|  |  |
| --- | --- |
|  | |
| **Evoluční výpočetní techniky s variabilní délkou jedince** | |
|  | |
| Bc. Jozef Varhaník | |
|  | |
|  |  |
| Diplomová práce  2025 |  |
|  |  |

Obrázok, na ktorom je text, snímka obrazovky, písmo, dokument

Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.

Obrázok, na ktorom je text, list, snímka obrazovky, dokument

Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.

**PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE**

**Beru na vědomí, že**

* odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
* diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
* jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
* na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
* podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
* podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
* pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
* pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

**Prohlašuji, že**

* jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
* odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne .............................. ...............................................................

podpis autora

ABSTRAKT

Text abstraktu v jazyce práce.

Klíčová slova: klíčové slovo, klíčové slovo

ABSTRACT

Text abstraktu ve světovém jazyce (angličtině).

Keywords: keyword, keyword

Zde je místo pro případné poděkování, popř. motto, úryvky knih atp.

OBSAH

[Úvod 15](#_Toc199289876)

1. [teoRetická část 18](#_Toc199289877)

[1 Základné princípy evolučných algoritmov 19](#_Toc199289878)

[1.1 Inšpirácia biologickou evolúciou 19](#_Toc199289879)

[1.1.1 Biologické operátory využívané v evolučných algoritmoch 19](#_Toc199289880)

[1.2 Jedinec a populácia v evolučných algoritmoch 20](#_Toc199289881)

[1.2.1 Reprezentácia parametrov jedinca 21](#_Toc199289882)

[1.3 Proces vývoja evolučných algoritmov 21](#_Toc199289883)

[1.3.1 Inicializácia populácie 22](#_Toc199289884)

[1.3.2 Vyhodnotenie kvality jedincov (fitness) 22](#_Toc199289885)

[1.3.3 Výber rodičov 22](#_Toc199289886)

[1.3.4 Kríženie 23](#_Toc199289887)

[1.3.5 Mutácia 24](#_Toc199289888)

[1.3.6 Výber novej generácie 24](#_Toc199289889)

[1.3.7 Kritéria ukončenia evolučného procesu 24](#_Toc199289890)

[2 Základné evolučné algoritmy 26](#_Toc199289891)

[2.1 Genetický algoritmus (GA) 26](#_Toc199289892)

[2.1.1 Jedinec v GA 27](#_Toc199289893)

[2.1.2 Klasický genetický algoritmus (SGA) 28](#_Toc199289894)

[2.1.3 Elitistický genetický algoritmus 28](#_Toc199289895)

[2.1.4 Multiobjektívny genetický algoritmus (MOGA) 28](#_Toc199289896)

[2.1.5 Adaptívny genetický algoritmus 29](#_Toc199289897)

[2.1.6 Paralelný genetický algoritmus 29](#_Toc199289898)

[2.1.7 Pseudokód GA 30](#_Toc199289899)

[2.2 Evolučné stratégie 31](#_Toc199289900)

[2.2.1 Jedinec v ES 31](#_Toc199289901)

[2.2.2 Mutácia v ES 32](#_Toc199289902)

[2.2.3 Rekombinácia v ES 32](#_Toc199289903)

[2.2.4 Selekcia v ES 32](#_Toc199289904)

[2.2.5 Varianty evolučných stratégií 33](#_Toc199289905)

[2.2.6 Pseudokód ES 33](#_Toc199289906)

[2.3 Diferenciálna evolúcia 34](#_Toc199289907)

[2.3.1 Postup evolúcie v DE 34](#_Toc199289908)

[2.3.2 Nastavenie parametrov v DE 35](#_Toc199289909)

[2.3.3 Mutačné stratégie v DE 35](#_Toc199289910)

[2.3.4 Druhy kríženia v DE 37](#_Toc199289911)

[2.3.5 Varianty DE 37](#_Toc199289912)

[2.3.6 Pseudokód DE 38](#_Toc199289913)

[2.4 Genetické programovanie 39](#_Toc199289914)

[2.4.1 Jedinec v GP 39](#_Toc199289915)

[2.4.2 Operátory v GP 40](#_Toc199289916)

[2.4.3 Problém preplnenia kódu 41](#_Toc199289917)

[2.4.4 Aplikácie GP 42](#_Toc199289918)

[2.4.5 Pseudokód GP 42](#_Toc199289919)

[2.5 Gramatická evolúcia 43](#_Toc199289920)

[2.5.1 Jedinec v GE 44](#_Toc199289921)

[2.5.2 Operátory v GE 44](#_Toc199289922)

[2.5.3 Aplikácie GE 44](#_Toc199289923)

[2.5.4 Pseudokód GE 45](#_Toc199289924)

[2.6 Analytické programovanie 46](#_Toc199289925)

[2.6.1 Jedinec v AP 47](#_Toc199289926)

[2.6.2 Aplikácie AP 48](#_Toc199289927)

[2.6.3 Pseudokód AP 48](#_Toc199289928)

[2.7 Porovnanie evolučných algoritmov 50](#_Toc199289929)

[2.7.1 Genetické algoritmy (GA) 50](#_Toc199289930)

[2.7.2 Diferenciálna evolúcia (DE) 50](#_Toc199289931)

[2.7.3 Evolučné stratégie (ES) 50](#_Toc199289932)

[2.7.4 Genetické programovanie (GP) 50](#_Toc199289933)

[2.7.5 Analytické programovanie (AP) 51](#_Toc199289934)

[2.7.6 Gramatická evolúcia (GE) 51](#_Toc199289935)

[2.7.7 Tabuľka porovnania 51](#_Toc199289936)

[3 Evolučné výpočetné techniky s variabilnou dĺžkou jedinca 54](#_Toc199289937)

[3.1 Úvod do techník s variabilnou dĺžkou jedinca 55](#_Toc199289938)

[3.2 Reprezentácie jedinca s variabilnou dĺžkou 55](#_Toc199289939)

[3.2.1 Stromová reprezentácia (Genetické programovanie) 56](#_Toc199289940)

[3.2.2 Lineárna reprezentácia s mapovaním (Gramatická evolúcia) 56](#_Toc199289941)

[3.2.3 3.2.3 Grafická reprezentácia (Kartézske genetické programovanie) 56](#_Toc199289942)

[3.2.4 Symbolická reprezentácia (Analytické programovanie) 57](#_Toc199289943)

[3.3 Evolučné algoritmy podporujúce variabilnú dĺžku 57](#_Toc199289944)

[3.3.1 Genetické programovanie (GP) 57](#_Toc199289945)

[3.3.2 Gramatická evolúcia (GE) 58](#_Toc199289946)

[3.3.3 Kartézske genetické programovanie (CGP) 58](#_Toc199289947)

[3.3.4 Analytické programovanie (AP) 58](#_Toc199289948)

[3.3.5 Ďalšie algoritmy 59](#_Toc199289949)

[3.4 Evolučné operátory pre variabilnú dĺžku jedinca 59](#_Toc199289950)

[3.4.1 Mutácia 59](#_Toc199289951)

[3.4.2 Kríženie 60](#_Toc199289952)

[3.5 Problémy evolučných techník s variabilnou dĺžkou jedinca 60](#_Toc199289953)

[3.5.1 Problém preplnenia kódu (Code bloat) 61](#_Toc199289954)

[3.5.2 Balansovanie medzi exploráciou a exploitáciou 61](#_Toc199289955)

[3.5.3 Optimalizácia veľkosti riešení 62](#_Toc199289956)

[3.5.4 Robustnosť genotyp-fenotyp mapovania 62](#_Toc199289957)

[3.5.5 Evolúcia štruktúr s rôznymi typmi uzlov 62](#_Toc199289958)

[3.6 Techniky kontroly dĺžky riešení 62](#_Toc199289959)

[3.6.1 Parsimónia 63](#_Toc199289960)

[3.6.2 Dynamické limity veľkosti (bloat control) 63](#_Toc199289961)

[3.6.3 Prerušenie rastu stromov 63](#_Toc199289962)

[3.7 Aplikácie evolučných techník s variabilnou dĺžkou 64](#_Toc199289963)

[3.7.1 Symbolická regresia a modelovanie 64](#_Toc199289964)

[3.7.2 Evolučná syntéza programov a algoritmov 64](#_Toc199289965)

[3.7.3 Evolučný návrh štruktúr a topológií 64](#_Toc199289966)

[3.7.4 Evolučné algoritmy v spracovaní obrazu a počítačovom videní 65](#_Toc199289967)

[4 Testovacie funkcie pre evolučné algoritmy 67](#_Toc199289968)

[4.1 Klasifikácia testovacích funkcií 67](#_Toc199289969)

[4.1.1 Unimodálne funkcie 67](#_Toc199289970)

[4.1.2 Multimodálne funkcie 68](#_Toc199289971)

[4.1.3 Škálované funkcie 68](#_Toc199289972)

[4.1.4 Nelineárne funkcie 68](#_Toc199289973)

[4.2 Najčastejšie používané testovacie funkcie 68](#_Toc199289974)

[4.2.1 Sphere 69](#_Toc199289975)

[4.2.2 Rosenbrock 69](#_Toc199289976)

[4.2.3 Rastrigin 69](#_Toc199289977)

[4.2.4 Ackley 70](#_Toc199289978)

[4.2.5 Griewank 70](#_Toc199289979)

[4.2.6 Schwefel 70](#_Toc199289980)

[4.2.7 Michalewicz 71](#_Toc199289981)

[4.3 Testovacie funkcie s variabilnou zložitosťou 71](#_Toc199289982)

[4.3.1 Funkcie s kontrolovaným počtom extrémov 71](#_Toc199289983)

[4.3.2 Dynamicky sa meniace funkcie (dynamic fitness landscapes) 71](#_Toc199289984)

[4.4 Zložitejšie benchmarky a súťaže 72](#_Toc199289985)

[4.4.1 CEC benchmark sets 72](#_Toc199289986)

[4.4.2 Black-Box Optimization Benchmarking (BBOB) 73](#_Toc199289987)

[4.5 Výber vhodnej testovacej funkcie v závislosti od typu algoritmu 74](#_Toc199289988)

1. [praktická část 77](#_Toc199289989)

[5 Výber testovacích funkcií pre experimentálne testovanie 78](#_Toc199289990)

[5.1 Rozdiely medzi testovaním s fixnou a variabilnou dĺžkou jedinca 78](#_Toc199289991)

[5.1.1 Obmedzenia klasických testovacích funkcií pri variabilnej dĺžke 78](#_Toc199289992)

[5.1.2 Potreba benchmarkových problémov s prirodzenou variabilitou 78](#_Toc199289993)

[5.2 Kritéria výberu testovacích funkcií 79](#_Toc199289994)

[5.2.1 Pokrytie unimodálnych a multimodálnych prostredí 79](#_Toc199289995)

[5.2.2 Podpora škálovateľnosti dimenzionality 79](#_Toc199289996)

[5.2.3 Možnosť adaptácie na dynamické zmeny dĺžky genómu 79](#_Toc199289997)

[5.2.4 Zahrnutie numerických aj symbolických problémov 80](#_Toc199289998)

[5.3 Predstavenie vybraných testovacích funkcií a benchmarkov 80](#_Toc199289999)

[5.3.1 Sphere 80](#_Toc199290000)

[5.3.2 Rastrigin 80](#_Toc199290001)

[5.3.3 Griewank 81](#_Toc199290002)

[5.3.4 Ackley 81](#_Toc199290003)

[5.3.5 Santa Fe Trial 81](#_Toc199290004)

[5.3.6 Koza benchmark 82](#_Toc199290005)

[5.3.7 Knapsack problem 82](#_Toc199290006)

[5.3.8 N-bit parity problem 82](#_Toc199290007)

[5.4 Zdôvodnenie výberu použitých testovacích funkcií a benchmarkov 83](#_Toc199290008)

[5.5 Tabuľka použitých testovacích funkcií a benchmarkov 84](#_Toc199290009)

[6 Implementácia existujúcich evolučných výpočetných techník s variabilnou dĺžkou jedinca 87](#_Toc199290010)

[6.1 Spoločné parametre pre implementáciu 87](#_Toc199290011)

[6.1.1 Použitý programovací jazyk a vývojové prostredie 88](#_Toc199290012)

[6.1.2 Použité moduly a knižnice 88](#_Toc199290013)

[6.1.3 Nastavenie experimentov 89](#_Toc199290014)

[6.1.4 Použité testovacie benchmarky 90](#_Toc199290015)

[6.2 Implementácia genetického programovania 91](#_Toc199290016)

[6.2.1 Jedinec v implementácií 94](#_Toc199290017)

[6.2.2 Účelové funkcie 96](#_Toc199290018)

[6.2.3 Operátory ovplyvňujúce dĺžku jedinca 98](#_Toc199290019)

[6.2.4 Funkcia evolučného cyklu 101](#_Toc199290020)

[6.2.5 Zhrnutie použitia variabilnej dĺžky jedinca v implementácií 102](#_Toc199290021)

[6.3 Implementácia gramatickej evolúcie 104](#_Toc199290022)

[6.3.1 Reprezentácia genómu a dekódovanie 105](#_Toc199290023)

[6.3.2 Inicializácia populácie 108](#_Toc199290024)

[6.3.3 Účelové funkcie 108](#_Toc199290025)

[6.3.4 Evolučné operátory 110](#_Toc199290026)

[6.3.5 Regulácia dĺžky 111](#_Toc199290027)

[6.3.6 Evolučný cyklus 111](#_Toc199290028)

[6.3.7 Zhrnutie implementácie 112](#_Toc199290029)

[7 Úprava štandardných evolučných algoritmov pre variabilnú dĺžku jedinca 114](#_Toc199290030)

[7.1 Motivácia pre úpravu dĺžky jedinca 114](#_Toc199290031)

[7.2 Spoločné parametre pre implementácie 115](#_Toc199290032)

[7.3 Genetický algoritmus s variabilnou dĺžkou jedinca 116](#_Toc199290033)

[7.3.1 Reprezentácia jedinca 116](#_Toc199290034)

[7.3.2 Operátory meniace dĺžku jedinca 116](#_Toc199290035)

[7.3.3 Kríženie rodičov rôznych dĺžok 118](#_Toc199290036)

[7.3.4 Evolučný cyklus 118](#_Toc199290037)

[7.3.5 Zhrnutie GA-V 120](#_Toc199290038)

[7.4 Diferenciálna evolúcia s variabilnou dĺžkou 121](#_Toc199290039)

[7.4.1 Dynamický mutant variabilných rozmerov 121](#_Toc199290040)

[7.4.2 Operátor pridania a odstránenia prvku 121](#_Toc199290041)

[7.4.3 Kríženie 122](#_Toc199290042)

[7.4.4 Zachovanie škálovania parametrov F a CR 122](#_Toc199290043)

[7.4.5 Selektívna náhrada a penalizačný termín 123](#_Toc199290044)

[7.4.6 Zhrnutie implementácie 123](#_Toc199290045)

[7.5 CMA-ES s variabilnou dĺžkou 124](#_Toc199290046)

[7.5.1 Dynamická zmena dimenzie 124](#_Toc199290047)

[7.5.2 Aktualizácia hyper-parametrov 125](#_Toc199290048)

[7.5.3 Udržiavanie kovariančnej matice 125](#_Toc199290049)

[7.5.4 Adaptívna kroková veľkosť σ 125](#_Toc199290050)

[7.5.5 Penalizácia dĺžky verzus adaptácia 125](#_Toc199290051)

[7.5.6 Zhrnutie implementácie 126](#_Toc199290052)

[8 Experimentálne testovanie a vyhodnocovanie 127](#_Toc199290053)

[8.1 Ciele experimentov 127](#_Toc199290054)

[8.2 Štatistické metódy použité pri testovaní 128](#_Toc199290055)

[8.2.1 Friedmanov test 128](#_Toc199290056)

[8.2.2 Nemenyi post-hoc test 128](#_Toc199290057)

[8.2.3 Sledované metriky 129](#_Toc199290058)

[8.3 Implementácia zberu dát v algoritmoch 129](#_Toc199290059)

[8.3.1 Štruktúra výstupu v algoritmoch 129](#_Toc199290060)

[8.3.2 Riadenie experimentu a identifikácia replík 130](#_Toc199290061)

[8.3.3 Zápis výsledkov 130](#_Toc199290062)

[8.3.4 Zabezpečenie konzistencie a prenositeľnosti dát 131](#_Toc199290063)

[8.4 Implementácia vyhodnocovania algoritmov 131](#_Toc199290064)

[9 analýza výsledkov 133](#_Toc199290065)

[Závěr 135](#_Toc199290066)

[Seznam použité literatury 137](#_Toc199290067)

[Seznam obrázků 143](#_Toc199290068)

[Seznam tabulek 144](#_Toc199290069)

[Seznam použitých symbolů a zkratek 145](#_Toc199290070)

[Seznam příloh 146](#_Toc199290071)

Úvod

Evolučné výpočtové techniky patria medzi pokročilé optimalizačné metódy inšpirované princípmi prírodnej evolúcie, ako sú selekcia, kríženie a mutácia. Tieto techniky sa úspešne využívajú na riešenie komplexných optimalizačných úloh v rôznych oblastiach, ako sú umelá inteligencia, strojové učenie, návrh algoritmov či inžinierska optimalizácia. Jedným z moderných smerov vo vývoji evolučných algoritmov je využitie variabilnej dĺžky jedinca, ktorá umožňuje dynamickú zmenu štruktúry riešenia počas evolučného procesu. Tento prístup ponúka väčšiu flexibilitu a efektivitu pri riešení problémov s nefixnou štruktúrou riešení, ako sú symbolická regresia, návrh neurónových sietí či automatický návrh programov.

Cieľom tejto diplomovej práce je preskúmať a analyzovať evolučné výpočtové techniky s variabilnou dĺžkou jedinca, implementovať existujúce metódy, navrhnúť úpravy pre štandardné evolučné algoritmy a experimentálne vyhodnotiť ich efektivitu na vhodnej množine testovacích funkcií.

V prvej fáze práce bude vypracovaná literárna rešerš na danú tému, ktorá poskytne teoretický základ a prehľad súčasného stavu výskumu evolučných výpočtových techník. Zameria sa na evolučné algoritmy s variabilnou dĺžkou jedinca, vrátane genetického programovania, gramatickej evolúcie a ďalších adaptívnych prístupov. Zároveň budú identifikované výzvy a otvorené problémy, ako napríklad riadenie komplexity, prevencia prehnaného nárastu dĺžky (bloating) a efektívne operátory kríženia a mutácie pre variabilné štruktúry.

Ďalším krokom bude výber vhodnej množiny testovacích funkcií pre experimentálne vyhodnotenie. Tieto funkcie musia byť dostatočne rozmanité, aby pokryli rôzne optimalizačné scenáre, vrátane multimodálnych a vysoko dimenzionálnych problémov.

Následne budú implementované existujúce evolučné výpočtové techniky s variabilnou dĺžkou jedinca, aby sa overila ich efektivita a správanie na zvolených testovacích funkciách. Pre porovnanie budú tiež upravené štandardné evolučné algoritmy na prácu s variabilnou dĺžkou jedinca, pričom dôraz bude kladený na zachovanie ich pôvodnej výpočtovej efektivity a robustnosti.

V ďalšej fáze budú testované a štatisticky vyhodnotené všetky implementované metódy. Analýza bude riešiť porovnanie výkonnosti v rámci rôznych metrík, ako sú rýchlosť konvergencie, kvalita nájdených riešení a robustnosť voči lokálnym optimám. Na štatistické vyhodnotenie budú použité štandardné metódy, aby boli zabezpečené spoľahlivé a správne výsledky.

Záverečná časť práce bude venovaná formulácii záverov z experimentov, kde budú zosumarizované hlavné zistenia a vyvodené odporúčania pre ďalší výskum.

|  |  |
| --- | --- |
|  | teoRetická část |

Základné princípy evolučných algoritmov

Evolučné algoritmy (EA) predstavujú skupinu optimalizačných metód inšpirovaných biologickými procesmi evolúcie, ako sú prirodzený výber, kríženie a mutácia. Jedná sa o poddruh softcomputingu a umelej inteligencie. Tieto algoritmy sa používajú na riešenie komplexných optimalizačných problémov, ktoré sú pre tradičné analytické postupy ťažko riešiteľné alebo dokonca neriešiteľné. Evolučné algoritmy napodobňujú procesy prebiehajúce v prírode, kde jedince v populácii podstupujú selekciu na základe svojej schopnosti adaptovať sa na prostredie. Táto schopnosť je v kontexte optimalizácie hodnotená pomocou tzv. fitness funkcie (tiež aj účelová funkcia, alebo angl. objective function), ktorá ohodnocuje kvalitu každého riešenia [1][2][5].

Inšpirácia biologickou evolúciou

Evolučné algoritmy sú založené na princípoch biologickej evolúcie, ktoré prvýkrát formalizoval Charles Darwin v teórii prirodzeného výberu. Základná myšlienka spočíva v tom, že jedince, ktoré sú lepšie prispôsobené prostrediu, majú vyššiu šancu prežiť a rozmnožovať sa, čím odovzdávajú svoje genetické informácie nasledujúcim generáciám. Evolučné algoritmy túto myšlienku aplikujú na populáciu kandidátnych riešení, kde každý jedinec predstavuje možné riešenie daného optimalizačného problému. Kvalita jednotlivých riešení je hodnotená pomocou fitness funkcie, ktorá určuje ich schopnosť „prežiť“ v rámci evolučného procesu [2][6].

Biologické operátory využívané v evolučných algoritmoch

Proces evolúcie v prírode obsahuje nasledujúce operátory, ktoré sa adaptovali aj v evolučných algoritmoch. V závislosti od druhu a použitia evolučného algoritmu sa môže meniť napríklad poradie týchto operátorov, prípadne sa v niektorých prípadoch môžu vynechávať. V základe sa ale implementujú tieto operátory:

* **selekcia** – vyberajú sa jedince s vyššou hodnotou účelovej funkcie, ktoré majú vyššiu pravdepodobnosť reprodukcie. Tento mechanizmus zabezpečuje prenos kvalitných genetických informácií do ďalších generácií;
* **kríženie (rekombinácia)** – spojením genetického materiálu dvoch rodičov vznikajú noví potomkovia, ktorí zdedia vlastnosti oboch rodičov. Tento proces zvyšuje genetickú diverzitu populácie a podporuje efektívnejšie prehľadávanie priestoru riešení;
* **mutácia** –náhodné zmeny v genetickom kóde jedincov zabezpečujú genetickú variabilitu populácie, čo umožňuje algoritmu uniknúť z lokálnych optimálnych riešení a priblížiť sa k globálnemu optimu [1][2][6].

Tieto operátory umožňujú evolučným algoritmom efektívne prehľadávať rozsiahle a zložité priestory riešení, čím sa zvyšuje pravdepodobnosť nájdenia globálneho optima.

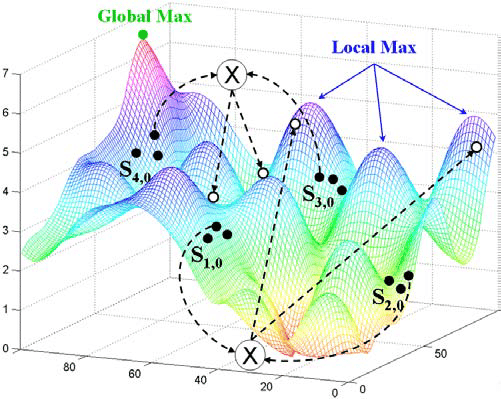
Jedinec a populácia v evolučných algoritmoch

V evolučných algoritmoch každý jedinec predstavuje potenciálne riešenie optimalizačného problému. Jedinec je reprezentovaný geometricky ako vektor v n-rozmernom priestore, kde každá jeho zložka zodpovedá jednej premennej optimalizačného problému. Matematicky je jedinec reprezentovaný nasledovne:

kde xi predstavuje argument (parameter) riešenia a n je počet parametrov. Ďalej môže byť jedinec reprezentovaný pomocou rôznych spôsobov. V závislosti od spôsobu reprezentácie sa mení aj spôsob reprezentácie parametrov. V základe je jedinec reprezentovaný binárne takže hodnoty jeho parametrov môžu nadobúdať hodnoty 0 a 1 [2][6].

Celková populácia je reprezentovaná množinou všetkých jedincov:

kde n je počet jedincov v populácii. Geometricky môžeme populáciu chápať ako množinu bodov v n-rozmernom priestore, kde každý bod predstavuje jedno možné riešenie. Pri vizualizácii v dvoj- alebo trojrozmernom priestore je možné sledovať rozloženie jedincov a dynamiku ich pohybu počas evolučného procesu. Každý jedinec je hodnotený fitness funkciou a jeho hodnotenie sa používa pri aplikovaní výberu pre ďalšie generácie. Hodnotenie býva vo forme celého čísla, ktoré je zapísané ako D+1 parameter jedinca. Je možné konštatovať, že pri hľadaní optimálneho riešenia je hľadaná najmenšia (najväčšia) hodnota v D+1-priestore (n-priestore) [1][5]. Vývoj jedincov v priestore je zachytený na nasledujúcom obrázku [1].



Obrázok 1 Geometrická reprezentácia vývoja jedincov v priestore

Reprezentácia parametrov jedinca

Parametre jedinca sa pri niektorých genetických algoritmoch nazývajú chromozómy. Jednotlivé chromozómy jedincov môžu byť zakódované rôznymi spôsobmi:

1. **binárne kódovanie** – vektor binárnych hodnôt;
2. **reálne kódovanie** – vektor reálnych čísel;
3. **stromová štruktúra** – používa sa pri genetickom programovaní, kde jedinec predstavuje syntaktický strom matematického výrazu;
4. **gramatické kódovanie** – aplikované pri gramatickej evolúcii, kde jedinec predstavuje reťazec produkčných pravidiel formálnej gramatiky [2][6].

Proces vývoja evolučných algoritmov

Evolučné algoritmy sa vyznačujú špecifickou štruktúrou, ktorá napodobňuje prirodzené evolučné procesy. Táto štruktúra pozostáva z niekoľkých základných krokov, ktoré sa opakujú v cykle, až kým nie je dosiahnuté kritérium ukončenia. Nasledujúci text poskytuje podrobný popis jednotlivých častí štruktúry evolučných algoritmov.

Inicializácia populácie

Proces evolučných algoritmov začína inicializáciou počiatočnej populácie jedincov. Populácia predstavuje množinu možných riešení daného optimalizačného problému. Každý jedinec v populácii je reprezentovaný v podobe chromozómu, ktorý môže byť tvorený binárnym reťazcom, reálnymi číslami alebo inými dátovými štruktúrami v závislosti od konkrétneho typu algoritmu a povahy riešeného problému.

Veľkosť počiatočnej populácie sa volí na základe kompromisu medzi výpočtovou náročnosťou a rozmanitosťou riešení. Väčšia populácia zabezpečuje lepšie pokrytie vyhľadávacieho priestoru, ale zvyšuje výpočtovú záťaž. Naopak, menšia populácia znižuje výpočtovú zložitosť, ale môže viesť k predčasnej konvergencii [1].

Inicializácia sa najčastejšie vykonáva náhodným generovaním jedincov v rámci povoleného vyhľadávacieho priestoru. Existujú však aj pokročilé prístupy, ako napríklad použitie heuristík na vytvorenie počiatočných jedincov s lepšími vlastnosťami, čím sa urýchli proces konvergencie [2].

Vyhodnotenie kvality jedincov (fitness)

Po inicializácii populácie nasleduje krok vyhodnotenia kvality jednotlivých riešení pomocou fitness funkcie. Fitness funkcia je matematická reprezentácia cieľovej úlohy, ktorá kvantifikuje kvalitu každého jedinca. Cieľom evolučného algoritmu je maximalizovať alebo minimalizovať túto hodnotu v závislosti od povahy optimalizačného problému [1].

Fitness funkcia (angl. objective function) môže byť jednoduchá lineárna funkcia alebo zložitý matematický model, v závislosti od typu problému. Napríklad pri optimalizácii funkcií sa často používajú štandardné testovacie funkcie ako Sphere, Rastrigin alebo Ackley [3]. V prípade problémov v kombinatorike môže byť fitness funkciou napríklad dĺžka cesty v probléme obchodného cestujúceho [1].

Výber rodičov

Výber rodičov je krokom, ktorý určuje, ktoré jedince sa zúčastnia tvorby novej generácie. Cieľom je zabezpečiť, aby kvalitnejšie riešenia mali vyššiu pravdepodobnosť prežitia a rozmnožovania, čím sa zabezpečuje postupné zlepšovanie populácie. Existuje viacero metód výberu rodičov:

* **turnajový výber –** náhodne sa vyberie niekoľko jedincov z populácie a víťazom sa stáva jedinec s najlepšou fitness hodnotou. Tento postup sa opakuje, až kým sa nevyberie požadovaný počet rodičov;
* **Ruletová selekcia –** pravdepodobnosť výberu jednotlivca je úmerná jeho fitness hodnote. Tento prístup pripomína otáčanie rulety, kde jedinci s vyššou kvalitou zaberajú väčšiu časť rulety;
* **Stochastický výber –** modifikácia proporcionálneho výberu, kde sa naraz vyberie viacero rodičov, čím sa znižuje šanca na výber extrémne dobrých alebo zlých jedincov;
* **Elitizmus –** niektorí najlepší jedinci sa priamo prenášajú do ďalšej generácie bez zmeny, čím sa zabezpečuje, že najlepšie riešenia nie sú stratené [4].

Kríženie

Kríženie predstavuje základný genetický operátor, ktorým sa vytvárajú noví potomkovia kombináciou genetickej informácie dvoch alebo viacerých rodičov. Cieľom kríženia je zabezpečiť prenos výhodných vlastností do novej generácie a tým zlepšovať kvalitu populácie [1].

Existuje niekoľko typov kríženia:

* **jednobodové kríženie** – chromozómy rodičov sa rozdelia na dvoch miestach a časti sa navzájom vymenia;
* **dvojbodové kríženie** – rozdelenie sa uskutoční na dvoch miestach a stredná časť sa vymení;
* **uniformné kríženie** – každý gén sa nezávisle vyberá od jedného z rodičov s určitou pravdepodobnosťou;
* **aritmetické kríženie** – používa sa v prípadoch, keď sú chromozómy reprezentované reálnymi číslami. Nové gény sa vypočítajú ako lineárna kombinácia génov rodičov [5].

Mutácia

Mutácia je operátor, ktorý mení hodnoty génov u jedincov s nízkou pravdepodobnosťou. Cieľom mutácie je zabezpečiť diverzitu v populácii a predísť uviaznutiu algoritmu v lokálnom optime. Slúži vlastne ako poistka proti takémuto uviaznutiu.

* **bitová mutácia** – používa sa pri binárnej reprezentácii chromozómov, kde sa náhodne zmení hodnota bitu (napr. z 0 na 1 alebo naopak);
* **gaussova mutácia** – používa sa pri reálnej reprezentácii, kde sa k hodnote génu pripočíta náhodná hodnota vygenerovaná z normálneho rozdelenia;
* **Inverzia** – prevráti sa poradie génov v určitej časti chromozómu. Tento typ mutácie je obzvlášť užitočný pri kombinatorických problémoch [6].

Výber novej generácie

Po vykonaní operácií kríženia a mutácie vznikne nová množina jedincov – potomkovia. Aby evolučný algoritmus pokračoval v ďalšej iterácii, musí byť vytvorená nová generácia jedincov. Tento krok zahŕňa výber jednotlivcov, ktorí budú pokračovať do nasledujúcej generácie. Existuje viacero prístupov, ako tento výber uskutočniť:

* **generačná selekcia** – celá rodičovská populácia je nahradená potomkami. Tento prístup zabezpečuje vysokú mieru diverzity, avšak môže dôjsť k strate kvalitných riešení;
* **Elitizmus** – najlepší jedinci z rodičovskej populácie sú priamo prenesení do novej generácie. Tento prístup zabezpečuje, že najkvalitnejšie riešenia nie sú stratené;
* **turnajová selekcia** – jedinci z rodičovskej aj potomkovej populácie sú náhodne vybraní a medzi sebou súťažia o postup do ďalšej generácie. Tento prístup umožňuje zachovanie rozmanitosti a zároveň zvyšuje pravdepodobnosť prežitia kvalitných riešení;
* **(μ + λ) selekcia** – používa sa v evolučných stratégiách, kde μ jedincov je vybraných z kombinácie rodičov (μ) a potomkov (λ). Tento prístup zaisťuje, že najlepší jedinci z oboch populácií budú pokračovať do ďalšej generácie [1].

