvoří nová složka, do které se budou ukládat všechny výsledky spojené s daným experimentem. S těmito výsledky se také uloží všechny hodnoty parametrů z toho důvodu, aby je bylo možné později jednoduchým způsobem kategorizovat a analyzovat. Součástí této přípravy je také vygenerování počáteční populace a výpočet hodnot fitness jedinců, kteří spadají do této „nulté“ generace.

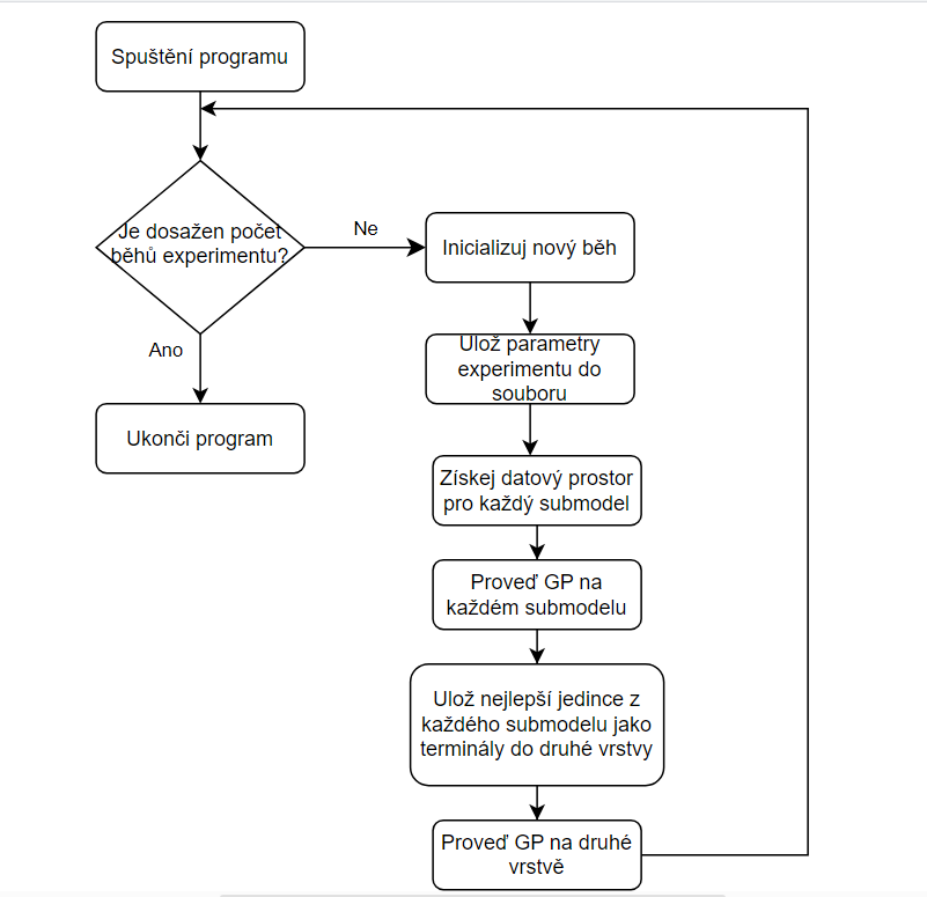
Poté již přichází na řadu samotný běh genetického algoritmu, který se bude opakovat, dokud nebude vyčerpán počet generací alokován na daný běh algoritmu, nebo není dosažena podmínka ukončení. Podmínka ukončení byla nastavena na událost, kdy se v aktuální populaci nachází perfektní řešení daného problému (jedinec má nejlepší možnou hodnotu fitness funkce).

Běh samotného GP se skládá z několika tradičních operátorů (selekce, křížení, mutace, evaluace). K těmto genetickým operátorům byl také přidán operátor ořezávání, který kontroluje, zda má daný potomek, který vznikl z předešlých operátorů, větší hloubku, než je specifikovaná v parametrech běhu a pokud ano, tak nalezne všechny uzly, které se nachází v této specifikované hloubce a všechny tyto uzly vymění za terminály. Tato operace tedy způsobí, že se na dané úrovni budou nacházet pouze listy, protože terminály žádné další potomky mít nemohou a celková hloubka jedince se tedy zmenší.

V poslední řadě byla do běhu GP přidána implementace elitismu, která byla využívána ve velkém množství v předchozích studiích a ukázala se jako přínosná.

Součástí běhu je také ukládání nejlepšího potomka v každé generaci, aby bylo možno vidět, zda dochází k vylepšení populace v průběhu dalších generací a také nastavení ukončovací podmínky, která může ukončit provádění genetického programování dříve, než je dosažen limit generací. Tato podmínka je nastavena z důvodu zamezení zbytečného generování dalších generací běhů, které již dosáhly stanovenou hodnotu fitness.

Nyní přejdeme k zobrazení vývojového diagramu pro dvouvrstvé genetické programování.



Obrázek 11 – Vývojový diagram dvouvrstvého GP, zdroj: vlastní

V tomto diagramu již není zobrazen celý běh GP jak tomu bylo u jednovrstvého GP, abychom již nezobrazovali informace, které byly představeny u jednovrstvého GP.

Vývojový diagram pro dvouvrstvý model je obdobný s jednovrstvým, obsahuje ale několik rozdílů. V první řadě se v tomto typu GP nachází operace „Získej datový soubor pro každý submodel“, kde se získá datový set, na kterém se bude daný GP v první vrstvě provádět. Postup, jak se tento datový prostor získává je mezi experimenty odlišný. Používá se například metoda bootstrappingu, samplingu a v některých případech je tento krok úplně přeskočen a používá se stejný datový set jako u následující druhé vrstvy. Všechny modely první vrstvy je možné vykonávat paralelně, protože na sobě nejsou žádným způsobem závislé. Jakmile jsou všechny GP v první vrstvě dokončeny, předají se do inicializace druhé vrstvy všichni nejlepší jedinci, kteří byli získání z provádění první vrstvy, jako terminály. Poté už je možno začít provádění druhé vrstvy ze které vzejde finální výsledek. Tento proces je opakován obdobně jako u jednovrstvého GP do stanoveného počtu experimentů.

5 EXPERIMENTY

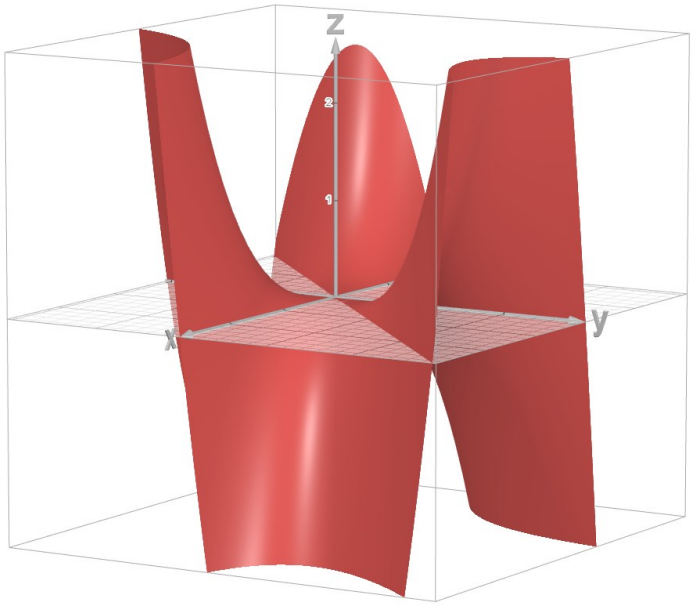
V této kapitole budou prováděny experimenty, které budou mít za cíl porovnat jednovrstvé genetické programování s dvouvrstvým. Pro toto porovnání bylo zvoleno několik problémů na aproximaci pomocí symbolické regrese.

5.1 Jednoduchá funkce s dvěma proměnnými

První datová sada, která byla zvolena pro aproximaci, je následující rovnice (2)

$$(x * y) * (y - x) \qquad (2)$$

Tato rovnice byla převzata z článku The Evolutionary Buffet Method od autorů Arend Hintze, Jory Schossau a Clifford Bohm, kde byla použita jako jeden z benchmarkových problémů pro jejich vyvíjený algoritmus.



Obrázek 12 – 3D reprezentace první aproximované rovnice, zdroj: vlastní pomocí [41]

5.1.1 Iniciální parametry

V této kapitole budou představeny všechny iniciální parametry použité pro počáteční běhy. Parametry společné pro všechny základní běhy jsou zobrazeny v tabulce níže.

Tabulka 1 – Obecné parametry pro aproximaci první rovnice, zdroj: vlastní

Použité křížení	Křížení podstromů s 2 výslednými potomky
Použitá mutace	Jednobodová mutace
Použitá selekce	Turnajový výběr
Velikost turnaje	2
Počet elit	1
Pravděpodobnost na křížení	0.9
Pravděpodobnost na mutaci	0.01
Maximální hloubka stromu při provádění genetických operací	17
Minimální hloubka stromu při inicializaci	2
Maximální hloubka stromu při inicializaci	6
Minimální hodnota *x* a *y* pro aproximaci	-2
Maximální hodnota *x* a *y* pro aproximaci	2
Velikost kroku, po kterém jsou body aproximovány	0.1
Seznam terminálů	Konstanty -1.0, 1.0, 2.0, 3.0 a 2 vstupní proměnné

Nejprve bude proveden základní běh jednoduchého jednovrstvého genetického algoritmu, který bude sloužit jako základní měřítko, které bude určovat, zda je dvouvrstvá reprezentace v porovnání s jednovrstvou výsledkově horší nebo lepší. Každé nastavení parametrů genetického programování bude vyzkoušeno na 500 nezávislých bězích. Tento počet běhů se zdál jako dostatečný pro ustálení průměrné kvadratické chyby u testovacích běhů provedených na tomto problému.

Pro jednovrstvé genetické programování byly zvoleny následující parametry.

Tabulka 2 – Specifické parametry pro jednovrstvé genetické programování u prvního problému, zdroj: vlastní

Velikost populace	200
Počet generací	100
Seznam funkcí	+, -, *, *sqrt*, *pow2*, *pow3*

Ostatní parametry je možno vidět v tabulce znázorňující společné parametry pro všechny typy algoritmů.

Pro dvouvrstvou reprezentaci budou zvoleny jako počáteční bod parametry vyobrazeny v následující tabulce.

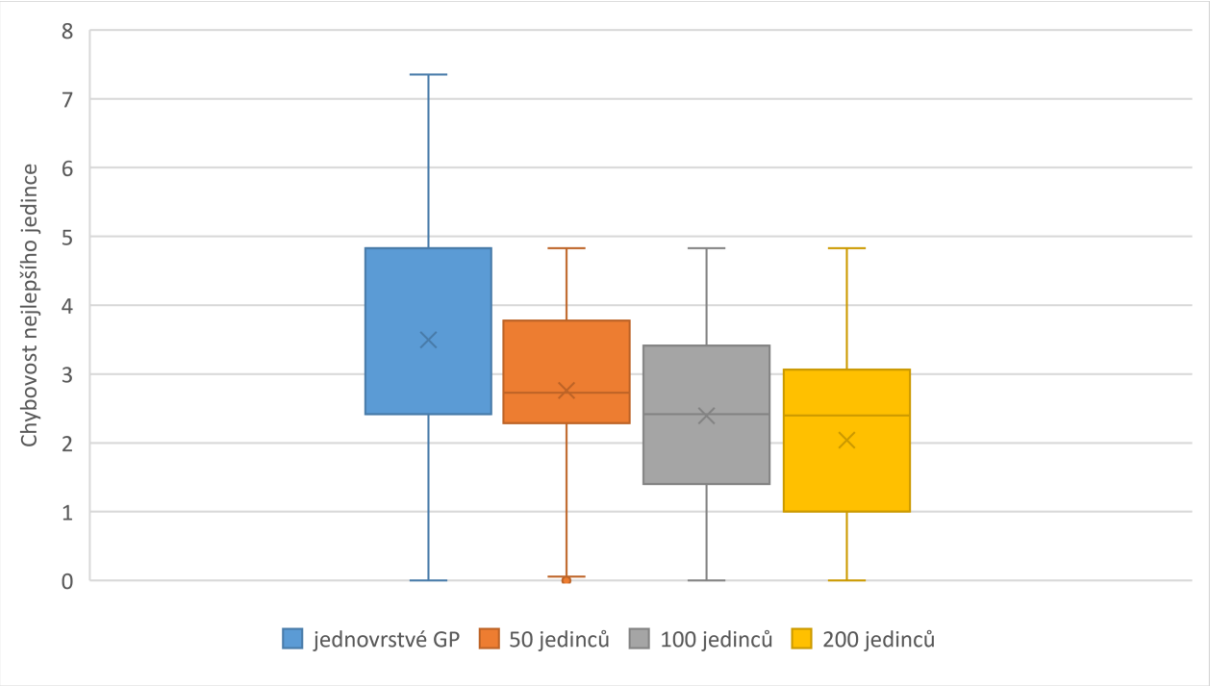
Tabulka 3 – Specifické parametry pro dvouvrstvé genetické programování u prvního problému, zdroj: vlastní

Počáteční velikost populace v první vrstvě	50
Počet generací v první vrstvě	3
Počet submodelů a subsetů	30
Seznam funkcí v první vrstvě	+, -, *
Seznam funkcí ve druhé vrstvě	+, -, *, *sqrt*, *pow2*, *pow3*
Velikost populace ve druhé vrstvě	200
Počet generací ve druhé vrstvě	10

5.1.2 Experimenty s velikostí populace v první vrstvě

Nejprve budou prováděny experimenty s velikostí populace v první vrstvě. Jako základ byl stanoven počet jedinců v jednom subsetu v první vrstvě na hodnotu 50 jedinců. Pokud bychom si tedy spočítali celkový počet evaluací jedinců v první vrstvě (počet submodelů * počet generací * velikost populace) dostali bychom hodnotu 4500, když tuto hodnotu sečteme s počtem evaluací v druhé vrstvě dostaneme 4500 + 2000 = 6500. Když spočítáme počet evaluací v jednovrstvém GP, dostaneme hodnotu 20 000. Budeme se tedy pokoušet zjistit, zda bude zvyšování velikosti populace v první vrstvě, dokud nedosáhneme celkové hodnoty evaluací rovné jednovrstvému GP, zlepšovat celkovou úspěšnost dvouvrstvého GP.

Byly provedeny celkem 3 různé experimenty s počtem jedinců v první vrstvě nastaveným na hodnoty 50, 100 a 200. Níže je možno vidět výsledky všech těchto experimentů.



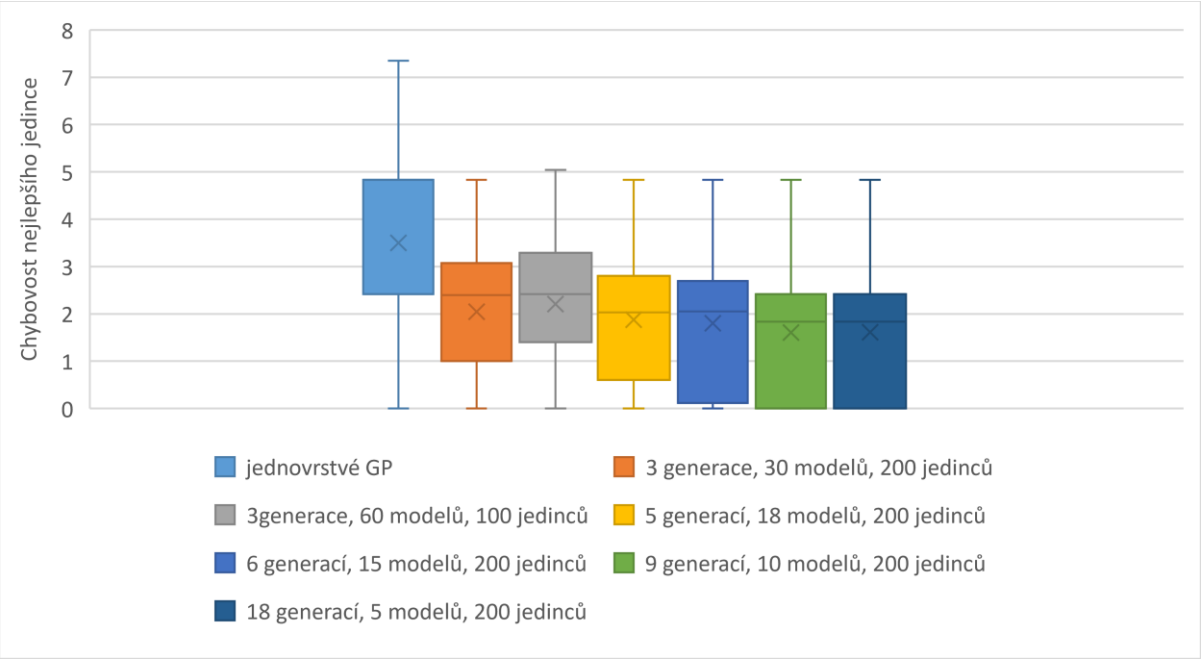
Obrázek 13 – Porovnání jednovrstvého GP se zvyšujícím se počtem jedinců v první vrstvě u dvouvrstvého přístupu, zdroj: vlastní

Z obrázku je možno vidět, že se zvyšujícím se počtem jedinců v první vrstvě u dvouvrstvého GP se průměrná chybovost modelu postupně snižuje. Při zvýšení velikosti populace na hodnotu 200 dosáhneme nejlepšího výsledku, proto pro další experiment použijeme právě tuto hodnotu s cílem dosáhnout ještě lepších výsledků.