Kritéria ukončenia evolučného procesu

Evolučné algoritmy sú iteratívne procesy, ktoré pokračujú až do splnenia definovaného kritéria ukončenia. Výber vhodného kritéria je dôležitý pre zabezpečenie toho, aby sa zabezpečilo efektívne využitie výpočtovej techniky a zároveň sa dosiahlo optimálne riešenie. Najčastejšie používané kritériá zahŕňajú:

* **maximálny počet generácií** – algoritmus sa ukončí po vykonaní vopred stanoveného počtu generácií. Toto kritérium zabezpečuje kontrolu nad výpočtovou náročnosťou;
* **konvergencia hodnotenia účelovej funkcie** – ak sa hodnota kvality najlepšieho jedinca nemení počas niekoľkých generácií, algoritmus sa ukončí. Tento prístup naznačuje, že algoritmus dosiahol lokálne alebo globálne optimum;
* **dosiahnutie cieľovej hodnoty účelovej funkcie** – algoritmus sa ukončí, ak je dosiahnutá alebo prekročená vopred definovaná cieľová hodnota;
* **časové obmedzenie** – evolučný algoritmus sa ukončí po uplynutí určitého času, čo je užitočné pri výpočtoch s obmedzenými časovými zdrojmi;
* **manuálne ukončenie** – v niektorých prípadoch môže byť algoritmus manuálne zastavený, napríklad ak používateľ zistí, že konvergencia je príliš pomalá alebo výsledky nie sú uspokojivé [2].

Základné evolučné algoritmy

Evolučné algoritmy implementujú rôzne spôsoby vývoja populácie, hodnotenia, alebo ukončovania vývoja. Na základe tohto je implementovaných niekoľko typov evolučných algoritmov, ktoré sa ďalej vyvinuli na rôzne podtypy. V nasledujúcej časti budú predstavené používané typy evolučných algoritmov.

Genetický algoritmus (GA)

Genetické algoritmy (GA) patria medzi najznámejšie a najpoužívanejšie evolučné algoritmy. Boli prvýkrát predstavené Johnom Hollandom v 60. rokoch 20. storočia a sú inšpirované Darwinovou teóriou evolúcie prirodzeným výberom [3]. Hlavným cieľom genetických algoritmov je hľadanie optimálneho riešenia daného problému prostredníctvom simulácie evolučného procesu, ktorý zahŕňa mechanizmy selekcie, kríženia a mutácie.

Genetické algoritmy pracujú s populáciou jedincov, kde každý jedinec reprezentuje jedno potenciálne riešenie problému. Jedinci sú reprezentovaní vo forme chromozómov, ktoré sú vo väčšine prípadov zakódované ako binárne reťazce (0 a 1), no môžu byť aj reálne čísla, permutácie alebo stromové štruktúry [7].

Každý jedinec je hodnotený pomocou fitness funkcie, ktorá určuje kvalitu riešenia, ktoré reprezentuje daný jedinec. Cieľom genetických algoritmov je maximalizovať alebo minimalizovať hodnotu fitness funkcie. Vývoj populácie prebieha v iteráciách nazývaných generácie, pričom v každej generácii sa aplikujú evolučné operátory:

* **selekcia** – výber jedincov na základe ich kvality, pričom lepší jedinci majú vyššiu pravdepodobnosť reprodukcie;
* **kríženie (crossover)** – kombinácia genetickej informácie dvoch jedincov za účelom vytvorenia nových potomkov;
* **mutácia** – náhodná zmena génov jedinca, ktorá zabezpečuje genetickú diverzitu v populácii.

Proces pokračuje až do splnenia ukončovacej podmienky (podmienok), ktoré môžu byť definované ako:

* Dosiahnutie určitého počtu generácií;
* Nájdenie riešenia s požadovanou kvalitou (hodnotou fitness);
* Stagnácia populácie – ak sa hodnota účelovej funkcie nemení počas niekoľkých generácií [8].

Proces vývoja algoritmu GA je možné zhrnúť do nasledujúcich krokov:

1. **Inicializácia populácie** – Náhodné vytvorenie počiatočnej populácie jedincov.
2. **Vyhodnotenie fitness funkcie** – Ohodnotenie kvality každého jedinca pomocou fitness funkcie.
3. **Selekcia** – Výber rodičov na základe ich fitness hodnoty.
4. **Kríženie a mutácia** – Vytvorenie novej populácie pomocou kríženia a mutácie.
5. **Nahradenie populácie** – Pôvodná populácia je nahradená novou generáciou.
6. **Kontrola ukončovacej podmienky** – Ak je splnené kritérium zastavenia, algoritmus sa ukončí, inak sa opakuje krok 2.

Genetické algoritmy sa vyznačujú robustnosťou a schopnosťou efektívne prehľadávať veľké a zložité priestorové oblasti riešení. Ich nevýhodou môže byť riziko predčasnej konvergencie a vyššie nároky na výpočtový výkon pri komplexných problémoch [5].

Jedinec v GA

Jedinec a jeho reprezentácia sú dôležitým aspektom genetických algoritmov, pretože určuje, ako bude riešenie zakódované a aké evolučné operátory budú aplikované. Existuje niekoľko rôznych spôsobov reprezentácie jedincov:

1. **Binárna reprezentácia** – Najčastejšie používaná reprezentácia, kde sú chromozómy zakódované ako reťazce bitov (0 a 1). Táto forma je jednoduchá na implementáciu a efektívna pri riešení diskrétnych problémov [8].
2. **Reálna reprezentácia** – Používa sa pri optimalizačných problémoch s reálnymi číslami. Chromozómy obsahujú skutočné hodnoty, čo umožňuje presnejšiu manipuláciu s číselnými údajmi [4].
3. **Celočíselná reprezentácia** – Využíva sa pre optimalizačné problémy, kde premenné nadobúdajú diskrétne hodnoty v určitom intervale
4. **Permutačná reprezentácia** – Je vhodná pre úlohy, kde je dôležité usporiadanie prvkov, napr. problém obchodného cestujúceho (TSP). Chromozóm obsahuje permutáciu čísel, ktoré predstavujú poradové čísla prvkov [2].

Genetické algoritmy prešli od svojho vzniku mnohými modifikáciami a vylepšeniami. Existuje niekoľko rôznych variant, ktoré sú optimalizované pre špecifické typy problémov alebo využívajú iné evolučné operátory na zlepšenie výkonu a efektivity algoritmu. V nasledujúcej časti je popísaných niekoľko variant genetických algoritmov.

Klasický genetický algoritmus (SGA)

Klasický genetický algoritmus je základnou verziou genetického algoritmu, ako ju definoval John Holland [3]. Tento variant využíva binárnu reprezentáciu chromozómov a aplikáciu troch základných operátorov:

* **ruletová selekcia** – pravdepodobnosť výberu jedinca je úmerná jeho kvalite;
* **jednobodové alebo dvojbodové kríženie** – chromozómy rodičov sa rozdelia na základe jedného alebo dvoch náhodne vybraných bodov a časti chromozómov sa medzi rodičmi vymenia;
* **jednoduchá bitová mutácia** – náhodná zmena bitu v chromozóme (napr. z 0 na 1 a naopak) s nízkou pravdepodobnosťou.

Hoci je klasický genetický algoritmus jednoduchý na implementáciu, jeho nevýhodou je vyššia tendencia k predčasnej konvergencii, čo môže viesť k lokálnym optimám [8].

Elitistický genetický algoritmus

Elitistický genetický algoritmus pridáva k jednoduchému genetickému algoritmu koncept elitizmu, ktorý zabezpečuje, že najlepší jedinci z aktuálnej generácie sa prenesú do nasledujúcej generácie bez akejkoľvek zmeny. Tento prístup pomáha udržiavať najlepšie nájdené riešenie počas celého evolučného procesu a znižuje riziko jeho straty v dôsledku náhodných mutácií alebo nevhodných kombinácií pri krížení [5].

Mechanizmus funguje tak, že po vyhodnotení fitness funkcie sa vyberie n najlepších jedincov (tzv. elita), ktorí sa priamo prenesú do ďalšej generácie. Zvyšok populácie sa generuje pomocou štandardných operátorov selekcie, kríženia a mutácie. Tento prístup zrýchľuje konvergenciu, avšak znižuje genetickú diverzitu v populácii.

Multiobjektívny genetický algoritmus (MOGA)

Mnoho reálnych optimalizačných problémov zahŕňa viacero cieľových funkcií, ktoré sú často vo vzájomnom konflikte (napr. minimalizácia nákladov a maximalizácia kvality). Multiobjektívne genetické algoritmy sa snažia optimalizovať tieto cieľové funkcie naraz a nájsť tzv. Pareto optimálne riešenia, kde žiadne iné riešenie nie je lepšie vo všetkých cieľoch naraz [12].

Najznámejším algoritmom v tejto kategórii je NSGA-II (Non-dominated Sorting Genetic Algorithm II), ktorý využíva:

* **nedominované triedenie** – jedince sú triedené podľa toho, koľko riešení ich dominuje (riešenie A dominuje riešeniu B, ak je lepšie vo všetkých cieľových funkciách);
* **crowding distance** – používa sa na udržanie rozmanitosti populácie v rámci Pareto čela.

Tento prístup umožňuje získanie sady riešení predstavujúcich kompromisy medzi cieľovými funkciami, z ktorých si používateľ môže vybrať podľa svojich preferencií [11].

Adaptívny genetický algoritmus

Adaptívne genetické algoritmy dynamicky menia pravdepodobnosti evolučných operátorov počas výpočtu. To znamená, že napríklad pravdepodobnosť mutácie alebo kríženia sa mení v závislosti od vývoja populácie. Tento prístup pomáha predísť predčasnej konvergencii a umožňuje lepšiu adaptáciu na rôzne fázy optimalizačného procesu [4].

Adaptívna mutácia – Pravdepodobnosť mutácie sa zvyšuje, ak sa populácia začína koncentrovať okolo jedného riešenia (nízka diverzita). Naopak, ak je populácia dostatočne rozmanitá, pravdepodobnosť mutácie sa znižuje.

Adaptívne kríženie – Pravdepodobnosť kríženia sa mení na základe fitness hodnoty rodičov. Ak sú rodičia veľmi odlišní, je vyššia šanca na vytvorenie kvalitného potomka, a preto sa pravdepodobnosť kríženia zvyšuje [5].

Paralelný genetický algoritmus

Paralelné genetické algoritmy využívajú viacero populácií na paralelné prehľadávanie priestoru riešení. Jednotlivé populácie sa vyvíjajú nezávisle od seba, pričom medzi nimi prebieha migrácia jedincov po určitom počte generácií. Tento prístup umožňuje efektívnejšie využitie výpočtových zdrojov a zvyšuje diverzitu populácie [6].

Existujú rôzne topológie pre komunikáciu medzi populáciami, ako napríklad:

* **kruh** – každá populácia komunikuje len so svojimi dvoma susedmi.
* **hyperkocka** – populácie sú usporiadané do n-rozmernej kocky.
* **globálna komunikácia** – každá populácia môže komunikovať s ktoroukoľvek inou.

Tento variant sa často používa na distribuovaných a paralelných výpočtových platformách, čo umožňuje efektívnejšie riešenie veľkých a zložitých optimalizačných problémov [13].

Tieto varianty genetických algoritmov rozširujú základný koncept a umožňujú ich aplikáciu na širokú škálu problémov od jednoduchých optimalizačných úloh až po komplexné viacúčelové a distribuované systémy.

Pseudokód GA

Nasledujúci pseudokód predstavuje základnú implementáciu genetického algoritmu. Bol vytvorený použitím zdrojov [5][8].

Inicializácia:

Inicializuj populáciu P = {x1, x2, ..., xN}, kde xi ∈ Ω (prehľadávaný priestor)

Nastav parametre: Pc (pravdepodobnosť kríženia), Pm (pravdepodobnosť mutácie), max\_generácie

kým nie je splnená ukončovacia podmienka vykovaj

Vyhodnoť vhodnosť jednotlivcov v populácii P pomocou cieľovej funkcie f(x)

Normalizuj vhodnosť pre výber rodičov

Nová populácia Q = {}

kým |Q| < N vykonaj

// Výber rodičov

Vyber dvoch rodičov x1 a x2 pomocou metódy výberu (napr. ruleta alebo turnaj)

// Kríženie

Ak rand(0, 1) < Pc potom

Vykonaj kríženie na x1 a x2 za vzniku potomkov y1 a y2

Inak

y1 = x1

y2 = x2

Koniec ak

// Mutácia

pre každého potomka y vykonaj

pre každý génu yj vykonaj

Ak rand(0, 1) < Pm potom

Mutuj gén yj (napr. inverziou alebo náhodnou zmenou hodnoty)

Koniec ak

Koniec pre

Koniec pre

Pridaj y1 a y2 do novej populácie Q

Koniec kým

Aktualizuj populáciu: P = Q

Koniec kým

Evolučné stratégie

Evolučné stratégie (ES) sú špecifický typ evolučných algoritmov, ktoré sú primárne zamerané na optimalizáciu reálnych parametrov. Ich základnou charakteristikou je využívanie mutácie ako hlavného evolučného operátora a selekcie na báze princípu prežitia najvhodnejších jedincov. Tento prístup sa ukázal ako veľmi efektívny pri riešení optimalizačných problémov v spojitých priestoroch, čo robí evolučné stratégie vhodnými pre široké spektrum inžinierskych aplikácií [2][5].

Evolučné stratégie vznikli v 60. rokoch 20. storočia v Nemecku ako výsledok výskumu Ingo Rechenberga a Hansa-Paula Schwefela. Pôvodne boli vyvinuté na optimalizáciu aerodynamických tvarov a inžinierskych návrhov [2]. Rechenberg zaviedol koncept jednoprvkovej evolučnej stratégie (1 + 1), kde sa z jedného rodiča mutáciou vytvorí jeden potomok a medzi nimi prebieha kompetícia o prežitie. Tento jednoduchý model sa neskôr rozvinul na zložitejšie varianty, ako sú (μ, λ) a (μ + λ) stratégie, ktoré umožňujú využívať väčšie populácie jedincov a efektívnejšie prehľadávanie hľadacieho priestoru [2][5][6].

Jedinec v ES

V evolučných stratégiách je každý jedinec reprezentovaný ako vektor reálnych čísel, kde každá zložka vektora predstavuje jednu optimalizovanú premennú. Tento spôsob reprezentácie je vhodný pre problémy, kde je potrebné optimalizovať spojité parametre, napríklad fyzikálne alebo technické vlastnosti navrhovaného riešenia [2][6].

Mutácia v ES

Mutácia je kľúčovým operátorom v evolučných stratégiách a zodpovedá za náhodné perturbácie parametrov jedinca, čo umožňuje prehľadávanie hľadaného priestoru a nachádzanie optimálnych riešení.

V klasických ES sa mutácia realizuje pridaním náhodného šumu z normálového rozdelenia:

Kde x‘ je mutant, inak označovaný aj ako potomok, x je rodič, σ je vektor mutačných krokov a N(0,1) je normálové rozdelenie.

Veľkosť mutačného kroku môže byť konštantná alebo adaptívna. Adaptívna mutácia, nazývaná aj „self-adaptation“, umožňuje dynamickú zmenu veľkosti mutačného kroku počas evolúcie, čo zlepšuje schopnosť algoritmu prehľadávať daný priestor [2][5].

Rekombinácia v ES

Rekombinácia v evolučných stratégiách sa používa na kombinovanie parametrov viacerých rodičov za účelom vytvorenia nového potomka.

Existujú rôzne typy rekombinácie:

* globálna rekombinácia, kde sa každý parameter potomka vyberá z náhodného rodiča;
* aritmetická rekombinácia, kde sú parametre potomka vypočítané ako priemery hodnôt rodičov.

Rekombinácia zvyšuje variabilitu populácie a umožňuje lepšie prehľadávanie hľadacieho priestoru [2][5][6].

Selekcia v ES

Evolučné stratégie využívajú selekciu na základe princípu prežitia najvhodnejších jedincov (angl. survival of the fittest). Najčastejšie sa používajú dva hlavné prístupy:

* **(μ, λ)** – z λ potomkov sa vyberie μ najlepších jedincov pre nasledujúcu generáciu. Rodičia sa však neprenášajú do ďalšej generácie, čo podporuje vyššiu mieru inovácie;
* **(μ + λ)** – μ jedincov sa vyberá zo spojenej množiny μ rodičov a λ potomkov. Tento prístup zaručuje elitizmus, teda zachovanie najlepších jedincov v populácii [2][5].

Varianty evolučných stratégií

Evolučné stratégie najčastejšie využívajú nasledujúce varianty.

* **(1 + 1) ES** – najjednoduchšia forma s jedným rodičom a jedným potomkom;
* **(μ + λ) ES** – využíva väčšiu populáciu a selekciu s elitizmom;
* **(μ, λ) ES** – bez elitizmu, vhodná pre diverzifikáciu populácie;
* **CMA-ES** (Covariance Matrix Adaptation ES) – pokročilá verzia, ktorá dynamicky prispôsobuje kovariančnú maticu mutácií, čo zlepšuje prehľadávanie nelineárnych priestorov [9].

Evolučné stratégie sa využívajú v širokom spektre aplikácií, najmä na riešenie inžinierskych optimalizačných problémov, strojové učenie a výpočtovej technike [2][6].

Pseudokód ES

Nasledujúci pseudokód ilustruje základnú implementáciu evolučnej stratégie. Kód bol vytvorený použitím zdrojov [2][5][12].

Inicializácia:

Inicializuj populáciu P = { (x1, σ1), (x2, σ2), ..., (xN, σN) }, kde xi ∈ Ω a σi je smerodajná odchýlka mutácie

Nastav parametre: μ (počet rodičov), λ (počet potomkov), max\_generácie

kým nie je splnená ukončovacia podmienka vykonaj

Nová populácia Q = {}

// Generovanie potomkov

pre i = 1 to λ vykonaj

Vyber rodiča (alebo rodičov) z populácie P náhodne alebo na základe vhodnosti

// Mutácia

Potomok y = xi + σi \* N(0,1) // Normálové rozdelenie s priemerom 0 a smerodajnou odchýlkou 1

Nová odchýlka σ' = σi \* exp(τ' \* N(0,1) + τ \* N(0,1))

Pridaj (y, σ') do populácie Q

Koniec pre

// Selekcia

Ak (μ + λ) stratégia potom

Spoj populácie P a Q a vyber μ najlepších podľa hodnoty f(x)

Inak ak (μ, λ) stratégia potom

Vyber μ najlepších iba z potomkov v Q

Koniec ak

Aktualizuj populáciu: P = vybrané najlepšie jedince

Koniec kým

Diferenciálna evolúcia

Diferenciálna evolúcia (DE) je evolučný algoritmus, ktorý navrhli Price, Storn a Lampinen [4]. Tento algoritmus je určený na riešenie globálnych optimalizačných problémov v kontinuálnych prehľadávaných priestoroch a vyznačuje sa jednoduchosťou, robustnosťou a efektivitou [1]. V porovnaní s inými evolučnými algoritmami, ako sú genetické algoritmy, DE nevyžaduje predpoklady o deriváciách alebo špecifickom tvare („hladkosti“) cieľovej funkcie, čo jej umožňuje úspešne sa vysporiadať aj s nelineárnymi a multimodálnymi úlohami [5].

Algoritmus DE pracuje s pevne veľkou populáciou kandidátnych riešení, ktoré sú reprezentované ako vektory s reálnymi číslami. V rámci každej iterácie sa na evolúciu populácie používajú tri hlavné operátory: mutácia, kríženie a selekcia. Operátory sa používajú presne v tomto poradí, takže sa nejedná o tradičné použitie operátorov ako napríklad pri genetických algoritmoch [4].

Postup evolúcie v DE

Prvým krokom evolúcie je inicializácia počiatočnej populácie. Populácia sa inicializuje náhodným rozmiestnením jednotlivcov (vektorov) v rámci povoleného hľadaného priestoru. Každý vektor predstavuje jedno možné riešenie optimalizačnej úlohy [6].

Následne jednotlivci mutujú. Pre každý vektor rodiča xi​ sa generuje mutačný vektor vi pomocou kombinácie troch náhodne vybraných vektorov xr1, xr2a xr3 pri implementovaní základnej mutačnej stratégie podľa vzorca [4]:

kde F je faktor mutácie, ktorý je typicky v rozsahu od 0,4 do 1,0 [2]. Tento parameter určuje mieru znásobenia rozdielu medzi dvoma vektormi, čím ovplyvňuje diverzitu populácie.

V ďalšom kroku prebieha kríženie. Mutačný vektor vi​ (mutant) sa následne kombinuje s pôvodným vektorom rodiča xi ​ prostredníctvom krížovej operácie. Výsledkom je tzv. skúšobný vektor ui​ (angl. trial vector), ktorý je tvorený náhodným výberom komponentov z mutanta a rodičovského vektora. Kríženie je riadené pravdepodobnosťou CR (angl. crossover rate), ktorá určuje pravdepodobnosť, že sa zvolí hodnota z mutačného vektora [4].

Vektor ui​ sa následne porovnáva s pôvodným rodičovským vektorom xi​ na základe hodnoty cieľovej (fitness) funkcie. Ak skúšobný vektor prináša lepšie riešenie, nahradí cieľový vektor v novej generácii populácie [5].

Nastavenie parametrov v DE

Úspešnosť DE algoritmu je výrazne ovplyvnená správnym nastavením jeho parametrov. Týmito parametrami sú: veľkosť populácie (NP), faktor mutácie (F) a pravdepodobnosť kríženia (CR) [4].

Typicky sa odporúča nastaviť veľkosť populácie na 5 až 10 násobok dimenzie problému. Väčšia populácia poskytuje lepšie preskúmanie prehľadávaného priestoru, ale zvyšuje výpočtovú náročnosť [2].

Hodnota faktora mutácie zvyčajne spadá do intervalu <0,4; 1,0>. Vyššie hodnoty FFF zvyšujú diverzitu populácie, no môžu znížiť konvergenciu algoritmu [4].

Pravdepodobnosť kríženia riadi, aká časť skúšobného vektora bude pochádzať z vektoru mutanta. Bežné hodnoty sú v rozsahu <0,6; 1,0> [6].

Mutačné stratégie v DE

Mutačné stratégie sú dôležitým prvkom diferenciálnej evolúcie (DE) a zásadne ovplyvňujú výkon algoritmu. Diferenciálna evolúcia využíva niekoľko variantov mutácie, ktoré sa líšia spôsobom výberu vektorov a výpočtu vektoru mutanta [4]. Každá stratégia má svoje špecifické vlastnosti a je vhodná pre rôzne typy optimalizačných úloh [5].

Mutant sa vo všeobecnosti generuje podľa vzorca, ktorý bol popísaný pri základnom postupe diferenciálnej evolúcie. Existuje však viacero stratégií, ako tieto vektory vyberať a kombinovať. Stratégie v DE sú zapisované vo formáte DE/m/n/c, kde m je typ populácie, n je počet rozdielov jedincov, ktorí sú vybraní v mutácií a c je typ kríženia. Nasledujúce stratégie sú bežne pri diferenciálnej evolúcií používané.

* **DE/rand/1** je definovaná vzorcom:

Táto stratégia je najbežnejšia a využíva náhodne vybrané vektory. Je jednoduchá na implementáciu a poskytuje dobrú diverzitu populácie, ale niekedy môže viesť k pomalšej konvergencii [4].

* **DE/best/1**, ktorá je definovaná vzorcom:

V tomto prípade sa na tvorbu mutačného vektora používa najlepší vektor populácie xbest . Stratégia zvyšuje rýchlosť konvergencie, avšak môže spôsobiť predčasnú konvergenciu do lokálneho optima [6].

* **DE/rand-to-best/1**, podľa vzorca:

Táto stratégia kombinuje informácie z najlepšieho vektora a náhodne vybraných vektorov, čím dosahuje lepšiu rovnováhu medzi preskúmavaním a využívaním priestoru riešení [2].

* **DE/rand/2** používa vzorec:

Pri tejto stratégii sa využívajú štyri náhodne vybrané vektory, čo zvyšuje diverzitu, ale aj výpočtovú náročnosť [4].

* **DE/best/2** podľa vzorca:

Kombinuje informácie z najlepšieho vektora s dvoma rozdielovými vektormi, čím zvyšuje konvergenciu, ale zároveň riziko predčasnej konvergencie [5].

Druhy kríženia v DE

Kríženie v DE sa používa na tvorbu vektora ui kombináciou rodičovského vektora xi a vektoru mutanta vi. Existujú dva hlavné typy kríženia:

1. **binomické kríženie (angl. binomial crossover) –** pri tomto type kríženia sa pre každý komponent skúšobného vektora náhodne rozhoduje, či sa hodnota prevezme z mutanta alebo rodičovského vektora. Pravdepodobnosť výberu hodnoty z mutanta je určená pomocou parametra CR.
2. **exponenciálne kríženie (angl. exponential crossover)** – exponenciálne kríženie vyberá niekoľko po sebe nasledujúcich komponentov z mutačného vektora, začínajúc náhodným indexom. Dĺžka tohto segmentu je určená opäť pomocou parametra CR. Tento typ kríženia je menej častý, ale môže byť efektívny pri problémoch, kde sú optimálne riešenia usporiadané v susedných dimenziách [6].

Varianty DE

Diferenciálna evolúcia má mnoho variant, ktoré vznikli s cieľom vylepšiť výkon a adaptovať algoritmus na špecifické typy optimalizačných úloh [4]. Najvýznamnejšie varianty sú:

1. **Self-adaptive Differential Evolution (SaDE)** – v tomto variante sa parametre ako faktor mutácie F a crossover rate CR adaptujú počas behu algoritmu na základe ich historickej výkonnosti. Cieľom je zvýšiť robustnosť a znížiť citlivosť na počiatočné nastavenia [5];
2. **Differential Evolution with Local Search (DELS)** – táto verzia kombinuje globálne vyhľadávanie DE s lokálnym prehľadávaním, čím dosahuje rýchlejšiu a presnejšiu konvergenciu, najmä pri multimodálnych úlohách [4];
3. **Opposition-based Differential Evolution (ODE)** – ODE využíva myšlienku opozitných riešení, kde sa spolu s každým kandidátnym riešením hodnotí aj jeho náprotivok. To vedie k efektívnejšiemu preskúmavaniu prehľadávaného priestoru a zvyšuje pravdepodobnosť nájdenia globálneho optima [6];
4. **Multi-objective Differential Evolution (MODE)** – MODE je navrhnutá pre viacúčelové optimalizačné problémy. Namiesto jedného cieľa optimalizuje niekoľko cieľových funkcií súčasne a hľadá Pareto-optimálny front riešení [4];
5. **Hybrid Differential Evolution (HDE)** – HDE kombinuje DE s inými optimalizačnými algoritmami, ako sú napr. genetické algoritmy alebo rojové algoritmy (PSO), aby využila výhody viacerých metód a zlepšila konvergenciu [1].

Pseudokód DE

Nasledujúci pseudokód predstavuje implementáciu základného algoritmu diferenciálnej evolúcie so stratégiou DE/rand/1/bin. Kód bol vytvorený pomocou zdrojov [4][5][6].

Inicializácia:

Inicializuj populáciu P = {x1, x2, ..., xN}, kde xi ∈ Ω (prehľadávaný priestor)

Nastav parametre: F (faktor mutácie), CR (pravdepodobnosť kríženia), max\_generácie

kým nie je splnená ukončovacia podmienka vykonaj

pre i = 1 to N vykonaj

// Mutácia

Vyber náhodné indexy r1, r2, r3 z {1, ..., N}, kde r1 ≠ r2 ≠ r3 ≠ i

Vypočítaj vektor mutanta:

vi = xr1 + F \* (xr2 - xr3)

// Kríženie

Inicializuj vektor ui

pre j = 1 to D vykonaj

Ak (rand(0, 1) < CR alebo j = j\_rand) potom

uij = vij

Inak

uij = xij

Koniec ak

Koniec pre

// Selekcia

Ak f(ui) ≤ f(xi) potom

yi = ui

Inak

yi = xi

Koniec ak

Vlož yi do novej populácie Q

Koniec pre

Aktualizuj populáciu: P = Q

Koniec kým

Genetické programovanie

Genetické programovanie (GP) je evolučná výpočtová technika, ktorá sa zameriava na automatickú tvorbu počítačových programov prostredníctvom procesov inšpirovaných biologickou evolúciou. GP je rozšírením genetických algoritmov, avšak namiesto optimalizácie číselných hodnôt hľadá optimálnu štruktúru programu alebo matematického výrazu, ktorý najlepšie rieši zadaný problém [3][7].

Genetické programovanie operuje s populáciou jedincov, ktorí predstavujú stromové štruktúry kódu. Tieto stromy obsahujú:

* **funkčné uzly** – predstavujú operátory (napr. aritmetické operácie +, -, \*, /) alebo logické funkcie (AND, OR, NOT);
* **terminálne uzly** – predstavujú vstupné premenné alebo konštanty.

Jedinec v GP je teda stromová štruktúra, ktorá reprezentuje matematický výraz alebo program.

Jedinec v GP

V GP je jedinec reprezentovaný ako stromová štruktúra, čo umožňuje prirodzené vyjadrenie programov a matematických výrazov.

Tieto stromy môžu mať:

* **fixnú hĺbku** – kde maximálna hĺbka stromu je vopred určená, čím sa obmedzuje zložitosť programov;
* **premenlivú hĺbku** – kde strom môže rásť a vyvíjať sa bez striktne stanovených obmedzení, čo umožňuje vyššiu flexibilitu, ale môže viesť k problémom s preplnením kódu (angl. code bloat);

Jedinci v štruktúre sú ďalej reprezentovaní pomocou notácií:

* **prefixová notácia** – operátor je pred operandmi (napr. + x y);
* **postfixová notácia** – operátor je za operandmi (napr. x y +);
* **infixová notácia** – klasická matematická notácia (napr. x + y).

Operátory v GP

V genetickom programovaní (GP) zohrávajú evolučné operátory kľúčovú úlohu pri vytváraní nových jedincov a prehľadávaní hľadacieho priestoru možných riešení. Medzi hlavné operátory patrí mutácia, kríženie a selekcia, pričom každý z nich má špecifickú funkciu a význam v procese evolúcie populácie programov. Tieto operátory sa prispôsobujú štruktúre stromov, ktoré v GP reprezentujú programy, a zabezpečujú tak dostatočnú diverzitu a efektívnu optimalizáciu riešení [3][5].

Mutácia v genetickom programovaní je riešená pomocou náhodnej zmeny časti stromu, čím sa dosahuje variabilita v populácii a zabraňuje sa predčasnej konvergencii na lokálne optimá. Existuje niekoľko foriem mutácií, ktoré sa v GP používajú. Zmena terminálneho uzla spočíva v náhrade konštanty alebo premennej inou konštantou alebo premennou, čo môže zmeniť výstup programu bez zmeny jeho štruktúry. Zmena funkčného uzla sa realizuje napríklad zmenou operátora „+“ na „\*“, čím sa upravuje logika výpočtu v danej časti programu. Najvýznamnejšou formou mutácie je však náhrada podstromu, kde sa náhodne vybraný podstrom nahradí novým, náhodne vygenerovaným podstromom. Táto operácia umožňuje výrazné zmeny v štruktúre a správaní programu, čím sa výrazne rozširuje prehľadávanie hľadacieho priestoru [3][5][7].

Kríženie (crossover) je základným mechanizmom GP na kombinovanie genetického materiálu dvoch rodičovských stromov. Tento operátor umožňuje vytváranie nových potomkov, ktorí kombinujú vlastnosti oboch rodičov, čím sa podporuje explorácia hľadacieho priestoru. Proces kríženia prebieha nasledovne: najprv sa vyberú dva rodičovské stromy na základe ich fitness hodnoty. Následne sa v oboch stromoch náhodne určia krížové body, čo sú uzly, kde dôjde k výmene podstromov medzi rodičmi. Podstromy od týchto krížových bodov sa následne vymenia, čím vzniknú dvaja noví potomkovia. Tento proces umožňuje prenášanie genetického materiálu medzi jedincami, čo vedie k efektívnejšej kombinácii existujúcich funkcií a štruktúr v populácii. Výsledkom kríženia je vznik nových programov, ktoré môžu mať lepšiu výkonnosť ako ich rodičia [3][5].