5.1.3 Experimenty s počty submodelů

V dalších experimentech se pokusíme optimalizovat počet submodelů, které budou použity v první vrstvě. Jako počáteční bod si stanovíme parametry z předchozího experimentu, nastavíme tedy velikost populace na hodnotu 200 a počet generací v první vrstvě na hodnotu 3. Pro různé počty submodelů budeme muset hodnotu těchto parametrů měnit, abychom dosáhli stejných počtů evaluací a měli tak možnost porovnat jednovrstvé GP s dvouvrstvým přístupem.

Níže je možno vidět porovnání všech experimentů, které byly provedeny s myšlenkou změny počtů submodelů v první vrstvě.



Obrázek 14 – Porovnání různých počtů submodelů v první vrstvě u dvouvrstvého GP, zdroj: vlastní

Nejprve byl proveden pokus, ve kterém se počet modelů zvýšil na hodnotu 60, protože by byl počet evaluací dvojnásobný než původní přístup s 30 modely, bylo nutné také zmenšit počet jedinců v populace u GP v první vrstvách, proto byla tato hodnota nastavena na 100. Nicméně zvýšení počtu submodelů nepřineslo snížení průměrné chybovosti nejlepšího jedince, ale způsobilo pravý opak, tedy zhoršení. Dále bylo tedy vyzkoušeno snížení počtu těchto modelů. Zde bylo nutné změnit počet generací v první vrstvě, aby byl dodržen stejný počet generací s jednovrstvým přístupem, počet jedinců u těchto experimentů, kde byl celkový počet modelů snižován zůstával neměnný na hodnotě 200. Postupným snižováním je možno vidět, že se průměrná chyba s menším počtem modelů snižuje zejména díky většímu počtu generací a s tím větší přesností výsledných jedinců první vrstvy. Po zmenšení počtu modelů na 10 již další zmenšování neposkytlo výrazně lepší výsledek, proto pro další experiment budou změněny parametry první vrstvy na 10 submodelů a 9 generací v první vrstvě.

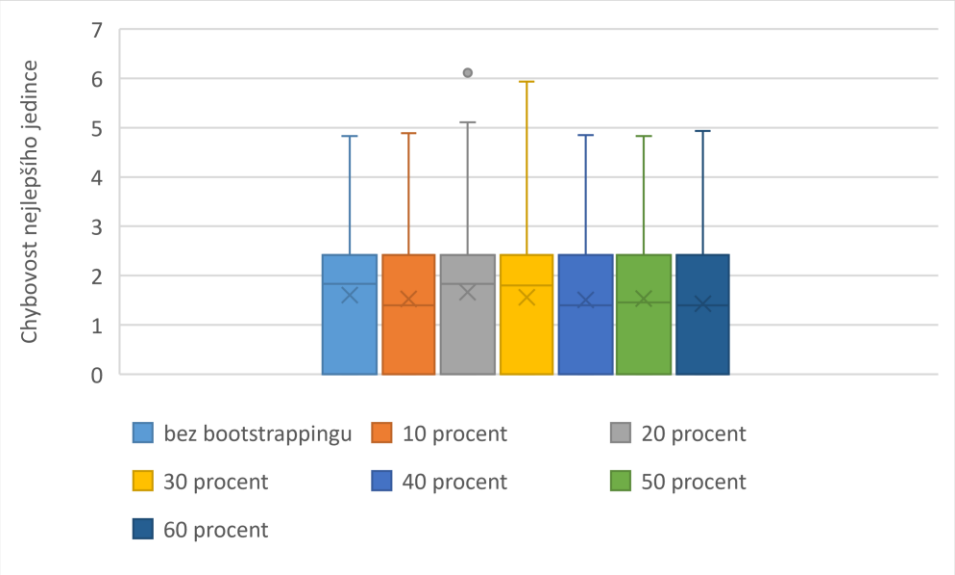
5.1.4 Experimenty s bootstrappingem

Další parametr, pomocí kterého se dále pokusíme zmenšit chybovost dvouvrstvého GP je bootstrapping.

Ve statistice a strojovém učení je bootstrapping technikou opakovaného výběru vzorků, která zahrnuje opakovaný výběr vzorků ze zdrojových dat s možností jejich nahrazení, Výrazem "s nahrazením" máme na mysli, že stejný datový bod může být zahrnut do našeho vzorkovaného souboru dat vícekrát. [42]

V našem případě provedeme 6 experimentů s procentem bootstrappingu nastaveném na 10, 20, 30, 40, 50 a 60 %. Hodnoty získané z těchto experimentů porovnáme s výsledkem nejlepší konfigurace z předchozího příkladu, kde jsme vytvářeli submodely bez bootstrappingu.

Níže je možno vidět porovnání všech těchto metod.



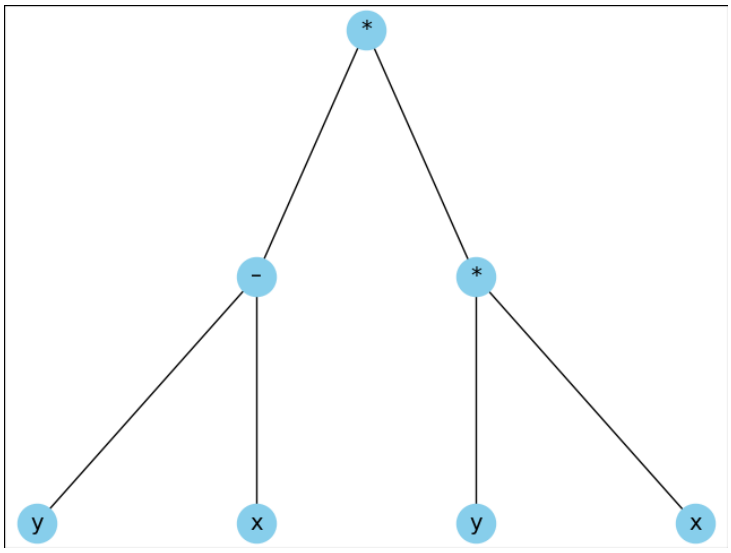
Obrázek 15 – Porovnání různých procent bootstrappingu u dvouvrstvého GP, zdroj: vlastní

Z názorného porovnání chybovostí všech těchto přístupů je možno vidět, že nebylo v žádném z přístupů dosaženo žádného výrazného zlepšení a krabicové grafy vypadají téměř totožně. Nicméně nejlepší hodnoty průměru se nachází v přístupu s 60 procentech bootstrappingu.

5.1.5 Programy s nejlepší hodnotou fitness

Pro zajímavost si v této podkapitole zobrazíme stromy, které během všech provedených experimentů dosáhly nejlepší hodnoty fitness, což v tomto případě je hodnota 0, protože se v žádném aproximovaném bodě nelišily s původní funkcí.

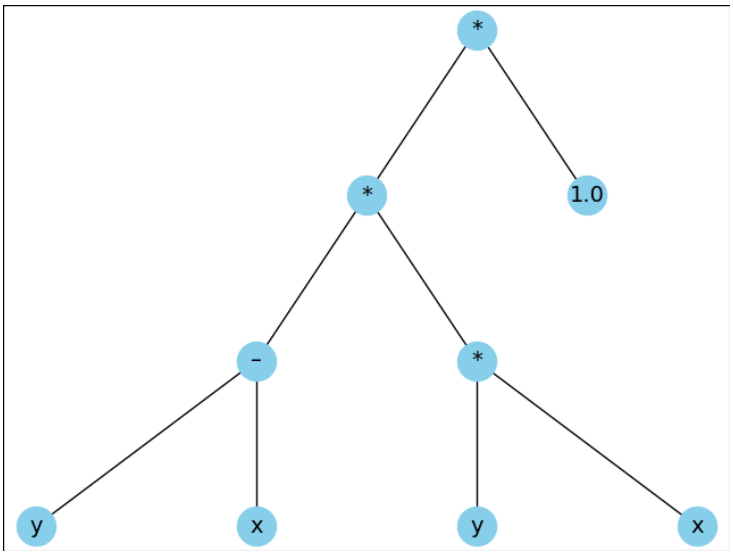
Nejprve si představíme strom, který přesně určuje rovnici, kterou se snažíme aproximovat.



Obrázek 16 – Nejlepší dosažitelný jedinec pro rovnici 1, zdroj: vlastní

Samozřejmě, protože násobení je komutativní operace, nezáleží na pořadí *x* a *y* a výsledek bude stejný s aproximovanou rovnicí.

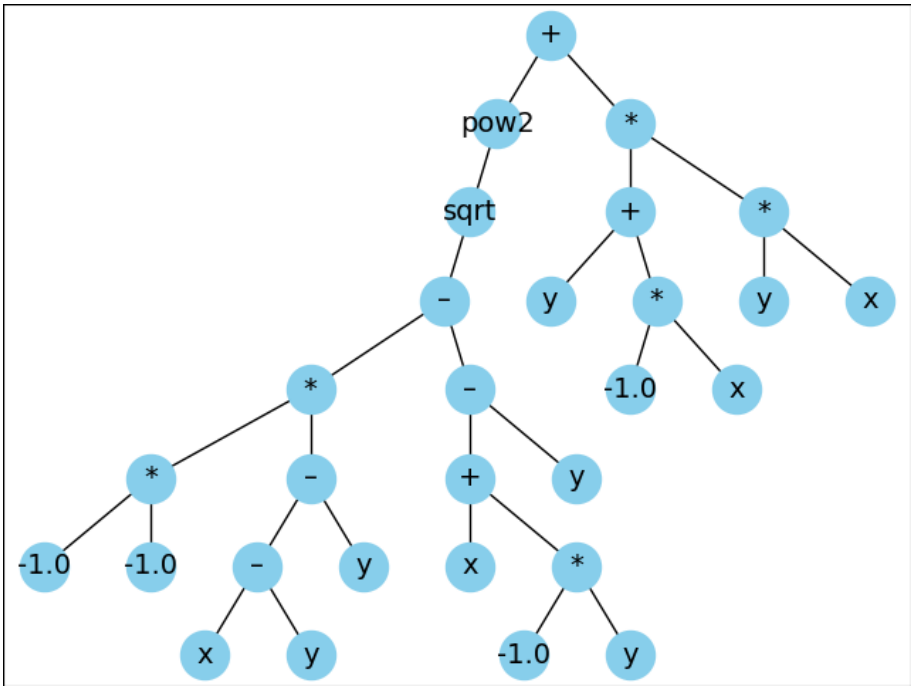
Často ale genetické programování nekonverguje přesně k tomuto řešení, většinou mají výsledné stromy větší hloubku a obsahují zbytečné operace, které žádným způsobem nezmění výsledek. Například v následujícím příkladu se u daného jedince, který také dosáhl nejlepší možné hodnoty fitness, levý podstrom násobí 1, což poskytne stejný výsledek, a tudíž je zbytečná operace.



Obrázek 17 – Jedinec se zbytečným násobením u rovnice 1, zdroj: vlastní

Těchto zbytečných operací je ve stromech často více a někdy je jich takové množství, že na první pohled nejsme schopni jednoznačně konstatovat, jestli daný strom skutečně poskytuje

řešení k danému problému, nebo ne. Příklad stromu se spousty zbytečných operací je možno vidět níže.



Obrázek 18 – Jedinec se spoustou zbytečných operací u rovnice 1, zdroj: vlastní

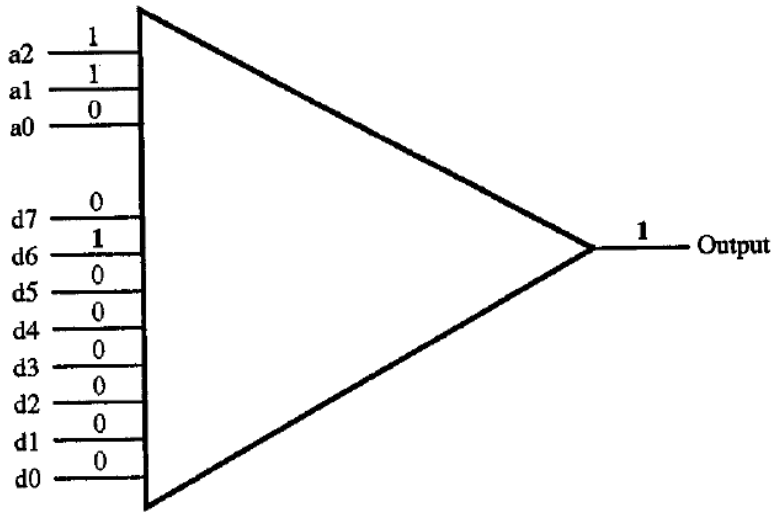
Pokud bychom daný strom blíže prozkoumali, zjistili bychom, že celá levá část stromu od kořene přes metodu *pow2* atd. se rovná hodnotě 0 a výsledná rovnice se tedy získá pouze z pravého podstromu. Ve stromě také můžeme vidět spoustu operací, které by bylo možno zjednodušit (například násobení -1 * -1 by se mohlo rovnou zapsat jako 1) a také operace, které vlastně nic nedělají (například druhá mocnina z odmocniny).

5.2 Booleovský multiplexor

Multiplexor je kombinační logický obvod určený k přepínání jednoho z několika vstupních vedení na jedno společné výstupní vedení.

Multiplexování je obecný termín používaný k popisu posílání jednoho nebo více analogových nebo digitálních signálů přes společnou přenosovou linku v různé časy nebo rychlosti a zařízení, které toto zprostředkovává se nazývá multiplexor. [43]

Pro prostředky genetického programování si je možno multiplexor představit jako určitý počet vstupních bitů, které se skládají z k adresních bitů a 2^k datových bitů ze kterých vzejde $n = k + 2^k$ multiplexor. Níže je možno vidět reprezentaci jedenácti vstupového multiplexoru, na kterém budou další vlastnosti konkrétně vysvětleny.



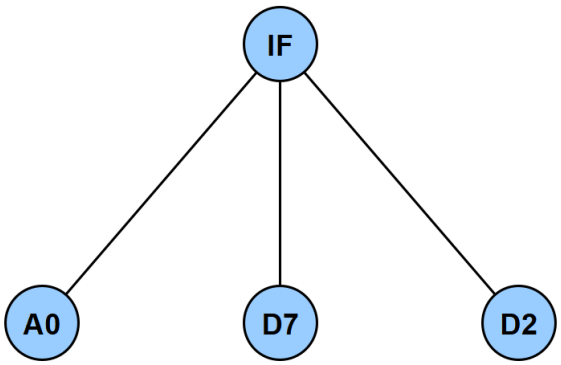
Obrázek 19 – Reprezentace jedenácti vstupového booleovského multiplexoru [17]

Pro aproximaci pomocí genetického programování bude použit právě tento multiplexor. Tento typ obsahuje 3 adresní bity, které rozhodují o tom, jaký z datových bitů bude použit pro výstup z multiplexoru. V tomto případě je adresních bitů celkem 8, z důvodu povahy multiplexoru, která byla již popsána výše. Jak můžeme z obrázku vidět, v tomto případě jsou bity *a1* a *a2* rovny 1, to znamená, že pokud převedeme binární číslo 011, které vzejde z adresních bitů na číslo v desítkové soustavě, dostaneme číslo 6. Toto číslo odpovídá indexu datového bitu, který bude sloužit jako výstup z multiplexoru. Zaměříme se tedy na datový bit *d6*. Tento bit v daném příkladu nabývá hodnoty 1 a tím pádem bude i výstup z multiplexoru roven této hodnotě.

5.2.1 Implementace funkce IF v S-výrazovém stromě

Před samotnými experimenty se nejprve musí vyřešit implementace funkce *if*. Jelikož se jedná o problém, ve kterém se celý strom skládá z booleovských funkcí a proměnných, lze funkci *if* implementovat jednoduše jako funkci s aritou rovnou 3, kde první potomek bude určovat podmínku a pokud tato podmínka bude platit, tak se provede druhý potomek. Pokud tato podmínka neplatí tak se provede potomek třetí. [17]

Jednodušší to možná bude pochopit na praktickém příkladu. Představme si jednoduchý strom s kořenem a třemi potomky.



Obrázek 20 – Příklad stromu s funkcí *if,* zdroj: vlastní

V tomto příkladu bude funkce *if* fungovat tak, že pokud bude potomek *A0* pravdivý, potom bude výsledek funkce uzel *D7*. Pokud ale bude *A0* nepravdivý, potom bude výsledek funkce uzel *D2*.

5.2.2 Iniciální parametry

Stejně jako u předchozího problému si v tabulkové formě zobrazíme všechny iniciální parametry, použité pro aproximaci booleovského multiplexoru.

Tabulka 4 – Obecné parametry pro problém aproximace multiplexoru, zdroj: vlastní

Použité křížení	Křížení podstromů s 2 výslednými potomky
Použitá mutace	Jednobodová mutace
Použitá selekce	Turnajový výběr
Velikost turnaje	2
Minimální hloubka stromu při inicializaci	2
Maximální hloubka stromu při inicializaci	6
Počet elit	1
Maximální hloubka stromu při provádění genetických operací	17
Pravděpodobnost na mutaci	0.05
Pravděpodobnost na křížení	0.9
Seznam funkcí	*AND*, *OR*, *NOT*, *IF*
Seznam terminálů	11 vstupních argumentů (3 adresové, 8 datových)

Pro jednovrstvé genetické programování zaměřené na tento problém budou k obecným parametrům přidány ještě parametry následující, které jsou specifické právě pro tento typ:

Tabulka 5 – Specifické parametry pro jednovrstvé genetické programování u druhého problému, zdroj: vlastní

Počet generací	100
Velikost populace	200

Pro dvouvrstvou reprezentaci budou zvoleny jako počáteční bod parametry vyobrazeny v následující tabulce.

Tabulka 6 – Specifické parametry pro dvouvrstvé genetické programování u druhého problému, zdroj: vlastní

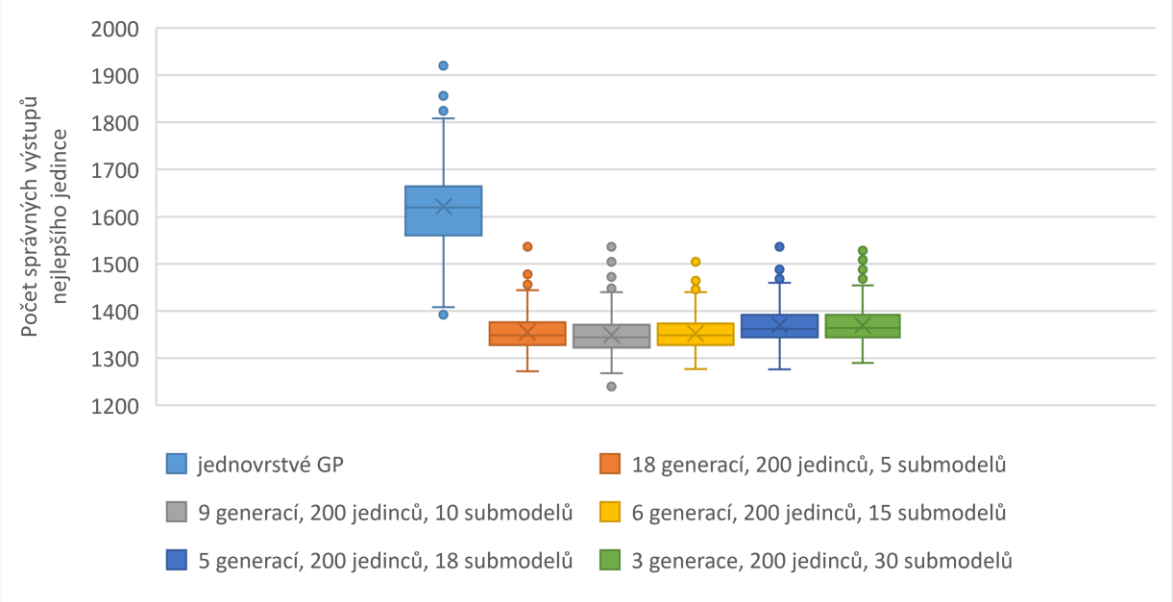
Počáteční velikost populace v první vrstvě	200
Počet generací v první vrstvě	3
Počet submodelů a subsetů	15
Velikost populace ve druhé vrstvě	200
Počet generací ve druhé vrstvě	10

Stejně jako u prvního problému i u tohoto problému je kladen důraz na to, aby byly oba přístupy porovnatelné. Celkový počet generací u prvního přístupu je tedy stejný s počtem generací u přístupu druhého a totéž platí pro celkový počet provedených evaluací jedinců u kterého se o tomto faktu lze ujistit provedením podobného výpočtu jaký byl načrtnut u prvního problému.

5.2.3 Počáteční experimenty

Nejprve budou provedeny počáteční experimenty, které se nebudou soustředit na žádný specifický parametr, abychom nalezli nejlepší počáteční nastavení parametrů, které bude sloužit pro další experimenty.

Nastavení pro jednovrstvé GP nebude v průběhu experimentů žádným způsobem alterováno, proto bude platit nastavení které bylo popsáno v předchozí kapitole.



Obrázek 21 – Porovnání výsledků počátečních experimentů problému multiplexoru, zdroj: vlastní

Před samotným porovnáním je nutné podotknout, že hodnoty fitness jsou u tohoto typu problému vypočítány podle počtů správných výstupů nejlepšího jedince, proto jsou pro nás

větší hodnoty fitness v tomto příkladu „lepší“. Toto bude platit u všech dalších experimentů, které se budou týkat tohoto problému.

Z porovnání je možno vidět, že všechny vyzkoušené iniciální parametry dvouvrstvého genetického programování jsou o poznání horší než jednovrstvé programování. Nejlepší výsledek pro dvouvrstvé genetické programování vzešel z kombinace 18 submodelů, 200 jedinců a 5 generací pro první vrstvu, proto budou původní nastavení parametrů pro další experiment změněny na tyto hodnoty.

Tabulka 7 – Změněné parametry pro první vrstvu po iniciálních experimentech, zdroj: vlastní

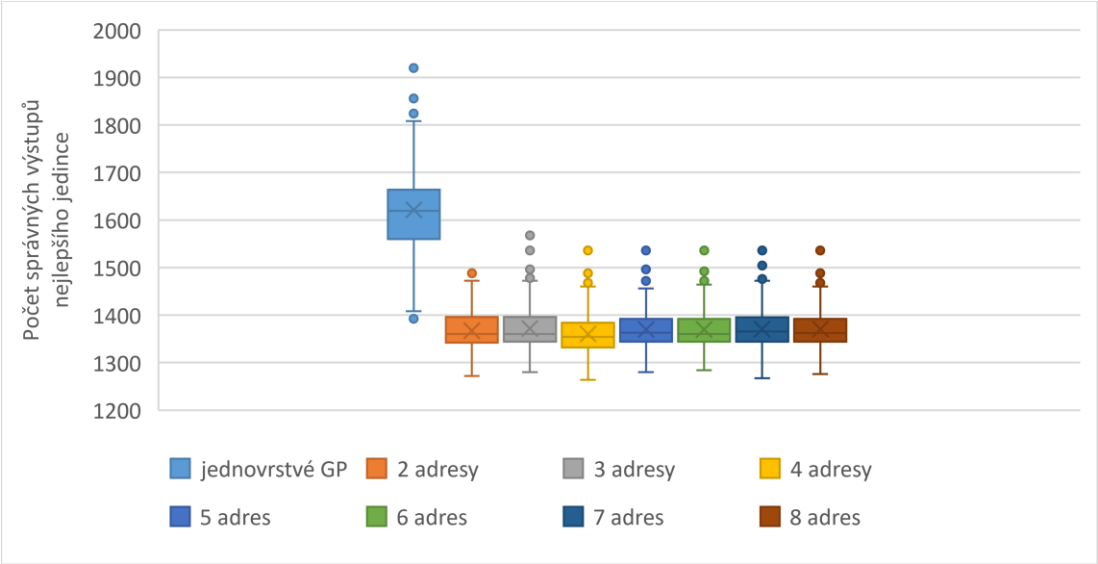
Počáteční velikost populace v první vrstvě	200
Počet generací v první vrstvě	5
Počet submodelů a subsetů	18

5.2.4 Porovnání různých počtů adres použitých pro aproximaci submodelů v první vrstvě

V předchozích experimentech bylo v první vrstvě vždy použito všech 8 adresních prostorů pro generování submodelů a díky relativně nízkému počtu generací oproti jednovrstvému genetickému programování byly získané submodely horší z hlediska hodnoty fitness. Tyto experimenty se budou snažit zjistit, zda by nebylo výhodné aproximovat submodely první vrstvy na limitovanou množinu adres. Tímto způsobem by mohly mít modely v první vrstvě možnost vygenerovat lepší množinu terminálů pro druhou vrstvu.

Pro přiřazení adresních prostorů bude využita variace samplingu, která bude zajišťovat to, že budou všechny adresní prostory rovnoměrně rozděleny mezi modely. Tato modifikace byla vytvořena z důvodu zajištění, aby byly všechny adresní prostory v první vrstvě v nějakém modelu zahrnuty a nemohlo se tak stát že například adresní prostor *010* by nebyl v žádném modelu přítomen, což by u tohoto problému mohlo být nežádoucí.

Níže je možno vidět porovnání výsledků pro různé počty adres pro aproximaci v modelech první vrstvy.



Obrázek 22 – Porovnání výsledků různých počtů aproximovaných adres v první vrstvě u problému multiplexoru, zdroj: vlastní

Jak je vidět z porovnání výše, změna počtu adres pro aproximaci submodelů první vrstvy bohužel nezpůsobila žádné výrazné zlepšení.

5.2.5 Přidání více generací do druhé vrstvy

Z důvodu neúspěšných výsledků z předchozích experimentů bude v tomto experimentu vyzkoušeno zvýšení celkového počtu generací a s tím souvisejících evaluací ve druhé vrstvě.

Tabulka 8 – Změněné parametry u první vrstvy pro experiment s přidáním více generací do druhé vrstvy u problému multiplexoru, zdroj: vlastní

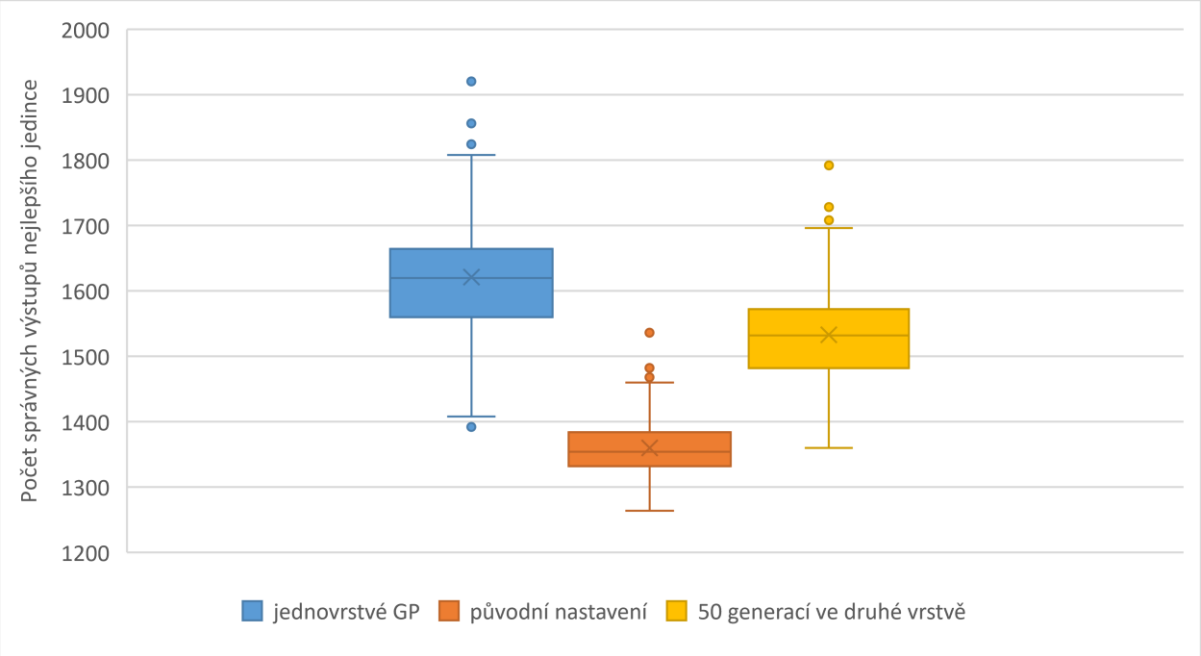
Velikost populace v první vrstvě	200
Počet generací v první vrstvě	10
Počet submodelů a subsetů	5

Tabulka 9 – Změněné parametry u druhé vrstvy pro experiment s přidáním více generací do druhé vrstvy u problému multiplexoru, zdroj: vlastní

Velikost populace v druhé vrstvě	200
Počet generací v druhé vrstvě	50

Počet generací u modelů první vrstvy byl nastaven na hodnotu 10 a počet submodelů byl snížen na hodnotu 5. U druhé vrstvy byl naopak počet generací zvýšen na hodnotu 50.

Níže je možno vidět výsledek, který byl získán pomocí těchto nových parametrů pro 4 aproximované adresy u první vrstvy společně s nejlepším výsledkem původního nastavení, tedy 5 generací, 18 submodelů a 200 jedinců v první vrstvě.



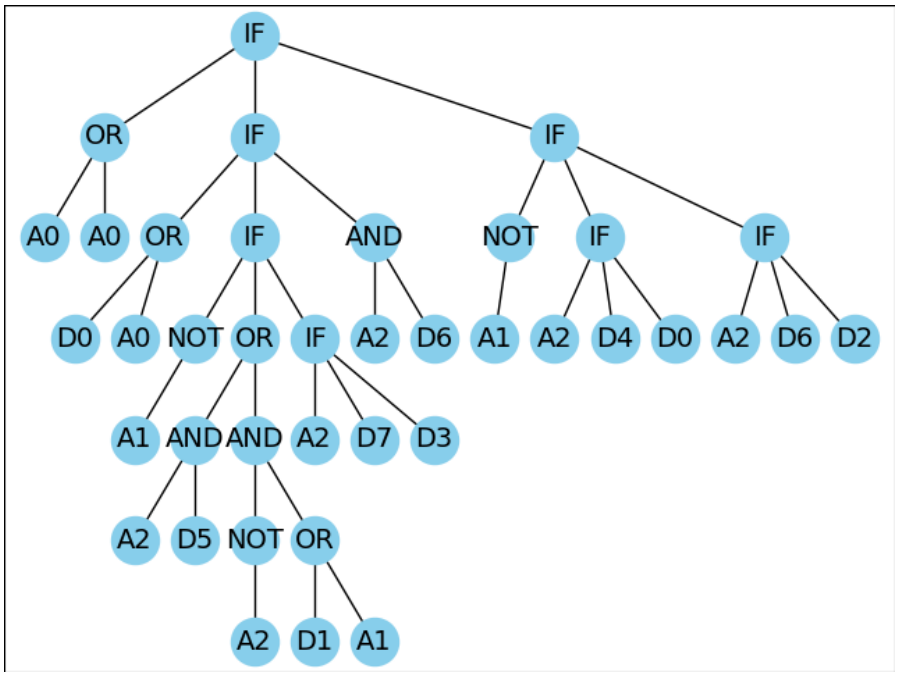
Obrázek 23 – Porovnání výsledků při větším počtu generací ve druhé vrstvě u problému multiplexoru, zdroj: vlastní

Jak je vidět z porovnání, přidání více generací do druhé vrstvy skutečně poskytlo celkem výrazné zlepšení a výsledky jsou již alespoň porovnatelné s jednovrstvou verzí i přesto je ale jednovrstvé genetické programování v tomto případě mírně lepší. Bohužel jsme tedy u tohoto problému nebyli schopni nalézt takovou konfiguraci dvouvrstvého genetického programování, která by poskytovala lepší výsledky než jednovrstvé GP.

5.2.6 Program s nejlepší hodnotou fitness

Maximální hodnotu fitness, kterou jedinec mohl pro tento problém dosáhnout, je hodnota 2048, kde výsledný program vrátí správnou hodnotu pro všech 2048 možných kombinací.

Níže je možné vidět jedince, který měl ze všech dosažených perfektních řešení zároveň i nejmenší hloubku a je proto nejefektivnější.