Selekcia v genetickom programovaní určuje, ktorí jedinci budú použití na reprodukciu a ktorí prežijú do ďalšej generácie. Hlavným cieľom selekcie je zvýhodniť jedincov s vyššou fitness hodnotou, čím sa zabezpečuje evolučný tlak smerom k lepším riešeniam. Fitness funkcia v GP hodnotí kvalitu každého programu na základe jeho výstupu a schopnosti riešiť daný problém. Existuje viacero selekčných metód používaných v GP. Turnajová selekcia spočíva v náhodnom výbere niekoľkých jedincov, z ktorých najlepší postúpi do reprodukcie. Táto metóda zabezpečuje dostatočnú selekčnú tlak a zároveň zachováva variabilitu v populácii. Roulette wheel selection prideľuje každému jedincovi pravdepodobnosť výberu úmernú jeho fitness hodnote, čím zvýhodňuje lepších jedincov, ale zároveň ponecháva šancu aj menej výkonným riešeniam, čo podporuje diverzitu. Rank selection zoradí jedincov podľa ich fitness hodnoty a pravdepodobnosť výberu závisí od ich poradia. Tým sa eliminuje prílišná dominancia najlepších jedincov a zabraňuje sa predčasnej konvergencii [3][5][7].

Tieto operátory spolu tvoria základný mechanizmus genetického programovania, ktorý umožňuje efektívnu evolúciu programov a optimalizáciu riešení v rôznych aplikačných oblastiach, ako sú automatizované návrhy algoritmov, modelovanie systémov a symbolická regresia [3][5].

Problém preplnenia kódu

V genetickom programovaní (GP) je preplnenie kódu (angl. Code Bloat) problémom, pri ktorom stromy reprezentujúce programy rastú do nadmernej veľkosti bez zlepšenia fitness hodnoty. Tento jav vedie k zníženiu efektivity výpočtu, pretože väčšie stromy vyžadujú viac výpočtového výkonu a času na vyhodnotenie. Zároveň komplikuje ladenie a interpretáciu kódu, keďže nadmerná veľkosť stromov zahmlieva logiku riešenia a zhoršuje jeho čitateľnosť [3][5][7].

Na zvládnutie tohto problému sa používajú rôzne techniky regulácie veľkosti stromov. Jednou z nich sú tresty za dĺžku programu v rámci fitness funkcie, ktoré penalizujú príliš veľké stromy a tým motivujú algoritmus k vytváraniu kompaktnejších riešení. Ďalším prístupom sú pravidlá orezávania stromov, ktoré obmedzujú maximálnu hĺbku alebo veľkosť stromov, čím sa zabraňuje nekontrolovanému rastu [3][5][7].

Dôležitou stratégiou je aj parsimónia, ktorá uprednostňuje jednoduchšie riešenia pri rovnakej fitness hodnote, čím sa prirodzene podporuje tvorba efektívnych a prehľadných programov. Tieto techniky umožňujú udržiavať evolučný proces efektívny bez straty schopnosti riešiť komplexné úlohy [3][5][7].

Aplikácie GP

Genetické programovanie nachádza široké uplatnenie v rôznych oblastiach vedy a techniky. Symbolická regresia využíva GP na automatickú tvorbu matematických modelov na základe experimentálnych údajov, čo umožňuje objavovanie vzorcov a vzťahov bez nutnosti preddefinovanej štruktúry modelu [3][5].

V oblasti automatickej tvorby algoritmov sa GP používa na vývoj počítačových programov, ktoré riešia špecifické problémy, ako sú napríklad optimalizačné úlohy alebo spracovanie dát [7]. Okrem toho nachádza uplatnenie pri optimalizácii riadiacich systémov, kde sa využíva na návrh efektívnych regulátorov v inžinierskych aplikáciách [6].

Genetické programovanie sa tiež osvedčilo v strojovom učení na tvorbu predikčných modelov a klasifikátorov, pričom umožňuje automatickú evolúciu štruktúr neurónových sietí alebo rozhodovacích stromov [8]. V robotike a evolučnom návrhu sa využíva na optimalizáciu trajektórií a správania robotov, čím prispieva k efektívnejšiemu pohybu a adaptívnemu správaniu v dynamických prostrediach [11].

Pseudokód GP

Nasledujúci pseudokód implementovaný podľa zdrojov [3][8] predstavuje základnú implementáciu použitia GP.

Inicializácia:

Vytvor počiatočnú populáciu P náhodných stromových štruktúr (programov)

Nastav parametre: veľkosť populácie N, max\_generácie, pravdepodobnosť mutácie pm, pravdepodobnosť kríženia pc

kým nie je splnená ukončovacia podmienka vykonaj

Ohodnoť každého jedinca v populácii pomocou fitness funkcie

Nová populácia Q = {}

// Selekcia

kým veľkosť(Q) < N vykonaj

Vyber dvoch rodičov z populácie P na základe ich fitness (napr. turnajová selekcia)

// Kríženie (Crossover)

ak náhodné\_číslo() < pc potom

Vyber náhodné podstromy v oboch rodičoch

Vymeniť podstromy medzi rodičmi na získanie dvoch potomkov

inak

Potomkovia = rodičia (bez zmeny)

koniec ak

// Mutácia

pre každého potomka vykonaj

ak náhodné\_číslo() < pm potom

Vyber náhodný uzol v strome potomka

Nahraď ho náhodne vygenerovaným podstromom

koniec ak

koniec pre

Pridaj potomkov do populácie Q

koniec kým

Aktualizuj populáciu: P = Q

koniec kým

Gramatická evolúcia

Gramatická evolúcia (GE) je evolučný algoritmus, ktorý využíva formalizované gramatiky na generovanie počítačových programov alebo matematických výrazov. Na rozdiel od genetického programovania, ktoré operuje so stromovými štruktúrami, gramatická evolúcia pracuje s binárnymi reťazcami, ktoré sa následne prevádzajú na syntaktické stromy pomocou pravidiel formálnej gramatiky. Tento prístup umožňuje flexibilné generovanie programov v rôznych programovacích jazykoch a riešenie zložitých problémov [10][3].

Základným princípom GE je oddelenie genotypu (binárny reťazec) od fenotypu (program alebo matematický výraz). Transformácia genotypu na fenotyp prebieha pomocou pravidiel bezkontextovej gramatiky vo forme Backus-Naurovej formy (BNF) [10].

Genotyp je reprezentovaný binárnym alebo celočíselným reťazcom, ktorý obsahuje genetickú informáciu potrebnú na zostavenie programu. Fenotyp je výsledný program alebo matematický výraz, ktorý vznikne po aplikovaní pravidiel gramatiky na genotyp [3][10].

Transformácia genotypu na fenotyp prebieha nasledovne [10]:

1. Genotyp sa rozdelí na gény (napr. 8-bitové alebo 32-bitové sekvencie).
2. Každý gén sa prevedie na celočíselnú hodnotu.
3. Celočíselné hodnoty sa použijú na výber pravidiel gramatiky pomocou modulárneho operátora.
4. Pravidlá sa aplikujú postupne, až kým nevznikne kompletný program alebo výraz.

Tento prístup umožňuje dynamické a flexibilné generovanie rôznych štruktúr na základe definovaných gramatických pravidiel.

Jedinec v GE

Jedinec v GE je reprezentovaný binárnym alebo celočíselným reťazcom, ktorý obsahuje genetickú informáciu potrebnú na generovanie fenotypu. Táto reprezentácia má niekoľko výhod [10]:

* **Oddelenie genotypu a fenotypu** – umožňuje flexibilnú evolúciu a jednoduchú zmenu gramatických pravidiel bez zásahu do evolučného procesu.
* **Modulárny výber pravidiel** – výber pravidiel sa realizuje pomocou operácie modulo, čo zabezpečuje konzistentné mapovanie génov na pravidlá gramatiky.

Operátory v GE

V gramatickej evolúcii sa operátory aplikujú na genotypovej úrovni, pričom sa využíva najmä mutácia, kríženie a selekcia. Mutácia spočíva v náhodnej zmene častí genotypu. Môže ísť o bitovú mutáciu, kde sa mení hodnota náhodného bitu v binárnom reťazci, alebo o celočíselnú mutáciu, ktorá zahŕňa zmenu hodnoty génu v rámci definovaného rozsahu. Ďalším variantom je náhrada génu, čo vedie k zmene výberu pravidla v gramatike [10].

Kríženie v gramatickej evolúcii taktiež prebieha na genotypovej úrovni. Najčastejšie sa používa jednobodové kríženie, pri ktorom sa genotypy dvoch rodičov vymenia na náhodne zvolenom mieste. Viacbodové kríženie umožňuje výmenu viacerých segmentov, zatiaľ čo uniformné kríženie využíva náhodnú masku na výmenu jednotlivých génov [10].

Selekcia sa vykonáva na základe účelovej funkcie, ktorá hodnotí kvalitu fenotypu generovaného z genotypu. Najčastejšie sa využíva turnajová selekcia, kde sa porovnáva niekoľko náhodne vybraných jedincov, a najlepší z nich postupuje do ďalšej generácie. Ďalej sa používa ruletová selekcia, kde je pravdepodobnosť výberu úmerná fitness hodnote jedinca, a rank selection, ktorá zohľadňuje poradie jedincov podľa ich kvality [10].

Aplikácie GE

Gramatická evolúcia nachádza široké uplatnenie v rôznych oblastiach vedy a techniky. Jednou z hlavných aplikácií je symbolická regresia, kde sa využíva na automatické generovanie matematických modelov z experimentálnych údajov. Táto metóda umožňuje objavovať vzťahy medzi premennými bez predpokladu konkrétneho tvaru rovnice [10].

Ďalšou významnou oblasťou je automatická tvorba algoritmov, kde sa gramatická evolúcia používa na návrh programov a riešení pre špecifické problémy. Tento prístup umožňuje evolučný vývoj kódu, čo vedie k inovatívnym riešeniam [10].

V optimalizácii riadiacich systémov sa využíva na návrh regulačných štruktúr a kontrolérov, ktoré sú schopné adaptívne sa prispôsobovať zmenám v prostredí. Tieto aplikácie sú bežné najmä v inžinierstve a automatizácii [10].

V oblasti strojového učenia sa gramatická evolúcia využíva na evolučný návrh štruktúr neurónových sietí alebo na tvorbu klasifikátorov. Umožňuje optimalizovať nielen parametre, ale aj architektúru modelov, čo vedie k efektívnejšiemu riešeniu komplexných úloh [10].

Modelovanie biologických systémov je ďalšou oblasťou, kde sa GE využíva na simulácie genetických a biochemických procesov. Pomocou evolučných princípov je možné skúmať dynamiku zložitých biologických sietí a lepšie pochopiť ich fungovanie [10].

Pseudokód GE

Nasledujúci pseudokód predstavuje základnú implementáciu gramatickej evolúcie. Kód bol vytvorený pomocou zdrojov [3][8][10].

Inicializácia:

Načítaj gramatiku vo forme BNF (Backus-Naurova forma)

Vytvor počiatočnú populáciu P náhodných chromozómov (reťazcov čísel)

Nastav parametre: veľkosť populácie N, max\_generácie, pravdepodobnosť mutácie pm, pravdepodobnosť kríženia pc

kým nie je splnená ukončovacia podmienka vykonaj

Pre každý chromozóm v populácii vykonaj

Generuj syntaktický strom použitím gramatiky a chromozómu

Ak je strom platný potom

Ohodnoť ho pomocou účelovej funkcie

Inak

Priraď mu najhoršiu hodnotu kvality

koniec ak

koniec pre

Nová populácia Q = {}

// Selekcia

kým veľkosť(Q) < N vykonaj

Vyber dvoch rodičov z populácie P na základe ich kvality (napr. turnajová selekcia)

// Kríženie (Crossover)

ak náhodné\_číslo() < pc potom

Vyber náhodný bod kríženia v oboch rodičoch

Vymeniť časti chromozómov medzi rodičmi na získanie dvoch potomkov

inak

Potomkovia = rodičia (bez zmeny)

koniec ak

// Mutácia

pre každého potomka vykonaj

ak náhodné\_číslo() < pm potom

Vyber náhodný gén v chromozóme

Nahradiť ho náhodne vygenerovanou hodnotou v rámci povoleného rozsahu

koniec ak

koniec pre

Pridaj potomkov do populácie Q

koniec kým

Aktualizuj populáciu: P = Q

koniec kým

Analytické programovanie

Analytické programovanie (AP) predstavuje modernú evolučnú optimalizačnú techniku, ktorá bola zavedená ako odpoveď na obmedzenia tradičných prístupov ku generovaniu symbolických štruktúr, najmä genetického programovania [14]. Jeho hlavným cieľom je hľadanie matematických výrazov, ktoré opisujú daný problém optimálnym spôsobom [14].

Základná idea analytického programovania spočíva v oddelení vývoja symbolickej štruktúry od jej explicitného zakódovania do pevnej reprezentácie. Riešenie je kódované ako lineárna sekvencia čísel alebo operátorov, bez potreby pevnej stromovej alebo grafovej štruktúry, pričom samotná interpretácia a tvorba výrazov prebieha dynamicky pri ich vyhodnocovaní. Tento prístup prispieva k zlepšeniu problém nadmerného rastu riešení (už spomenutý Code bloat), ktorý je charakteristický pre klasické genetické programovanie [14].

Analytické programovanie pracuje s tromi základnými prvkami:

* **množina terminálov** (napr. premenné, konštanty);
* **množina operátorov** (matematické operácie, logické funkcie);
* **regulačné pravidlá** na syntézu a interpretáciu sekvencií do syntakticky správnych analytických výrazov;

Evolučný proces v AP pozostáva z nasledovných hlavných krokov:

1. **Inicializácia** – náhodné generovanie sekvencií reprezentujúcich potenciálne riešenia.
2. **Dekódovanie** – premena sekvencií na konkrétne matematické výrazy podľa definovaných pravidiel.
3. **Vyhodnotenie** – hodnotenie výrazu pomocou účelovej funkcie, ktorá zohľadňuje napríklad presnosť odhadu alebo jednoduchosť riešenia.
4. **Aplikácia evolučných operátorov** – výber, mutácia a kríženie sekvencií s cieľom generovať nové a lepšie riešenia.

Jedinečnou vlastnosťou analytického programovania je tzv. chaotické dekódovanie, ktoré využíva chaotické mapy alebo generátory (napr. logistická mapa) na riadenie procesu výberu operátorov počas interpretácie riešení. Tento spôsob výberu zvýšenú diverzitu v populácii a umožňuje lepšie preskúmanie priestoru možných riešení [14].

Jedinec v AP

V analytickom programovaní (AP) je jedinec reprezentovaný ako symbolická štruktúra, ktorá je výsledkom kombinácie funkčných a terminálnych symbolov. Tieto symboly sú ukladané do lineárneho poľa, ktoré následne slúži na zostavenie výsledného matematického alebo logického výrazu [14].

V porovnaní s inými evolučnými algoritmami, ako je napríklad genetické programovanie, sa AP vyznačuje tým, že nevyužíva explicitné stromy (ako GP), ale naopak využíva postupnú interpretáciu prvkov z poľa podľa predom definovaných pravidiel syntaktickej analýzy. Každý prvok poľa môže predstavovať:

* **funkciu** (napr. sčítanie, násobenie, trigonometrické operácie);
* **premennú** (vstupnú veličinu problému);
* **konštantu** (pevne danú hodnotu).

Jedinec teda nie je pevne definovaná stromová štruktúra, ale dynamický zápis, ktorý je dekódovaný počas vyhodnocovania. Tento spôsob reprezentácie umožňuje vyššiu flexibilitu a eliminuje niektoré problémy typické pre iné techniky, napríklad problém s preplnením kódu (code bloat) známy z genetického programovania [14].

Pri dekódovaní sa prechádza lineárne pole sprava doľava (alebo podľa definovaného poradia), pričom sa postupne skladá syntakticky správny výraz. Výhodou je, že aj pri náhodne generovanom poli je možné zabezpečiť, že výsledný jedinec bude vždy syntakticky správny.

Aplikácie AP

Analytické programovanie je univerzálny nástroj schopný riešiť široké spektrum problémov, najmä v oblastiach, kde je cieľom nájsť symbolické vzťahy alebo funkčné závislosti. V literatúre [14] sú popísané najmä tieto oblasti aplikácie:

* **symbolická regresia** – hľadanie explicitného matematického vzťahu medzi vstupnými a výstupnými dátami. AP v tomto prípade automaticky syntetizuje rovnice, ktoré najlepšie aproximujú skúmané dáta;
* **riešenie diferenciálnych rovníc** – AP dokáže generovať uzatvorené formy riešení diferenciálnych rovníc bez potreby numerických aproximácií.
* **modelovanie komplexných systémov** – využitie AP v oblasti riadenia, predikcie alebo diagnostiky, kde je potrebné nájsť optimálne symbolické modely systémov (napríklad v strojárstve alebo bioinformatike)
* **evolučná tvorba algoritmov** – pomocou AP je možné automatizovať návrh algoritmov alebo rozhodovacích pravidiel v prípade, že existujú vhodné funkčné a terminálne množiny;
* **oblasti s neznámou štruktúrou riešenia** – AP je efektívne v situáciách, kde nie je známa presná podoba optimálneho riešenia a kde je potrebné skúmať veľké a zložité priestory možných vzorcov.

Pseudokód AP

Nasledujúci pseudokód bol vytvorený s použitím zdrojov [2][4][12][14] a ilustruje základnú implementáciu analytického programovania.

Inicializácia:

Definuj množinu funkcií F a terminálov T

Vytvor počiatočnú populáciu P náhodných riešení

Nastav parametre: veľkosť populácie N, max\_generácie, pravdepodobnosť mutácie pm, pravdepodobnosť kríženia pc

kým nie je splnená ukončovacia podmienka vykonaj

Pre každý jedinec v populácii vykonaj

Dekóduj jedinca na syntaktický strom pomocou gramatických pravidiel

Ak je syntaktický strom platný potom

Ohodnoť ho pomocou účelovej funkcie

Inak

Priraď mu najhoršiu hodnotu kvality

koniec ak

koniec pre

Nová populácia Q = {}

// Selekcia

kým veľkosť(Q) < N vykonaj

Vyber dvoch rodičov z populácie P na základe ich fitness (napr. turnajová selekcia)

// Kríženie (Crossover)

ak náhodné\_číslo() < pc potom

Vyber náhodný bod kríženia v oboch rodičoch

Vymeniť podstromy medzi rodičmi na získanie dvoch potomkov

inak

Potomkovia = rodičia (bez zmeny)

koniec ak

// Mutácia

pre každého potomka vykonaj

ak náhodné\_číslo() < pm potom

Vyber náhodný uzol v syntaktickom strome

Nahradiť ho náhodne vygenerovaným podstromom na základe gramatických pravidiel

koniec ak

koniec pre

Pridaj potomkov do populácie Q

koniec kým

Aktualizuj populáciu: P = Q

koniec kým

Porovnanie evolučných algoritmov

V nasledujúcom texte sú porovnávané genetické algoritmy (GA), diferenciálna evolúcia (DE), evolučné stratégie (ES), genetické programovanie (GP), analytické programovanie (AP) a gramatická evolúcia (GE).

Genetické algoritmy (GA)

Genetické algoritmy sú inšpirované princípmi prirodzeného výberu a genetických operácií, ako je kríženie a mutácia. Sú univerzálne a efektívne pri riešení optimalizačných úloh na diskrétnych aj spojitých priestoroch. Ich nevýhodou môže byť predčasná konvergencia do lokálnych optím. GA sú flexibilné a ľahko implementovateľné, avšak výber vhodných parametrov (napr. veľkosť populácie, mutačná a krížová pravdepodobnosť) výrazne ovplyvňuje ich výkonnosť. [7][8].

Diferenciálna evolúcia (DE)

Diferenciálna evolúcia využíva rozdiely medzi vektorovými riešeniami na generovanie nových kandidátov. Vyniká jednoduchosťou implementácie a schopnosťou efektívne optimalizovať spojité funkcie. DE má dobrú globálnu prieskumnú schopnosť a je robustná voči predčasnej konvergencii vďaka mutačným stratégiám. Na druhej strane môže vyžadovať viac výpočtového času pri vysokorozmerných úlohách. [4][5].

Evolučné stratégie (ES)

Evolučné stratégie sa zameriavajú na optimalizáciu spojitých parametrov prostredníctvom adaptívnej mutácie a selekcie. Hlavnou výhodou je schopnosť dynamicky prispôsobovať krok mutácie, čo zlepšuje konvergenciu. ES sú však menej účinné pri diskrétnych problémoch. Typické varianty zahŕňajú (μ, λ) a (μ+λ) selekciu. [2][6].

Genetické programovanie (GP)

Genetické programovanie rozširuje genetické algoritmy na evolúciu programových štruktúr, čo umožňuje automatickú tvorbu algoritmov a modelov. Hlavným prínosom je schopnosť objavovať nelineárne vzťahy a riešiť problémy, kde nie je známa explicitná funkcia cieľa. GP však trpí vysokou výpočtovou náročnosťou a zložitosťou implementácie. [3][5].

Analytické programovanie (AP)

Analytické programovanie je podobné genetickému programovaniu, ale využíva symbolické štruktúry na generovanie kandidátnych riešení. Výhodou je schopnosť presne definovať štruktúru a formu riešení, čo znižuje priestor vyhľadávania. Na druhej strane vyžaduje podrobnú znalosť problému a vhodne definovanú štruktúru. [11][12][14].

Gramatická evolúcia (GE)

Gramatická evolúcia využíva gramatiku na generovanie programových štruktúr, čo umožňuje flexibilnú reprezentáciu riešení v ľubovoľnom programovacom jazyku. Výhodou je oddelenie vyhľadávacieho priestoru od syntaktických pravidiel, čo zvyšuje univerzálnosť. GE však môže trpieť redundanciou kódu a vyššou výpočtovou náročnosťou. [10].

Tabuľka porovnania

Nasledujúca tabuľka obsahuje porovnanie najdôležitejších vlastností spomenutých evolučných algoritmov. Tabuľka bola vytvorená podobne ako porovnanie vyššie použitím zdrojov [1][2][3][4][5][6][10][11][12].

Tabuľka 1 Porovnanie základných evolučných algoritmov

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Parameter** | **GA** | **DE** | **ES** | **GP** | **AP** | **GE** |
| **Priestor** | Diskrétny aj spojitý | Spojitý | Spojitý | Programové štruktúry | Matematické výrazy | Jazyk pomocou gramatiky |
| **Operátory** | Mutácia, kríženie, selekcia | Mutácia (dif.) kríženie, selekcia | Mutácia, kríženie, selekcia | Mutácia, kríženie, selekcia (stromové) | Formálne pravidlá | Mutácia, kríženie, selekcia |
| **Adaptivita** | Nie | Áno | Áno | Nie | Nie | Nie |
| **Náročnosť výpočtová** | Stredná | Stredná | Nízka až stredná | Vysoká | Stredná | Vysoká |
| **Jedinec** | Bin/real | Reálny | Reálny | Stromový | Symbolický | Gramatika (binárna) |

Evolučné výpočetné techniky s variabilnou dĺžkou jedinca

Vývoj evolučných algoritmov prešiel v posledných dekádach významným pokrokom, pričom sa postupne presúval od jednoduchých, pevne štruktúrovaných modelov k flexibilnejším systémom umožňujúcim dynamickú modifikáciu štruktúry riešení počas evolučného procesu. Jednou z dôležitých vlastností evolučných výpočtových techník je schopnosť pracovať s variabilnou dĺžkou jedinca, čím sa otvárajú nové možnosti pre riešenie komplexných problémov, ako sú automatická syntéza programov, optimalizácia štruktúr alebo návrh symbolických modelov [1][5][16].

V klasických evolučných algoritmoch, ako je genetický algoritmus, sú jedinci často reprezentovaní ako fixnej veľkosti binárne alebo reálne kódované vektory [5][8]. Tento prístup je efektívny pre úlohy optimalizácie nad pevne definovanými priestorovými parametrami, avšak limituje schopnosť algoritmu adaptovať sa na úlohy, kde riešenia majú prirodzene rôznu zložitosť alebo veľkosť. Evolučné techniky s variabilnou dĺžkou jedinca, ako sú genetické programovanie (GP) [3], gramatická evolúcia (GE) [10], analytické programovanie (AP) [14] alebo kartézske genetické programovanie (CGP) [18], tento nedostatok prekonávajú umožnením tvorby riešení s dynamickou štruktúrou a veľkosťou počas behu algoritmu.

Jedincov s variabilnou dĺžkou najčastejšie reprezentujú stromové štruktúry, lineárne sekvencie alebo grafové modely [16][18]. Tento prístup však prináša aj nové výzvy, medzi ktoré patrí predovšetkým už spomínaný problém nekontrolovaného rastu štruktúr (tzv. bloat) [15][17]. Nekontrolovaný nárast veľkosti riešení vedie k vyššej výpočtovej náročnosti, zhoršenej generalizácii výsledkov a strate efektivity evolučného procesu. Riešenie týchto problémov si vyžaduje nasadenie špecifických techník, ako sú napríklad dynamické limity veľkosti, parsimónia, penalizácia veľkých štruktúr alebo modifikované operátory na kríženie a mutáciu [17].

V tejto kapitole budú popísané základné koncepty súvisiace s prácou s jedincami s variabilnou dĺžkou. Pozornosť bude venovaná rôznym typom reprezentácií, špecifickým evolučným technikám, operátorom ovplyvňujúcim štruktúru jedincov a metódam kontroly rastu riešení.

Úvod do techník s variabilnou dĺžkou jedinca

Ako už bolo spomenuté vyššie evolučné algoritmy tradične reprezentujú riešenia ako štruktúry s pevne stanovenou veľkosťou, typicky vektormi fixnej dĺžky, ktoré podliehajú mutácii a kríženiu [5][8]. Tento prístup je však limitujúci pri riešení problémov, kde optimálne riešenie nie je známe vopred a kde jeho veľkosť alebo štruktúra môže dynamicky variovať v závislosti od vývoja populácie. Techniky s variabilnou dĺžkou jedinca preto predstavujú prirodzený krok vpred v rámci evolučných výpočtových prístupov [1][5].

Základnou myšlienkou týchto techník je umožniť jedincovi, teda potenciálnemu riešeniu problému, aby počas evolučného procesu menil svoju veľkosť, zložitosť či topológiu. Tento mechanizmus sa ukázal ako kľúčový najmä pri riešení úloh, kde samotné vyjadrenie riešenia má neznámu alebo dynamickú štruktúru, ako sú napríklad symbolická regresia, návrh algoritmov, generovanie pravidiel alebo modelovanie systémov [3][10][14].

Medzi najvýznamnejšie techniky využívajúce variabilnú dĺžku jedinca patria:

* Genetické programovanie (GP);
* Gramatická evolúcia (GE);
* Analytické programovanie (AP);
* Kartézske genetické programovanie (CGP).

Výhodou práce s variabilnou dĺžkou je vyššia flexibilita reprezentácie a schopnosť objavovať nielen optimálne parametre, ale aj samotné štruktúry riešení [5][16]. Na druhej strane však tento spôsob reprezentácie prináša špecifické problémy, ako je code bloat (nekontrolovaný rast veľkosti jedincov), ktorý môže viesť k zbytočne zložitým riešeniam a zvýšeniu výpočtovej náročnosti [15][17].

Pri navrhovaní a implementácii evolučných techník s variabilnou dĺžkou je preto potrebné venovať pozornosť nielen reprezentácii jedincov, ale aj špecifickým evolučným operátorom, stratégiám kontroly rastu riešení a vhodnému hodnoteniu fitness funkcie, ktoré spoločne ovplyvňujú efektivitu celého evolučného procesu [5][8].

Reprezentácie jedinca s variabilnou dĺžkou

Kľúčovým aspektom evolučných výpočtových techník s variabilnou dĺžkou je samotný spôsob, akým sú jedinci reprezentovaní. Rôzne prístupy k reprezentácii umožňujú riešiť rôzne typy problémov a zásadne ovplyvňujú efektivitu evolučného procesu [1][5]. V nasledujúcich podkapitolách sú rozpracované najvýznamnejšie modely reprezentácie.

Stromová reprezentácia (Genetické programovanie)

Stromová reprezentácia je typická najmä pre genetické programovanie (Genetic Programming – GP), kde každý jedinec predstavuje stromovú štruktúru [3][16]. Uzly stromu reprezentujú funkcie (neterminálne symboly) a listy predstavujú terminálne symboly (konštanty alebo premenné).

Výhodou tohto prístupu je vysoká flexibilita, pretože vetvenie, hĺbka a tvar stromu môžu dynamicky rásť alebo sa zmenšovať počas evolúcie. Jednoduchá manipulácia s „podstromami“ umožňuje efektívne implementovať kríženie a mutáciu [3][16].

Typickým problémom je však nárast zbytočne veľkých stromov bez významného zlepšenia kvality riešenia (tzv. code bloat), čo si vyžaduje zavedenie techník na kontrolu veľkosti jedincov [15][17].

Lineárna reprezentácia s mapovaním (Gramatická evolúcia)

V prípade gramatickej evolúcie (Grammatical Evolution – GE) je jedinec reprezentovaný ako binárny reťazec alebo vektor celých čísel, ktorý je následne dekódovaný pomocou formálnej gramatiky do syntakticky správnej štruktúry, napríklad programu alebo výrazu [10].

Tento prístup prináša výhodu oddelenia evolučného procesu od syntaktickej štruktúry výsledku, čo umožňuje flexibilné definovanie pravidiel a jednoduché rozšírenie na nové domény. Navyše, vďaka oddeleniu genotypu od fenotypu je možné použiť klasické operátory známe z genetických algoritmov [10].

Rizikom tejto reprezentácie je, že malé zmeny v genotypovom kóde môžu spôsobiť výrazné zmeny vo fenotypovej štruktúre, čo môže negatívne ovplyvniť evolúciu.

3.2.3 Grafická reprezentácia (Kartézske genetické programovanie)

Kartézske genetické programovanie (Cartesian Genetic Programming – CGP) používa orientované acyklické grafy (Directed Acyclic Graphs – DAG) ako reprezentáciu jedincov [18]. Každý uzol grafu vykonáva určitú funkciu a je napojený na vstupy z iných uzlov alebo zo samotného vstupu problému.

Táto reprezentácia je veľmi vhodná na návrh digitálnych obvodov, neurónových sietí alebo iných štruktúr, kde je výhodné využívať znovupoužitie častí riešenia (modularita). CGP umožňuje meniť počet uzlov aj ich prepojenia, pričom zachováva orientovanú acyklickosť grafu [18].

Nevýhodou môže byť vyššia zložitosť implementácie evolučných operátorov, predovšetkým zabezpečenie správnosti a efektívnosti grafových mutácií.

Symbolická reprezentácia (Analytické programovanie)

Analytické programovanie (Analytical Programming – AP) predstavuje flexibilný prístup, kde sú jedinci vytváraní kombináciou funkčných a terminálnych symbolov zo špecifikovanej množiny, pričom výsledné riešenie môže byť reprezentované ako symbolický výraz bez explicitného stromového usporiadania [14].

Táto reprezentácia umožňuje tvorbu výrazov rôznej zožitosti s adaptívnym využívaním matematických funkcií, operátorov a konštánt. Významným prínosom je možnosť riadiť stavbu riešení pomocou špecifických pravidiel, ktoré podporujú efektívne vyhľadávanie v priestore možných riešení [14].

Obdobne ako pri stromovej reprezentácii, aj tu sa vyskytuje problém code bloatu, ktorý je potrebné kontrolovať pomocou vhodných evolučných stratégií a penalizácií zložitosti riešenia [14].

Evolučné algoritmy podporujúce variabilnú dĺžku

V oblasti evolučných výpočtových techník existuje viacero algoritmov, ktoré podporujú alebo priamo vyžadujú variabilnú dĺžku jedinca. Takéto algoritmy boli navrhnuté predovšetkým pre problémy, kde je dôležitá schopnosť dynamickej tvorby štruktúr, optimalizácie programov, modelov alebo iných komplexných reprezentácií, ktorých vhodná veľkosť nie je vopred známa [3][5][8].

Nasledujúce evolučné algoritmy patria medzi najvýznamnejšie predstaviteľov techník s variabilnou dĺžkou jedinca.

Genetické programovanie (GP)

Genetické programovanie (GP) patrí medzi najstaršie a najznámejšie techniky pracujúce s variabilnou dĺžkou jedincov. V GP sú jedinci reprezentovaní ako stromy, kde uzly reprezentujú funkcie a terminály [3][15].

Variabilita dĺžky je prirodzeným dôsledkom stromovej štruktúry a aplikačných operátorov ako je kríženie a mutácia, ktoré môžu meniť veľkosť a tvar stromov. Táto vlastnosť umožňuje GP efektívne hľadať riešenia, ktorých optimálna zložitosť nie je vopred známa.