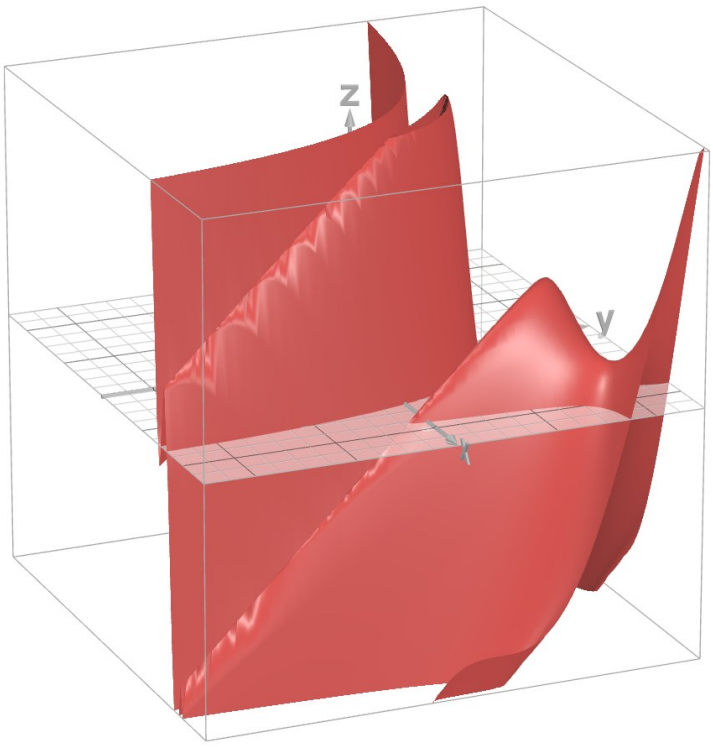
Obrázek 24 – Nejlepší nalezený jedinec pro řešení problému multiplexoru, zdroj: vlastní

5.3 Rovnice obsahující goniometrické funkce

Další datová sada, která byla zvolena pro aproximaci bude obsahovat goniometrickou funkci sinus. Bude se jednat o následující rovnici.

$$(x-3)*(y-3)+2*sin((x-4)*(y-4)) \qquad (3)$$

Tato rovnice byla převzata z článku Order of Nonlinearity as a Complexity Measure for Models Generated by Symbolic Regression via Pareto Genetic Programming od Ekateriny Vladislavlevy včetně výchozích parametrů pro jednovrstvé genetické programování z kterých se u našich experimentů vycházelo. [44]



Obrázek 25 – 3D reprezentace aproximované rovnice s goniometrickou funkcí, zdroj: vlastní pomocí [41]

5.3.1 Iniciální parametry

V této kapitole budou představeny všechny parametry, které byly použity pro iniciální experimenty. Všechny parametry společné pro iniciální běhy jsou obdobně jako u předchozích problémů zobrazeny v tabulce, kterou je možno vidět níže.

Tabulka 10 – Obecné parametry použité pro aproximaci rovnice s goniometrickou funkcí, zdroj: vlastní

Použité křížení	Křížení podstromů s 2 výslednými potomky
Použitá mutace	Jednobodová mutace
Použitá selekce	Turnajový výběr
Velikost turnaje	5
Počet elit	1
Pravděpodobnost na křížení	0.95
Pravděpodobnost na mutaci	0.05
Maximální hloubka stromu při provádění genetických operací	8
Minimální hloubka stromu při inicializaci	2
Maximální hloubka stromu při inicializaci	6
Minimální hodnota *x* a *y* pro aproximaci	0.05
Maximální hodnota *x* a *y* pro aproximaci	6.05
Velikost kroku, po kterém jsou body aproximovány	0.25
Seznam terminálů	Konstanty -4.0, -3.0, -2.0, -1.0, 1.0, 2.0, 3.0, 4.0 a 2 vstupní proměnné
Seznam funkcí	+, -, *, /, sqrt, pow2, e^x, e^{-x}, sin, cos

Stejně jako u ostatních problémů bude nejprve proveden základní běh jednoduchého jednovrstvého genetického algoritmu, který bude sloužit jako základní měřítko. Každé nastavení parametrů genetického bude opět vyzkoušeno na 500 nezávislých bězích.

Pro jednoduché genetické programování byly zvoleny následující podrobnější parametry.

Tabulka 11 – Specifické parametry pro jednovrstvé GP pro aproximaci rovnice s goniometrickou funkcí, zdroj: vlastní

Velikost populace	100
Počet generací	250

Pro dvouvrstvou reprezentaci budou zvoleny jako počáteční bod parametry vyobrazeny v následující tabulce.

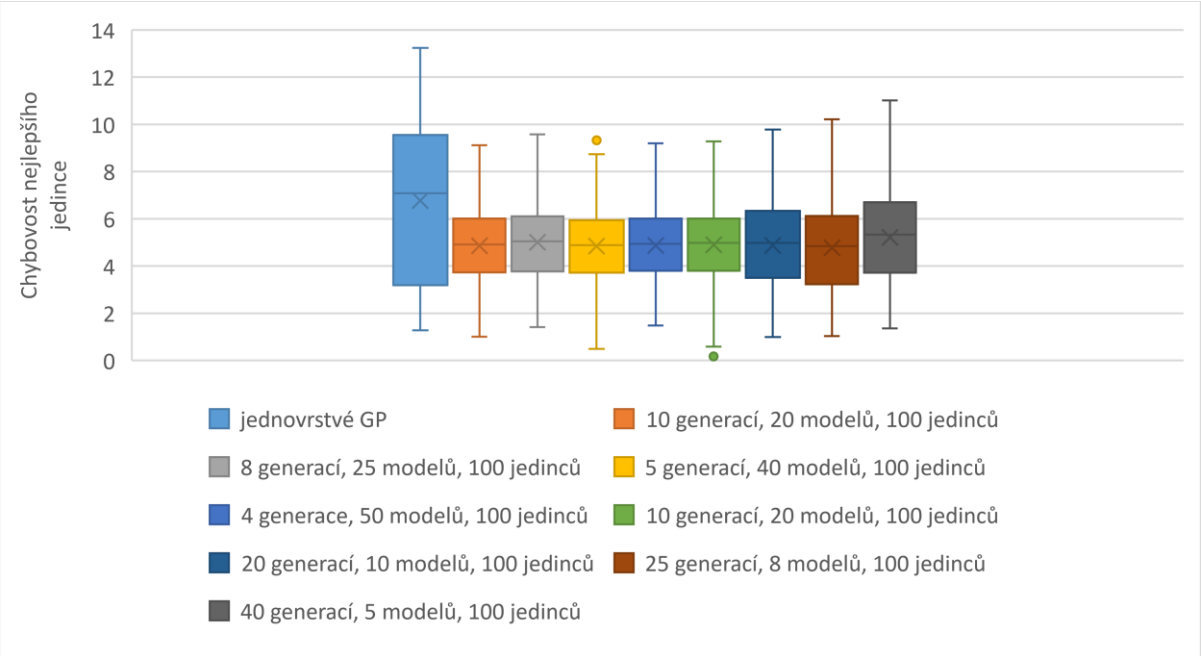
Tabulka 12 – Specifické parametry pro dvouvrstvé GP pro aproximaci rovnice s goniometrickou funkcí, zdroj: vlastní

Počáteční velikost populace v první vrstvě	100
Počet generací v první vrstvě	10
Počet submodelů a subsetů	20
Velikost populace ve druhé vrstvě	200
Počet generací ve druhé vrstvě	25

Parametry byly u těchto iniciálních konfiguracích a dalších experimentů opět zvoleny tak, aby byl celkový počet evaluací jedinců v obou přístupech stejný. U tohoto problému byl nastaven celkový součet generací ve dvouvrstvém přístupu na hodnotu 225, což je o 25 generací méně než u prvního přístupu.

5.3.2 Počáteční experimenty

Nejprve budou provedeny počáteční experimenty, které budou porovnávat různé konfigurace iniciálních parametrů s jednovrstvým genetickým programováním. Z předchozích dvou problémů je zřejmé, že je nutné vyzkoušet velké množství různých kombinací parametrů abychom nalezli co možná nejlepší konfiguraci pro dvouvrstvé genetické programování. Nejprve ale začneme pouze různými kombinacemi počtu generací, submodelů a jedinců v první vrstvě a tyto konfigurace mezi sebou porovnáme.



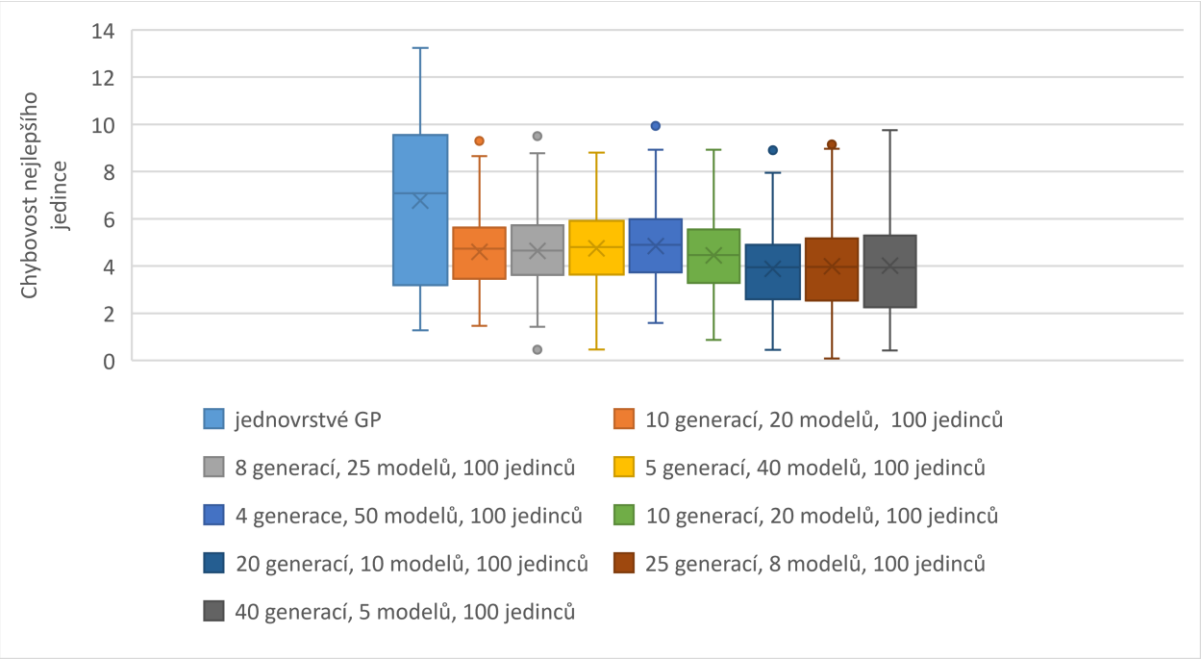
Obrázek 26 – Porovnání počátečních experimentů s různými konfiguracemi první vrstvy u rovnice s goniometrickou funkcí, zdroj: vlastní

Z porovnání, které je možno vidět na obrázku výše je na první pohled vidět, že již naše iniciální experimenty s dvouvrstvým GP mají lepší průměrnou hodnotu fitness nejlepších jedinců než jednovrstvé GP. Jediná vlastnost, kterou má jednovrstvé GP „lepší“ je že první kvartil (spodní stranu boxu v boxplotu) je blíže nule než u dvouvrstvých experimentů. Je také možno vidět, že konfigurace s 10 generacemi, 20 modely a 100 jedinci v první vrstvě (zobrazenou v grafu zelenou barvou) byla schopna získat dva zatím nejlepší dosažené výsledky, jejichž hodnota fitness se nejvíce blíží 0.

5.3.3 Snížení maximální velikosti stromu v první vrstvě

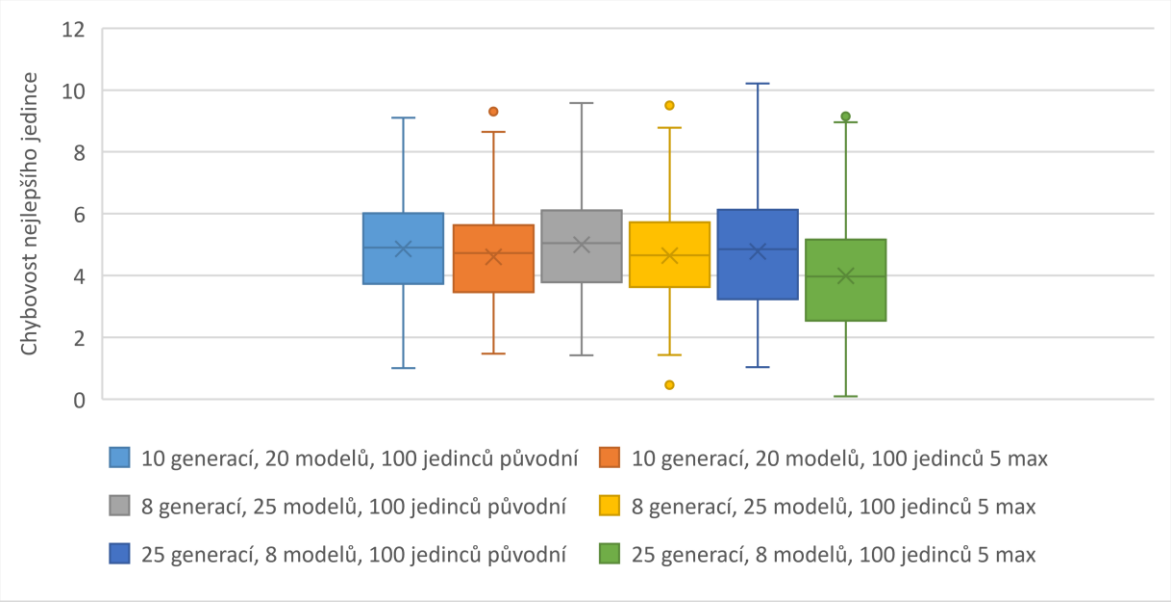
Při provádění předchozího experimentu byla pozorována tendence vytváření hlubokých stromů v první vrstvě, které byly poté přenášeny do druhé vrstvy jako terminály a kvůli jejich velikosti znemožňovaly úspěšnou konvergenci ve druhé vrstvě. Proto bude v dalším experimentu nastavena maximální hloubka stromu v první vrstvě na hodnotu 5 místo iniciálních 8.

Níže je možno vidět porovnání těchto nových experimentů s původním jednovrstvým genetickým programováním.



Obrázek 27 – Porovnání experimentů s nižší maximální hloubkou stromu u rovnice s goniometrickou funkcí, zdroj: vlastní

Na první pohled není vidět žádný výrazný rozdíl s původními konfiguracemi z minulých experimentů, vyberme tedy některé z nich a porovnejme pouze je, bez porovnání s jednovrstvým genetickým programováním, aby byly rozdíly více patrné.



Obrázek 28 – Porovnání experimentů s nižší maximální hloubkou stromu s původními experimenty u rovnice s goniometrickou funkcí, zdroj: vlastní

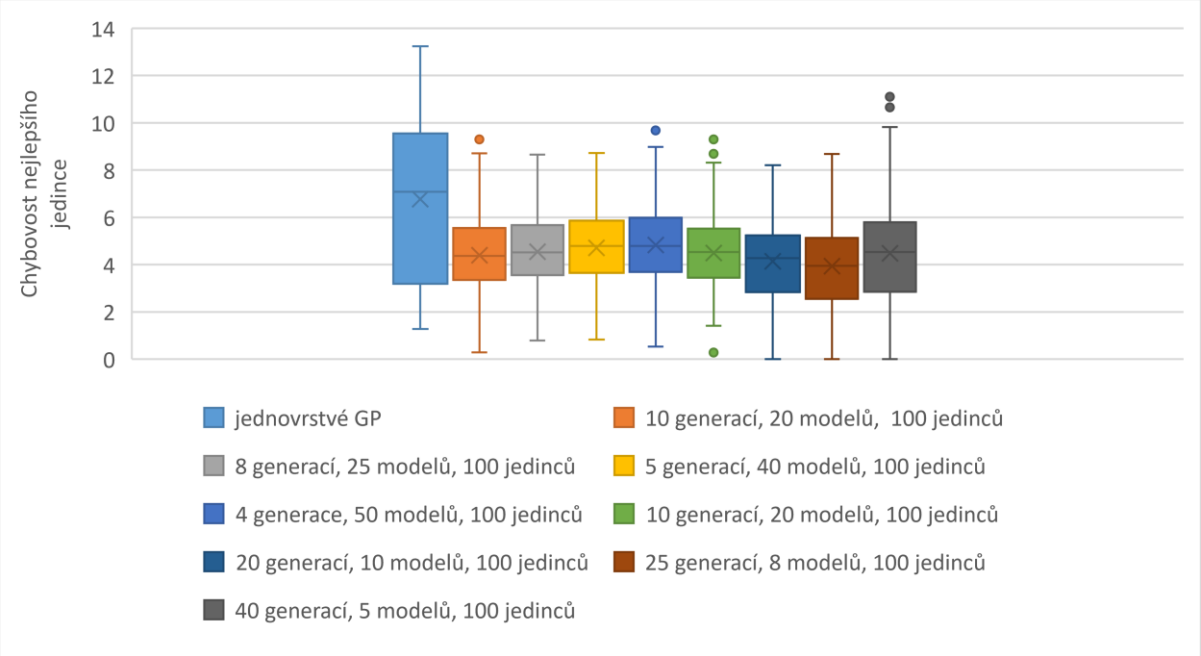
Pokud porovnáme pouze experimenty mezi sebou bez porovnání s jednovrstvým genetickým programováním, jsou rozdíly již jednoduše viditelné. V obrázku je vždy zobrazena dvojice

stejných konfigurací lišící se pouze v maximální povolené výšce stromu v první vrstvě, kde vlevo je původní konfigurace a vpravo je konfigurace s nastavenou nižší hodnotou maximální velikosti stromu v první vrstvě. Pokud tedy tyto dvojice porovnáme, zjistíme že limitace hloubky stromu na hodnotu 5 v první vrstvě ve všech zobrazených konfiguracích přináší lepší výsledky jak z hlediska průměrné hodnoty fitness, tak i rozložení kvartilů. Pro další experimenty bude tedy nastavena maximální hloubka stromů (největší hloubka před ořezáváním stromu) v první vrstvě na hodnotu 5.

5.3.4 Experimenty s 50 % bootstrappingem

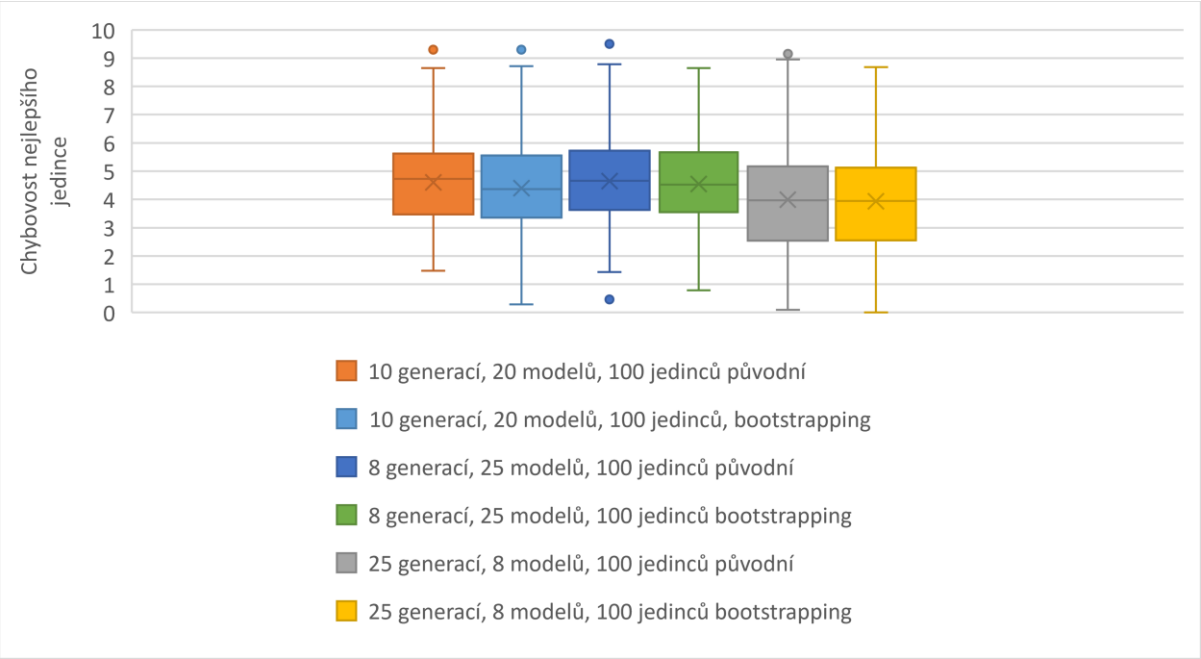
V tomto experimentu se pokusíme zjistit, zda by nastavení hodnoty boostrappingu u první vrstvy na hodnotu 50 % bylo schopno poskytnout lepší výsledky, než jsme získali z předchozích experimentů.

Opět tedy nejprve porovnáme výsledky s nově nastaveným bootstrappingem na hodnotu 50 % s jednovrstvým genetickým programováním.



Obrázek 29 – Porovnání experimentů s 50 % bootstrappingem u rovnice s goniometrickou funkcí, zdroj: vlastní

Z porovnání opět není na první pohled zřetelné, zda tato změna poskytla lepší nebo horší výsledky než v předchozím experimentu, ve kterém jsme nastavili maximální hloubku stromu v první vrstvě na hodnotu 5, proto opět mezi sebou porovnáme oba tyto experimenty v následujícím obrázku.



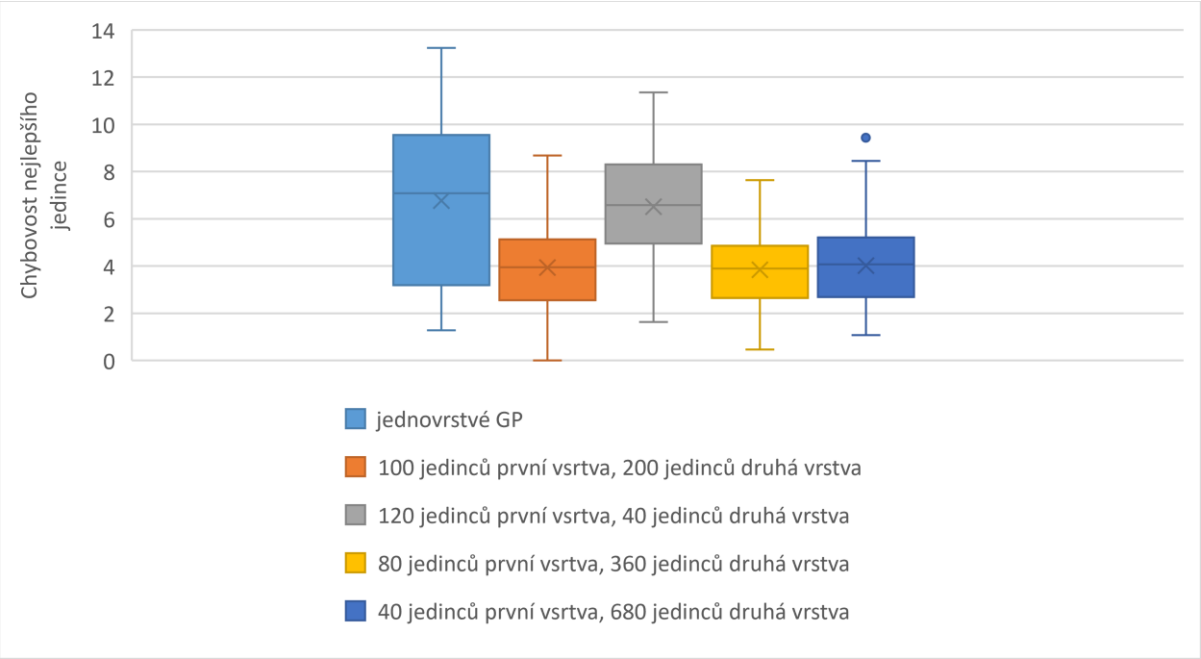
Obrázek 30 – Porovnání experimentů s nastaveným bootstrappingem na hodnotu 50 % s předchozím experimentem s maximální hloubkou stromu v první vrstvě nastavenou na 5 u rovnice s goniometrickou funkcí, zdroj: vlastní

Pokud porovnáme oba přístupy, tedy původní bez jakéhokoliv bootstrappingu a nový s 50 % boostrappingem, tak můžeme opět jako u předešlého experimentu vidět v tomto případě pouze mírné zlepšení v 50 % bootsrappingu oproti původnímu experimentu bez boostrappingu, je tedy zřejmé že aplikace této metody je pro nás obdobně jako nastavení odlišné výšky stromů výhodná.

5.3.5 Experimenty s různými počty jedinců ve vrstvách

V těchto experimentech se pokusíme vyzkoušet různé velikosti populací v první a druhé vrstvě dvouvrstvého GP. Jako počáteční bod si stanovíme konfiguraci, která byla zatím schopna dosáhnout nejlepšího výsledku ze všech vyzkoušených konfigurací. Touto konfigurací je: 25 generací v první vrstvě, 8 submodelů, 100 jedinců v první vrstvě a 25 generací s 200 jedinci ve druhé vrstvě. Bude také nastaven bootstrapping na hodnotu 50 % a maximální velikost stromu v první vrstvě na hodnotu 5, protože jejich pozitivní efekt na průměrnou fitness bylo možno vidět v předchozích experimentech.

Níže je možno vidět porovnání všech vyzkoušených kombinací různých velikostí populací ve vrstvách.

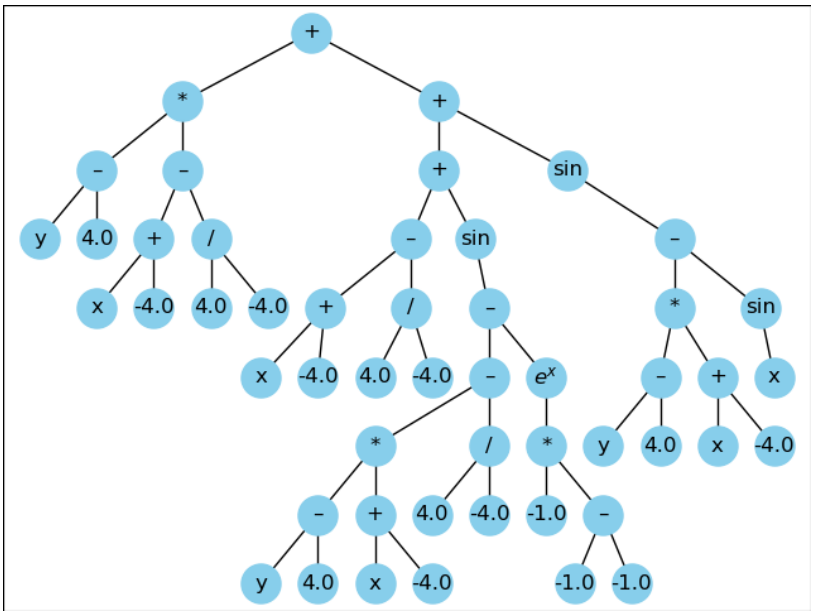


Obrázek 31 – Porovnání různých velikostí populací v první a druhé vrstvě u rovnice s goniometrickou funkcí, zdroj: vlastní

Jak je možno z porovnání vidět, při zvýšení velikosti populace ve druhé vrstvě na úkor první vrstvy se průměrná hodnota fitness a rozptyl téměř nezmění a při nastavení 680 jedinců ve druhé vrstvě mírně zhorší. Na druhou stranu ale při zvýšení počtu jedinců v první vrstvě na hodnotu 120 nám umožní stanovit počet jedinců na druhou vrstvu pouze na 40. Toto drastické snížení musí být nastaveno, abychom nezměnili celkový součet počtu evaluací v obou vrstvách a zachovali porovnatelnost výsledků. Při tomto počtu jedinců je možno vidět, že je průměrná hodnota fitness znatelně horší. Toto chování je očekávané, protože 40 jedinců ve druhé vrstvě není schopno prohledat dostatečně velký stavový prostor na to, aby byla nalezena lepší řešení.

5.3.6 Nejlepší jedinci získaných ze všech experimentů

Všechny stromy získané z prováděných experimentů, které měly hodnotu fitness (celkovou střední kvadratickou chybu) menší než 1 byly velikostně poměrně velké a složité, proto si představíme pouze jednoho jedince, který měl celkovou hodnotu fitness rovnou okolo 0.25. Níže je možno vidět jeho podoba.



Obrázek 32 – První jedinec s dobrou hodnotou fitness u rovnice s goniometrickou funkcí, zdroj: vlastní

Pokud bychom tohoto jedince chtěli popsat ve tvaru rovnice, dostali bychom rovnici o následujícím tvaru:

$$sin\big((y-4)*(x-4)\big) - sin(x) + 2xy - 7y - 7x + 25 \qquad (4)$$

Pokud bychom tuto rovnici porovnali s původní rovnicí, zjistili bychom, že genetické programování bylo schopno nalézt nejpodstatnější část rovnice, tedy

$$sin\big((y-4)*(x-4)\big), \qquad (5)$$

tato část nejspíše pomohla jedinci získat tak dobrou hodnotu fitness. Druhá část rovnice už se s původní poněkud rozchází, zejména v nepodstatném sinu z x, který se v původní rovnici vůbec nenacházel. Na druhou stranu se zde ale objevují obě vstupní proměnné x a y, které se nachází i v původní rovnici, takže tímto také není jedinec úplně zcestný.

5.4 Rovnice s větším počtem proměnných

Poslední datová sada, která byla zvolena pro aproximaci bude obsahovat tři vstupní proměnné, to z toho důvodu, aby se ukázala účinnost dvouvrstvého genetického programování i na funkci s větším množstvím proměnných. Bude se jednat o následující rovnici:

$$30\,\frac{(x-1)(z-1)}{y^2(x-10)} \qquad (6)$$

Tato rovnice byla, stejně jako rovnice (2) převzata z článku Order of Nonlinearity as a Complexity Measure for Models Generated by Symbolic Regression via Pareto Genetic Programming od Ekateriny Vladislavlevy.

5.4.1 Iniciální parametry

Výchozí parametry byly u tohoto problému vzhledem k původním experimentů provedených od Ekateriny Vladislavlevy mírně alterovány, z důvodu větší náročnosti evaluace jedinců. Z tohoto důvodu byl také celkový počet nezávislých běhů, ze kterých se získávají výsledné grafy, snížen na počet 300.

Tabulka 13 – Obecné parametry použité pro aproximaci rovnice větším počtem proměnných, zdroj: vlastní

Použité křížení	Křížení podstromů s 2 výslednými potomky
Použitá mutace	Jednobodová mutace
Použitá selekce	Turnajový výběr
Velikost turnaje	5
Počet elit	1
Pravděpodobnost na křížení	0.95
Pravděpodobnost na mutaci	0.05
Maximální hloubka stromu při provádění genetických operací	12
Minimální hloubka stromu při inicializaci	2
Maximální hloubka stromu při inicializaci	6
Minimální hodnota proměnných *x* a z pro aproximaci	-0.05
Maximální hodnota proměnných *x* a z pro aproximaci	2.05
Minimální hodnota proměnné *y* pro aproximaci	0.95
Maximální hodnota proměnné *y* pro aproximaci	1.95
Velikost kroku, po kterém byly aproximovány proměnné *x* a *z*	0.15
Velikost kroku, po kterém byla aproximována proměnná *y*	0.1
Seznam terminálů	Konstanty -5.0, -4.0, -3.0, -2.0, -1.0, 1.0, 2.0, 3.0, 4.0, 5.0 a 3 vstupní proměnné
Seznam funkcí	+, -, *, /, sqrt, pow2, e^x

Oproti ostatním problémům, je u tohoto problému mírně komplikovanější generování bodů, ze kterých se počítá celková chyba vygenerované funkce genetickým programováním proti

aproximované funkci. Je zde nastavený různý krok pro proměnné, aby nebylo nutné počítat celkovou chybu pro velké množství bodů a tím následně ještě více prodloužit běh algoritmu.

Pro jednovrstvé GP, byly zvoleny následující hodnoty pro velikost populace a počet generací.

Tabulka 14 – Specifické parametry pro jednovrstvé GP pro aproximaci rovnice s větším počtem proměnných, zdroj: vlastní

Velikost populace	125
Počet generací	100

Pro dvouvrstvou reprezentaci budou zvoleny jako počáteční bod parametry vyobrazeny v následující tabulce.