GP sa uplatňuje napríklad pri automatickej tvorbe algoritmov, v symbolickej regresii, v evolučnom návrhu systémov alebo v oblasti riadiacich štruktúr [3][16].

Gramatická evolúcia (GE)

Gramatická evolúcia (GE) rozširuje myšlienku genetického programovania o použitie formálnych gramatík pre riadenie tvorby štruktúr. Genotyp jedinca v GE je tvorený zoznamom čísel (často celé čísla), ktoré sa pomocou gramatických pravidiel transformujú na fenotyp — syntakticky korektnú štruktúru [10].

Variabilná dĺžka genotypu a možnosť tvorby fenotypov rôznych veľkostí umožňujú GE riešiť veľmi rozmanité problémy, vrátane syntézy programov, tvorby modelov alebo návrhu heuristík [10].

Významnou výhodou GE je oddelenie genotypu a fenotypu, čo umožňuje flexibilnú a robustnú manipuláciu s riešeniami.

Kartézske genetické programovanie (CGP)

Kartézske genetické programovanie (CGP) reprezentuje jedincov ako orientované acyklické grafy, kde uzly môžu byť pripojené ľubovoľným spôsobom k predchádzajúcim uzlom alebo vstupom [18].

Aj keď pôvodné CGP používalo pevne daný počet uzlov, moderné varianty CGP (ako napr. dynamické CGP alebo adaptívne CGP) podporujú variabilnú dĺžku prostredníctvom aktívnych a neaktívnych uzlov. Evolučné operácie môžu meniť počet aktívnych častí grafu, čím efektívne upravujú zložitosť jedinca počas behu algoritmu [18].

CGP je využívané predovšetkým v oblasti návrhu digitálnych obvodov, symbolickej regresii a evolučného návrhu softvérových modulov.

Analytické programovanie (AP)

Analytické programovanie (AP) je ďalšia technika založená na variabilnej dĺžke riešení, ktorá sa odlišuje použitím všeobecného analytického zápisu riešení pomocou funkčných a terminálnych symbolov [14].

Jedinci v AP predstavujú postupnosti volaní funkcií a operátorov, pričom ich dĺžka a štruktúra sa môžu voľne meniť počas evolúcie[14].

AP je vhodné predovšetkým pre úlohy symbolickej regresie, optimalizácie výrazov a evolučnej tvorby analytických modelov.

Ďalšie algoritmy

Okrem vyššie uvedených techník existujú aj ďalšie metódy podporujúce variabilnú dĺžku jedinca:

* Lineárne genetické programovanie (LGP), kde sa programy skladajú zo sekvencií inštrukcií a dĺžka programov môže dynamicky rásť alebo klesať [16];
* Evolučné algoritmy založené na grafovej evolúcii, kde štruktúra grafov (počet vrcholov a hrán) podlieha evolučným zmenám [18];
* Hybridné systémy, kombinujúce viaceré reprezentácie a operátory, aby dosiahli vyššiu adaptabilitu na dynamické prostredie problému [5][13].

Evolučné operátory pre variabilnú dĺžku jedinca

Evolučné operátory, ako mutácia a kríženie, zohrávajú zásadnú úlohu v evolučných výpočtových technikách. V prípade techník s variabilnou dĺžkou jedinca sa však vyžaduje ich špecifické prispôsobenie, keďže operácie musia zvládať meniace sa štruktúry a veľkosti jedincov [3][5][8].

Mutácia

Mutácia v technikách s variabilnou dĺžkou neznamená len jednoduchú zmenu hodnôt, ale často je nutné zasahovať aj do samotnej štruktúry. Nasledujú typické druhy mutácií.

* **vloženie prvku (insertion)** – do existujúcej štruktúry (napríklad stromu alebo reťazca) sa vloží nový funkčný alebo terminálny symbol, čím sa predĺži dĺžka jedinca [3][16];
* **odstránenie prvku (deletion)** – vybraný prvok alebo celá podštruktúra sa odstráni, čím sa dĺžka jedinca skráti. Táto operácia pomáha kontrolovať nárast zložitosti riešenia [17];
* **nahradenie (replacement)** – nahradenie existujúcej časti riešenia inou, ktorá môže mať odlišnú veľkosť, čo umožňuje jemné alebo zásadné zmeny vo fenotype [5].

Pri stromovej reprezentácii sa mutácia často realizuje náhodnou výmenou „podstromov“ alebo nahradením uzlu iným funkčným alebo terminálnym symbolom [3].

Mutácia v gramatickej evolúcii môže znamenať modifikáciu jednotlivých génov v genotypovom reťazci, čo po dekódovaní vedie k zmene štruktúry fenotypu [10].

Mutácie v kartézskom genetickom programovaní spôsobujú zmeny funkcií v uzloch, ich pripojení alebo hodnotách parametrov [18].

Kríženie

Kríženie (crossover) v evolučných algoritmoch s variabilnou dĺžkou je náročnejšie v porovnaní so štandardným krížením fixných reťazcov, pretože musia byť rešpektované štrukturálne vlastnosti jedincov.

Medzi hlavné stratégie patria nasledujúce.

* **kríženie podštruktúr (subtree crossover)** – typické najmä pre genetické programovanie. Náhodne sa vyberú podstromy v oboch rodičoch a vzájomne sa vymenia [3][16];
* **jednobodové a viacbodové kríženie** – pri lineárnych reprezentáciách (napr. v gramatickej evolúcii) sa používajú klasické jednobodové alebo viacbodové kríženia, pričom body delenia môžu byť flexibilne volené tak, aby umožňovali variabilné výsledné dĺžky potomkov [10];
* **kríženie grafov** – v kartézskom genetickom programovaní sa môžu vymieňať jednotlivé časti grafu, pričom musí byť zachovaná acyklická štruktúra [18].

Hlavnou výzvou pri krížení je zabezpečiť, aby vzniknutí potomkovia boli syntakticky a sémanticky korektní a aby bol zachovaný ich funkčný potenciál [5].

Problémy evolučných techník s variabilnou dĺžkou jedinca

Evolučné výpočtové techniky využívajúce variabilnú dĺžku jedinca prinášajú oproti technikám s fixnou dĺžkou viaceré výhody, predovšetkým vyššiu flexibilitu a schopnosť modelovať komplexné štruktúry. Zároveň však so sebou nesú aj množstvo špecifických problémov a výziev, ktoré je potrebné adekvátne riešiť, aby bola evolúcia efektívna a stabilná [3][5][8][15].

Problém preplnenia kódu (Code bloat)

Jedným z najvážnejších problémov je prerastanie štruktúr (angl. code bloat), ktoré označuje nárast veľkosti riešení bez primeraného zlepšenia ich kvality (hodnoty účelovej funkcie). Tento jav spôsobuje:

* zvýšenie výpočtovej náročnosti hodnotenia riešení;
* väčšiu pravdepodobnosť preťaženia pamäte;
* ťažšiu interpretáciu výsledných riešení, čo je kritické v oblasti symbolickej regresie alebo automatizovanej tvorby programov [3][15].

Existujú ale metódy na potláčanie preplnenia kódu. Nižšie sú spomenuté tie najčastejšie [17].

* zavedenie penalizácie veľkosti do účelovej funkcie;
* dynamické obmedzovanie veľkosti stromov alebo iných štruktúr počas evolúcie;
* použitie takzvanej „parsimony pressure“, kde sa uprednostňujú menšie štruktúry pri rovnakom výkone;
* explicitné orezávanie (angl. pruning) nadbytočných častí štruktúr.

Balansovanie medzi exploráciou a exploitáciou

Pri variabilnej dĺžke jedinca sa zvyšuje priestor možných riešení, čo komplikuje balansovanie medzi exploráciou (hľadanie nových oblastí riešení) a exploitáciou (jemné zlepšovanie už známych dobrých riešení) [5].

Dynamické riadenie operátorov (napr. adaptívne pravdepodobnosti mutácie a kríženia) je preto dôležité na udržanie zdravého evolučného procesu. Príliš veľké preferovanie rozmerných štruktúr môže viesť k predčasnej konvergencii alebo k stagnácii [5][8].

Optimalizácia veľkosti riešení

V technikách s variabilnou dĺžkou nie je cieľom len maximalizácia fitness (hodnoty účelovej funkcie), ale často aj optimalizácia veľkosti riešenia. Malé a kompaktné štruktúry sú vo všeobecnosti:

* rýchlejšie na vyhodnocovani;
* lepšie interpretovateľné;
* menej náchylné na pretrénovanie (angl. overfitting) [15][17].

Existujú aj viackritériové prístupy, kde sa hodnota účelovej funkcie kombinuje s kritériami veľkosti (napr. pomocou Pareto-optimálnych stratégií) [5].

Robustnosť genotyp-fenotyp mapovania

Pri technikách ako gramatická evolúcia alebo analytické programovanie je kľúčové, aby mapovanie medzi genotypom a fenotypom bolo:

* robustné (malé zmeny v genotypovom priestore nespôsobujú dramatické zmeny vo fenotypoch);
* hladké (umožňujúce postupné vylepšovanie riešení).

Zlá kvalita mapovania môže viesť ku „krehkosti“ evolúcie a znižovať pravdepodobnosť nájdenia optimálneho riešenia [10][14].

Evolúcia štruktúr s rôznymi typmi uzlov

Pri stromových alebo grafových reprezentáciách (napr. v genetickom programovaní alebo kartézskom genetickom programovaní) môže byť problémom správne riadenie typov uzlov:

* funkčné uzly musia mať vhodný počet vstupov a výstupov;
* kríženie a mutácie musia zachovávať syntaktickú správnosť [3][18].

Na riešenie týchto problémov sa používajú typované systémy alebo kontextovo senzitívne operátory, ktoré zohľadňujú typovú informáciu uzlov [15].

Techniky kontroly dĺžky riešení

Pri evolučných algoritmoch s variabilnou dĺžkou riešení je častým problémom nekontrolovaný rast veľkosti jedincov bez adekvátneho zlepšenia ich kvality. Tento už spomínaný jav, známy ako code bloat, môže výrazne znižovať efektivitu evolučného procesu, zvyšovať výpočtovú náročnosť a znižovať schopnosť algoritmu generalizovať [3][15][17].

Na obmedzenie tohto javu sa využívajú špeciálne techniky kontroly dĺžky riešení.

Parsimónia

Parsimónia (z lat. *parsimonia* – šetrnosť) je princíp, podľa ktorého sú preferované riešenia s menšou zložitosťou, ak majú podobnú alebo rovnakú kvalitu ako zložitejšie riešenia. V evolučných algoritmoch sa tento princíp implementuje rôznymi spôsobmi [15][17].

* Penalizácia: k hodnoteniu jedinca (fitnes) sa pridáva penalizačný člen úmerný jeho veľkosti. Typicky má formu:

kde λ je váhový koeficient vyjadrujúci silu penalizácie.

* Preferenčný výber: pri výbere medzi dvoma jedincami s rovnakým alebo veľmi podobným hodnotením je uprednostnený kratší jedinec.

Dynamické limity veľkosti (bloat control)

Ďalším prístupom je dynamické obmedzovanie maximálnej prípustnej dĺžky jedincov počas evolúcie [17]:

* počas behu algoritmu sa adaptívne nastavuje maximálna prípustná veľkosť jedinca;
* ak riešenia dosahujú požadovanú kvalitu, limit sa môže sprísňovať, čím sa podporuje hľadanie kompaktnejších riešení;
* tento mechanizmus pomáha udržať rozumnú rovnováhu medzi kvalitou riešení a ich zložitosťou.

Prerušenie rastu stromov

V genetickom programovaní a príbuzných technikách sa používa aj metóda prerušenia rastu stromov pri generovaní alebo mutácii [3][15]:

* stromy sa generujú do vopred stanovenej maximálnej hĺbky alebo veľkosti;
* ak sa dosiahne tento limit, ďalší rast je zakázaný a proces sa ukončí.

Aplikácie evolučných techník s variabilnou dĺžkou

Evolučné techniky s variabilnou dĺžkou jedincov nachádzajú široké uplatnenie v mnohých oblastiach, kde je potrebné automaticky hľadať štruktúry, modely alebo riešenia variabilnej veľkosti. Ich schopnosť adaptívne meniť zložitosť riešení počas evolúcie je mimoriadne užitočná v úlohách, kde nie je vopred známa optimálna štruktúra riešenia alebo kde sa počas optimalizácie mení prostredie a požiadavky [1][3][5][13].

Symbolická regresia a modelovanie

Jednou z hlavných aplikácií je symbolická regresia, kde cieľom je nájsť matematický výraz, ktorý čo najlepšie popisuje závislosť medzi vstupnými a výstupnými dátami [3][15]. Genetické programovanie (GP) a analytické programovanie (AP) sa v tejto oblasti často využívajú práve vďaka schopnosti generovať výrazy variabilnej dĺžky a zložitosti.

Praktickými aplikáciami sú napríklad:

* automatické vytváranie predikčných modelov v inžinierskych systémoch;
* modelovanie biologických procesov;
* ekonomické a finančné predikcie [1][13].

Evolučná syntéza programov a algoritmov

Ďalšou významnou oblasťou je automatická syntéza programov, kde sa evolučnými technikami generujú celé programy alebo algoritmy riešiace daný problém [3][16]. Tu je flexibilita dĺžky kľúčová – výsledné programy môžu byť krátke alebo komplexné podľa potreby problému.

Príklady aplikácií:

* evolučné návrhy riadiacich systémov;
* generovanie heuristík pre zložité optimalizačné problémy;
* automatické ladenie softvérových komponentov.

Evolučný návrh štruktúr a topológií

Evolučné algoritmy s variabilnou dĺžkou sa úspešne využívajú aj pri návrhu topológií neurónových sietí, elektrických obvodov či logických štruktúr [11][18]. V týchto prípadoch nie je daná pevná štruktúra, ale je žiaduce, aby algoritmus objavil optimálny počet prvkov a ich prepojenie.

Praktické príklady:

* návrh architektúr hlbokých neurónových sietí;
* syntéza digitálnych a analógových obvodov;
* evolúcia robotických riadiacich systémov [11][18].

Evolučné algoritmy v spracovaní obrazu a počítačovom videní

V oblasti spracovania obrazu a počítačového videnia sú techniky s variabilnou dĺžkou využívané na evolúciu filtrov, deskriptorov a klasifikátorov [13]. Štruktúra riešenia môže dynamicky rásť a prispôsobovať sa podľa potreby úlohy, napríklad pri detekcii objektov, segmentácii obrazu či analýze scény.

Príklady:

* evolúcia programov na spracovanie obrazových dát;
* návrh adaptívnych klasifikátorov pre rozpoznávanie obrazov;
* automatické generovanie pravidiel pre počítačové videnie.

Testovacie funkcie pre evolučné algoritmy

Testovacie funkcie zohrávajú podstatnú úlohu pri vývoji, ladení a porovnávaní evolučných algoritmov. Vďaka nim je možné objektívne hodnotiť schopnosti algoritmov riešiť optimalizačné problémy rôznych charakteristík, ako je počet extrémov, separabilita, hladkosť alebo dimenzionalita vyhľadávacieho priestoru.

Táto kapitola sa zameriava na predstavenie základnej klasifikácie testovacích funkcií a opis najčastejšie využívaných funkcií v oblasti evolučnej optimalizácie.

Následne budú popísané špecifické skupiny testovacích funkcií umožňujúce variabilitu zložitosti problému, ako aj benchmarkové súbory úloh, ktoré sa často používajú na medzinárodných súťažiach.

V závere kapitoly bude uvedené, aké faktory by mali byť zohľadnené pri výbere vhodnej testovacej funkcie v závislosti od typu a charakteru optimalizačného algoritmu.

Klasifikácia testovacích funkcií

Vzhľadom na široké spektrum problémov, ktoré môžu evolučné algoritmy riešiť, existuje množstvo rôznych typov testovacích funkcií. Ich klasifikácia sa obvykle zakladá na charakteristikách, ktoré ovplyvňujú náročnosť optimalizácie a spôsob hľadania globálneho extrému.

Testovacie funkcie používané pri vývoji a hodnotení evolučných algoritmov sa líšia podľa viacerých charakteristík, ktoré priamo ovplyvňujú zložitosť optimalizácie. Rozdelenie týchto funkcií slúži na efektívnejšie posúdenie výkonnosti algoritmov v rôznych typoch prostredí [5][6].

Unimodálne funkcie

Predstavujú kategóriu problémov, kde existuje iba jeden globálny extrém (maximum alebo minimum). Vzhľadom na absenciu lokálnych extrémov je ich optimalizácia relatívne jednoduchšia. Tento typ funkcií je vhodný na testovanie schopnosti algoritmu rýchlo a presne konvergovať do optimálneho bodu. Typickým príkladom unimodálnej funkcie je napríklad funkcia Sphere [5][23].

Multimodálne funkcie

Obsahujú viacero lokálnych extrémov, čo značne komplikuje proces hľadania globálneho extrému. Evolučné algoritmy musia pri týchto problémoch preukázať schopnosť vyhýbať sa predčasnej konvergencii do lokálne optimálnych riešení. Funkcie ako Rastrigin či Ackley sú známe svojou komplexnou multimodálnou štruktúrou [5][6][23].

Škálované funkcie

Sú charakteristické tým, že jednotlivé dimenzie problému majú rôznu mieru vplyvu na výsledok hodnotenia. Tento typ testovacích funkcií je vhodný na overovanie schopnosti algoritmov prispôsobiť sa nehomogénnej topológii problému. Príkladom je Rosenbrock funkcia, ktorá testuje schopnosť algoritmu riešiť úlohy s úzkymi a zakrivenými územiami optimálnych riešení [5][23].

Nelineárne funkcie

Predstavujú ďalšiu dôležitú skupinu testovacích problémov. Tieto funkcie majú zložité, často zakrivené plochy optimalizácie, ktoré môžu obsahovať zdanlivo náhodné prechody medzi jednotlivými oblasťami krajiny riešení. Príkladom môže byť Schwefel funkcia [6][23].

Okrem týchto základných kategórií existujú aj funkcie s dynamickými vlastnosťami, pri ktorých sa vlastnosti optimalizačnej krajiny menia v čase. Tieto funkcie sa používajú na testovanie adaptívnych schopností evolučných algoritmov v prostrediach, kde sa podmienky vyvíjajú počas optimalizácie [5].

Najčastejšie používané testovacie funkcie

Pri vývoji a porovnávaní výkonnosti evolučných algoritmov sa bežne využívajú štandardizované testovacie funkcie, ktoré svojimi vlastnosťami simulujú rôzne typy optimalizačných problémov. Tieto funkcie sú dôležité pre objektívne posudzovanie robustnosti, rýchlosti konvergencie a schopnosti algoritmov vyhľadávať globálne optimum v zložitých prostrediach [5][6]. V nasledujúcich podkapitolách budú predstavené niektoré z najčastejšie používaných funkcií.

Sphere

Sphere je jednou z najjednoduchších a najčastejšie používaných testovacích funkcií v oblasti optimalizácie. Definovaná je ako súčet druhých mocnín jednotlivých vstupných premenných.

kde x = (x1​,x2​,…,xn​) predstavuje vektor vstupných premenných [5][23].

Táto funkcia je unimodálna, hladká a symetrická okolo globálneho minima, ktoré sa nachádza v bode x=(0,0,…,0), kde f(x)=0. Sphere sa používa predovšetkým na testovanie základných schopností algoritmov konvergovať do optimálneho riešenia v prostredí bez lokálnych extrémov [5][6][23].

Rosenbrock

Rosenbrock funkcia, známa aj ako „banana function“, je definovaná nasledovne:

Táto funkcia je charakteristická zakriveným údolím, ktorého dno predstavuje globálne minimum [5][23].

Napriek tomu, že globálne minimum je jednoduché, úzka a zakrivená cesta k nemu robí optimalizáciu náročnou, pretože vyžaduje vysokú presnosť v krokoch optimalizačného algoritmu [5]. Rosenbrock je preto často využívaná na testovanie schopností algoritmov adaptovať sa na zložité topológie.

Rastrigin

Rastrigin funkcia je známa svojou periodickou a multimodálnou štruktúrou. Jej matematická definícia je:

Rastriginova funkcia má veľké množstvo lokálnych miním, pričom globálne minimum sa nachádza v bode x=(0,0,…,0) [5][23].

Vďaka svojej štruktúre je ideálna na testovanie schopností algoritmov vyhľadávať globálne optimum v prostredí s početnými pascami v podobe lokálnych extrémov [5][6].

Ackley

Ackley funkcia je ďalším príkladom náročného testovacieho problému, kombinujúcim veľké množstvo lokálnych extrémov a hladkú globálnu štruktúru. Funkcia je definovaná takto:

Ackleyho funkcia testuje schopnosť algoritmov vyhľadávať globálne minimum v prostredí so zdanlivo plochými oblasťami obklopenými viacerými lokálnymi extrémami [5][23].

Griewank

Griewank funkcia je kombináciou súčtu a súčinu komponentov vstupného vektora, a definuje sa nasledovne:

Funkcia obsahuje veľké množstvo lokálnych miním, avšak tie sú menej výrazné ako pri funkcií Rastrigin. Globálne minimum sa nachádza v bode x=(0,0,…,0) [5][23].

Griewank funkcia je vhodná pre testovanie schopnosti algoritmov preskúmavať rozsiahle optimalizačné prostredie s menej strmými zmenami hodnoty fitness [5].

Schwefel

Schwefel funkcia je známa svojou veľmi zložitou topológiou s množstvom lokálnych extrémov. Je definovaná ako:

Globálne minimum sa nachádza ďaleko od stredu definovanej oblasti, čo robí optimalizáciu náročnou, najmä pre algoritmy náchylné na predčasnú konvergenciu [5][6][23].

Michalewicz

Michalewicz funkcia je určená nasledovne:

Kde m je parameter určujúci ostrosť lokálnych miním (často sa používa m=10) [5][23].

Táto funkcia je náročná na optimalizáciu, pretože obsahuje veľmi hlboké a úzke lokálne minimá. Vhodná je predovšetkým na testovanie presnosti a jemnosti hľadania optimálnych riešení.

Testovacie funkcie s variabilnou zložitosťou

Štandardné testovacie funkcie majú často pevnú štruktúru a daný počet extrémov. V praxi však optimalizačné problémy zvyčajne vykazujú zložitosť, ktorá sa môže meniť v čase alebo v závislosti od špecifických parametrov problému. Z tohto dôvodu boli navrhnuté špeciálne triedy testovacích funkcií, ktoré umožňujú simulovať meniace sa alebo kontrolované prostredia [5][6].

Funkcie s kontrolovaným počtom extrémov

Funkcie s kontrolovaným počtom extrémov umožňujú explicitne nastavovať počet globálnych a lokálnych miním v optimalizačnom priestore. Takéto funkcie sú cenné pre štúdium správania algoritmov v závislosti od hustoty lokálnych extrémov a zložitosti účelovej funkcie [5].

Príkladom môže byť funkcia, kde parametre, ako sú počet vrcholov, ich poloha a výška, môžu byť priamo ovládané. Tento prístup umožňuje analyzovať, ako efektívne sa algoritmy dokážu vysporiadať so zvyšujúcou sa komplexnosťou problému [5][11].

Takéto funkcie sú tiež vhodné na testovanie odolnosti algoritmov voči premnoženiu lokálnych optím, kde hrozí riziko uviaznutia v lokálne optimálnych riešeniach [6].

Dynamicky sa meniace funkcie (dynamic fitness landscapes)

Dynamické optimalizačné problémy obsahujú situácie, kde sa topológia fitness mení v čase. Zmena môže byť periodická, náhodná alebo závislá od určitých udalostí v prostredí [5].

Testovanie evolučných algoritmov na dynamických účelových funkciách je dôležité pre aplikácie, kde podmienky optimalizácie nie sú statické, napríklad v oblasti riadenia, adaptívnych systémov alebo v modelovaní reálnych prírodných procesov [11].

Príkladmi dynamických benchmarkov sú „Moving Peaks Benchmark“ alebo rôzne modifikované verzie známych funkcií, kde sa extrémy pohybujú alebo menia svoju hodnotu v čase [5].

Úspešné riešenie dynamických problémov vyžaduje, aby algoritmy boli nielen efektívne v hľadaní optimálnych riešení, ale aj dostatočne flexibilné a adaptabilné na meniace sa podmienky [5][11].

Zložitejšie benchmarky a súťaže

Pre komplexnejšie testovanie evolučných algoritmov nestačia len jednoduché analytické funkcie. Preto boli vytvorené rozsiahle súbory benchmarkov a špeciálne súťaže, ktoré poskytujú náročné testovacie prostredia s cieľom realistickejšie simulovať podmienky reálnych optimalizačných problémov [5][21][22].

Tieto benchmarky obsahujú rôznorodé funkcie kombinujúce rôzne aspekty zložitosti, ako je napríklad multimodalita, variabilná mierka, rotované a korelované vstupné priestory alebo dynamické zmeny topológie. Využívajú sa najmä v akademických kruhoch aj v praktických aplikáciách na porovnávanie výkonnosti optimalizačných metód.

CEC benchmark sets

Súťaže organizované v rámci Congress on Evolutionary Computation (CEC) sa stali kľúčovým zdrojom štandardizovaných benchmarkov pre hodnotenie výkonnosti evolučných algoritmov. Medzi najznámejšie patria súťažné sady ako CEC 2005, CEC 2013 a CEC 2017, ktoré poskytujú špecificky navrhnuté optimalizačné problémy reflektujúce rôzne výzvy, s ktorými sa evolučné algoritmy stretávajú.

CEC 2005 sa zameriaval na problémy optimalizácie reálnych parametrov a bol jedným z prvých pokusov o štandardizáciu testovacích úloh v tejto oblasti [22]. Táto sada obsahuje 25 testovacích funkcií rozdelených do niekoľkých kategórií podľa zložitostí, multimodalít a podmienok optimalizačného priestoru.

CEC 2013 špecificky cielený na veľkorozmernú globálnu optimalizáciu (large-scale global optimization, LSGO) priniesol benchmarkovú sadu funkcií, ktoré obsahujú tisíce premenných [24]. Cieľom tejto súťaže bolo overiť robustnosť a efektivitu algoritmov pri riešení problémov so značným počtom dimenzií, kde tradičné algoritmy často narážajú na problémy so škálovateľnosťou a konvergenciou.

CEC 2017 sa sústredil na evolučnú optimalizáciu s viacerými cieľmi (many-objective optimization) [25]. Benchmarková sada obsahovala 10 úloh špeciálne navrhnutých pre testovanie schopnosti algoritmov zvládať problémy s vysokým počtom optimalizačných cieľov, často 10 a viac. Táto súťaž priniesla nové výzvy v oblasti kompromisného riešenia medzi rôznymi cieľmi a vyváženosti medzi exploráciou a exploitáciou v riešeniach.

Tieto benchmarkové súbory sa stali štandardom v komunite a často slúžia ako základné porovnávacie testy pri vývoji a hodnotení nových evolučných metód. Zároveň umožňujú porovnávanie rôznych algoritmov na rovnakých testovacích podmienkach, čo je dôležité pre objektívne hodnotenie výkonnosti.

Black-Box Optimization Benchmarking (BBOB)

Ďalšou významnou iniciatívou v oblasti hodnotenia optimalizačných algoritmov je projekt Black-Box Optimization Benchmarking (BBOB), ktorý vznikol v rámci GECCO a CEC komunít. BBOB sa zameriava na testovanie optimalizačných algoritmov v kontexte problémov, kde nie je známa analytická forma cieľovej funkcie, a teda algoritmus môže s funkciou komunikovať iba prostredníctvom dotazovania sa na hodnoty — tzv. čierna skrinka [21].

BBOB ponúka rozsiahlu sadu benchmarkových funkcií, ktoré boli navrhnuté tak, aby pokrývali široké spektrum charakteristík optimalizačných úloh, konkrétne rôznych druhov multimodalít, variabilnej škálovateľnosti, prítomnosti zložitých korelácií medzi premennými, ako aj funkcií s rôznou hladkosťou a štruktúrou krajiny hľadania.

Výhodou BBOB benchmarkov je, že každá testovacia funkcia je definovaná v niekoľkých verziách s rôznymi transformáciami (rotácie, posuny, škálovania), čo zabezpečuje, že algoritmy sa nemôžu „naučiť“ riešenie konkrétnej fixnej formy problému. Každá funkcia je navyše podrobne štatisticky analyzovaná, čo umožňuje hlboké vyhodnotenie správania sa algoritmov vrátane ich robustnosti a konvergenčných vlastností [21].

BBOB sa často používa v kombinácii s nástrojmi ako COCO (COmparing Continuous Optimizers), ktoré poskytujú štandardizované metodiky na vykonávanie experimentov a spracovanie výsledkov. Tento prístup výrazne zjednodušuje reprodukovateľnosť experimentov a objektívne porovnávanie rôznych optimalizačných stratégií.

Vzhľadom na svoju modularitu a dôraz na objektívne a dôkladné hodnotenie je BBOB benchmark sada považovaná za jednu z najspoľahlivejších a najširšie akceptovaných v komunite výskumu evolučných algoritmov.

Výber vhodnej testovacej funkcie v závislosti od typu algoritmu

Výber vhodnej testovacej funkcie je zásadným krokom pri hodnotení a porovnávaní výkonnosti evolučných algoritmov. Nie každá funkcia je rovnako vhodná pre každý typ algoritmu, keďže rôzne optimalizačné techniky môžu byť citlivé na špecifické vlastnosti optimalizačného problému, ako sú multimodalita, korelácia parametrov alebo prítomnosť klamných extrémov [5][6].

Unimodálne funkcie, ako napríklad Sphere alebo Rosenbrockova funkcia, sú často používané na testovanie základných schopností algoritmu rýchlo konvergovať k optimu. Tieto funkcie umožňujú analyzovať napríklad rýchlosť konvergencie alebo schopnosť algoritmu vyhýbať sa stagnácii [5].

Multimodálne funkcie, ako Rastrigin, Ackley či Schwefel funkcia, kladú na algoritmy vyššie nároky tým, že obsahujú veľké množstvo lokálnych extrémov. Tieto funkcie sú vhodné pre testovanie robustnosti algoritmov, ich schopnosti vyhýbať sa predčasnej konvergencii a ich efektivity pri globálnom vyhľadávaní [5][7].

Funkcie s vysokou koreláciou parametrov (napr. zakrivený Rosenbrock) slúžia na overenie schopnosti algoritmov efektívne prehľadávať šikmé alebo deformované priestory riešení. Evolučné algoritmy, ktoré využívajú adaptívne stratégie alebo evolučné operátory berúce do úvahy korelácie, sa na týchto funkciách typicky ukazujú ako výkonnejšie [4][5].

Vysoko rozmerné problémy (Large-Scale Optimization) testujú škálovateľnosť algoritmov, teda ako dobre dokážu riešiť úlohy so stovkami až tisíckami premenných. Pre takéto prípady sa využívajú špecifické benchmarky, ako CEC 2013 alebo CEC 2017, kde sú optimalizačné problémy navrhnuté s dôrazom na efektívne spracovanie veľkých rozmerov [24][25].

Dynamicky sa meniace funkcie sú vhodné pre testovanie algoritmov v oblasti prispôsobenia sa v prostrediach, kde sa cieľová funkcia alebo podmienky optimalizácie časom menia. Evolučné algoritmy, ktoré využívajú mechanizmy pamäte alebo rýchle adaptívne operátory, dosahujú v týchto prípadoch spravidla lepšie výsledky [5][11].

Výber testovacích funkcií by preto mal zodpovedať charakteristikám cieľovej aplikácie, typu optimalizovaného problému, ako aj požiadavkám na analýzu konkrétnych aspektov správania algoritmu (rýchlosť konvergencie, robustnosť, presnosť, škálovateľnosť atď.). Kombinovanie viacerých typov testovacích funkcií umožňuje získať komplexný prehľad o silných a slabých stránkach skúmaných optimalizačných metód.

|  |  |
| --- | --- |
|  | praktická část |

Výber testovacích funkcií pre experimentálne testovanie

V tejto kapitole budú predstavené a odôvodnené testovacie funkcie a benchmarkové úlohy, ktoré budú použité pre hodnotenie navrhnutých evolučných algoritmov s variabilnou dĺžkou jedinca. Taktiež budú vysvetlené rozdiely oproti štandardným testom s fixnou dĺžkou. V závere bude vysvetlený výber jednotlivých funkcií a vytvorená tabuľka s vybranými testovacími funkciami a benchmarkmi.