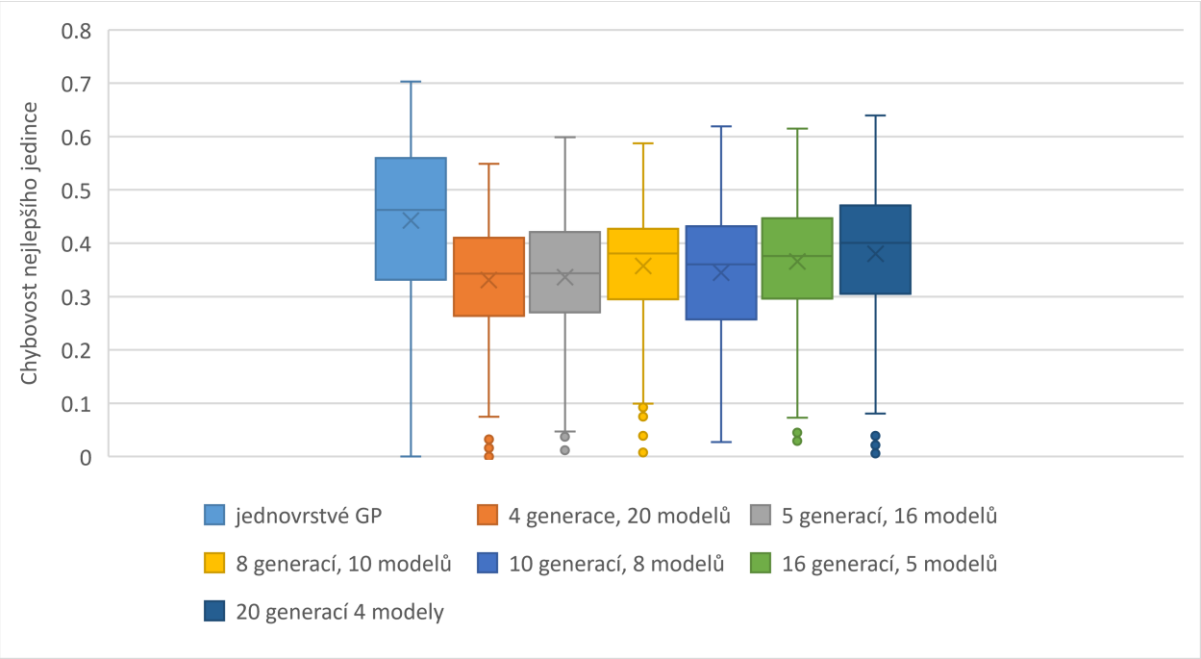
Tabulka 15 – Specifické parametry pro dvouvrstvé GP pro aproximaci rovnice s větším počtem proměnných, zdroj: vlastní

Počáteční velikost populace v první vrstvě	100
Počet generací v první vrstvě	10
Počet submodelů a subsetů	20
Velikost populace ve druhé vrstvě	225
Počet generací ve druhé vrstvě	20

Parametry byly zvoleny u těchto iniciálních parametrů a dalších experimentů tak, aby byl celkový počet generací a počet evaluací jedinců v obou přístupech stejný.

5.4.2 Počáteční experimenty

Nejprve, podobně jako u ostatních problémů vyzkoušíme různé konfigurace počtů generací a počtů modelů v první vrstvě, abychom měli nějaký základní referenční bod, který budeme moci použít u dalších experimentů. Porovnání těchto počátečních experimentů s jednovrstvým přístupem je možno vidět na obrázku níže.

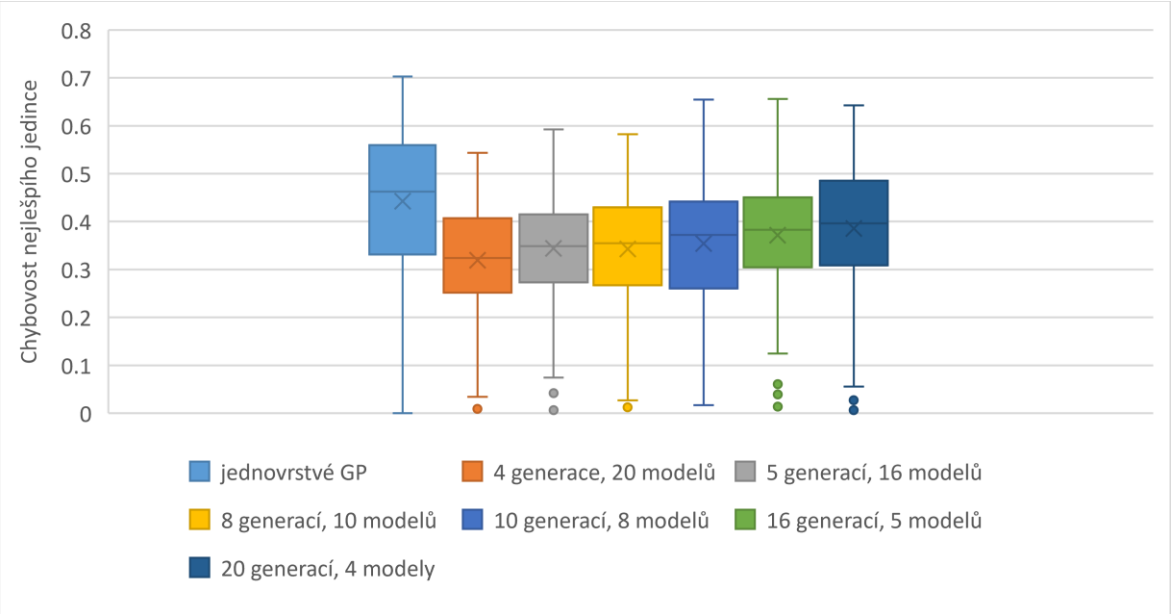


Obrázek 33 – Porovnání počátečních experimentů s různými konfiguracemi první vrstvy u rovnice s větším počtem proměnných, zdroj: vlastní

Z porovnání je vidět, že celkový průměr je ve dvouvrstvém přístupu mírně lepší než v jednovrstvém. Celková chybovost je u tohoto problému velice nízká (okolo 0.4), oproti ostatním funkcím, které byly aproximovány je chybovost výrazně nižší. Nejlepší výsledek dosáhla konfigurace s 10 generacemi a 8 modely v první vrstvě.

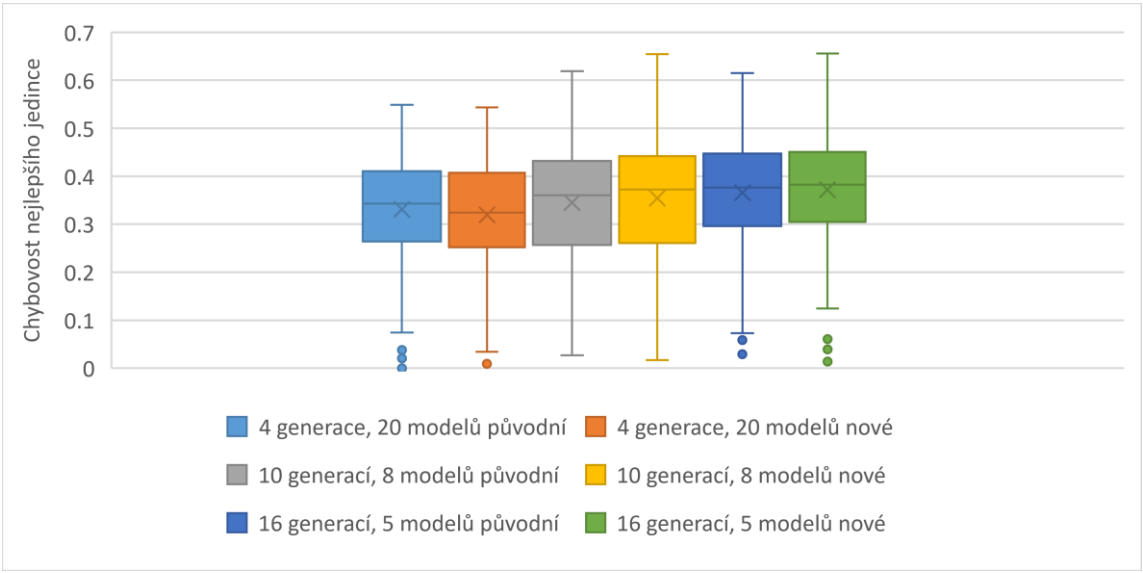
5.4.3 Nastavení maximální velikosti stromu a bootstrappingu

Z důvodu úspěšného zjištění z minulého problému, že nastavení maximální velikosti stromu v první vrstvě na hodnotu menší, než je maximální velikost stromu ve druhé vrstvě jsme schopni dostat lepší výsledky, se i u tohoto příkladu aplikuje tato konfigurace. V tomto případě se maximální velikost stromu nastaví na hodnotu 8. Bootstrapping bude nejprve nastaven na hodnotu 50 %.



Obrázek 34 – Porovnání experimentů s nižší maximální velikostí stromu a boostrappingem u první vrstvy u rovnice s větším počtem proměnných, zdroj: vlastní

Z grafu není na první pohled vidět zlepšení oproti předchozím experimentům, které byly provedeny s původní maximální velikostí stromu a bez jakéhokoli bootstrappingu. Níže tedy zvolíme některé z těchto kombinací a porovnáme je mezi sebou. Toto porovnání je možno vidět níže.



Obrázek 35 – Porovnání iniciálních konfigurací s konfiguracemi se maximální velikostí stromu v první vrstvě rovnou 8 a bootstrappingem 50 %, zdroj: vlastní

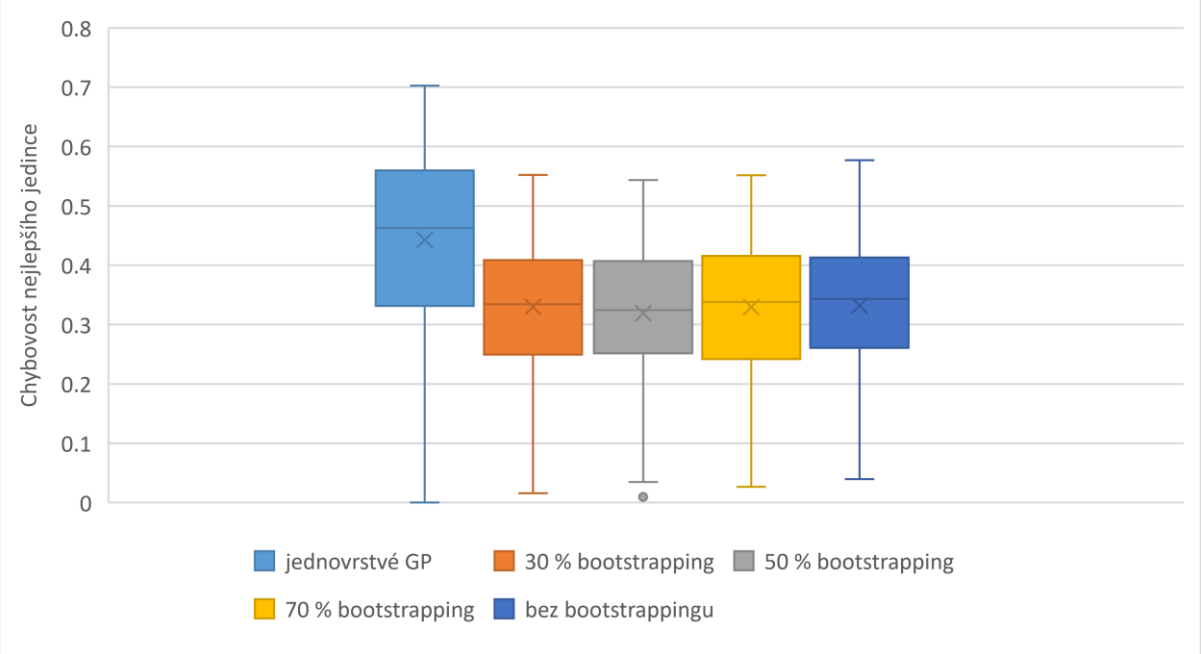
Z porovnání je možné vidět, že změna parametrů poskytla mírné zlepšení pro konfiguraci s malým počtem generací a velkým počtem modelů. Naopak ale pro velký počet generací a

malý počet modelů jsou výsledky téměř totožné a u konfigurace s 16 generacemi je i 10 % kvartil mírně horší než u původní konfigurace. I přesto ale pro další experimenty ponecháme nastavení maximální hloubky stromu v první vrstvě na hodnotu 8, z důvodu mírného zlepšení u některých konfigurací a žádného zhoršení u konfigurací ostatních.

5.4.4 Nastavení různých procent bootstrappingu na konfiguraci s větším množstvím modelů

Z důvodu poměrně nepřesvědčivého výsledku z předchozího experimentu se v tomto experimentu pokusíme zjistit, jaký vliv by na výsledky mělo nastavení jiného procenta bootstrappingu.

Pro tento experiment bude nejprve vyzkoušena konfigurace s 4 generacemi a 20 modely v první vrstvě. Konfigurace druhé vrstvy nebude v těchto experimentech žádným způsobem alterována. Níže je možno vidět porovnání různých procent bootstrappování pro tuto konfiguraci.



Obrázek 36 – Porovnání různých procent bootstrappingu pro konfiguraci s 4 generacemi a 20 modely v první vrstvě a nastavenou maximální velikost stromu v první vrstvě na 8, zdroj: vlastní

Z porovnání nelze vidět žádné velké rozdíly mezi konfiguracemi. Všechny konfigurace jsou z hlediska získaného nejlepšího jedince porovnatelné. Jeden fakt, který stojí za to v tomto případě vyzdvihnout je menší mezikvartilové rozpětí výsledků konfigurace bez bootstrappingu. Výsledky jsou tedy koncentrovanější okolo průměrné hodnoty fitness než u ostatních konfigurací.

Ve všech následných porovnávaných konfiguracích bude také přiložena tabulka porovnání rychlostí provedení běhů, jelikož tato skutečnost také velmi úzce souvisí s použitým procentem bootstrappingu. Níže je tato tabulka, které porovnává celkové doby vykonávání všech 300 nezávislých běhů u různých konfigurací, přiložena. Všechny běhy byly testovány na jednom a tom samém zařízení, aby byly výsledky co možná nejvíce porovnatelné.

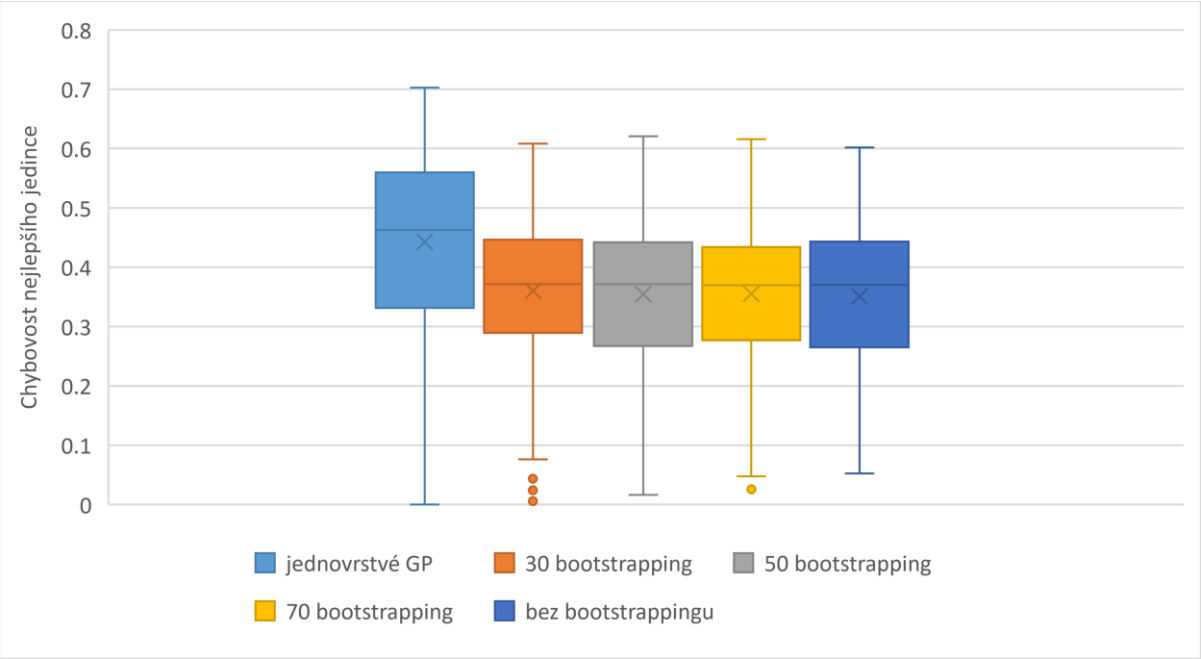
Tabulka 16 – Porovnání rychlostí provedení běhů s různými procenty bootstrappingu u 4 generacích, 20 modelech a 100 jedincích v první vrstvě, zdroj: vlastní

Procento bootstrappingu	Čas provedení
30 %	269 min
50 %	272 min
70 %	294 min
Žádný bootstrapping (celý dataset)	300 min

Z porovnání lze vidět, že při zvyšování celkového procenta bootstrappingu se zvyšuje časová náročnost daných běhů. Toto nastává, protože je nutné pro ohodnocení jedinců projít více body a spočítat celkovou chybovost na více bodech.

5.4.5 Nastavení různých procent bootstrappingu na konfiguraci s menším množstvím modelů

Následně vyzkoušíme vliv bootstrappingu na obráceném rozložení, tedy konfiguraci, kde bude větší množství generací a malý počet modelů. V tomto případě se bude jednat o konfiguraci s 20 generacemi a 4 modely. Výsledky tohoto porovnání je možno vidět níže.



Obrázek 37 – Porovnání různých procent bootstrappingu pro konfiguraci s 20 generacemi a 4 modely v první vrstvě a nastavenou maximální velikost stromu v první vrstvě na 8, zdroj: vlastní

Z porovnání lze vidět mírnou tendenci zlepšování hodnot fitness pro konfigurace s větším počtem bootstrappingu. Tento jev je nejspíš způsoben tím, že z důvodu většího počtů generací na první vrstvě se při větších procentech bootstrappingu mají genetická programování v první vrstvě větší možnost přizpůsobit na větší rozsah bodů, takže jsou celkově pro druhou vrstvu použitelnější než modely u menších procent bootstrappingu, které nemusí být v celkovém kontextu dostatečně přesné.

Níže je také možno vidět porovnání časů provedení pro všechny porovnávané konfigurace.

Tabulka 17 – Porovnání rychlostí provedení běhů s různými procenty bootstrappingu u 20 generacích, 4 modelech a 100 jedincích v první vrstvě, zdroj: vlastní

Procento bootstrappingu	Čas provedení
30 %	297 min
50 %	324 min
70 %	329 min
Žádný bootstrapping (celý dataset)	352 min

Z porovnání rychlostí je možno vidět, že čas provedení je oproti předchozímu experimentu s větším počtem modelů v první vrstvě o poznání pomalejší. Například pro procento bootstrappingu rovné 30 byl čas provedení u tohoto experimentu roven 297 minutám a naproti

tomu pro předchozí experiment pro tu samou hodnotu bootstrappingu byl 269 minut, což je téměř půlhodinový rozdíl. Tento jev nejspíše nastal z důvodu, že při konfiguraci s pouhými 4 modely nebyla využita všechna dostupná vlákna procesoru, takže využití procesoru nebylo tak efektivní jako při konfiguraci s 20 modely.

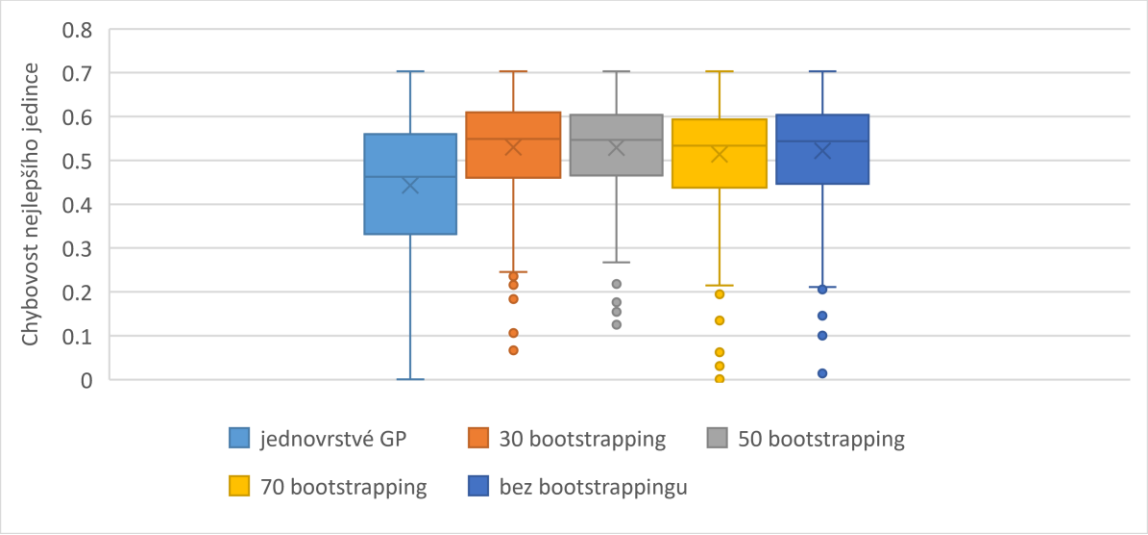
5.4.6 Testování metody bootstrappingu pro větší velikost populace v první vrstvě na konfiguraci s menším množstvím modelů

Stejné experimenty, které jsme prováděli v předchozích dvou kapitolách s procenty bootstrappingu nyní zopakujeme na konfiguracích, které budou mít větší počet jedinců v první vrstvě. Tento přístup by měl být schopen poskytnout lepší modely pro následující druhou vrstvu. Abychom ale neporušili podmínky, které jsme si stanovili na počátku ohledně možnosti porovnání dvouvrstvého a jednovrstvého přístupu bude nutné abychom snížili celkový počet jedinců v druhé vrstvě.

Nejprve provedeme experiment se 150 jedinci v první vrstvě, kde pro druhou vrstvu zbude pouze 25 jedinců. Poté vyzkoušíme konfiguraci, kde budeme mít stejný počet jedinců v první a ve druhé vrstvě, tedy 125. Oba tyto experimenty budou prováděny na konfiguraci s 20 generacemi a 4 modely v první vrstvě.

150 jedinců v první vrstvě, 25 jedinců ve druhé

Níže je možno vidět porovnání různých procent bootstrappingu pro tuto konfiguraci.



Obrázek 38 – Porovnání experimentů s bootstrappingem pro 150 jedinců v první vrstvě a 25 jedinců ve druhé vrstvě pro 20 generací a 4 modely v první vrstvě, zdroj: vlastní

Z porovnání lze vidět mírně se zlepšující tendenci nejlepších dosažených u 70 a 100 procent bootstrappingu. Kromě této skutečnosti jsou výsledky téměř stejné. Průměrné hodnoty fitness

jsou ale v tomto případě horší než u našich původních běhů, kde se hodnoty jedinců pohybovaly okolo 0.3 až 0.4 chybovosti. Toto nastalo nejspíše z důvodu nedostatku jedinců pro dostatečnou konvergenci ve druhé vrstvě. Na druhou stranu jsou ale běhy této konfigurace velmi rychlé, což je možno vidět z tabulky níže.

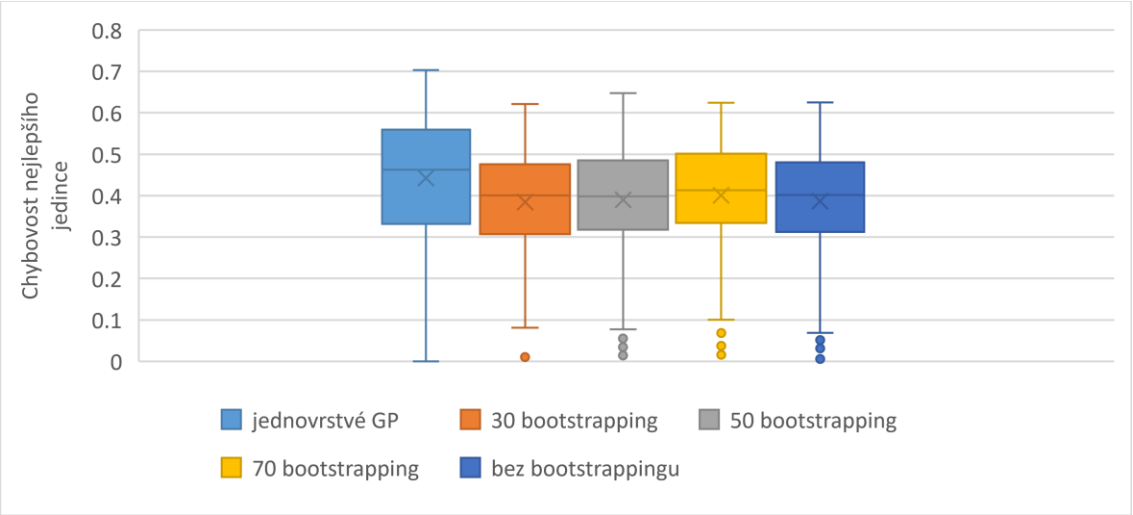
Tabulka 18 – Porovnání rychlostí provedení běhů s různými procenty bootstrappingu u 20 generací, 4 modelech a 150 jedincích v první vrstvě, zdroj: vlastní

Procento bootstrappingu	Čas provedení
30 %	62 min
50 %	63 min
70 %	64 min
Žádný bootstrapping (celý dataset)	63 min

U těchto konfigurací jsou všechny časy provedení téměř identické. V tomto případě je tedy lepší zvolit možnost bez bootstrappingu z důvodu větší přesnosti a téměř žádným dopadem na časovou náročnost.

125 jedinců v první vrstvě, 125 ve druhé

Níže je možno vidět porovnání různých procent bootstrappingu pro tuto konfiguraci se 125 jedinci.



Obrázek 39 – Porovnání experimentů s bootstrappingem pro 125 jedinců v první vrstvě a 125 jedinců ve druhé vrstvě pro 20 generací a 4 modely v první vrstvě, zdroj: vlastní

V tomto případě nelze říci, že by mezi různými procenty bootstrappingu byl na první pohled velký rozdíl, proto se pouze podíváme na porovnání rychlostí různých procent bootstrappingu v tabulce níže.

Tabulka 19 – Porovnání rychlostí provedení běhů s různými procenty bootstrappingu u 20 generací, 4 modelech a 125 jedincích v první vrstvě, zdroj: vlastní

Procento bootstrappingu	Čas provedení
30 %	172 min
50 %	180 min
70 %	178 min
Žádný bootstrapping (celý dataset)	222 min

V tabulce je opět možno pozorovat trend, který byl patrný z předchozích experimentů. V tomto případě zde ale vznikl paradox, kde 70 % bootstrapping byl proveden rychleji než 50 %. Toto mohlo být způsobeno nějakým nečekaným zatížením zařízení, na kterém byly běhy prováděny, která způsobila tento výkyv. Zajímavý je také časový skok mezi konfigurací se 70 % bootstrappingu a konfigurací bez bootstrappingu. Zde se jednalo o rozdíl 44 minut, což je již docela zásadní zpoždění.

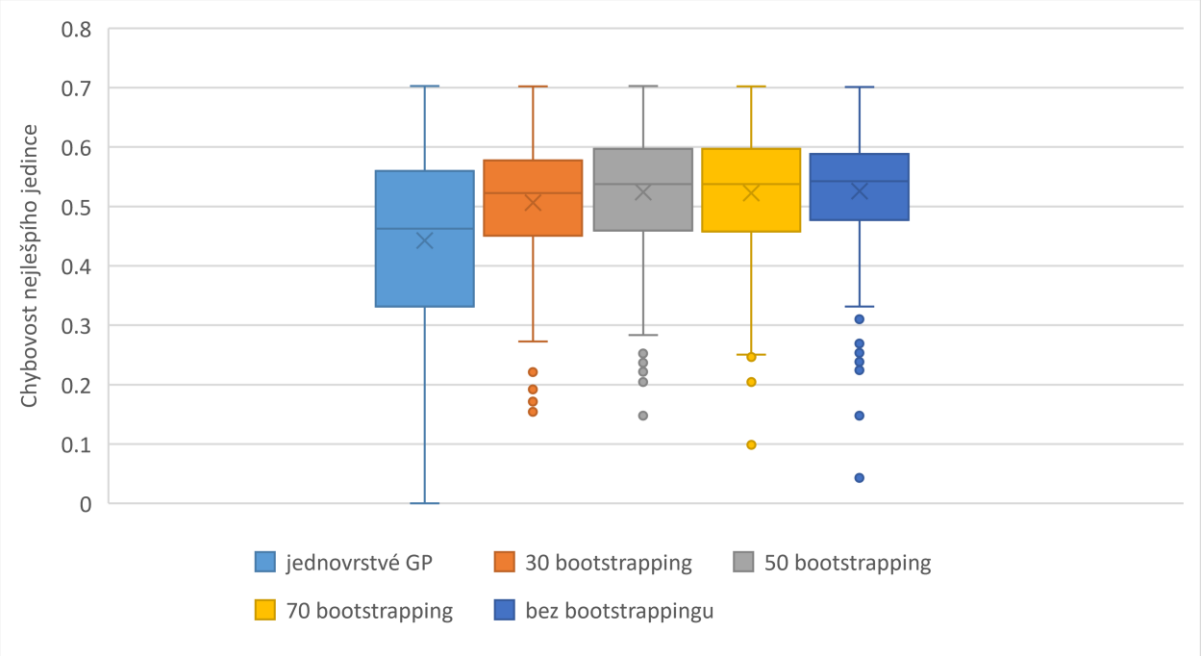
5.4.7 Testování metody bootstrappingu pro větší velikost populace v první vrstvě na konfiguraci s větším množstvím modelů

Jelikož jsme prozkoumali vliv různých procent bootstrappingu pro větší velikost populace v první vrstvě u modelu, který měl menší počet modelů, vyzkoušíme tento přístup i pro opačný

stav, tedy menší počet generací a větší počet modelů. V tomto případě se bude jednat o konfigurace s 4 generacemi a 20 modely v první vrstvě.

150 jedinců v první vrstvě, 25 jedinců v druhé vrstvě

Nejprve tedy vyzkoušíme stav pro 150 jedinců v první vrstvě a 25 jedinců v druhé. Porovnání všech zkoušených hodnot bootstrappingu je možno vidět níže.



Obrázek 40 – Porovnání experimentů s bootstrappingem pro 150 jedinců v první vrstvě a 25 jedinců ve druhé vrstvě u rovnice s větším počtem proměnných, zdroj: vlastní

Z porovnání není na první pohled zřejmé, jaký přístup poskytuje lepší řešení. Všechny nastavené hodnoty bootstrappingu přinesly velice podobné výsledky. Je nutné ale opět podotknout, že s nižší hodnotou bootstrappingu se snižuje celkový čas nutný pro provedení všech 300 nezávislých běhů. Níže je možno vidět tabulka srovnání času vykonání všech těchto procent bootstrappingu.

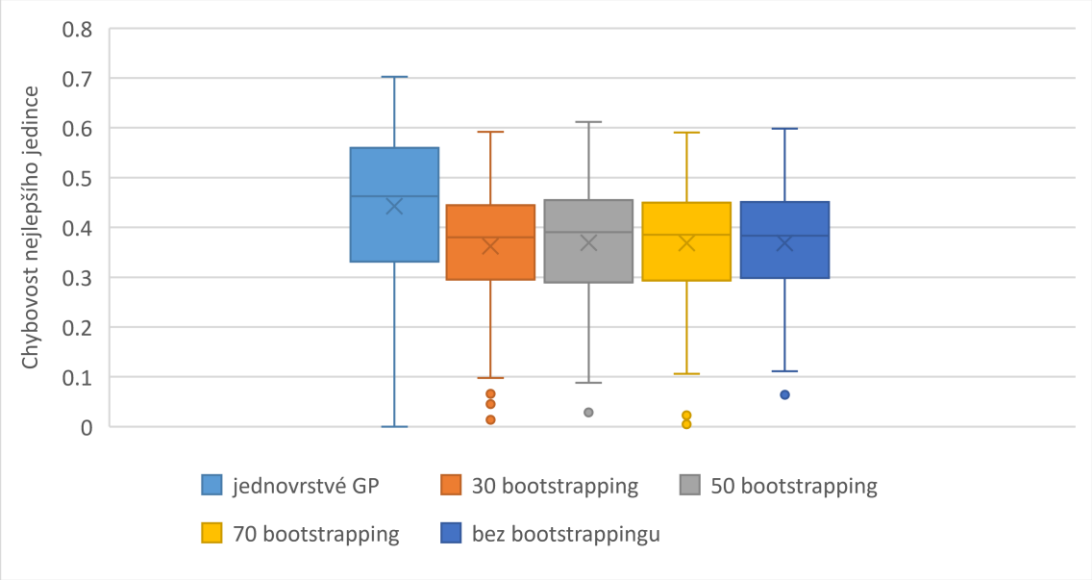
Tabulka 20 – Porovnání rychlostí provedení běhů s různými procenty bootstrappingu u 4 generacích, 20 modelech a 150 jedincích v první vrstvě, zdroj: vlastní

Procento bootstrappingu	Čas provedení
30 %	48 min
50 %	61 min
70 %	75 min
Žádný bootstrapping (celý dataset)	93 min

Z tabulky je možné vidět, že například provést experiment s celkovým procentem bootstrappingu nastaveným na 30 % trval pouhých 48 min, což je zhruba polovina času trvání experimentu s žádným boostrappingem. Z tohoto důvodu a z důvodu podobnosti výsledných hodnot fitness pro všechny hodnoty bootstrappingu by mohlo být výhodné pro tuto konfiguraci využít pouze 30 % bootstrappingu a získat podobné výsledky s tou výhodou, že doba nutná na dokončení 300 běhů bude téměř poloviční.

125 jedinců v první vrstvě, 125 jedinců v druhé vrstvě

Následně tedy vyzkoušíme různá procenta bootstrappingu i pro 125 jedinců v první vrstvě a se stejným počtem i ve vrstvě druhé.



Obrázek 41 – Porovnání experimentů s bootstrappingem pro 125 jedinců v první vrstvě a 125 jedinců ve druhé vrstvě u rovnice s větším počtem proměnných, zdroj: vlastní

Z porovnání, podobně jako v předchozím experimentu, není mezi různými procenty bootstrappinu téměř žádný rozdíl ve výsledcích. Obrátíme se tedy pouze na celkové časy provedení, které je možno vidět porovnané v tabulce níže.