Rozdiely medzi testovaním s fixnou a variabilnou dĺžkou jedinca

V klasických evolučných algoritmoch (napr. genetický algoritmus, diferenciálna evolúcia) je jedinec reprezentovaný ako vektor pevnej dĺžky n, kde každá pozícia zodpovedá jednému parametru optimalizácie [5]. Táto „fixná dimenzionalita“ umožňuje jednoduché použitie analytických testovacích funkcií, ktoré majú presne definovaný počet vstupov. Naopak, techniky ako genetické programovanie, gramatická evolúcia či analytické programovanie pracujú s jedincami, ktorých štruktúra a dĺžka (počet uzlov v strome, dĺžka programu či výrazu) sa môže počas evolúcie meniť [3][10][14]. Takáto variabilná dĺžka jedinca prináša schopnosť objavovať riešenia rôznej komplexity bez preddefinovania finálnej štruktúry.

Obmedzenia klasických testovacích funkcií pri variabilnej dĺžke

Analytické testovacie funkcie, ako Sphere, Rastrigin, Ackley či Griewank, sú navrhnuté pre vstupný vektor dimenzie n a bez úprav neumožňujú meniť počet premenných počas jedného behu algoritmu [6]. Pokus o aplikáciu týchto funkcií na jedincov rozdielnej dĺžky vedie k nejednoznačnému hodnoteniu: nedostatočný počet parametrov môže znamenať chýbajúce dimenzie, nadbytočné zas penalizáciu či ignorovanie časti genotypu. Výsledkom je, že variabilná dĺžka genómu nie je skutočne využitá a evolučný postup degeneruje na optimalizáciu pevne stanovenej či čiastočne ignorovanej podmnožiny génov [1][5].

Potreba benchmarkových problémov s prirodzenou variabilitou

Aby bolo možné plne otestovať schopnosť evolučných algoritmov využiť variabilnú dĺžku jedinca, je nutné zaradiť do testovacieho setu úlohy, v ktorých štruktúra genómu priamo ovplyvňuje kvalitu riešenia. Typickými príkladmi sú symbolická regresia či Santa Fe Trail v genetickom programovaní, kde dĺžka stromu či programu určuje výrazový tvar riešenia [15][16]. Tieto benchmarkové problémy umožňujú, aby krátki jedinci boli príliš jednoduchí a dlhí príliš zložití, čím vzniká prirodzená selekcia optimálnej veľkosti štruktúry bez umelých penalizácií. Takéto úlohy reprezentujú skutočné scenáre, v ktorých nie je pevne dané, koľko parametrov či uzlov je potrebných, a algoritmus tak musí samostatne zisťovať vhodnú mieru komplexity [14][16].

Kritéria výberu testovacích funkcií

Pri zostavovaní súboru testovacích funkcií pre experimentálne hodnotenie evolučných algoritmov s variabilnou dĺžkou jedinca je potrebné zohľadniť niekoľko kľúčových kritérií. Správna kombinácia vlastností testovacích úloh zabezpečí, že vyhodnotenie bude komplexné, objektívne a odhalí silné i slabé stránky navrhnutých metód.

Pokrytie unimodálnych a multimodálnych prostredí

Unimodálne funkcie (napr. Sphere) obsahujú jedno globálne optimum, bez lokálnych pascí, a slúžia na overenie základnej schopnosti algoritmu konvergovat k riešeniu [6]. Multimodálne funkcie (ako Rastrigin či Ackley) disponujú mnohými lokálnymi optimami, ktoré testujú robustnosť metódy proti predčasnej konvergencii [1][5]. Výber oboch typov zabezpečuje, že algoritmus bude skúmaný v jednoduchých i zložitých topológiach fitnes.

Podpora škálovateľnosti dimenzionality

Pre algoritmy s variabilnou dĺžkou jedinca je nevyhnutné testovať správanie pri rôznom počte parametrov. Testovacie funkcie musia byť definované tak, aby ich dimenzionalitu (počet vstupov) bolo možné meniť bez zmeny základnej charakteristiky úlohy [5]. Adaptované verzie Sphere, Rastrigin, Griewank a Ackley umožňujú práve takúto škálovateľnosť, pričom fitness sa vypočítava pre aktuálny počet génov jedinca.

Možnosť adaptácie na dynamické zmeny dĺžky genómu

Efektívne testovanie variabilnej dĺžky vyžaduje, aby funkcia korektne vyhodnotila jedincov s rôznym počtom génov bez špeciálnych výnimiek. Preto upravené funkcie musia prijať ako parameter aktuálnu dĺžku a prispôsobiť výpočet sumy, súčinu či exponenciálnych členov tejto veľkosti [6]. Tým sa zabezpečí, že evolúcia môže navyšovať i znižovať dĺžku genómu počas behu bez straty konzistencie hodnoty účelovej funkcie.

Zahrnutie numerických aj symbolických problémov

Klasické numerické funkcie overujú správanie v spojitých priestoroch, no pre variabilnú dĺžku je rovnako dôležité testovať diskrétne a štruktúrované úlohy. Symbolická regresia a Santa Fe Trail reprezentujú situácie, v ktorých jedinec nie je vektor, ale programová alebo výrazová štruktúra s variabilným počtom uzlov [14][16]. Knapsack problem a N-bit parity potom dopĺňajú kombinatoriku a boolean logiku. Týmto spôsobom sú pokryté všetky dôležité kategórie problémov, na ktorých môže variabilná dĺžka priniesť výhodu.

Predstavenie vybraných testovacích funkcií a benchmarkov

V tejto časti sú predstavené jednotlivé testovacie funkcie a benchmarky vybrané na experimentálne hodnotenie evolučných algoritmov s variabilnou dĺžkou jedinca. Pri každej funkcii je popísaná jej základná charakteristika, dôvod zaradenia do súboru, a špecifiká pri použití v prostredí s variabilnou dĺžkou genómu.

Sphere

Sphere funkcia v základnej verzii predstavuje jednoduchú unimodálnu optimalizačnú úlohu. Aby bolo možné jej použitie pre jedincov s variabilnou dĺžkou genotypu, bola upravená pridaním penalizácie za rozdiel medzi skutočnou dĺžkou jedinca a predpokladanou cieľovou dimenziou dtarget​. Upravená forma funkcie je:

Kde λ je penalizačný koeficient, ktorého odporúčaná hodnota je 10 až 20 [26]. Táto úprava umožňuje hodnotiť aj jedincov s kratším alebo dlhším genotypom bez potreby ich násilnej konverzie na fixnú dĺžku.

Rastrigin

Rastrigin funkcia je známa svojou výraznou multimodalitou. Úprava pre variabilnú dĺžku je analogická ako v prípade Sphere funkcie:

Penalizačný člen zvyšuje hodnotu fitness v prípade odchýlky od cieľovej dimenzie [26].

Griewank

Griewank funkcia kombinuje aditívne a multiplikatívne členy a je vhodná na testovanie komplexnejších charakteristík optimalizačných algoritmov. Upravená verzia:

Takáto modifikácia umožňuje testovať schopnosť algoritmov správne sa adaptovať nielen na hodnoty parametrov, ale aj na ich počet [26].

Ackley

Ackley funkcia je často používaná pre testovanie algoritmov v prostredí s veľkým množstvom lokálnych extrémov. Upravená verzia so zaradením penalizácie je:

Kde P(x) je penalizácia daná ako v predošlých funkciách vzorcom:

Týmto spôsobom je možné efektívne posudzovať variabilitu štruktúry jedincov v prostrediach s vysokou zložitosťou optimalizácie [26].

Santa Fe Trial

Santa Fe Trail je klasická úloha využívaná predovšetkým v genetickom programovaní. Ide o navigačný problém, kde agent (mravec) prechádza dvojrozmernou mriežkou a zbiera potravu rozmiestnenú na preddefinovanej trase. Úloha je vhodná pre testovanie algoritmov s variabilnou dĺžkou genotypu, pretože optimálne riešenie závisí od schopnosti vyvíjať komplexné, dynamické stratégie, a nie od fixnej dĺžky vstupného vektora [3][14][15].

Fitness funkcia hodnotí jedinca podľa počtu zozbieranej potravy v obmedzenom počte krokov. Riešenia môžu mať rôzne veľkosti a štruktúru, čo umožňuje prirodzený rozvoj variabilnej dĺžky jedinca.

Koza benchmark

Koza benchmark je štandardná testovacia sada úloh v oblasti genetického programovania, ktorú pôvodne definoval John R. Koza [3][16]. Obsahuje rôzne problémy, ako sú symbolická regresia, logické úlohy (napr. návrh parity alebo multiplexer funkcií) a ďalšie typické príklady, ktoré testujú schopnosť algoritmov nielen optimalizovať parametre, ale aj vyvíjať štruktúry riešení.

Účelová funkcia sa v rámci týchto úloh definuje individuálne podľa konkrétnej úlohy, pričom cieľom je minimalizácia chyby (v prípade regresie) alebo maximalizácia správnosti riešenia (v prípade logických problémov).

Koza benchmark je osobitne vhodný na hodnotenie evolučných algoritmov s dynamickou reprezentáciou, pretože podporuje tvorbu riešení s rôznou dĺžkou a štruktúrou, čo umožňuje testovať aj schopnosti explorácie a exploitácie v zložitých priestoroch riešení [3][16].

Knapsack problem

Knapsack problem je známy kombinatorický optimalizačný problém, kde cieľom je vybrať podmnožinu položiek s danými váhami a hodnotami tak, aby celková hodnota bola maximalizovaná bez prekročenia kapacity batohu [2][5][6].

Formálna definícia účelovej funkcie môže byť upravená nasledovne:

Kde vi​ je hodnota položky a P(x) je penalizácia za prekročenie kapacity batohu. Variabilná dĺžka jedinca je priamo žiaduca, pretože umožňuje rôzny počet vybraných položiek, a teda rôzne dĺžky genotypov.

N-bit parity problem

N-bit parity problem je klasická úloha v oblasti evolučných algoritmov a genetického programovania, kde cieľom je naučiť sa funkciu parity – t. j. určiť, či počet jednotiek v binárnom vektore danej dĺžky je párny alebo nepárny [3][5][8].

Účelová funkcia je zvyčajne definovaná ako počet správne klasifikovaných binárnych vstupov. Variabilná dĺžka reprezentácie umožňuje vyvíjať rôzne logické štruktúry (stromy, programy) schopné riešiť problém pre rôzne veľkosti N bez fixného počtu operátorov alebo premenných.

Zdôvodnenie výberu použitých testovacích funkcií a benchmarkov

Výber konkrétnych testovacích funkcií a benchmarkov bol realizovaný na základe niekoľkých dôležitých faktorov súvisiacich s cieľmi experimentálnej časti tejto práce. Primárnym cieľom bolo zabezpečiť, aby boli testovacie úlohy dostatočne rôznorodé z hľadiska vlastností optimalizovanej funkcie, typu problémového priestoru a požiadaviek na schopnosti testovaných evolučných algoritmov.

Základné optimalizačné funkcie (Sphere, Rosenbrock, Rastrigin, Griewank) boli vybrané, pretože predstavujú štandardné benchmarky používané v literatúre na overovanie výkonnosti optimalizačných algoritmov, ako je uvedené v [1][5][21][22]. Tieto funkcie sú známe svojimi rôznymi vlastnosťami.

* sphere – jednoduchá konvexná funkcia s jedným globálnym minimom, vhodná na overenie základnej schopnosti konvergencie algoritmu;
* rosenbrock – úloha so zakriveným údolím, ktorá preverí schopnosť algoritmu hľadať v úzkych optimálnych oblastiach;
* rastrigin – multimodálna funkcia s veľkým počtom lokálnych miním, čím simuluje náročnosť hľadania globálneho optima v prítomnosti mnohých „lokálnych pascí“;
* griewank – funkcia s kombinovaným súčtovým a súčinovým efektom, ktorá zvyšuje nároky na presnosť a globálne prehľadávanie priestoru.

Benchmarkové úlohy (Santa Fe Trail, Koza benchmark, Knapsack problem, N-bit parity problem) boli zvolené s cieľom otestovať algoritmy na zložitejších, reálnejších a diskrétnych problémoch, ktoré lepšie vyobrazujú praktické využitie evolučných algoritmov.

* Santa Fe Trail [3] umožňuje overiť schopnosti algoritmu v prostredí, kde je potrebné hľadať sekvenciu rozhodnutí v dynamickom a neúplne známom priestore;
* Koza benchmark [3][16] reprezentuje súbor štandardných problémov používaných v genetickom programovaní, ktorých príkladmi sú syntéza funkčných výrazov, logických operácií aj riešenie kombinatorických úloh. Overuje schopnosť algoritmov efektívne pracovať v priestore stromových štruktúr, operátorov a variabilných dĺžok riešení, čo je charakteristické pre oblasť genetického programovania;
* Knapsack problem [2][5] slúži ako klasický problém diskrétnej optimalizácie, kde je potrebné vyvažovať viacero kritérií vo vzájomných konfliktoch pod obmedzením kapacity;
* N-bit parity problem [3][8] predstavuje náročnú úlohu, kde je potrebné modelovať komplexné logické vzťahy, čo je testom schopnosti algoritmu generalizovať.

Testovacie funkcie a benchmarky boli vyberané tak, aby výsledný súbor testovacích úloh umožňoval testovanie kontinuálnych aj diskrétnych priestorov, overenie schopnosti hľadania globálneho optima v prítomnosti viacerých lokálnych extrémov a simuláciu reálnych aplikácií, ako sú regresia, optimalizácia s obmedzeniami a logické úlohy.

Ďalším dôležitým aspektom výberu bolo, že všetky použité funkcie a benchmarky sú dobre zdokumentované v literatúre a bežne používané v porovnávacích štúdiách, čo umožňuje nielen validáciu výsledkov tejto práce, ale aj ich porovnateľnosť s existujúcimi výskumnými prácami [1][3][5][21][23].

Tabuľka použitých testovacích funkcií a benchmarkov

Pre lepšiu prehľadnosť boli všetky vybrané testovacie funkcie a benchmarky zhrnuté v tabuľke 2. Ich výber bol realizovaný na základe zdrojov literatúry [1][2][3][5][8][14][16][21][22].

Tabuľka 2 Porovnanie základných evolučných algoritmov

|  |  |  |
| --- | --- | --- |
| **Názov** | **Typ úlohy** | **Charakteristika** |
| Sphere | Kontinuálna optimalizácia | Jednoduchá konvexná funkcia s jedným globálnym minimom. |
| Rosenbrock | Kontinuálna optimalizácia | Funkcia s úzkym zakriveným údolím, náročná na konvergenciu. |
| Rastrigin | Kontinuálna optimalizácia | Multimodálna funkcia s veľkým počtom lokálnych miním. |
| Griewank | Kontinuálna optimalizácia | Kombinácia súčtu a súčinu, test schopnosti globálneho hľadania. |
| Santa Fe Trail | Pathfinding | Dynamická úloha so sekvenčným rozhodovaním v čiastočne známom prostredí. |
| Koza benchmark | Evolučná syntéza programov | Testovanie schopnosti pracovať so stromovými štruktúrami a funkčnými operátormi. |
| Knapsack problem | Diskrétna optimalizácia | Výber optimálnej podmnožiny položiek pri kapacitnom obmedzení. |
| N-bit parity problem | Logická úloha | Modelovanie a generalizácia komplexných logických vzťahov. |

Implementácia existujúcich evolučných výpočetných techník s variabilnou dĺžkou jedinca

Táto kapitola sa zaoberá dokumentáciou implementácií dvoch evolučných algoritmov, ktoré prirodzene podporujú variabilnú dĺžku reprezentácie jedinca – genetického programovania (GP) [3] a gramatickej evolúcie (GE) [10]. V úvodných častiach práce už boli stručne predstavené základné princípy týchto algoritmov, preto bude pozornosť v tejto kapitole upriamená predovšetkým na implementačné detaily, reprezentácie jedincov a praktické aspekty kontroly variabilnej dĺžky počas evolučného procesu.

Oba algoritmy boli implementované v jazyku Python, pričom dôležitým nástrojom v prípade implementácie GP bola knižnica deap, ktorá poskytuje flexibilný rámec na realizáciu evolučných výpočtových metód. Použitím tejto knižnice bolo možné efektívne vytvárať, manipulovať a vyhodnocovať rôzne typy jedincov, ako aj jednoducho sledovať zmeny ich štruktúry počas evolučného behu [5].

Dôležitou súčasťou implementácie bolo zabezpečenie reprodukovateľnosti experimentov, ktoré bolo dosiahnuté dôsledným nastavením a zaznamenávaním všetkých použitých parametrov vrátane počiatočných semien generátorov náhodných čísel. Výsledky experimentov boli zaznamenávané do .csv súborov, čo umožnilo neskoršiu podrobnú analýzu priebehu evolúcie, vrátane sledovania zmien veľkosti a kvality jedincov.

Táto kapitola poskytuje dôkladný popis implementačných krokov a použitých techník na efektívne zvládnutie výziev spojených s variabilnou dĺžkou jedinca v evolučných algoritmoch a tvorí základ pre porovnávaciu analýzu uvedených prístupov s upravenými štandardnými evolučnými algoritmami, ktoré budú predmetom nasledujúcej časti práce.

Spoločné parametre pre implementáciu

V tejto časti sú podrobne popísané spoločné prvky implementácie, ktoré sú zdieľané medzi oboma testovanými evolučnými technikami – genetickým programovaním (GP) a gramatickou evolúciou (GE). Týmito prvkami sú konkrétne použité softvérové nástroje, základné parametre riadiace evolučný proces a spôsob, akým boli výsledky zaznamenávané. Cieľom tohto zjednotenia bolo dosiahnuť konzistentnosť naprieč implementáciami všetkých použitých algoritmov, čo je nutné pre objektívne porovnávanie ich vlastností a výkonu [5].

Použitý programovací jazyk a vývojové prostredie

Implementácia evolučných techník bola realizovaná v programovacom jazyku Python. Tento jazyk patrí medzi vysokoúrovňové, interpretované jazyky so silnou podporou pre vedecké výpočty, vývoj prototypov a dátovú analýzu. Jeho zrozumiteľná syntax, rozsiahla štandardná knižnica a množstvo dostupných rozšírení z neho robia vhodnú voľbu pre výskumné úlohy v oblasti evolučných algoritmov [40].

Ako vývojové prostredie bolo použité JetBrains DataSpell, ktoré je priamo optimalizované pre prácu s dátami a výpočtami v prostredí Jupyter Notebook. Zdrojové súbory boli vytvorené vo formáte .ipynb, čo umožňuje kombinovať výpočtový kód, výstupy a textové poznámky v jednom dokumente. Tento prístup uľahčuje experimentovanie, dokumentovanie a reprodukovateľnosť výskumu [41].

Použité moduly a knižnice

V rámci implementácie boli okrem samotného jazyka Python využité aj viaceré knižnice a moduly, ktoré zohrávajú úlohu pri definovaní správania algoritmov, spracovaní údajov a zabezpečení reprodukovateľnosti experimentov. Jednou z hlavných knižníc bola deap (Distributed Evolutionary Algorithms in Python), ktorá poskytuje komplexný framework na návrh a konfiguráciu evolučných algoritmov. DEAP ponúka flexibilné objektovo orientované rozhranie na reprezentáciu jedincov, populácií a evolučných operátorov, ako sú selekcia, mutácia či kríženie [5].

Na podporu numerických výpočtov bola použitá knižnica NumPy, ktorá umožňuje efektívnu prácu s vektorovými údajmi, matícami a ďalšími číselnými štruktúrami. V kontexte evolučných algoritmov slúžila najmä na generovanie vstupných hodnôt, výpočty v rámci účelových funkcií a zabezpečenie nastavenia vstupných bodov generátora náhodných čísel (angl. seed) [42].

Generovanie náhodných hodnôt, ktoré je nevyhnutné pri stochastickej povahe evolučných techník, bolo zabezpečené pomocou štandardného modulu random. Tento modul bol využitý na inicializáciu populácií, výber rodičov pri turnajovej selekcii, náhodný výber bodov kríženia a realizáciu mutácií [34].

Na zaznamenávanie výsledkov experimentov do výstupných súborov bol použitý modul csv, ktorý umožňuje zapisovanie dát vo formáte vhodnom pre ďalšie spracovanie, napríklad v tabuľkových procesoroch alebo pri štatistickej analýze [36]. Každý experimentálny beh bol identifikovaný pomocou unikátneho identifikátora vytvoreného modulom uuid, čo zabezpečuje jednoznačnosť záznamov a zjednodušuje ich spätné vyhľadávanie [37].

Meranie času výpočtu jednotlivých behov prebiehalo prostredníctvom modulu time, konkrétne funkciou perf\_counter(), ktorá poskytuje vysokú presnosť merania a je vhodná pre profilovanie výkonnosti algoritmov [35].

Na správu súborového systému, vytváranie priečinkov pre výsledky a prácu s cestami k súborom slúžil modul os. Jeho využitie zabezpečilo, že každý experiment bol zaznamenaný do samostatného adresára s jedinečným názvom obsahujúcim časovú pečiatku, čím bola zabezpečená organizácia a prehľadnosť dát [38].

Napokon, pri definovaní funkcií použitých v evolučných stromoch a pravidlách gramatiky bol použitý modul operator. Tento modul poskytuje základné aritmetické, logické a porovnávacie operácie vo forme funkcií, ktoré je možné jednoducho registrovať ako súčasť primitívnych množín v rámci knižnice deap [39].

Nasledujúci blok zdrojového kódu predstavuje spoločnú časť vloženia („importu“) použitých knižníc pre obidve implementácie.

import random

import numpy as np

import time

import uuid

import csv

import os

Nastavenie experimentov

Oba implementované algoritmy boli navrhnuté s cieľom zabezpečiť rovnaké podmienky pre porovnanie výkonnosti. Z tohto dôvodu boli použité identické parametre určujúce veľkosť populácie, rozsah vyhodnocovania a počet behov (replikácií). Tieto parametre sú definované nasledovne:

* POP\_SIZE = 100 – veľkosť populácie, ktorá bola počas celého behu udržiavaná konštantná;
* N\_EVALS = 100000 – maximálny počet vyhodnotení fitness funkcie, ktorý slúži ako hlavné zastavovacie kritérium evolučného procesu.
* N\_REPLICATES = 30 – počet nezávislých opakovaní každého experimentu pre zabezpečenie štatistickej významnosti výsledkov.

Týmto nastavením sa dosiahlo, že každý benchmark bol riešený 30-krát nezávisle, pričom každý beh prebiehal dovtedy, kým nebolo vyhodnotených 100 000 jedincov. Veľkosť populácie 100 zodpovedá štandardným odporúčaniam pre evolučné algoritmy, ktoré balansujú medzi diverzitou a rýchlosťou konvergencie [5][6].

Každému behu bol pred jeho spustením priradený náhodný seed (štartovací bod na generovanie sekvencie náhodných čísel). Seed bol nastavený osobitne pre modul random aj numpy.random, čím bola zabezpečená plná kontrola nad stochastickým správaním systému.

POP\_SIZE = 100

N\_EVALS = 100000

N\_REPLICATES = 30

Použité testovacie benchmarky

Z pôvodne navrhnutej množiny ôsmich testovacích úloh boli v tejto časti implementácie použité tri vybrané benchmarky: Koza benchmark, N-bit parity problem a Santa Fe Trail. Rozhodnutie o výbere týchto úloh bolo podložené ich vhodnosťou vzhľadom na vlastnosti testovaných evolučných techník, ako aj požiadavkou na konzistentnosť a interpretovateľnosť výsledkov.

Prvým dôvodom výberu je povaha reprezentácie jedinca v genetickom programovaní a gramatickej evolúcii. Obe tieto techniky využívajú stromové alebo gramatické štruktúry, ktorých dĺžka a tvar sa počas evolúcie dynamicky menia. Takéto reprezentácie sú prirodzene vhodné pre úlohy, kde riešenie predstavuje výrazy, logické funkcie alebo programové sekvencie. Benchmarky Koza, Parity a Santa Fe túto požiadavku napĺňajú – jedinec tu nepredstavuje fixný vektor, ale dynamickú funkčnú alebo procedurálnu štruktúru [3][10][14].

Naopak, analytické optimalizačné funkcie ako Sphere, Rastrigin, Griewank a Ackley sú definované v spojitom priestore Rn a ich použitie je typické pre algoritmy s pevnou dĺžkou jedinca, ako sú genetické algoritmy, diferenciálna evolúcia alebo CMA-ES [1][5]. Pri snahe použiť ich v GP alebo GE by bolo potrebné zaviesť zložité mechanizmy na kódovanie algebraických operácií (napr. súčtu štvorcov) do programových štruktúr, čo by viedlo k neefektívnemu a neprirodzenému evolučnému procesu. Navyše, výsledky získané takýmto spôsobom by neboli priamo porovnateľné s metódami využívajúcimi reálny vektor ako základnú reprezentáciu.

Kombinatorické úlohy, ako napríklad Knapsack problem, síce umožňujú použitie gramatickej evolúcie (napr. kódovaním výberu položiek), avšak nie sú prirodzeným cieľom genetického programovania, ktoré sa zameriava na vývoj výrazových alebo procedurálnych riešení. Oproti tomu problémy ako Parity či Santa Fe Trail predstavujú logické a diskrétne úlohy, ktoré sú v GP a GE bežne riešené pomocou logických operátorov, podmienkových konštrukcií a sekvencií krokov. Umožňujú tak naplno využiť potenciál týchto metód [3][16].

Výber iba troch „programových“ benchmarkov taktiež napomáha logickému rozdeleniu experimentálnej časti diplomovej práce. GP a GE sú v tejto práci testované výlučne na úlohách, kde variabilná dĺžka a štruktúra jedinca zohráva rolu a je vhodná pre dané reprezentácie jedincov. Takéto rozdelenie umožňuje vytvoriť konzistentné a metodologicky čisté porovnanie výkonu algoritmov v doménach, na ktoré sú navrhnuté.

Implementácia genetického programovania

Táto podkapitola sa zameriava na analýzu praktickej implementácie genetického programovania (GP) v prostredí programovacieho jazyka Python s využitím knižnice deap. Teoretický základ GP, vrátane základných princípov, ako sú reprezentácia jedinca pomocou stromových štruktúr, aplikácia genetických operátorov a problém tzv. bloatu, je podrobne popísaný v predchádzajúcich častiach práce. V tejto časti sa práca venuje výlučne spôsobu, akým je GP implementované v kóde. Dôraz je taktiež kladený na variabilnú dĺžku jedinca a spôsob akým pracuje v tejto konkrétnej implementácií [3][15].

Implementácia rozoberaná v tejto kapitole je navrhnutá tak, aby umožňovala aplikáciu GP na rôzne úlohy, pričom v konkrétnom skripte sú definované tri už spomínané benchmarkové problémy – Koza benchmark, Parity problem a Santa Fe trail. Každý benchmark využíva vlastnú množinu primitív a účelovú funkciu. Napriek tomu majú tieto implementácie spoločnú základnú štruktúru, ktorá sa opiera o nasledovné komponenty knižnice deap [36]:

* gp.PrimitiveTree ako stromová reprezentácia jedinca;
* base.Toolbox pre definovanie operácií evolučného algoritmu;
* tools pre výber, kríženie, mutáciu a zaznamenávanie najlepších riešení;
* algorithms pre riadenie hlavného evolučného cyklu.

Z pohľadu tejto práce je dôležité, že každý jedinec je reprezentovaný ako strom s premennou dĺžkou, ktorá sa mení v priebehu evolučného procesu v dôsledku aplikácie genetických operátorov. Tento aspekt je úzko spätý s javom známeho ako bloat – nekontrolovaný rast programových stromov – a preto implementácia obsahuje niekoľko mechanizmov na jeho obmedzenie [17].

Na úvod sú do prostredia importované potrebné knižnice, medzi nimi aj štandardné moduly jazyka Python ako random, operator, uuid, time, csv a os (spomínané v spoločných parametroch), a komponenty z balíka deap:

from deap import base, creator, gp, tools, algorithms

Z pohľadu implementácie GP sú podstatné moduly gp (obsahujúci operácie pre stromy), tools (sada preddefinovaných evolučných operátorov) a creator, ktorý umožňuje dynamickú tvorbu nových typov objektov, vrátane definícií jedincov a účelových funkcií.

Následne sú definované špecifické množiny primitív – základných stavebných blokov, z ktorých sa skladajú jedinci (tzv. PrimitiveSet) pre jednotlivé úlohy. Každá z nich je zostavená tak, aby poskytovala operátory a terminály adekvátne povahe riešeného problému. Napríklad pre Koza benchmark sa vytvára množina [39]:

pset\_sr = gp.PrimitiveSet('MAIN\_SR', 1)

pset\_sr.addPrimitive(operator.add, 2)

pset\_sr.addPrimitive(operator.sub, 2)

pset\_sr.addPrimitive(operator.mul, 2)

pset\_sr.addPrimitive(protectedDiv, 2)

pset\_sr.addEphemeralConstant('rand101', lambda: random.uniform(-1, 1))

pset\_sr.renameArguments(ARG0='x')

Každý operátor je tu pridaný s určeným počtom vstupov, ktoré vyžaduje na svoje vykonanie (napríklad operator.add potrebuje dva vstupy, teda je binárny operátor). Funkcia protectedDiv slúži na zabezpečenie stability pri delení, aby sa predišlo chybe pri delení nulou. Týmto spôsobom sa dosahuje vyššia robustnosť evolučne generovaných výrazov, čo je v rámci genetického programovania bežná a odporúčaná prax [3][15]. Kriticky dôležitý krok pre podporu stromovej štruktúry s premenlivou dĺžkou spočíva v definícii nových typov jedincov. Tento krok je realizovaný pomocou creator.create [39]:

creator.create('IndividualKOZA',gp.PrimitiveTree,fitness=creator.FitnessMin)

Tu je vytvorený nový typ jedinca pomenovaný ako IndividualKOZA, ktorý dedí zo základnej triedy gp.PrimitiveTree. Táto trieda predstavuje usporiadaný strom reprezentovaný ako lineárny zoznam inštrukcií v prefixovej notácii. Dĺžka tohto zoznamu, ktorú je možné získať príkazom len(ind), priamo zodpovedá počtu uzlov v strome. Vzhľadom na to, že každý uzol môže byť funkciou, alebo terminálom (vstupná premenná, konštanta), výsledná dĺžka jedinca môže byť rozdielna v priebehu evolúcie.

Táto vlastnosť – možnosť meniť dĺžku (veľkosť) jedinca v čase – je zásadnou vlastnosťou genetického programovania a odlišuje ho od klasických genetických algoritmov, kde je dĺžka chromozómu typicky fixná [1][3][5].

K vytváraniu jedincov v populácii slúžia inicializačné funkcie, ktoré sú registrované v Toolboxe. Napríklad pre benchmark Koza je definovaná nasledovne [39]:

toolbox\_koza = base.Toolbox()

toolbox\_koza.register('expr', gp.genHalfAndHalf, pset=pset\_sr, min\_=1, max\_=3)

toolbox\_koza.register('individual', tools.initIterate, creator.IndividualKOZA, toolbox\_koza.expr)

toolbox\_koza.register('population', tools.initRepeat, list, toolbox\_koza.individual)

Použitie metódy gp.genHalfAndHalf, ktorá slúži na náhodné generovanie stromov rôznej štruktúry a veľkosti, v rozsahu výšok od min\_ po max\_, je dôležitá pri inicializácií. Vďaka tejto inicializácii nie je dĺžka jedincov v populácii jednotná, ale už od prvého kroku evolúcie je rozdeľovaná podľa výšky a tvaru stromu.

Rovnako variabilnú dĺžku zabezpečujú aj ďalšie dve toolboxy: pri Parite sa používajú stromy generované genFull (min\_=1, max\_=3), kým Santa Fe povoľuje širší rozsah min\_=2, max\_=6, takže oba benchmarky štartujú s populáciou, kde sa veľkosť stromov líši už v momente inicializácie.

Tento spôsob tvorby počiatočnej populácie vytvára priaznivé predpoklady pre efektívne prehľadávanie priestoru riešení, pričom zároveň zvyšuje robustnosť algoritmu voči predčasnej konvergencii, čo je známy problém v kontexte evolučných algoritmov [5][8][11].

Už v základnom nastavení implementácie sú aplikované princípy, ktoré podporujú a umožňujú prácu s jedincami rôznej veľkosti. Úloha gp.PrimitiveTree spolu s generátormi stromov a náhodnými funkciami zabezpečuje, že genetický program je schopný operovať so stromami s premenlivým počtom uzlov. Tieto mechanizmy sú neskôr obohatené o operátory kríženia, mutácie a kontrolu rastu.