Tabulka 21 – Porovnání rychlostí provedení běhů s různými procenty bootstrappingu u 4 generací, 20 modelech a 125 jedincích v první vrstvě, zdroj: vlastní

Procento bootstrappingu	Čas provedení
30 %	157 min
50 %	166 min
70 %	173 min
Žádný bootstrapping (celý dataset)	193 min

Z tabulky je možno vidět podobný trend, který se vyskytoval i u ostatních konfigurací. Kupodivu ale v této konfiguraci se 125 jedinci v porovnání s konfigurací s 20 generacemi a 4 modely již tato konfigurace neposkytuje takové zrychlení jako tomu bylo u konfiguraci se 100 jedinci.

ZÁVĚR

Cílem diplomové práce bylo prozkoumat možný přínos dvouvrstvého přístupu ke genetickému programování. Toto porovnání bylo celkem vyzkoušeno na čtyřech problémech symbolické regrese. Celkem u tří z těchto problémů lze přímo konstatovat, že při použití specifických konfigurací dosáhl dvouvrstvý přístup lepších výsledků než přístup jednovrstvý. Jediný problém, ve kterém bylo dvouvrstvé genetické programování výkonnostně horší, byl problém jedenáctivstupového booleovského multiplexoru. Jeden z hlavních důvodů, proč v tomto problému dvouvrstvý přístup neobstál je podle mého názoru ten, že v daném případě není jednoduché rozdělit datový set do subsetů prvních vrstev tak, aby byly výsledné submodely použitelné pro vrstvu druhou. Poté tedy zbývá pouze možnost daný dataset nerozdělovat a poskytnout modelům genetického programování v první vrstvě dataset celý. V tomto případě ale poté běhy genetického programování nemají dostatečné množství prostředků, aby konvergovaly k jedincům s dobrou hodnotou fitness, proto nejspíše na tento problém nebude dvouvrstvý přístup ideální.

Dvouvrstvý přístup s sebou ale přinesl spoustu výhod. Jednou z hlavních výhod je možnost jednoduše paralelizovat první vrstvu. Běhy dvouvrstvého genetického programování jsou tedy schopny velmi efektivně využít procesor a tím pádem je celková časová náročnost oproti jednovrstvému přístupu menší a běhy časově kratší. Tuto skutečnost lze ještě více zvýraznit zavedením nějaké formy resamplingu. V této diplomové práci byla využívaná metoda bootstrappingu, je ale možné použít i jiné metody rozdělení datového setu na subsety. Tento přístup u některých konfiguracích dokázal zmenšit čas provádění dokonce i na polovinu, bez téměř žádného dopadu na výsledné hodnoty fitness nejlepších potomků z druhé vrstvy.

Další výhoda, která byla pozorována u všech zkoušených problémů, je menší mezikvartilový rozptyl získaných nejlepších jedinců. Díky tomuto faktu je možné konstatovat, že výsledky získané z dvouvrstvého přístupu jsou více konzistentní a často spadají do menšího intervalu než u jednovrstvé architektury.

V budoucích experimentech s pomocí dvouvrstvého přístupu by bylo vhodné vyzkoušet jiné typy problémů než pouze symbolickou regresi. Existuje široká škála problémů, na které je možné genetické programování využít a bylo by dobré zjistit, zda by byla dvouvrstvá architektura stejně úspěšná jako na vyzkoušených problémech symbolické regrese.

To ale neznamená, že jsou všechny problémy spojené se symbolickou regresí touto diplomovou prací vyřešeny. I problémy, které byly v této práci řešeny pomocí dvouvrstvého přístupu

nejspíše obsahují prostor pro zlepšení, zejména problém booleovského multiplexoru. Je totiž nemožné porovnat všechny možné kombinace konfigurací a určit jednu konfiguraci, která by poskytnula nejlepší výsledky. Jeden z parametrů, který by stálo za to v budoucnu prozkoumat více do hloubky, je nastavení odlišných funkcí v první a ve druhé vrstvě. S pomocí tohoto nastavení by tak teoreticky bylo možné například zamezit využití funkce sinus v první vrstvě a pouze ji využít ve vrstvě druhé a tím následně snížit komplexnost submodelů, které budou využity jako terminály ve druhé vrstvě.

ZDROJE

[1] GOLDBERG, David E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Illinois: Addison-Wesley Professional, 1989. ISBN 978-0201157673.

[2] CARR, Jenna. *An Introduction to Genetic Algorithms*. Senior Project. Washington: Whitman College, 2014.

[3] ABDULHAMED, Ahmed A.; TAWFEEK, Medhat A. a KESHK, Arabi E. *A genetic algorithm for service flow management with budget constraint in heterogeneous computing*. Online. *Future Computing and Informatics Journal*. 2018, roč. 3, č. 2, s. 341-347. ISSN 23147288. Dostupné z: https://doi.org/10.1016/j.fcij.2018.10.004. [cit. 2024-03-02].

[4] ROETZEL, Wilfried; LUO, Xing a CHEN, Dezhen. *Design and Operation of Heat Exchangers and their Networks*. Academic Press, 2019. ISBN 978-0-12-817894-2.

[5] WHITLEY, Darrell. *An overview of evolutionary algorithms: practical issues and common pitfalls*. Online. S. 817-831. Dostupné z: https://doi.org/ISSN 0950-5849. [cit. 2024-03-04].

[6] ALSPAUGH, Thomas A. *Binary Strings*. Online. 2010. Dostupné z: https://ics.uci.edu/~alspaugh/cls/shr/binaryString.html. [cit. 2024-03-04].

[7] WALLET, Bradley C.; MARCHETTE, David J.; SOLKA, Jeffrey L. a SADJADI, Firooz A. *Matrix representation for genetic algorithms*. Online. *Proceedings of the SPIE*. 1996, roč. 2756, č. Automatic Object Recognition VI, s. 206-214. Dostupné z: https://doi.org/10.1117/12.241153. [cit. 2024-03-09].

[8] EIBEN, A. E. a SMITH, Jim. *Introduction To Evolutionary Computing*. Second edition. Heidelberg: Springer, 2015. ISBN 978-3-662-44873-1.

[9] CHAPLIN, Ryan. *Selection Methods of Genetic Algorithms*. Stipendijní práce. Illinois: Olivet Nazarene University, 2018.

[10] JEBARI, Khadil a MADIAFI, Mohammed. *Selection Methods for Genetic Algorithms*. Online. *International journal of emerging sciences*. 2013, roč. 3, č. 4, article 3, s. 333–344. Dostupné z: https://doi.org/10.35940/ijrte.2277-3878. [cit. 2024-03-11].

[11] *Genetic Algorithms, Tournament Selection, and the Effects of Noise*. Online. Complex Systems. 1995, roč. 9, č. 3, article 3, s. 193–212. ISSN 0891-2513. Dostupné z: https://content.wolfram.com/sites/13/2018/02/09-3-2.pdf. [cit. 2024-03-13].

[12] POLI, Riccardo; LANGDON, W. B.; MCPHEE, Nicholas F. a KOZA, John R. *A field guide to genetic programming*. [S.l.: Lulu Press], 2008. ISBN 1409200736.

[13] KUBALÍK, Jiří. *Kybernetika a umělá inteligence 12. Evolutionary Algorithms: GA & GP*. Online, prezentace. Praha: České vysoké učení technické v Praze, Fakulta elektrotechnická, Katedra kybernetiky, 2012. Dostupné z: https://cw.fel.cvut.cz/old/_media/courses/a3b33kui/prednasky/prednaska_12.pdf?cache=nocache. [cit. 2024-03-13].

[14] AIBEN, Agoston E. a SMITH, James E. *Introduction to Evolutionary Computing*. Second Edition. Springer, 2015. ISBN 978-3-662-44873-1.

[15] STAATS, Kai. About Genetic Programming. Online. STAATS, Kai. *Genetic Programming*. 2019, 1. 6. 2019. Dostupné z: https://geneticprogramming.com. [cit. 2024-03-21].

[16] WONG, Man Leung a LEUNG, Kwong Sak. *Data Mining Using Grammar Based Genetic Programming and Application*. 2002nd Edition. Springer, 2000. ISBN 978-0792377467.

[17] KOZA, John R. *Genetic programming as a means for programming computers by natural selection*. Online. *Statistics and Computing*. 1994, roč. 4, č. 2, s. 87–112. ISSN 0960-3174. Dostupné z: https://doi.org/10.1007/BF00175355. [cit. 2024-03-21].

[18] DEEPAI. *What is Genetic Programming?* Online. DEEPAI. Genetic Programming. 2017. Dostupné z: https://deepai.org. [cit. 2024-03-21].

[19] ANGELIS, Dimitrios; SOFOS, Filippos a KARAKASIDIS, Theodoros E. Artificial Intelligence in Physical Sciences: Symbolic Regression Trends and Perspectives. Online. *Archives of Computational Methods in Engineering*. 2023, roč. 30, č. 6, s. 3845-3865. ISSN 1134-3060. Dostupné z: https://doi.org/10.1007/s11831-023-09922-z. [cit. 2024-03-21].

[20] KOZA, John R. *Genetic programming 1: on the programming of computers by means of natural selection*. Complex adaptive systems. Cambridge, Mass.: MIT Press, 1992. ISBN 02-621-1170-5.

[21] BANZHAF, Wolfgang; NORDIN, Peter; KELLER, Robert E. a FRANCONE, Frank D. *Genetic Programming: An Introduction (The Morgan Kaufmann Series in Artificial Intelligence).* Morgan Kaufmann, 1997. ISBN 978-1558605107.

[22] PAGE, Jonathan; POLI, Riccardo a LANGDON, William B. Smooth Uniform Crossover with Smooth Point Mutation in Genetic Programming: A Preliminary Study. Online. *Lecture Notes in Computer Science*. 1999, roč. 1999, č. 1598, article 1598, s. 39–48. Dostupné z: https://doi.org/10.1007/3-540-48885-5. [cit. 2024-03-27].

[23] ANGELINE, Peter J. Comparing subtree crossover with macromutation. Online. *Evolutionary Programming VI*. Lecture Notes in Computer Science. 1997, roč. 1997, č. 1213, s. 101-111. ISBN 978-3-540-62788-3. Dostupné z: https://doi.org/10.1007/BFb0014804. [cit. 2024-03-27].

[24] TRUJILLO, Leonardo; MUÑOZ, Luis; GALVÁN-LÓPEZ, Edgar a SILVA, Sara. Neat Genetic Programming: Controlling bloat naturally. Online. *Information Sciences*. 2016, roč. 2016, č. vol. 333, s. 21-43. ISSN 00200255. Dostupné z: https://doi.org/10.1016/j.ins.2015.11.010. [cit. 2024-03-27].

[25] WOODWARD, John; MARTIN, Simon a SWAN, Jerry. Benchmarks that matter for genetic programming. Online. *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. 2014-07-12, roč. 2014, s. 1397-1404. ISBN 9781450328814. Dostupné z: https://doi.org/10.1145/2598394.2609875. [cit. 2024-03-30].

[26] ANGELINE, Peter J. a KINNEAR, Kenneth E. The Royal Tree Problem, a Benchmark for Single and Multiple Population Genetic Programming. Online. In: *Advances in Genetic Programming*. Vol. 2. MIT Press, 1996, s. 299–316. ISBN 9780262290791. Dostupné z: https://ieeexplore.ieee.org/document/6277533. [cit. 2024-03-30].

[27] MITCHELL, Melanie; FORREST, Stephanie a HOLLAND, John H. *The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance*. Online. Michigan, 1994. Dostupné z: https://www.researchgate.net/publication/2247968_The_Royal_Road_for_Genetic_Algorithms_Fitness_Landscapes_and_GA_Performance/citations. [cit. 2024-03-30].

[28] SOULE, Terence a HECKENDORN, Robert B. An Analysis of the Causes of Code Growth in Genetic Programming. Online. *Genetic Programming and Evolvable Machines*. 2002, roč. 3, č. 3, s. 283-309. ISSN 13892576. Dostupné z: https://doi.org/10.1023/A:1020115409250. [cit. 2024-03-30].

[29] MCDERMOTT, James; WHITE, David R.; LUKE, Sean; MANZONI, Luca; CASTELLI, Mauro et al. Genetic programming needs better benchmarks. Online. *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. 2012, roč. 2012, s. 791-798. ISBN 9781450311779. Dostupné z: https://doi.org/10.1145/2330163.2330273. [cit. 2024-03-30].

[30] MCDERMOTT, James; KRONBERGER, Gabriel; ORZECHOWSKI, Patryk; VANNESCHI, Leonardo; MANZONI, Luca et al. Genetic programming benchmarks: looking back and looking forward. Online. *ACM SIGEVOlution*. 2022, roč. 15, č. 3, s. 1-19. ISSN 1931-8499. Dostupné z: https://doi.org/10.1145/3578482.3578483. [cit. 2024-03-30].

[31] HELMUTH, Thomas a KELLY, Peter. PSB2: The Second Program Synthesis Benchmark Suite. Online. *Proceedings of the Genetic and Evolutionary Computation Conference*. 2021, roč. 2021, s. 785-794. Dostupné z: https://doi.org/10.48550/arXiv.2106.06086. [cit. 2024-03-30].

[32] CAVA LAB. *SRBench: A Living Benchmark for Symbolic Regression*. Online. CAVA LAB. SRBench. 2018. Dostupné z: https://cavalab.org/srbench/. [cit. 2024-03-30].

[33] LA CAVA, William; ORZECHOWSKI, Patryk; BURLACU, Bogdan; DE FRANÇA, Fabrício Olivetti; VIRGOLIN, Marco et al. Contemporary Symbolic Regression Methods and their Relative Performance. Online. *ArXiv e-prints*. 2021, roč. 2021. Dostupné z: https://doi.org/10.48550/arXiv.2107.14351. [cit. 2024-04-02].

[34] CAVA LAB. *SRBench: A Living Benchmark for Symbolic Regression*. Online. CAVA LAB. SRBench. 2018. Dostupné z: https://cavalab.org/srbench/competition-2022. [cit. 2024-04-02].

[35] ORZECHOWSKI, Patryk a MOORE, Jason H. Generative and reproducible benchmarks for comprehensive evaluation of machine learning classifiers. Online. *Science Advances*. 2022, roč. 8, č. 47. Dostupné z: https://doi.org/10.48550/arXiv.2107.06475. [cit. 2024-04-02].

[36] GITHUB. *GitHub*. Online. EPISTASISLAB. DIGEN. 2022. Dostupné z: https://github.com/. [cit. 2024-04-02].

[37] MERTA, Jan a BRANDEJSKÝ, Tomáš. Two-layer genetic programming. Online. Neural Network World. 2022, roč. 32, č. 4, s. 215-231. ISSN 23364335. Dostupné z: https://doi.org/10.14311/NNW.2022.32.013. [cit. 2024-04-02]

[38] HEURISTICLAB. *HeuristicLab A Paradigm-Independent and Extensible Environment for Heuristic Optimization*. Online. HEURISTICLAB. Symbolic Regression Benchmark Functions. 2012. Dostupné z: https://dev.heuristiclab.com/trac.fcgi/. [cit. 2024-04-06].

[39] FORTIN, Félix-Antoine; DE RAINVILLE, Francois-Michel; GARDNER, Marc-André; PARIZEAU, Marc a GAGNÉ, Christian. *DEAP documentation*. Online. Dostupné z: https://deap.readthedocs.io/en/master/index.html. [cit. 2024-04-06].

[40] FORTIN, Félix-Antoine; DE RAINVILLE, Francois-Michel; GARDNER, Marc-André; PARIZEAU, Marc a GAGNÉ, Christian. *DEAP documentation*. Online. Genetic Programming. 2023. Dostupné z: https://deap.readthedocs.io/en/master/index.html. [cit. 2024-04-06].

[41] DESMOS STUDIO. *Desmos*. Online. C2024. Dostupné z: https://www.desmos.com/3d. [cit. 2024-04-06].

[42] POLLARD, Tom a PERU, Giacomo. *Bootstrapping*. Online. Introduction to Machine Learning in Python. 2024. Dostupné z: https://carpentries-incubator.github.io/machine-learning-novice-python/07-bootstrapping/index.html. [cit. 2024-04-07].

[43] ASPENCORE. *The Multiplexer*. Online. ASPENCORE. ElectronicsTutorials. C2024. Dostupné z: https://www.electronics-tutorials.ws/combination/comb_2.html. [cit. 2024-04-07].

[44] VLADISLAVLEVA, E.J.; SMITS, G.F. a DEN HERTOG, D. Order of Nonlinearity as a Complexity Measure for Models Generated by Symbolic Regression via Pareto Genetic Programming. Online. *IEEE Transactions on Evolutionary Computation*. 2009, roč. 13, č. 2, s. 333-349. ISSN 1941-0026. Dostupné z: https://doi.org/10.1109/TEVC.2008.926486. [cit. 2024-04-07].