Jedinec v implementácií

Ako bolo spomenuté, jednou z vlastností genetického programovania (GP) je schopnosť pracovať s jedincami, ktorých reprezentácia má dynamickú, teda premennú dĺžku. Tento aspekt priamo súvisí s používaním stromových štruktúr ako dátového modelu pre jedinca, čím sa GP odlišuje od väčšiny ostatných evolučných algoritmov (napr. GA, DE), kde je dĺžka chromozómu zväčša fixná [2][3][15]. Táto sekcia sa zameriava na to, ako je v implementácii tento princíp uplatnený – ako je dĺžka jedinca definovaná, aké štruktúry ju podporujú a ako sa v priebehu evolúcie mení.

Základná reprezentácia jedinca je v deap-e implementovaná prostredníctvom triedy gp.PrimitiveTree, ktorá je podmnožinou objektu list, obohatenou o podporu syntaktických stromov [39]:

creator.create('IndividualKOZA',gp.PrimitiveTree,fitness=creator.FitnessMin)

Tu je IndividualKOZA definovaný ako prípad špecifickej stromovej štruktúry, ktorá je zároveň evolučným objektom (obsahuje atribút fitness).

Štruktúra PrimitiveTree sa ukladaním uzlov v prefixovej notácii podobá výrazovým stromom, ktoré reprezentujú výrazy v programovacích jazykoch. Každý uzol v takomto strome môže byť buď primitív, teda funkcia ako napríklad +, \*, protectedDiv, alebo terminál, čiže vstupná premenná (napríklad x, b0...b5, step) alebo konštanta.

Počet všetkých uzlov, získaný cez vstavanú funkciu len(ind), reprezentuje dĺžku jedinca. Táto dĺžka má priamy vzťah k zložitosti programu, ktorý daný jedinec reprezentuje [15][38][39].

Pre ilustráciu: jedinec so štruktúrou (+ x (\* x x)) obsahuje 5 uzlov (+, x, \*, x, x) a teda len(ind) == 5. Objekt PrimitiveTree po každom zásahu kríženia či mutácie okamžite aktualizuje svoj vnútorný zoznam uzlov, takže volanie len(ind) vždy odráža aktuálny stav stromu bez potreby ďalších operácií.

Variabilná dĺžka je v implementácii zabezpečená už pri tvorbe počiatočnej populácie. Použitím rôznych stratégií generovania stromov (napr. genHalfAndHalf, genFull) sa explicitne umožňuje, aby stromy mali rôznu výšku aj tvar [39]:

toolbox\_koza.register('expr', gp.genHalfAndHalf, pset=pset\_sr, min\_=1, max\_=3)

toolbox\_koza.register('individual',tools.initIterate,creator.IndividualKOZA,toolbox\_koza.expr)

Metóda genHalfAndHalf striedavo vytvára plné stromy, v ktorých všetky vetvy majú rovnakú dĺžku a rastové stromy (rôzne vetvy majú rôznu hĺbku).

V kombinácii s parametrami min\_ a max\_ sa zabezpečuje získanie rozdielnych štruktúr, pričom výsledné stromy sa líšia nielen výškou, ale aj celkovým počtom uzlov, čo vedie k variabilite dĺžky jedincov v počiatočnej populácii.

Táto implementácia sa založená na „filozofií“ GP, podľa ktorej nie je žiaduce obmedzovať veľkosť riešenia na jednu hodnotu vopred. Výrazný prínos tohto prístupu je najmä v prípadoch, keď optimálna dĺžka riešenia nie je známa – necháva sa na samotný evolučný proces, aby v priebehu času našiel rovnováhu medzi zložitosťou a účinnosťou riešenia [3][17].

To je obzvlášť dôležité pri použití symbolických operátorov, kde krátke jedince často nedokážu vyjadriť potrebnú zložitosť riešenia, zatiaľ čo priveľké stromy môžu viesť k bloatu [17][38].

V implementácii je dĺžka jedinca priamo prístupná cez vstavanú funkciu len(). Implementácia tak zohľadňuje dĺžku nie len ako pasívny atribút jedinca, ale aktívne s ňou pracuje – umožňuje jej vznik, sleduje jej vývoj a v niektorých prípadoch ju dokonca penalizuje.

Každý z troch benchmarkov používa vlastnú množinu funkcií, ktoré ovplyvňujú charakteristiku stromov a teda aj rozsah ich dĺžok:

Tabuľka 3 Charakteristiky stromov v použitých benchmarkoch pri GP

|  |  |  |
| --- | --- | --- |
| **Benchmark** | **Typ operácií** | **Typické dĺžky jedincov** |
| **Koza benchmark** | Aritmetické operácie (+, \*, protectedDiv) | Stredne veľké až väčšie stromy, rast s presnosťou modelu |
| **Parity (6-bit)** | Logické operácie (xor, and, or) | Skôr plytšie stromy, často krátke jedince |
| **Santa Fe** | Vetva (ifThen), unárne funkcie | Stromy majú tendenciu rýchlo rásť – penalizácia dĺžky nutná |

Tieto rozdiely sú dané najmä tým, aké funkcie a terminály sa používajú, a aká komplexita je potrebná pre adekvátne riešenie problému. Zatiaľ čo logické operácie v prípade parity často vystačia s menšími stromami, pri Santa Fe simulácii pohybu je potrebné riešiť viac stavových rozhodnutí, čím sa dĺžka programov zvyšuje.

V implementácii genetického programovania je jedinec reprezentovaný ako strom PrimitiveTree, ktorého dĺžka – definovaná ako počet uzlov – je dynamická. Táto dĺžka sa vytvára a mení počas inicializácie aj evolúcie, a v niektorých prípadoch sa stáva aj súčasťou selekčného „tlaku“. Premenná dĺžka jedinca je teda neoddeliteľnou súčasťou mechanizmu GP, a jej vhodné riadenie je dôležité pre úspešné hľadanie efektívnych riešení [3][17][38].

Účelové funkcie

V implementácii sú definované tri samostatné účelové funkcie pre jednotlivé benchmarky: Koza (symbolická regresia), Parity problem (6-bitová parita) a Santa Fe Trail. Každá z týchto funkcií je realizovaná ako samostatná funkcia (procedúra) v Pythone a je registrovaná do príslušného toolboxu. Všetky funkcie hodnotia skompilovaný stromový jedinec a vracajú číselnú hodnotu chyby, ktorú evolučný algoritmus minimalizuje.

Funkcia koza(ind) je definovaná nasledovne:

def koza(ind):

func = toolbox\_koza.compile(expr=ind)

err = 0.0

for x in x\_vals:

try:

val = func(x)

err += (val - (x \*\* 4 + x \*\* 3 + x \*\* 2 + x)) \*\* 2

except ZeroDivisionError:

return float('inf'),

return err,

Jedinec ind je najskôr transformovaný na funkciu pomocou gp.compile, pričom vstupom funkcie je reálna hodnota x. V rámci cyklu sa jedinec testuje na preddefinovanom rozsahu hodnôt x\_vals – v tomto prípade ide o 20 bodov v intervale ⟨−1, 1⟩. Výstup jedinca sa porovnáva s cieľovou funkciou, ktorá je definovaná nasledovne:

Počíta sa pritom súčet kvadratických chýb. V prípade, že dôjde k výnimke (napr. delenie nulou), vráti sa nekonečná chyba ako penalizácia [3][15].

Účelová funkcia parity(ind) sa následne definovaná takto:

def parity(ind):

func = toolbox\_par.compile(expr=ind)

errs = 0

for i in range(64):

bits = [(i >> j) & 1 for j in range(6)]

if func(\*bits) != (sum(bits) % 2):

errs += 1

return errs,

Hodnotenie prebieha testovaním všetkých 64 možných kombinácií šesťbitového vstupu. Každá kombinácia je rozdelená na zoznam jednotlivých bitov, ktoré sa odovzdajú ako vstupy do funkcie reprezentovanej jedincom. Výstup sa následne porovnáva s očakávanou hodnotou parity (súčet bitov modulo 2). Za každú nesprávnu odpoveď sa zvýši počet chýb. Funkcia vracia celkový počet chýb ako výslednú hodnotu [2][5][15].

Najkomplexnejšou účelovou funkciou v kóde je santafe(ind), ktorá simuluje autonómneho agenta (mravca) pohybujúceho sa po 2D mape s cieľom pozbierať čo najviac potravy:

def santafe(ind):

func = toolbox\_sf.compile(expr=ind)

x, y, d = START\_X, START\_Y, 0

food = 0

visited = world.copy()

mask = lambda xx, yy: (yy % H) \* W + (xx % W)

for step in range(400):

idx = mask(x, y)

if visited[idx] == 1:

food += 1

visited[idx] = 0

action = func(step) % 3

if action == 0:

dx, dy = [(0, 1), (1, 0), (0, -1), (-1, 0)][d]

x += dx

y += dy

elif action == 1:

d = (d + 1) % 4

else:

d = (d - 1) % 4

penalty = SF\_PEN\_LAM \* len(ind)

return -food + penalty,

Po inicializácii pozície a smeru pohybu sa simuluje 400 krokov, v ktorých sa mravec rozhoduje, akú akciu vykoná: pohyb vpred, otočenie doprava alebo doľava. Výstup skompilovanej funkcie func(step) určuje akciu v danom kroku. Ak je na súradnici potravina, počet zozbieraných položiek sa zvýši.

Dôležitou súčasťou účelovej funkcie je penalizácia dĺžky jedinca, ktorá sa vypočíta ako súčin SF\_PEN\_LAM \* len(ind). Tento príspevok je k výsledku pripočítaný a zabezpečuje, že sa zvýhodňujú kompaktnejšie riešenia. Tento spôsob kombinácie výkonnostného hodnotenia a penalizácie zložitosti je v literatúre popísaný ako efektívna stratégia boja proti tzv. bloatu [17], [38].

Treba zdôrazniť, že tento tlak pôsobí iba v bench­marku Santa Fe; v úlohách Koza a Parity sa dĺžka na hodnotení účelovej fukncie priamo nepodieľa, takže rast stromov tam obmedzuje len tvrdý výškový limit.

Každá účelová funkcia je zaregistrovaná do príslušného toolboxu takto:

toolbox\_koza.register('evaluate', koza)

toolbox\_par.register('evaluate', parity)

toolbox\_sf.register('evaluate', santafe)

To umožňuje univerzálnu prácu s rôznymi funkciami v rámci jednotného evolučného cyklu (runGP). Výstupom každej funkcie je n-ticová hodnota (tu 1-prvková), ktorá je priradená do atribútu fitness.values jedinca.

V implementácii sú účelové funkcie navrhnuté tak, aby korešpondovali s charakterom jednotlivých problémov a zároveň zabezpečovali robustné hodnotenie výkonnosti jedincov. Pre Koza benchmark sa hodnotí chyba aproximácie, pre paritu počet nesprávnych logických odpovedí a v prípade Santa Fe sa výsledok kombinuje s penalizáciou za dĺžku. Všetky funkcie sú konzistentne implementované s použitím mechanizmu gp.compile a ich výstupy sú kompatibilné s evolučným cyklom knižnice deap [36], čo umožňuje jednoduchú rozšíriteľnosť implementácie aj o ďalšie typy problémov.

Operátory ovplyvňujúce dĺžku jedinca

V tejto časti sa práca venuje analýze evolučných operátorov, ktoré v procese genetického programovania (GP) priamo ovplyvňujú dĺžku jedincov. V prípade stromovej reprezentácie – ako ju definuje gp.PrimitiveTree v knižnici DEAP – sa genetické operácie (najmä kríženie a mutácia) neobmedzujú len na hodnoty génov (uzlov stromu), ale môžu meniť topológiu a tým aj veľkosť stromu. Práve tieto operácie sú hlavným zdrojom variability dĺžky jedinca počas evolúcie, a ich správna implementácia je rozhodujúca pre kvalitu aj stabilitu výsledného riešenia [3][17][38].

Operátor kríženia je vo všetkých troch benchmarkoch implementovaný nasledovne [39]:

toolbox\_koza.register('mate', gp.cxOnePoint)

toolbox\_par.register('mate', gp.cxOnePoint)

toolbox\_sf.register('mate', gp.cxOnePoint)

Funkcia gp.cxOnePoint vykonáva jednobodové kríženie stromov, kde sa náhodne vyberie jeden uzol („podstrom“) v každom z dvoch rodičov, a tieto „podstromy“ sa následne vzájomne vymenia. Výsledkom sú dva nové jedince, ktorých dĺžky sa spravidla líšia od dĺžok pôvodných rodičov, nakoľko výmenou dochádza k nahradeniu časti stromu stromom inej veľkosti.

Príklad: Ak rodič A má 10 uzlov a rodič B má 20 uzlov, výmenou môže vzniknúť potomok s 12 a druhý s 18 uzlami.

Tento typ operácie teda prirodzene vedie k náhodnej zmene dĺžky jedincov, čím zvyšuje diverzitu populácie a rozširuje vyhľadávací priestor GP [3][15].

Aby sa zabránilo nekontrolovateľnému rastu stromov – čo je v literatúre známe ako bloat [17] – je každý operátor kríženia doplnený o tzv. „dekoratér“ statického obmedzenia výšky stromu [39]:

toolbox\_koza.decorate('mate', gp.staticLimit(key=operator.attrgetter('height'), max\_value=MAX\_TREE\_HEIGHT))

Dekorátor gp.staticLimit zabezpečí, že po aplikácii kríženia bude nový strom akceptovaný iba vtedy, ak jeho výška (t. j. maximálny počet úrovní v strome) neprekročí hodnotu definovanú v MAX\_TREE\_HEIGHT (v tomto prípade 25). Tento spôsob predstavuje takzvané tvrdé obmedzenie (hard limit) a patrí medzi štandardné nástroje bloat kontroly v GP [17][38][39].

Mutácia je rovnako registrovaná pre každý benchmark, napríklad v prípade Koza benchmark nasledovne [39]:

toolbox\_koza.register('expr\_mut', gp.genFull, min\_=0, max\_=2)

toolbox\_koza.register('mutate',gp.mutUniform,expr=toolbox\_koza.expr\_mut, pset=pset\_sr)

Použitý operátor gp.mutUniform vyberie náhodný uzol stromu a nahradí ho novým náhodne generovaným potomkom, pričom nový potomok sa vytvára pomocou funkcie expr\_mut (v tomto prípade genFull). Keďže nový potomok môže byť menší alebo väčší ako nahradzovaný uzol, výsledná dĺžka jedinca sa opäť mení dynamicky – mutácia tak môže jedinca skrátiť alebo predĺžiť.

Rovnako ako pri krížení, aj pri mutácií je použitý „dekoratér“ na obmedzenie výšky [39]:

toolbox\_koza.decorate('mutate',gp.staticLimit(key=operator.attrgette('height'), max\_value=MAX\_TREE\_HEIGHT))

Tento krok zabraňuje generovaniu príliš hlbokých stromov, ktoré by mohli negatívne ovplyvniť výpočtový čas a interpretovateľnosť výsledného riešenia.

Aj keď obmedzenie výšky kontroluje vertikálny rozmer stromu, neobmedzuje priamo počet uzlov – teda dĺžku. Preto v praxi môže dochádzať aj pri dodržaní výškového limitu k rastu dĺžky (napr. pri vetvení v nízkych úrovniach stromu).

Operátory cxOnePoint a mutUniform majú zásadný vplyv na schopnosť algoritmu efektívne prehľadávať priestor riešení. Skutočnosť, že nemajú vopred preferované veľkosti „podstromov“ – umožňuje prirodzený výber takých štruktúr, ktoré sú adaptívne vhodné pre riešenie danej úlohy.

Z evolučného hľadiska to znamená, že dĺžka jedinca je výsledkom súťaže medzi selekciou (fitness) a operátormi (náhodné zmeny štruktúry). Preto ak nie sú zavedené kontrolné mechanizmy (napr. výškové limity, penalizácia dĺžky), môžu sa v populácii objaviť nadmerne dlhé stromy s nízkou funkčnou hodnotou – bloat [17].

Implementácia poskytuje možnosť monitorovať dĺžku najlepšieho jedinca v čase, a to pomocou výpisu v konzole:

print(f'{name} Evals {evals}: Best = {best:.6f}, Len = {len(hof[0])}')

Táto výstupná informácia sa periodicky zaznamenáva počas behu hlavného evolučného cyklu (funkcia runGP) a umožňuje sledovať priebeh nárastu (alebo stabilizácie) dĺžky jedincov v populácii. Takýto monitoring je kľúčový pri ladení parametrov na kontrolovanie a zabránenie nadmerného nárastu riešení a výbere vhodných operátorov [17][38][39].

Dĺžka jedinca v genetickom programovaní je priamo ovplyvňovaná evolučnými operátormi – najmä krížením (gp.cxOnePoint) a mutáciou (gp.mutUniform). Tieto operácie umožňujú náhodnú modifikáciu stromovej štruktúry, čím dochádza k zmene počtu uzlov v strome. Aby bol tento rast udržaný pod kontrolou, implementácia využíva pevné obmedzenie výšky stromu (gp.staticLimit). Výsledkom je flexibilná, no zároveň regulovaná dynamika dĺžky jedincov počas evolúcie [3][17][38][39].

Funkcia evolučného cyklu

V tejto časti je popísaný hlavný evolučný cyklus implementovaný v rámci funkcie runGP, ktorá zabezpečuje priebeh genetického programovania (GP) – od inicializácie populácie cez aplikáciu evolučných operátorov až po selekciu a zber najlepších jedincov. Z pohľadu tejto práce je osobitne dôležité sledovať, ako sa v rámci tejto slučky uplatňuje mechanizmus variabilnej dĺžky jedinca, a akým spôsobom sa táto dĺžka vyvíja počas optimalizácie.

Implementácia hlavného optimalizačného cyklu je sústredená vo funkcii runGP, ktorá prijíma ako vstup príslušný toolbox (pre daný benchmark) a reťazec s menom benchmarku na výpis informácií [39]:

def runGP(toolbox, name):

pop = toolbox.population(n=POP\_SIZE)

hof = tools.HallOfFame(1)

evals = 0

best\_hist = []

gen = 0

Funkcia začína inicializáciou populácie, pamäte pre „sieň slávy“ (HallOfFame – skratka hof), a premennej evals, ktorá slúži na sledovanie celkového počtu vyhodnotených jedincov. Parametre POP\_SIZE a N\_EVALS sú globálne konfiguračné hodnoty určujúce veľkosť populácie a maximálny počet vyhodnotení.

Jadro evolučného procesu sa skladá z volania metódy algorithms.varAnd, ktorá kombinuje kríženie (mate) a mutáciu (mutate) s danými pravdepodobnosťami [39]:

offspring = algorithms.varAnd(pop, toolbox, cxpb=0.9, mutpb=0.1)

Práve v tomto kroku dochádza ku zmene štruktúry a dĺžky jedincov. Keďže obidva operátory (kríženie aj mutácia) môžu zmeniť počet uzlov v strome, výsledný potomkovia majú prirodzene rôzne dĺžky, čím sa obnovuje diverzita populácie.

Keďže algorithms.varAnd v jednom volaní vykoná obidva operátory (mate, mutate), nová dĺžka stromov vzniká priamo v tomto bloku a je zapísaná ešte pred výpočtom novej hodnoty účelovej funkcie.

Bezprostredne po aplikácii operátorov sa generujú nové hodnoty účelovej funkcie [39]:

fits = list(map(toolbox.evaluate, offspring))

for ind, fit in zip(offspring, fits):

ind.fitness.values = fit

evals += len(offspring)

Tento krok zároveň zvyšuje počet vyhodnotení, čím sa riadi ukončenie hlavného cyklu.

Nasleduje výber novej populácie pre ďalšiu generáciu [39]:

pop = toolbox.select(offspring, k=POP\_SIZE)

hof.update(pop)

Použitý výberový mechanizmus (selTournament) vyberá najvhodnejších jedincov z potomkov do ďalšej generácie. Okrem toho sa aktualizuje objekt HallOfFame, ktorý uchováva najlepšieho jedinca dosiahnutého v priebehu celej evolúcie. Tento jedinec je následne použitý na výpis štatistík [39]:

best = hof[0].fitness.values[0]

best\_hist.append(best)

if gen % 10 == 0:

print(f'{name} Evals {evals}: Best = {best:.6f}, Len = {len(hof[0])}')

Výpis každých 10 generácií (respektíve 1000 ohodnotení účelovej funkcie) obsahuje nielen aktuálne najlepšie skóre (Best), ale aj jeho dĺžku (Len), čo poskytuje dôležitý náhľad na vývoj zložitosti riešenia v čase. Táto informácia je nutná pre správne ladenie parametrov na kontrolu bloatu, ako aj pri analýze kompromisu medzi výkonnosťou a parsimóniou programu [17][38][39].

Po ukončení hlavného cyklu, t. j. po dosiahnutí stanoveného počtu vyhodnotení, funkcia vráti najlepší nájdený jedinec spolu s historickým záznamom jeho vývoja fitnes hodnoty:

return hof[0], best\_hist

V evolučnom cykle sa nachádza najdôležitejšia časť pre implementáciu GP. V jej priebehu dochádza k opakovanej aplikácii operátorov, ktoré menia štruktúru jedincov a tým dynamicky ovplyvňujú ich dĺžku. Implementácia zároveň poskytuje nástroje na sledovanie a vyhodnocovanie tejto dĺžky v čase, čím podporuje analýzu bloatu a efektivitu riešení [3][17][38][39].

Zhrnutie použitia variabilnej dĺžky jedinca v implementácií

Dokumentovaná implementácia genetického programovania je jednou z už existujúcich evolučných výpočetných techník s variabilnou dĺžkou jedinca, v ktorej sú jedinci reprezentovaní ako stromy. Táto vlastnosť je typická pre štandardné GP [3][15] a jedná sa teda o odlišnosť oproti evolučným algoritmom pracujúcim s fixnými dĺžkami genotypov, ako napr. klasický genetický algoritmus [1][5][8].

Dĺžka jedinca sa v tejto implementácii netvorí staticky, ale vzniká ako prirodzený dôsledok nasledovných procesov:

* inicializácia populácie pomocou genHalfAndHalf alebo genFull generuje rôzne typy stromov s výškou v definovanom rozsahu, čím už počiatočná populácia obsahuje jedincov s rôznou dĺžkou;
* genetické operácie (kríženie, mutácia) následne dynamicky menia štruktúru stromov, čo vedie k zmenám dĺžky – niekedy výrazným, inokedy minimálnym;
* selekcia ponecháva v populácii tých jedincov, ktorí dosahujú najlepší kompromis medzi výkonnosťou a dĺžkou (najmä ak je dĺžka penalizovaná).

Týmto spôsobom je zabezpečená evolúcia aj štruktúrnych vlastností riešenia – nie len jeho parametrov alebo hodnôt.

Aby variabilná dĺžka jedinca neprerástla do neefektívnych zložitostí, implementácia využíva dva typy kontrolných mechanizmov:

* tvrdé obmedzenie výšky stromu pomocou gp.staticLimit, ktoré zaručuje, že žiadny jedinec nemôže prekročiť maximálnu výšku definovanú v konštante MAX\_TREE\_HEIGHT;
* mäkká penalizácia dĺžky použitá v benchmarku Santa Fe, kde dĺžka jedinca priamo ovplyvňuje hodnotu účelovej funkcie cez lineárny penalizačný člen.

Táto kombinácia obmedzení umožňuje riadiť evolúciu tak, aby vznikali riešenia s primeranou zložitosťou, čím sa podporuje ich generalizačná schopnosť, výpočtová efektivita a interpretovateľnosť [17][38][39].

Počas evolúcie sú v pravidelných intervaloch (každých 10 generácií) do výpisu zaznamenávané informácie o aktuálnom najlepšom jedincovi, vrátane jeho dĺžky (len(hof[0])). Takýto výpis poskytuje okamžitú spätnú väzbu o vývoji zložitosti riešení, čo je dôležité z praktického aj výskumného hľadiska – napr. pri porovnávaní rôznych variant operátorov, parametrov, alebo úprav penalizácie.

Použitá stromová reprezentácia umožňuje prirodzenú zmenu veľkosti jedinca počas evolúcie, pričom dĺžka nie je fixná, ale dynamicky sa prispôsobuje. Kombináciou inicializačných stratégií, genetických operátorov a dvoch vrstiev kontroly je zabezpečené, že rast dĺžky zostáva pod kontrolou, bez toho, aby bola obmedzená evolučná flexibilita algoritmu [3][17][38].

Implementácia gramatickej evolúcie

Gramatická evolúcia (GE) sa v tejto časti využíva ako alternatíva k stromovému genetickému programovaniu popísanému v predošlej podkapitole. Namiesto manipulácie s uzlami stromu pracuje GE s lineárnym chromozómom – zoznamom tzv. kódonov (8-bitových celých čísel), ktorý sa počas dekódovania rozvíja podľa formálnej gramatiky BNF na syntakticky správny program [10]. Už z princípu ide o reprezentáciu s premennou dĺžkou – počet kódonov sa môže voľne meniť a nie je priamo viazaný na počet derivácií v gramatike. Flexibilitu však sprevádza riziko nekontrolovaného rastu, a preto skript kombinuje tvrdé limity dĺžky s mäkkou penalizáciou, čo tvorí hlavnú tému nasledujúceho popisu.

Hneď na začiatku sú deklarované globálne parametre, ktoré jednoznačne ohraničujú variabilitu:

MIN\_LEN = 3

MAX\_LEN = 100

CODON\_MAX = 255

Konštanty MIN\_LEN a MAX\_LEN sú tvrdými hranicami, to znamená, že žiadny jedinec nesmie kódonov ubrať pod tri ani presiahnuť sto. Každý kódon sa neskôr interpretuje „modulo počet alternatív“ daného neterminálu, a tým mapuje genotyp na konkrétnu produkciu gramatiky. Práve cyklické čítanie kódonov (codon\_ix % len(genome) v dekóderi) spôsobuje, že aj krátky genóm môže generovať dlhý fenotyp a naopak, čo je pre GE typické [10][40].

Chromozómy sa inicializujú funkciou:

def genInd():

return [random.randint(0, CODON\_MAX)

for \_ in range(random.randint(MIN\_LEN, MAX\_LEN))]

Dĺžka vzniká volaním random.randint(MIN\_LEN, MAX\_LEN), takže už počiatočná populácia vykazuje široké rozdelenie veľkostí a preto je diverzita prítomná od prvého hodnotenia [1]. Dekódovanie zabezpečuje procedúra decodeGenome, ktorá opakovane rozvíja neterminál najviac vľavo podľa hodnoty aktuálneho kódonu a gramatiky uložených v slovníku grammars. Ukážka položky pre 6-bit paritu:

'Parity': (

{'<expr>': [

['(', '<expr>', '^', '<expr>', ')'],

['(', '<expr>', '&', '<expr>', ')'],

['(', '<expr>', '|', '<expr>', ')'],

['~', '<expr>'],

['b0'], ['b1'], ['b2'], ['b3'], ['b4'], ['b5']

]},

'<expr>',

lambda tok: (''.join(tok), 'bool')

),

Tento slovník oddelením genetickej vrstvy (kódony) od syntaktickej (BNF) umožňuje vymeniť úlohu jednoduchou zmenou gramatiky bez zásahu do zvyšku algoritmu [42]. Variabilná dĺžka chromozómu sa však musí regulovať – okrem vyššie spomenutých tvrdých hraníc sa do každej účelovej funkcie vkladá penalizácia vytvorená volaním mkPenalizer(d\_target, lam). Táto funkcia pridáva k hodnoteniu účelovej funkcie lineárny príspevok úmerný odchýlke dĺžky od cieľovej hodnoty, pričom toleruje malú „mŕtvu zónu“ ±2 kódony. Regulácia tak kombinuje rázne zrezanie extrémov s jemným tlakom na optimálnu veľkosť, čo sa osvedčilo pri potláčaní bloatu v GE experimentoch [17][38].

Stručne povedané, v tejto implementácii funguje variabilná dĺžka nasledovne: náhodne sa vytvorí v počiatočnej populácii, mení sa v operátoroch (pridanie / odobratie kódonu v mutácii, kombinácia počiatočného a koncového segmentu pri krížení) a sleduje sa počas evolúcie. Kvôli pevne stanoveným limitom a penalizácii zostáva rast dĺžky pod kontrolou, zároveň však ponecháva algoritmu dostatok slobody preskúmať riešenia rôznej zložitosti [12].

Reprezentácia genómu a dekódovanie

Jadrom gramatickej evolúcie v kóde je lineárny chromozóm – zoznam kódonov typu int, ktorého dĺžka je dynamická a kontrolovaná iba hranicami MIN\_LEN a MAX\_LEN. Každý kódon nadobúda hodnotu v rozsahu <0, CODON\_MAX>; v implementácii je CODON\_MAX = 255, takže ide o 8-bitový priestor, ktorý poskytuje dostatočnú rozlišovaciu schopnosť a umožňuje jednoduchú prácu s operátormi typu kde sa náhodne mení jeden byte [10].

Volanie randint(MIN\_LEN, MAX\_LEN) zabezpečuje, že už počiatočná populácia obsahuje chromozómy rozdielnych dĺžok, čo vytvára rozdiel v populácií bez potreby ďalších zásahov. V priebehu evolúcie sa dĺžka genómu mení dvoma cestami, v jednej ceste môže mutácia pridať alebo odstrániť kódon, v druhej kríženie kombinuje počiatočnú časť jedného a koncovú časť druhého rodiča. Oba operátory sa však riadia rovnakými hranicami, takže dĺžka zostáva v povolenom intervale.

Dekódovanie genómu na fenotyp zabezpečuje funkcia decodeGenome:

def decodeGenome(genome, grammar, start\_symbol, max\_expansions=1000):

output = [start\_symbol]

codon\_ix = 0

expansions = 0

while expansions < max\_expansions and any(sym in grammar for sym in output):

for i, sym in enumerate(output):

if sym in grammar:

prods = grammar[sym]

cod = genome[codon\_ix % len(genome)]

choice = prods[cod % len(prods)]

output = output[:i] + choice + output[i+1:]

codon\_ix += 1

expansions += 1

break

final = []

for sym in output:

if sym in grammar:

for prod in grammar[sym]:

if not any(s in grammar for s in prod):

final.extend(prod)

break

else:

final.append(sym)

return final

Mechanizmus dekódovania využíva dva po sebe nasledujúce operátory „modulo“, ktoré zásadne ovplyvňujú prácu s dĺžkou chromozómu. Najprv sa uplatní cyklenie kódonov: index codon\_ix % len(genome) spôsobí, že po prečítaní posledného kódonu sa ukazovateľ vráti na začiatok zoznamu. Vďaka tomu dokáže aj veľmi krátky genóm rozvinúť ľubovoľne dlhú deriváciu, zatiaľ čo dlhší chromozóm ponúka bohatší výber pravidiel. Druhá modulo operácia ‒ výraz codon % len(prods) ‒ mapuje aktuálny kódon na konkrétnu produkciu v zozname alternatív daného neterminálu. Ak má napríklad symbol <expr> štyri alternatívy, kódon s hodnotou 17 vyberie produkciu s indexom 17 mod 4 = 1. Aby sa predišlo nekonečnému rozvíjaniu, dekóder sleduje aj maximálny počet rozšírení. Po prekročení limitu max\_expansions = 1000 ukončí deriváciu a doplní zvyšné neterminály najkratšími terminálnymi produkciami, čím zabráni veľkému rozširovaniu odvozených riešení pri nevhodných alebo príliš dlhých genómoch [42].

Po skončení hlavného cyklu zostanú v zozname output terminálne symboly alebo neterminály, pre ktoré už neexistuje ďalšia rozšíriteľná produkcia. Implementácia preto vstúpi do doplňovacej fázy, kde sa zvyšné neterminály nahradia prvou čisto terminálnou produkciou v ich zozname. Výsledkom je sekvencia symbolov, ktorú následne spracuje funkcia post na konkrétnu vykonateľnú podobu – lambda výraz pre Parity alebo zoznam číselných akcií pre Santa Fe.

V praxi teda na variabilnej dĺžke genómu závisí:

1. rozsah vyhľadávacieho priestoru – dlhší chromozóm prináša viac kódonov, t. j. viac možností voľby produkcií a vyššiu syntaktickú „jemnosť“ odvodenia, hoci za cenu väčšieho rozmeru genotypu;
2. Riziko redundancie – cyklenie znamená, že nadmerne dlhý genóm môže obsahovať nevyužité koncové úseky, ktoré nezmenia fenotyp. Tieto úseky síce znižujú účinnosť evolúcie, ale môžu chrániť užitočné časti pred deštrukčnými mutáciami [38];
3. interakciu s penalizátorom – funkcia mkPenalizer pridáva lineárny príspevok k fitnes, ak sa dĺžka odchýli od cieľa. Tlak na dĺžku je preto adaptívny. Pri Parity je d\_target = 64, čím sa podporuje bohatšia logická štruktúra, zatiaľ čo pri Santa Fe je cieľ nižší a preferuje sa kompaktnejší zoznam akcií.

Takéto nastavenie dekodéra napĺňa požiadavku, aby algoritmus flexibilne vytváral aj riadil variabilnú dĺžku. Krátke chromozómy sú životaschopné, dlhé sú selekčne penalizované len vtedy, keď ich nadmerná veľkosť neprináša prínos v ohodnotení.

Inicializácia populácie

Na vytvorenie počiatočnej populácie slúži dvojica funkcií genInd() a initPop(). Jedinec vzniká vo funkcií genInd() tak, že sa najprv náhodne zvolí dĺžka chromozómu veličinou random.randint(MIN\_LEN, MAX\_LEN) a potom sa každý jeho prvok naplní náhodným kódonom z intervalu 0 … CODON\_MAX.

Tento jednoduchý postup zabezpečí, že už v generácii 0 sú v populácii zastúpené chromozómy s celým rozsahom dĺžok od troch do stovky kódonov, čo výrazne podporuje rozdielnosť vyhľadávacieho priestoru [1]. Funkcia initPop() potom len opakuje volanie genInd() dovtedy, kým nedosiahne predpísanú veľkosť populácie POP\_SIZE:

def initPop():

return [genInd() for \_ in range(POP\_SIZE)]

Náhodné rozdelenie dĺžok je dôležité z dvoch dôvodov. Po prvé, krátke chromozómy reprezentujú kompaktné riešenia, ktoré sa rýchlo dekódujú a môžu slúžiť ako stavebný materiál pre dlhšie programy. Po druhé, dlhšie chromozómy prinášajú viac kódonov, a teda jemnejšie „riadkovanie“ pri výbere produkcií, čo zvyšuje syntaktickú variabilitu fenotypu [40]. V kombinácii s tvrdými hranicami MIN\_LEN a MAX\_LEN tak inicializácia nastavuje prirodzenú rovnováhu medzi jednoduchosťou a exploratívnym potenciálom populácie bez potreby zložitejšieho heuristického riadenia.

Premenná dĺžka jedincov sa od prvého kroku stáva atribútom, ktorý sa môže evolúciou vylepšovať. Následné operátory kríženia a mutácie ju môžu ďalej skracovať či predlžovať, no nikdy nemôžu prekročiť limity. Vďaka tomu môže algoritmus hneď na začiatku ťažiť zo širokého spektra veľkostí genotypov a následne adaptívne smerovať dĺžky tam, kde to vyžaduje účelová funkcia alebo funkcia penalizácie dĺžky.

Účelové funkcie

Účelové funkcie v skripte vznikajú volaním funkcie makeEvaluator(), ktorá pre každý benchmark zostaví anonymnú funkciu uzatvárajúcu prístup k zvolenej penalizácií dĺžky a k príslušnej gramatike. Penalizácia sa generuje pomocou mkPenalizer(d\_target, lam, deadzone). Logika je priamočiara. Ak sa aktuálna dĺžka chromozómu odchýli o viac než dva kódony od cieľovej hodnoty d\_target, pripočíta sa ku hodnoteniu účelovej funkcie lineárna penalizácie lam × |len(genome) − d\_target|. V opačnom prípade je príspevok nulový. Týmto spôsobom sa na dĺžku pôsobí mäkkým tlakom a nie absolútnym zákazom, takže evolúcia môže prechodne pracovať aj s dlhšími či kratšími jedincami, ak to prinesie výhodu v hlavnom kritériu [17].

def mkPenalizer(d\_target, lam=10, deadzone=2):

def \_pen(genome):

dev = abs(len(genome) - d\_target)

return 0 if dev <= deadzone else lam \* dev

return \_pen

Pre každý benchmark sa následne vytvorí konkrétny evaluátor. V prípade parity vyzerá jeho kostra takto:

def f(genome):

tokens = decodeGenome(genome, grammar, start)

expr, \_ = post(tokens)

fn = eval('lambda b0,b1,b2,b3,b4,b5: ' + expr, {'\_\_builtins\_\_':{}}, {})

errors = sum(fn(\*[(i>>j)&1 for j in range(6)]) != (bin(i).count("1") & 1)

for i in range(64))

return errors + pen(genome)

Ako prvé sa chromozóm dekóduje na reťazec logického výrazu, potom sa zostaví anonymná funkcia a otestuje na všetkých 64 možných kombináciách bitov. Fitness je súčtom počtu chýb a penalizácie dĺžky, takže dlhší jedinec prežije len vtedy, ak sa počet chýb zrazí dostatočne nízko. Pri benchmarku Santa Fe je princíp rovnaký – dekódovaný program je zoznam akcií, simulácia v prostredí vracia počet nazbieraných potravín, ku ktorému sa opäť pripočíta dĺžková penalizácia. Celý tlak na veľkosť je teda sústredený do jedinej riadenej zložky pen(genome), čo zjednodušuje porovnávanie rôznych cieľových dĺžok: pri Parity je d\_target = 64, Santa Fe pracuje s d\_target = 100, zatiaľ čo pre symbolickú regresiu Koza sa uvažuje hodnota 10.

Kód zároveň dbá na bezpečnosť. Pri konštrukcii lambda výrazov sa vyprázdňuje slovník \_\_builtins\_\_, takže dekódovaný výraz nemá prístup k nežiaducim funkciám Pythonu. V prípade výnimky (chybný výraz, delenie nulou) sa hodnota účelovej funkcie nastaví na float('inf'), čím sa jedinec, ktorý nie je validný okamžite diskvalifikuje.

Takáto architektúra spája dve odlišné ciele do jedného skalárneho kritéria – výkonnosť programu a primeranú dĺžku genómu. Evolúcia sa preto môže voľne rozhodovať, či zvýhodní kratší, ale možno menej presný program, alebo si „zaplatí“ niekoľko kódonov navyše za vyššiu presnosť. V praxi to vedie ku kompromisu medzi kompaktnosťou a funkčnosťou, čo je pri gramatickej evolúcii dôležité, pretože introny a redundancia bývajú časté [10][38].

Evolučné operátory

Evolučné operátory v tejto implementácii priamo manipulujú s lineárnym zoznamom kódonov a tým určujú, ako sa dĺžka chromozómu v priebehu evolúcie mení. Kríženie je realizované funkciou cross(p1, p2), ktorá pri pravdepodobnosti 0,9 vykoná jednobodové spojenie rodičov. Po náhodnom výbere bodov c1 a c2 vzniknú dvaja potomkovia tak, že sa počiatočný úsek prvého rodiča skombinuje s koncovým úsekom druhého a naopak:

c1 = random.randint(1, len(p1)-1)

c2 = random.randint(1, len(p2)-1)

o1 = p1[:c1] + p2[c2:]

o2 = p2[:c2] + p1[c1:]

Výsledná dĺžka potomka je teda súčtom dvoch častí odlišných rodičov. Môže byť kratšia, dlhšia alebo rovnaká ako dĺžky pôvodné. Aby sa vždy splnil interval <MIN\_LEN, MAX\_LEN>, kód okamžite kontroluje oba výstupy: príliš krátky chromozóm doplní náhodnými kódonmi, príliš dlhý oseká nadbytočné prvky. Takáto úprava zachováva genetickú informáciu získanú krížením, no zároveň zabraňuje explozívnemu rastu [10].

Mutácia, implementovaná vo funkcii mutate(genome), pôsobí na dĺžku ešte explicitnejšie. S pravdepodobnosťou p\_len = 0.3 sa rozhodne pre zmenu veľkosti. Ak náhodne bude hodnota pre pridanie a aktuálna dĺžka je menšia než MAX\_LEN, kódon sa pripojí na koniec zoznamu. Ak pripadne ma odobranie a chromozóm je dlhší než MIN\_LEN, jeden prvok sa odstráni. Následne prechádza celý zoznam a každý kódon s pravdepodobnosťou rate = 0.1 nahradí novou náhodnou hodnotou, čím sa mení genetický obsah bez ovplyvnenia dĺžky:

if random.random() < p\_len:

if random.random() < 0.5 and len(g) < MAX\_LEN:

g.append(random.randint(0, CODON\_MAX))

elif len(g) > MIN\_LEN:

del g[random.randrange(len(g))]

Tento dvojkrokový postup má dôležité dôsledky. Po prvé, dĺžka sa môže meniť inkrementálne. Po druhé, pravdepodobnosť pridania a odstránenia je symetrická, takže samotná mutácia nezavádza systematický rast ani skracovanie. Rozhodujúci smer určí selekcia v kombinácii s penalizáciou dĺžky [17].

V praxi sa tieto dve operácie dopĺňajú. Kríženie kombinuje existujúce bloky génov bez zámernej zmeny veľkosti, zatiaľ čo mutácia poskytuje jemné ladenie dĺžky i obsahu. Výsledkom je populácia, v ktorej dĺžka chromozómov kolíše, no vďaka limitom a penalizácii zostáva v rozumnej oblasti. Takáto dynamika umožňuje algoritmu využívať krátke, kompaktné reťazce tam, kde to stačí (napr. jednoduché logické výrazy pri parite), a naopak tolerovať dlhšie reťazce, ak prinášajú vyšší výkon, ako je to časté pri zložitejšej navigácii v úlohe Santa Fe [38].

Regulácia dĺžky

Kontrola rastu chromozómov sa v implementácii opiera o dvojúrovňový mechanizmus: kombináciu tvrdých hraníc MIN\_LEN / MAX\_LEN a penalizácie, ktorá dĺžku zapája priamo do hodnoty účelovej funkcie. Limity pôsobia okamžite pri každej manipulácii s genómom. Po génovej rekombinácii (krížení) alebo mutácii sa výsledný zoznam kódonov testuje. Ak je kratší ako tri prvky, doplní sa novými náhodnými kódonmi, ak presiahne sto, koniec zoznamu sa jednoducho oreže. Algoritmus tak nikdy nepracuje s neplatným chromozómom, čo zjednodušuje všetky následné operácie.

Jemnejšiu úpravu veľkosti zabezpečuje penalizačná funkcia, ktorá sa k hodnoteniu pripočíta až po vyhodnotení hlavnej úlohy. Jej parametre sa nastavujú osobitne pre každý benchmark. Parita má cieľovú dĺžku 64 kódonov, Santa Fe sto, zatiaľ čo Koza benchmark iba desať. V praxi to znamená, že evolúcia môže chromozóm skrátiť pod cieľ, ak tým výrazne zníži chybu, alebo ho predĺžiť, pokiaľ ďalšie kódony prinesú funkčný zisk – penalizácia za dĺžku rastie lineárne až za hranicou dvojkódonovej mŕtvej zóny, takže drobné odchýlky nie sú trestané [17].

Evolučný cyklus

Evolučný cyklus je sústredený vo funkcii runGE, ktorej logika sa odlišuje od prístupu využívaného pri knižnici deap v kapitole o GP. Počet generácií sa určí ako celočíselný podiel N\_EVALS // POP\_SIZE, takže celkový počet vyhodnotení je vždy presne dodržaný a nezávisí od prípadných odchýliek dĺžky chromozómov:

gens = N\_EVALS // POP\_SIZE

Každú generáciu sa najprv vypočíta hodnotenie pre všetkých jedincov v populácii. Kód potom nájde index najlepšej hodnoty best\_idx = argmin(fits). Elitizmus zabezpečí, že tento jedinec sa bez zmien kopíruje do novej populácie (new\_pop = [pop[best\_idx]]). Zvyšok populácie sa plní cyklom, v ktorom:

1. dvaja rodičia sa vyberú turnajom veľkosti 3 (tournamentSelection);
2. na dvojicu sa aplikuje cross() a následne mutate();
3. výsledný potomkovia sa vložia do new\_pop, kým nie je dosiahnutá veľkosť POP\_SIZE.

Pretože už mutácia môže pridávať alebo odoberať kódony a kríženie spája počiatočné a koncové úseky rôznych dĺžok, nová populácia vykazuje široké spektrum veľkostí. Vďaka bezprostrednej kontrole MIN\_LEN / MAX\_LEN je však každý potomok platný a pripravený na dekódovanie v nasledujúcom kole.

Sledovanie vývoja dĺžky prebieha priebežne. Pri každej desiatej generácií sa vypíše:

print(f'{name} Evals{gen\*POP\_SIZE:6d}: Best = {best:.6f}, Len = {len(pop[best\_idx])}')

V celom cykle sa dĺžka genómu mení výlučne prostredníctvom operátorov a následne pôsobí na:

* hodnotu účelovej funkcie (cez penalizáciu), takže selekčný tlak môže preferovať kratšie alebo dlhšie riešenia podľa parametra d\_target;
* elitizmus – ak sa najlepší jedinec stane zbytočne dlhým, stále prežije, no v ďalších generáciách môže byť kombinovaný s kratšími chromozómami a dĺžka populácie sa opäť vyváži.

V tomto evolučnom cykle je dostatočne jemné riadenie bloatu: limity bránia extrémom, penalizácia podporuje parsimóniu a priebežný monitoring poskytuje spätnú väzbu o tom, kam sa dĺžky chromozómov počas optimalizácie vyvíjajú [17][38].

Zhrnutie implementácie

Implementovaný zdrojový kód gramatickej evolúcie (GE) preukazuje, že lineárny chromozóm s premenlivým počtom kódonov dokáže pokryť široký okruh syntaktickej zložitosti pri relatívne jednoduchom kóde. Dĺžka jedinca je tu priamo totožná s počtom kódonov a môže sa plynulo meniť v intervale 3 – 100, pričom jemné prírastky zabezpečuje mutácia a hrubšie preskupenia poskytuje jednobodové kríženie spájajúce počiatočné a koncové úseky rodičov. Tvrdé limity MIN\_LEN/MAX\_LEN zaručujú, že sa populácia nikdy nevychýli mimo preddefinované hranice, zatiaľ čo lineárna penalizácia s malou mŕtvou zónou umožňuje evolúcii dočasne akceptovať väčšie či menšie chromozómy, pokiaľ to zlepšuje hlavný cieľ hodnotenia.

Dekódovanie s cyklením kódonov znamená, že aj krátky genóm môže generovať rozsiahly fenotyp, čím GE znižuje tendenciu k lineárnemu nárastu kódu. Evolučný cyklus s elitizmom je úsporný, no poskytuje všetko podstatné – turnajovú selekciu, výpis dĺžky víťazov každých desať generácií a logovanie finálnych hodnôt do CSV (bude bližšie popisované v kapitole 8).

Celkové správanie algoritmu potvrdzuje, že variabilná dĺžka je riadená tromi navzájom prepojenými mechanizmami: náhodnou inicializáciou, operátormi schopnými inkrementálnych zmien a kombinovanou reguláciou obmedzení. Takto zostavená gramatická evolúcia vyvažuje exploratívny potenciál s parsimóniou, zabraňuje bloatu a pritom zachováva dostatočnú voľnosť na hľadanie presných, syntakticky korektných programov pre odlišné úlohy [17][38][40][42].

Úprava štandardných evolučných algoritmov pre variabilnú dĺžku jedinca

Nasledujúca kapitola nadväzuje na analýzu genetického programovania a gramatickej evolúcie tým, že ukazuje, ako možno princíp variabilnej dĺžky genotypu preniesť aj do trojice „klasických“ populačných algoritmov – genetického algoritmu (GA), diferenciálnej evolúcie (DE) a CMA-ES. V pôvodnej podobe predpokladajú všetky tri metódy fixný počet rozhodovacích premenných. Ich operátory boli preto rozšírené tak, aby mohli jedinca predlžovať alebo skracovať a zároveň rešpektovali bezpečné medze <MIN\_LEN, MAX\_LEN>.

Kapitola postupne rozoberá zvolenú reprezentáciu jedinca, úpravy kríženia, mutácie, penalizačné a elitistické mechanizmy, ktoré držia rast dĺžky pod kontrolou, a dopady týchto zásahov na konvergenciu vo vybraných spojitých aj diskrétnych benchmarkoch. Hlavným cieľom je ukázať, že aj algoritmy pôvodne navrhnuté pre pevnú dimenziu je možné, pri zachovaní ich charakteristických prvkov, adaptovať na prostredia, kde optimálna veľkosť riešenia nie je vopred známa [5][8].

Motivácia pre úpravu dĺžky jedinca

Pri štandardných evolučných algoritmoch, akými sú napríklad genetický algoritmus (GA), diferenciálna evolúcia (DE) a CMA-ES, sa predpokladá fixná dimenzia riešenia: jedinec GA je modelovaný ako konštantný reťazec génov, vektor DE obsahuje vždy n prvkov a v CMA-ES sú stred m a kovariančná matica C definované v nemennom priestore Rn [5]. Takéto nastavenie síce zjednodušuje tvorbu operátorov a analytické odhady konvergencie, no stáva sa obmedzením v prípadoch, keď optimálny počet parametrov nie je vopred známy alebo sa počas učenia prirodzene mení – napríklad pri symbolických modeloch, kde môže jeden dodatočný člen výrazne zlepšiť aproximáciu, kým iný zostáva nadbytočný [1][6].

V tejto kapitole bude ukázané, že schopnosť pracovať s variabilnou dĺžkou jedinca nemusí zostať výsadou stromových metód. Bude predvedené, ako možno do algoritmov pôvodne určených pre fixnú dimenziu vložiť mechanizmy, ktoré umožnia predlžovanie a skracovanie vektora pri zachovaní ich pôvodných princípov. Konkrétne:

* bude predstavený minimálny zásah do reprezentácie, ktorým sa sprístupní pridávanie či odoberanie génov (resp. dimenzií) a ktorým sa zároveň zachová kompatibilita s existujúcimi operátormi;
* budú popísané limity a penalizácia dĺžky, pomocou ktorých sa predíde nekontrolovanému rastu („bloatu“), pričom definícia hlavnej účelovej funkcie zostane nezmenená;
* bude zdôraznené monitorovanie dĺžky ako súčasti experimentov, aby mohla byť vyhodnotená rovnováha medzi presnosťou a komplexnosťou riešení [17].

Takto navrhnuté rozšírenia budú demonštrované pre tri algoritmy – GA, DE a CMA-ES.

Spoločné parametre pre implementácie

Všetky tri upravené algoritmy, GA-V, DE-V a CMA-ES-V, boli implementované v programovacom jazyku Python, v IDE DataSpell so zhodnou sadou knižníc numpy, random, csv, uuid, time a os. Keďže identické verzie týchto modulov a ich konfiguračné detaily už boli podrobne popísané v podkapitole 6.1, opakovaný výklad nebude uvádzaný [27][28][29][30][31][32][33][34][35]. Rovnako boli použité rovnaké globálne premenné na počet ohodnotení, počet opakovaní a veľkosť populácie, aby sa vylúčili vplyvy odlišnej veľkosti populácie či počtu hodnotení na pozorovanú dynamiku dĺžky jedincov [5][6].

Vo všetkých troch kódoch sa používa identická funkcia mkPenalizer(d\_target, λ, deadzone), ktorá pripisuje lineárnu hodnotu („pokutu“), podľa:

Za odchýlku dĺžky mimo tolerancie ± povolenej zóny. Parametre penalizácie boli zvolené tak, aby:

* limity MIN\_LEN / MAX\_LEN zaručili, že každý potomok zostane syntakticky platný;
* bolo umožnené dočasné prekročenie cieľovej dĺžky, pokiaľ to prinesie výraznú úsporu hlavnej chyby hodnotenia [17].

Na rozdiel od algoritmov GP a GE, kde boli vybrané len tri „programové“ úlohy vhodné pre dané reprezentácie, bola v tejto časti použitá plná sada ôsmich benchmarkov (testovacích funkcií). Toto rozšírenie bolo nevyhnutné, pretože GA, DE aj CMA-ES pracujú s reálnymi vektormi prirodzene, a preto sú schopné riešiť aj klasické optimalizačné funkcie definované v priestore Rn [1][4]. Zahrnutím analytických a kombinatorických problémov sa umožnilo:

* preskúmať, ako sa variabilná dĺžka prejaví v kontinuálnom priestore, kde rast dimenzie priamo zvyšuje počet optimalizovaných parametrov;
* porovnať rovnaké mechanizmy dĺžku-meniacej mutácie aj v diskrétnych scenároch, kde dlhší vektor reprezentuje bohatšiu množinu rozhodnutí (Knapsack, Parity, Santa Fe) [2][26].

Z metodologického hľadiska bolo zabezpečené, že všetky algoritmy boli spúšťané na totožnej množine úloh a s rovnakým počtom vyhodnotení, čím sa vytvorili podmienky pre neskoršiu porovnanie ich schopnosti regulovať dĺžku jedinca a zároveň dosahovať nízku chybu účelovej funkcie.

Genetický algoritmus s variabilnou dĺžkou jedinca

V tejto časti je predstavená implementácia genetického algoritmu, ktorý umožňuje dynamickú zmenu dĺžky jedinca počas evolúcie. Popísané budú jeho hlavné komponenty, princíp práce s variabilnou dĺžkou a spôsob, akým je táto vlastnosť riadená v rámci výpočtu. Dôraz sa kladie na technickú realizáciu a konkrétne mechanizmy operátorov ovplyvňujúcich dĺžku jedincov.

Reprezentácia jedinca

Každý jedinec je uchovávaný ako obyčajný zoznam reálnych hodnôt typu float. Počet prvkov |x| sa generuje pri inicializácii volaním:

length = random.randint(MIN\_LEN, MAX\_LEN)

Týmto sa zabezpečí, že dĺžka leží v intervale <3, 100>. Hodnoty génov sú vzorkované z intervalu <-2, 2>. Takáto lineárna reprezentácia umožňuje priame použitie štandardných operácií GA a zároveň podporuje jednoduché pridávanie či odoberanie génov bez zmeny pôvodnej dátovej štruktúry [8].

Operátory meniace dĺžku jedinca

Najdôležitejším miestom, kde sa v GA-V modifikuje veľkosť chromozómu, je funkcia mutate(ind, p\_mut). Jej činnosť prebieha v nasledujúcich krokoch:

1. Príprava kópie – ind = ind[:]. Najskôr sa vytvorí kópia jedinca, aby nedošlo k nezamýšľanému zdieľaniu referencií v populácii.
2. Rozhodnutie o zmene dĺžky – if random.random() < 0.3. S pravdepodobnosťou 0,3 sa aktivuje „dĺžková“ mutácia. Táto hodnota bola zvolená empiricky, keďže literatúra odporúča, aby sa veľkosť genómu menila menej často než jeho obsah [5].
3. Pridanie génu – ind.append(random.uniform(gene\_low, gene\_high)). Pokiaľ je chromozóm kratší ako MAX\_LEN = 100, je na jeho koniec pripojený nový gén so zhodným doménovým rozsahom <gene\_low, gene\_high>. Týmto krokom sa dĺžka zvyšuje o jeden a do vyhľadávacieho priestoru sa dopĺňajú ďalšie dimenzie [2].
4. Odstránenie génu – del ind[random.randrange(len(ind))]. Ak dĺžka presahuje minimálnu hranicu MIN\_LEN = 3, je náhodne vybraný index odstránený. Obidve operácie sú riadené symetrickou pravdepodobnosťou 0,5, takže nastáva nulové zaujatie voči rastu či skracovaniu – výsledné rozdelenie dĺžok je ponechané na selekčný tlak penalizácie [6].
5. Gaussov posun génov – for i in range(len(ind)) a ind[i] += random.gauss(0,0.5). V druhej fáze sa pre každý existujúci gén nezávisle uplatňuje klasická mutácia hodnôt. Ak náhodná hodnota klesne pod p\_mut, k aktuálnej hodnote sa pripočíta realizácia z normálového rozdelenia N(0, 0.52). Takto sa dosahuje lokálny krok prehľadávania, ktorý je odporúčaný pri reálnom kódovaní GA [1][8].
6. Orezanie do domény ind[i] = max(min(ind[i], gene\_high), gene\_low). Hodnoty, ktoré by prekročili interval <gene\_low, gene\_high>, sú okamžite „odrezané“. Touto saturáciou sa zabraňuje posunu mimo akceptovateľné hranice a zároveň sa udržuje homogenita vyhľadávacieho priestoru [5].

def mutate(ind, p\_mut):

ind = ind[:]

if random.random() < 0.3:

if random.random() < 0.5 and len(ind) < MAX\_LEN:

ind.append(random.uniform(gene\_low, gene\_high))

elif len(ind) > MIN\_LEN:

del ind[random.randrange(len(ind))]

for i in range(len(ind)):

if random.random() < p\_mut:

ind[i] += random.gauss(0,0.5)

ind[i] = max(min(ind[i], gene\_high), gene\_low)

return ind

Popísaná stratégia úpravy dĺžky genómu zohľadňuje tri požiadavky, ktoré sú často kladené na algoritmy s premennou dĺžkou jedinca [6]. Prvou z nich je vykonanie zmien, pri ktorej sa dĺžka genómu mení vždy len o jeden gén. Táto stratégia zabezpečuje plynulý prechod v priestore riešení a zároveň minimalizuje riziko náhleho narušenia adaptovaných segmentov genetickej informácie. Druhou požiadavkou je dôsledné rešpektovanie pevných limitov dĺžky, pričom každá zmena je podmienená kontrolou voči vopred definovaným hraniciam MIN\_LEN a MAX\_LEN. Vďaka tomu sa aj pri opakovaných mutáciách predchádza prekročeniu prípustného rozsahu. Tretím aspektom je nepretržitá interakcia s penalizačným mechanizmom, keďže samotné mutácie nemenia dĺžku asymetricky. Smer vývoja štruktúry jedinca tak určujú výlučne selekčné procesy v kombinácii s penalizačnou funkciou mkPenalizer(), ktorá lineárne znižuje vhodnosť riešení s nadmernou dĺžkou. V prípadoch, keď predĺžený jedinec neprináša zodpovedajúce zlepšenie fitnes hodnoty, býva nahradený kompaktnejšou a efektívnejšou alternatívou. Toto priamo reflektuje poznatky týkajúce sa problému bloatu [17].

Kríženie rodičov rôznych dĺžok

Jednobodové kríženie crossover() je implementované ako výmena segmentov:

cut1 = random.randint(1, len(p1)-1)

cut2 = random.randint(1, len(p2)-1)

c1 = p1[:cut1] + p2[cut2:]

c2 = p2[:cut2] + p1[cut1:]

Výsledné potomstvo teda zdedí počiatočný segment jedného a koncový segment druhého rodiča. Ak by nová dĺžka klesla pod MIN\_LEN, zoznam sa doplní náhodnými génmi. Ak by presiahla MAX\_LEN, koncové hodnoty sú odstránené. Tvrdým orezaním sa bráni dosahovaniu extrémnych rozmerov a súčasne zostáva diverzita kombinácií zachovaná.

Evolučný cyklus

Celý optimalizačný proces sa realizuje vo funkcii runGA(). Jeho riadiaca logika bola usporiadaná tak, aby sa variabilná dĺžka stala súčasťou selekcie a aby mohla byť počas behu explicitne monitorovaná.

Po vytvorení počiatočnej populácie initPop() obsahuje každý kandidátny vektor náhodnú dĺžkou z intervalu <MIN\_LEN, MAX\_LEN>. Diverzita vo veľkosti je tak garantovaná od prvého hodnotenia – postup, ktorý je bežne odporúčaný pri GA-variantoch s reálnym kódovaním [2][5].

pop = initPop()

evals = 0

best\_hist = []

Slučka pokračuje, kým počet vyhodnotení evals nedosiahne konštantu N\_EVALS = 100 000. Index generácie sa odvádza z dĺžky histórie najlepších ohodnotení:

gen\_idx = len(best\_hist)

Na jeho základe sa lineárne adaptujú pravdepodobnosti evolučných operátorov ‒ princíp „exploration–exploitation“ sa týmto spôsobom uplatňuje priamo v kóde [1]:

p\_mut = 0.05 \* (1 - gen\_idx/(N\_EVALS/POP\_SIZE))

p\_cross = 0.8 \* (1 - 0.5\*gen\_idx/(N\_EVALS/POP\_SIZE))

Mutácia génov sa znižuje z 5 % na 0 %, čím sa s blížiacim sa koncom behu potláča náhodné rozptyľovanie dĺžok i hodnôt. Kríženie klesá z 0,8 na 0,4. Úplné zastavenie kríženia by síce urýchlilo konvergenciu, no znížilo by šancu korigovať nevhodnú dĺžku vzniknutú v skorších iteráciách [6].

Najlepší jedinec generácie (elite) sa identifikuje podľa fitnesu, ktorý už zahrnuje lineárnu penalizáciu dĺžky vytvorenú volaním mkPenalizer():

elite = min(pop, key=fit\_func)

new\_pop.append(elite)

Hodnota účelovej funkcie vytvára priamy konkurenčný vzťah medzi presnosťou a veľkosťou riešenia, čo predstavuje najjednoduchšiu formu parsimonického tlaku [17]. Bezpodmienečné kopírovanie elity chráni aktuálne najvhodnejšiu dĺžku pred deštruktívnymi mutáciami.

Nová populácia sa zapĺňa, kým nedosiahne POP\_SIZE:

while len(new\_pop) < POP\_SIZE and evals < N\_EVALS:

if selection=='tournament':

p1 = tournamentSelection(pop, fit\_func)

p2 = tournamentSelection(pop, fit\_func)

else:

p1 = rankSelection(pop, fit\_func)

p2 = rankSelection(pop, fit\_func)

c1, c2 = crossover(p1,p2,p\_cross)

new\_pop.append(mutate(c1,p\_mut))

if len(new\_pop)<POP\_SIZE:

new\_pop.append(mutate(c2,p\_mut))

Jednobodové kríženie kombinuje úseky rôznych dĺžok a okamžite ich orezáva na interval <MIN\_LEN, MAX\_LEN> – tak je zabezpečené, že ani kumulatívne účinky opakovaných krížení neporušia hranice. Mutačná procedúra pridáva alebo odoberá gény len po jednom, čím umožňuje jemnú reguláciu dĺžky.

Novovzniknutá populácia sa následne zoradí a odreže:

pop = sorted(new\_pop, key=fit\_func)[:POP\_SIZE]

Tým sa v každej generácii presadí kompromis s najnižšou penalizovanou chybou a postupne sa stabilizuje rozdelenie dĺžok okolo hodnoty, kde sa marža penalizácie vyrovná prírastku presnosti [5].

Po vyhodnotení celej generácie sa zvýši premenná evals += POP\_SIZE a do zoznamu best\_hist sa zaznamená aktuálna hodnota účelovej funkcie elity.

Každých desať generácií sa vypíše diagnostická správa:

if gen\_idx % 10 == 0:

print(f'{name} Evals {evals}: Best = {best:.6f}, Length = {len(elite)}')

Tento jednoduchý mechanizmus, ktorý je implementovaný aj v algoritmoch neskôr, poskytuje indikátor trendu dĺžky bez nákladného logovania celej populácie.

Kombináciou adaptívneho znižovania pravdepodobností (nižšia šanca na pridanie génu v neskorších fázach), elitnej ochrany a lineárnej penalizácie sa dosahuje stav, v ktorom:

* prebytočné gény nie sú dlhodobo selekčne udržateľné;
* krátkodobé odchýlky od cieľovej dĺžky sú stále povolené – čo zvyšuje šancu objaviť presnejšie riešenia;
* algoritmus konverguje k riešeniam, ktoré sú parsimoniálne a zároveň presné – čo korešponduje so všeobecne akceptovanými odporúčaniami pre kontrolu rastu dĺžky jedincov v GA s reálnym kódovaním [6][17].

Všetky uvedené prvky tak tvoria konzistentný rámec, v ktorom je premenná dĺžka genómu riadená priamo v srdci evolučného cyklu a jej vplyv na konvergenciu môže byť transparentne sledovaný aj vyhodnocovaný.

Zhrnutie GA-V

Navrhnutý genetický algoritmus bol rozšírený o minimalistické, no účinné prvky, ktoré umožnili dynamicky pridávať a odoberať gény bez zásahu do klasickej štruktúry GA. Dĺžka jedinca je riadená náhodným generovaním na začiatku, mutáciou (append/delete), krížením, limitmi a penalizáciou.

Takáto konfigurácia preukázala schopnosť prispôsobiť rozmer riešenia charakteru úlohy, pričom bolo zachované jednoduché kódovanie aj nízke výpočtové nároky. Za riziko sa považuje možný genetický „drift“ v „nevyužitých chvostoch“ dlhších vektorov, ktorý musí byť monitorovaný pri väčšom rozsahu parametrov [5].

Diferenciálna evolúcia s variabilnou dĺžkou

V nasledujúcej časti bude dokumentovaná úprava klasickej diferenciálnej evolúcie (DE) [4] na prácu s jedincom, ktorého počet súradníc sa smie počas optimalizácie meniť. Kód (funkcie DEMutate, DECrossover a run\_de) zachováva jadro algoritmu – diferenčné operácie v reálnom priestore – no dopĺňa ho o mechanizmy, ktoré dynamicky skracujú alebo predlžujú vektor riešenia a súčasne dodržiavajú pevné limity <MIN\_LEN, MAX\_LEN>.

Dynamický mutant variabilných rozmerov

Použitá mutačná stratégia DE/rand/1 vytvára „mutanta“ podľa nasledujúceho vzorca:

Ak sa dĺžky rodičov líšia, nemožno priamo odčítať vektory s rozdielnymi rozmermi [4]. V implementácii sa preto pracuje iba so súradnicami, ktoré majú všetci traja jedinci spoločné:

d\_mut = min(len(va), len(vb), len(vc))

mutant = [va[i] + F\*(vb[i] - vc[i]) for i in range(d\_mut)]

Počet komponentov d\_mut sa nastaví na najkratší z trojice. Tým je zabezpečená algebraická platnosť výrazu a zároveň sa implicitne rešpektuje princíp najmenšieho spoločného menovateľa – čím kratším vektorom sa umožní diktovať dimenziu mutanta, tým nižšia je pravdepodobnosť nežiaduceho nárastu rozmeru.

Operátor pridania a odstránenia prvku

Bezprostredne po vytvorení základného mutanta sa s pravdepodobnosťou P\_LEN\_CHANGE = 0.05 vloží zmena dĺžky (zvýšenie):

if random.random() < P\_LEN\_CHANGE:

if random.random() < 0.5 and len(mutant) < MAX\_LEN:

mutant.append(random.uniform(gene\_low, gene\_high))

elif len(mutant) > MIN\_LEN:

mutant.pop(random.randrange(len(mutant)))

Pridanie novej súradnice rozširuje vyhľadávací priestor práve o jeden rozmer. Hodnota sa nasadí z rovnakého rozsahu ako ostatné gény. Odstránenie náhodnej položky znižuje dimenziu o jeden, čím sa podporuje parsimónia. Symetrické rozdelenie pravdepodobností eliminuje prednastavenú tendenciu k rastu či k schudobneniu rozmeru [6].

Každá modifikácia je okamžite testovaná na hranice. Pri prípadnom podtečení sa mutant dopĺňa náhodnými génmi, pri opačnom prípade sa orezáva.

Kríženie

Klasický binárny operátor kríženia v DE požaduje identickú dĺžku cieľového vektora x a mutanta v [4]. V kóde sa preto vytvára dlhšia kostra s dĺžkou a chýbajúce hodnoty sa dosádzajú novými náhodnými génmi:

L = max(dt, dm)

j\_rand = random.randrange(L)

trial = []

for i in range(L):

if random.random() < CR or i == j\_rand:

if i < dm:

trial.append(mutant[i])

else:

trial.append(random.uniform(gene\_low, gene\_high))

else:

if i < dt:

trial.append(target[i])

else:

trial.append(random.uniform(gene\_low, gene\_high))

Takéto „spájanie s výplňou“ umožní, aby sa rozmery oboch rodičov preniesli do skúšobného vektora ľubovoľnej dĺžky, avšak výsledok je vždy prispôsobený pevným limitom. Opäť platí pravidlo minimálnej opravnej akcie – ak po dokončení dĺžka klesne pod MIN\_LEN, pridajú sa náhodné gény; ak presiahne MAX\_LEN, koniec sa skracuje.

Zachovanie škálovania parametrov F a CR

Verzie F a CR sa klesajúcou lineárnou funkciou generácie prispôsobujú podobne ako v GA-V:

F = F\_INIT \* (1 - gen\_idx/(N\_EVALS/POP\_SIZE))

CR = CR\_INIT \* (1 - 0.5\*gen\_idx/(N\_EVALS/POP\_SIZE))

Keďže normovanie rozdielu (xr2−xr3) závisí od počtu súradníc, nižšia hodnota F v neskorších fázach zamedzí príliš veľkým krokom v rozmerovo bohatom priestore. Zároveň sa tak stabilizuje adaptácia pri náhodných zmenách dĺžky, čo je odporúčané pri „self-adaptive“ DE algoritmoch [5].

Selektívna náhrada a penalizačný termín

Kritérium nahradenia v riadku:

if fit\_func(trial) <= fit\_func(x):

new\_pop.append(trial)

else:

new\_pop.append(x)

Využíva ohodnotenie penalizované dĺžkou mkPenalizer(). Skúšobný vektor teda nahradí cieľový jedinec iba vtedy, keď vylepší chybovú hodnotu aj po započítaní lineárnych nákladov na dĺžku. Dlhší vektor sa prijme len v prípade, že prínos presnosti preváži penalizačný prírastok, čo explicitne znižuje pravdepodobnosť bloatu [17].

Zhrnutie implementácie

Implementácia diferenciálnej evolúcie s variabilnou dĺžkou (DE-V) preukazuje, že klasický vektorový postup DE/rand/1/bin môže byť rozšírený na úroveň, v ktorej počet premenných nie je fixný. Najdôležitejším prvkom je mutant generovaný vždy iba po najkratšiu spoločnú dĺžku rodičov, čím sa zachová algebraická konzistentnosť diferenčného operátora a zároveň sa minimalizuje nekontrolovaný rast dimenzie. Pridanie alebo odobratie jediného génu – riadené pravdepodobnosťou P\_LEN\_CHANGE umožňuje vyhľadávaciemu procesu prispôsobovať si rozmer takmer nepretržite, pričom hranice <MIN\_LEN, MAX\_LEN> garantujú, že populácia nikdy neopustí vopred definovaný priestor riešení.

Rozšírené binárne kríženie dopĺňa chýbajúce súradnice náhodnými hodnotami a prípadné prekročenie limitov okamžite skracuje. Vektorová aritmetika teda ostáva plne kompatibilná s originálnou definíciou DE [4]. Adaptívne znižovanie faktorov F a CR v závislosti od počtu vyhodnotení prispieva k stabilite optimum-hľadania, aj keď dimenzia kolíše. Selektívny tlak na kompaktnosť je vynucovaný lineárnym penalizačným členom, ktorý vyvažuje presnosť voči zložitosti, a tak redukuje riziko bloatu [6][17].

Priebežný monitoring každých tisíc hodnotení poskytuje transparentné údaje o tom, ako sa dĺžka elitných riešení vyvíja.

Navrhnutá DE-V spája jednoduchosť klasickej diferenciálnej evolúcie s flexibilitou variabilnej reprezentácie. Trojica navzájom prepojených mechanizmov – dynamický mutant, jednoprvkový operátor zmeny dĺžky a lineárna penalizácia – umožňuje algoritmu udržať rovnováhu medzi exploráciou vysokorozmerných priestorov a parsimóniou riešení, bez nutnosti meniť základné princípy DE [5][11].

CMA-ES s variabilnou dĺžkou

V tejto implementácii bola klasická stratégia CMA-ES (Covariance Matrix Adaptation Evolution Strategy) rozšírená tak, aby sa počas behu mohla dynamicky meniť dimenzia optimalizovaného vektora. Základný algoritmus – samoadaptívna aktualizácia strednej hodnoty m, kovariančnej matice C a krokovej veľkosti σ – bol zachovaný podľa pôvodného postupu Hansen a Ostermeiera [43]. Do kódu však pribudli operácie, ktoré vkladajú alebo odrezávajú celé súradnice a súčasne udržiavajú všetky interné štatistiky konzistentné s novým rozmerom. V nasledujúcich odsekoch sa opisuje, ako sa táto úprava vykonáva a aký má vplyv na riadenie dĺžky jedinca.

Dynamická zmena dimenzie

Na začiatku každej generácie sa najprv s pravdepodobnosťou spustí blok, ktorý pridá alebo odstráni jeden rozmer:

* pripojenie novej súradnice – ak dĺžka ešte nedosiahla MAX\_LEN, vektor m sa rozšíri o náhodne inicializovanú hodnotu z intervalu <gene\_low, gene\_high>. Zároveň sa na kovariančnú maticu C aplikuje odsadenie, t. j. pribudne nový riadok aj stĺpec nastavený na nulu a hlavný diagonálny prvok sa inicializuje na 1.0, aby mala nová súradnica jednotkovú varianciu;
* odrezanie súradnice – ak je dĺžka väčšia než MIN\_LEN, posledná komponenta sa odstráni z vektorov m, p\_c, p\_s a zodpovedajúci riadok aj stĺpec sa vymažú z matice C.

if random.random()<P\_DIM\_CHANGE:

Takto je zabezpečené, že všetky stavové premenné zostávajú dimenzionálne kompatibilné a nikde nevznikajú nedefinované položky.

Aktualizácia hyper-parametrov

Okamžite po zmene počtu rozmerov n sa znova vyhodnotí funkcia, ktorá prepočítava váhy selekcie w, efektívnu veľkosť populácie μ\_eff a všetky adaptačné konštanty závislé od n. Tým sa zabráni tomu, aby pôvodné parametre (odvodené pre inú dimenziu) skreslili rýchlosť konvergencie alebo stabilitu evolúcie [43].

w, mu\_eff, c\_c, c\_s, c1, c\_mu, d\_s, chi\_n = computeHyperparams(n, mu)

Udržiavanie kovariančnej matice

Po každej iterácii sa C aktualizuje rovnicou klasickej CMA-ES [43]. Pred ďalším použitím sa matica:

1. Symetrizuje C = (C + C.T)/2.
2. Oreže alebo doplňuje podľa aktuálneho n.
3. Na diagonále sa hodnoty ohraničia do intervalu, <1e-8, 1e8>aby sa predišlo numerickému kolapsu.

Pri rozšírení rozmeru sa novovytvorený riadok/stĺpec necháva nulový, takže korelácie vzniknú prirodzene až po niekoľkých aktualizáciách. Pri skrátení sa nadbytočné korelácie odstránia, čím sa rýchlo eliminuje vplyv „zrušených“ súradníc.

Adaptívna kroková veľkosť σ

Vektor rýchlosti p\_s sa udržiava v priestore inv\_sqrtC. Preto po zmene dimenzie sa najprv prepočíta inverzná druhá odmocnina kovariančnej matice a následne sa aktualizuje:

p\_s = (1-c\_s)\*p\_s + sqrt(c\_s\*(2-c\_s)\*mu\_eff)\*(inv\_sqrtC@y)

Norma ||p\_s|| sa porovnáva s teoretickou strednou hodnotou γ\_n (ktorá je funkciou n), a exponent expnt upravuje σ. Vďaka tomuto postupu zostáva regulácia krokovej veľkosti konzistentná aj po náhlej zmene priestoru [43].

Penalizácia dĺžky verzus adaptácia

Účelová funkcia každého kandidáta vracia f(x) + pen(x) v prípade, ak sa dĺžka ocitne mimo hranice povolenej odchýlky („deadzone“) od cieľovej dĺžky. Lineárny prírastok penalizuje nadmerné zväčšenie aj prílišné skrátenie vektora, pričom „mŕtva zóna“ ± 5 komponentov ponechá CMA-ES-u voľnosť hľadať najvýhodnejší kompromis. Keďže váhy w uprednostňujú najúspešnejších μ potomkov, dodatočná dĺžková penalizácia sa automaticky premietne do pravdepodobnosti, s akou budú dlhé alebo krátke riešenia vyberané na formovanie novej strednej hodnoty m [43].

Zhrnutie implementácie

Rozšírená verzia CMA-ES bola navrhnutá tak, aby samoadaptívna štruktúra stratégie ostala nedotknutá, no vektor riešenia mohol počas optimalizácie meniť svoj rozmer. Kľúčom je blok pravdepodobného pridania/odstránenia dimenzie, okamžitá re-inicializácia všetkých rozmerovo závislých parametrov a uvoľnený (lineárny) penalizačný termín, ktorý vyvažuje presnosť proti zložitosti riešenia. Algoritmus tak kombinuje preslávenú robustnosť CMA-ES pri čiernej skrinke s flexibilitou variabilnej dĺžky, pričom riziko bloatu je tlmené pevným intervalom <MIN\_LEN, MAX\_LEN> a selekčným tlakom na parsimóniu [43].

Experimentálne testovanie a vyhodnocovanie

V tejto kapitole je opísaný spôsob, akým boli evolučné algoritmy implementované v rámci práce testované a vyhodnocované. Dôraz je kladený na objektívne a reprodukovateľné posúdenie ich správania na vybranom súbore benchmarkových problémov, pričom sú analyzované rôzne metriky súvisiace s kvalitou riešenia, dĺžkou výslednej reprezentácie a výpočtovou náročnosťou.

V úvode kapitoly sú stručne predstavené štatistické metódy, ktoré boli pri vyhodnocovaní použité, s dôrazom na neparametrické testy vhodné pre analýzu algoritmov pracujúcich s nelineárnymi a často neznámymi vlastnosťami optimalizačných funkcií. Následne je zdokumentovaný spôsob, akým boli generované výstupné údaje jednotlivými algoritmami, a detailne je opísaný skript, ktorým boli tieto údaje spracované, vizualizované a štatisticky analyzované. Celý proces bol navrhnutý tak, aby bola zabezpečená transparentnosť, modularita a možnosť opakovania experimentov.

Ciele experimentov

Experimentálna časť práce bola navrhnutá s cieľom porovnať správanie a výkonnosť vybraných evolučných algoritmov na rôznych typoch problémov. Dôraz bol kladený na zber kvantitatívnych údajov, ktoré umožňujú posúdiť kvalitu riešení, výpočtovú efektivitu a stabilitu dosahovaných výsledkov v opakovane spúšťaných experimentoch.

Zvolený experimentálny rámec vychádzal z nasledovných východísk:

* porovnateľnosť algoritmov – všetky algoritmy boli testované na rovnakom súbore benchmarkových úloh a pri identickom počte spustení (replikácií), čím boli vytvorené podmienky na spravodlivé porovnanie;
* zohľadnenie náhodnosti – keďže evolučné algoritmy sú založené na stochastických princípoch, boli všetky experimenty vykonané opakovane s rôznymi počiatočnými náhodnými stavmi, aby sa zachytila variabilita výsledkov;
* zachovanie objektivity a transparentnosti – experimenty boli navrhnuté tak, aby výsledky neboli ovplyvnené nepozorovanými premennými. Pre každý algoritmus bol použitý rovnaký mechanizmus zberu dát, rovnaký formát výstupu a identický postup spracovania výsledkov;
* zameranie na praktickú použiteľnosť – výber metrík reflektoval nielen optimalizačnú kvalitu, ale aj faktory, ktoré majú v praxi zásadný vplyv – veľkosť riešenia a výpočtový čas. Tým sa zabezpečilo, že porovnanie bude mať praktickú výpovednú hodnotu.

Celkovo bol návrh experimentov vedený snahou o dosiahnutie čo najvyššej mierky replikovateľnosti, s cieľom umožniť opakovanie analýzy aj v budúcnosti, prípadne jej rozšírenie o ďalšie algoritmy alebo benchmarky. Získané výsledky slúžili ako vstup pre následné štatistické vyhodnotenie a porovnanie výkonnosti jednotlivých metód.

Štatistické metódy použité pri testovaní

Pri vyhodnocovaní výsledkov experimentov s evolučnými algoritmami bola zvolená metodika, ktorá rešpektuje vlastnosti optimalizačných úloh a charakter analyzovaných dát. Keďže ide o porovnanie viacerých algoritmov na skupine benchmarkových problémov, pričom nie je možné predpokladať normálne rozdelenie meraných metrík a počet pozorovaní je obmedzený, bola použitá skupina neparametrických testovacích metód.

Friedmanov test

Na overenie existencie štatisticky významných rozdielov vo výkone algoritmov bol zvolený Friedmanov test pre opakované merania. Tento test predstavuje neparametrickú alternatívu k analýze rozptylu (ANOVA) a je založený na hodnotení poradia algoritmov v rámci jednotlivých benchmarkových úloh. Test bol pôvodne predstavený v práci Friedmana [44] a neskôr metodicky prehĺbený porovnaním alternatívnych testovacích štatistík a návrhom vhodných kritických hodnôt [46].

Friedmanov test sa osvedčil ako vhodný nástroj v prípadoch, keď sa porovnávajú algoritmy na rovnakých úlohách (blokový dizajn) a keď nie je možné predpokladať normalitu ani homogenitu rozptylu [1][2]. Takéto podmienky sú typické pre evolučné výpočtové techniky, kde môže byť variabilita medzi jednotlivými behmi výrazná a rozdelenie výsledkov často asymetrické [5].

Nemenyi post-hoc test

V prípade, že Friedmanov test odhalil štatisticky významný rozdiel medzi algoritmami (typicky pri hladine významnosti p < 0,05), bol následne aplikovaný Nemenyiho post-hoc test. Tento test slúži na identifikáciu konkrétnych dvojíc algoritmov, medzi ktorými existuje signifikantný rozdiel. Porovnávané sú priemerné poradia algoritmov a ich vzájomné rozdiely sa porovnávajú s tzv. kritickou diferenciu (Critical Difference, CD). Ak rozdiel prekročí túto hodnotu, považuje sa za štatisticky významný [45][46].

Metóda bola navrhnutá ako rozšírenie Friedmanovho rámca a patrí medzi najčastejšie používané techniky pri viacnásobnom porovnávaní algoritmov v oblasti strojového učenia a výpočtovej inteligencie [2][47].

Sledované metriky

Pre potreby hodnotenia boli vybraných päť kľúčové metrík:

* best – najlepšia hodnota cieľovej funkcie nájdená daným algoritmom;
* len\_final – dĺžka riešenia v jeho konečnej podobe (napr. dĺžka vektora či veľkosť stromu);
* cpu\_s – výpočtový čas v sekundách;
* delta\_len – absolútna odchýlka medzi dĺžkou riešenia a cieľovou referenčnou hodnotou stanovenou pre daný problém;
* fitness – pomocou nej bude vizualizovaný priemerný vývoj najlepšej hodnoty na každom benchmarku.

Tieto metriky umožňujú hodnotiť nielen kvalitu riešenia, ale aj jeho kompaktnosť a výpočtovú náročnosť, čo je v oblasti evolučného výpočtu často rozhodujúcim aspektom [1][5][6].

Implementácia zberu dát v algoritmoch

Aby mohli byť výkony jednotlivých algoritmov objektívne vyhodnocované a porovnávané, bol v rámci všetkých implementácií evolučných algoritmov vytvorený jednotný mechanizmus na zber, štrukturalizáciu a export výsledkov. Dôraz bol kladený na konzistentný formát, sledovanie relevantných metrík a zachytenie priebehu optimalizácie.

Štruktúra výstupu v algoritmoch

Výsledky boli zapisované do dvoch samostatných CSV súborov:

* súhrnný súbor (csv\_summary) obsahoval jeden riadok pre každú repliku experimentu a sumarizoval kľúčové výstupné metriky;
* s históriou hodnotení (csv\_history) zaznamenával vývoj fitness počas behu algoritmu, čím umožnil podrobnú analýzu konvergencie.

Oba súbory boli otvárané v režime zápisu cez kontextového správcu with, čím sa zabezpečilo ich bezpečné uzatvorenie po ukončení zápisu:

with open(csv\_summary, 'w', newline='') as fs, open(csv\_history, 'w', newline='') as fh:

writer\_sum=csv.DictWriter(fs, fieldnames=['run\_id','benchmark','replicate','seed','best','len\_final','cpu\_s'])

writer\_hist=csv.DictWriter(fh, fieldnames=['run\_id','benchmark','replicate','seed','evals','fitness'])

writer\_sum.writeheader()

writer\_hist.writeheader()

Zápis bol realizovaný pomocou objektov csv.DictWriter, ktoré umožňovali mapovanie medzi názvami stĺpcov a hodnotami v slovníkoch. Hlavička bola zapísaná automaticky prostredníctvom metódy writeheader().

Riadenie experimentu a identifikácia replík

Pre každý benchmarkový problém boli spúšťané viaceré nezávislé replikácie, pričom každý beh bol identifikovaný jedinečným identifikátorom run\_id vygenerovaným pomocou uuid.uuid4(). Replikácie boli kontrolované pomocou opakovaného cyklu:

for rep in range(N\_REPLICATES):

V rámci každej replikácie bol generovaný náhodný seed, ktorý bol následne aplikovaný súčasne pre generátor pseudonáhodných čísel v knižniciach random a numpy. Tým bola zabezpečená plná reprodukovateľnosť konkrétneho behu:

seed = random.randint(0,2\*\*31-1)

random.seed(seed); np.random.seed(seed)

Výpočtový čas behu algoritmu bol meraný pomocou funkcie time.perf\_counter().

Zápis výsledkov

Po ukončení behu bol do súhrnného súboru zapísaný jeden riadok obsahujúci:

* názov benchmarku;
* číslo repliky;
* použitý seed;
* najlepšiu nájdenú hodnotu cieľovej funkcie;
* dĺžku výsledného riešenia;
* a výpočtový čas v sekundách.

Príslušný zápis mal podobu:

writer\_sum.writerow({

'run\_id': run\_id,

'benchmark': name,

'replicate': rep,

'seed': seed,

'best': best\_val,

'len\_final': len(best\_ind),

'cpu\_s': round(cpu, 4)

})

Súbežne s tým bol do historického súboru zapisovaný priebeh vývoja riešenia počas optimalizácie. Každý záznam obsahoval počet vykonaných vyhodnotení a hodnotu účelovej fukncie v danom okamihu:

for ev, fitval in history:

writer\_hist.writerow({

'run\_id': run\_id,

'benchmark': name,

'replicate': rep,

'seed': seed,

'evals': ev,

'fitness': fitval

})

Zabezpečenie konzistencie a prenositeľnosti dát

Týmto spôsobom boli výstupy zo všetkých algoritmov generované v jednotnom formáte, čo umožnilo ich jednoduché načítanie a spracovanie v nadväzujúcich analytických krokoch. Údaje boli zároveň „verzované“ na úrovni priečinkov podľa času generovania, aby nedošlo k ich prepísaniu.

Implementácia vyhodnocovania algoritmov

----

analýza výsledkov

Závěr

Text závěru práce.

Seznam použité literatury

1. CUEVAS, Erik; OSUNA, Valentín a OLIVA, Diego. Evolutionary computation techniques: a comparative perspective. Studies in computational intelligence. Cham: Springer, 2017. Dostupné z: <https://doi.org/9783319511092>.
2. ZELINKA, Ivan. Evoluční výpočetní techniky: principy a aplikace. Praha: BEN - technická literatura, 2009. ISBN 978-80-7300-218-3. Dostupné také z: <http://krameriusndk.nkp.cz/search/handle/uuid:4ae2a4a0-4f76-11e4-a830-005056827e51>.
3. KOZA, John R. Genetic programming: on the programming of computers by means of natural selection. Complex adaptive systems. Cambridge, Mass: MIT Press, c1992. ISBN 0262111705.
4. PRICE, Kenneth V.; STORN, Rainer M. a LAMPINEN, Jouni A. Differential evolution: a practical approach to global optimization. Natural computing series. Berlin: Springer, 2005. ISBN 9783540209508. Dostupné také z: <https://proxy.k.utb.cz/login?url=https://doi.org/10.1007/3-540-31306-0>.
5. SIMON, Dan. Evolutionary optimization algorithms: biologically-Inspired and population-based approaches to computer intelligence. Hoboken, NJ: John Wiley & Sons, 2013. Dostupné také z: <https://ebookcentral.proquest.com/lib/natl-ebooks/detail.action?docID=1216196>.
6. MACH, Marián. Evolučné algoritmy: prvky a princípy. Edícia vedeckých spisov Fakulty elektrotechniky a informatiky TU Košice. Košice: Elfa, 2009. ISBN 978-80-8086-123-0. Dostupné také z: <https://www2.fiit.stuba.sk/~kvasnicka/Free%20books/Mach_Evolucne%20algoritmy_kniha.pdf>
7. Tanweer Alam. Shamimul Qamar. Amit Dixit. Mohamed Benaida. " Genetic Algorithm: Reviews, Implementations, and Applications.", International Journal of Engineering Pedagogy (iJEP). 2020.
8. SIVANANDAM, S. N. a DEEPA, S. N. Introduction to genetic algorithms. Berlin: Springer, c2008. ISBN 978-3-540-73189-4.
9. WIERSTRA, Daan; SCHAUL, Tom; PETERS, Jan a SCHMIDHUBER, Juergen. Natural Evolution Strategies. Online. In: 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence). IEEE, 2008, s. 3381-3387. ISBN 978-1-4244-1822-0. Dostupné z: https://doi.org/10.1109/CEC.2008.4631255. [cit. 2025-02-11].
10. O'NEILL, M. a RYAN, C. Grammatical evolution. Online. IEEE Transactions on Evolutionary Computation. Roč. 5, č. 4, s. 349-358. ISSN 1089778X. Dostupné z: https://doi.org/10.1109/4235.942529. [cit. 2025-02-11].
11. SLOWIK, Adam a KWASNICKA, Halina. Evolutionary algorithms and their applications to engineering problems. Online. Neural Computing and Applications. 2020, roč. 32, č. 16, s. 12363-12379. ISSN 0941-0643. Dostupné z: https://doi.org/10.1007/s00521-020-04832-8. [cit. 2025-02-11].
12. IBA, Hitoshi a NOMAN, Nasimul. New frontier in evolutionary algorithms: theory and applications. London: Imperial College Press, c2012. ISBN 978-1-84816-681-3.
13. BI, Ying; XUE, Bing; MESEJO, Pablo; CAGNONI, Stefano a ZHANG, Mengjie. A Survey on Evolutionary Computation for Computer Vision and Image Analysis: Past, Present, and Future Trends. Online. IEEE Transactions on Evolutionary Computation. 2023, roč. 27, č. 1, s. 5-25. ISSN 1089-778X. Dostupné z: https://doi.org/10.1109/TEVC.2022.3220747. [cit. 2025-02-22].
14. ZELINKA, Ivan; DAVENDRA, Donald; SENKERIK, Roman; JASEK, Roman a OPLATKOV, Zuzana. Analytical Programming - a Novel Approach for Evolutionary Synthesis of Symbolic Structures. Online. In: KITA, Eisuke (ed.). *Evolutionary Algorithms*. InTech, 2011. ISBN 978-953-307-171-8. Dostupné z: <https://doi.org/10.5772/16166>. [cit. 2025-03-11].
15. POLI, Riccardo; LANGDON, W. B. a MCPHEE, Nicholas F. *A field guide to genetic programming*. 2008. ISBN 978-1-4092-0073-4.
16. BANZHAF, Wolfgang; NORDIN, Peter; KELLER, Robert E. a FRANCONE, Frank. *Genetic programming: an introduction : on the automatic evolution of computer programs and its applications*. San Francisko: Morgan Kaufmann Publishers, c1998. ISBN 3920993586.
17. SILVA, Sara a COSTA, Ernesto. Dynamic Limits for Bloat Control. Online. In: DEB, Kalyanmoy (ed.). Genetic and Evolutionary Computation – GECCO 2004. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, s. 666-677. ISBN 978-3-540-22343-6. Dostupné z: <https://doi.org/10.1007/978-3-540-24855-2_74>. [cit. 2025-03-11].
18. MILLER, Julian Francis a HARDING, Simon L. Cartesian genetic programming. Online. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers. New York, NY, USA: ACM, 2009, s. 3489-3512. ISBN 9781605585055. Dostupné z: <https://doi.org/10.1145/1570256.1570428>. [cit. 2025-03-11].
19. DE JONG, Kenneth Alan. *An analysis of the behavior of a class of genetic adaptive systems*. Online, Ph.D. Dissertation. University of Michigan, USA: University of Michigan, 1975. Dostupné z: <https://deepblue.lib.umich.edu/bitstream/handle/2027.42/4507/bab6360.0001.001.pdf>. [cit. 2025-04-18].
20. XIN YAO; YONG LIU a GUANGMING LIN. Evolutionary programming made faster. Online. *IEEE Transactions on Evolutionary Computation*. Roč. 3, č. 2, s. 82-102. ISSN 1089778X. Dostupné z: <https://doi.org/10.1109/4235.771163>. [cit. 2025-04-19].
21. HANSEN, Nikolaus; FINCK, Steffen; ROS, Raymond a AUGER, Anne. Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions. [Výsledky výzkumu] RR-6829, INRIA, 2009. Dostupné z: <https://hal.inria.fr/inria-00362633>.
22. SUGANTHAN, P. N.; HANSEN, N.; LIANG, J. J.; CHEN, Y. -P.; AUGER, A. et al. *Problem Definitions and Evaluation Criteria for the CEC 2005 Special Session on Real-Parameter Optimization*. Online. ResearchGate. May 2005. Dostupné z: <https://www.researchgate.net/publication/235710019_Problem_Definitions_and_Evaluation_Criteria_for_the_CEC_2005_Special_Session_on_Real-Parameter_Optimization>. [cit. 2025-04-19].
23. *Optimization Test Problems*. Online. SIMON FRASER UNIVERSITY. Virtual Library of Simulation Experiments: Test functons and datasets. 2013. Dostupné z: <https://www.sfu.ca/~ssurjano/optimization.html>. [cit. 2025-04-19].
24. LI, Xiaodong; TANG, Ke; OMIDVAR, Mohammad Nabi; YANG, Zhenyu a QIN, Kai. Benchmark functions for the CEC’2013 Special Session and Competition on Large-Scale Global Optimization. Online. 2013. Dostupné z: <https://titan.csit.rmit.edu.au/~e46507/cec13-lsgo/competition/cec2013-lsgo-benchmark-tech-report.pdf>. [cit. 2025-04-19].
25. CHENG, Ran; LI, Miqing; TIAN, Yuandong; ZHANG, Xingyi; YANG, Shengxiang; JIN, Yao a YAO, Xin. Benchmark functions for the CEC’2017 Competition on Evolutionary Many-Objective Optimization. Online. In: Proceedings of the 2017 IEEE Congress on Evolutionary Computation (CEC), San Sebastián, Spain, 5.–8. júna 2017, s. 1–20. Dostupné z: <https://www.researchgate.net/profile/Yaochu-Jin/publication/312918385_Benchmark_Functions_for_the_CEC'2017_Competition_on_Evolutionary_Many-Objective_Optimization/links/5f8190cc92851c14bcbc18b9/Benchmark-Functions-for-the-CEC2017-Competition-on-Evolutionary-Many-Objective-Optimization.pdf>. [cit. 2025-04-19].
26. DEB, Kalyanmoy. An efficient constraint handling method for genetic algorithms. Online. In: Computer Methods in Applied Mechanics and Engineering. Volume 186, Issues 2–4. Amsterdam: Elsevier, 2000, s. 311-338. ISSN 0045-7825. Dostupné z: <https://doi.org/10.1016/S0045-7825(99)00389-8>. [cit. 2025-04-27].
27. PYTHON SOFTWARE FOUNDATION. Python Language Reference, version 3.12. Online. Python.org, 2024. Dostupné z: <https://docs.python.org/3/> [cit. 2025-05-22].
28. JETBRAINS s.r.o. DataSpell 2024.1 – Intelligent IDE for Data Science. Online. JetBrains, 2024. Dostupné z: <https://www.jetbrains.com/dataspell/> [cit. 2025-05-22].
29. HARRIS, Charles R.; MILLMAN, K. Jarrod; VAN DER WALT, Stéfan J.; GOMMERS, Ralf; VIRTANEN, Pauli et al. Array programming with NumPy. Online. Nature. 2020, roč. 585, č. 7825, s. 357-362. ISSN 0028-0836. Dostupné z: https://doi.org/10.1038/s41586-020-2649-2 [cit. 2025-05-22].
30. PYTHON SOFTWARE FOUNDATION. random – Generate pseudo-random numbers. Online. Python Standard Library, 2024. Dostupné z: <https://docs.python.org/3/library/random.html> [cit. 2025-05-22].
31. PYTHON SOFTWARE FOUNDATION. time – Time access and conversions. Online. Python Standard Library, 2024. Dostupné z: <https://docs.python.org/3/library/time.html> [cit. 2025-05-22].
32. PYTHON SOFTWARE FOUNDATION. csv – CSV File Reading and Writing. Online. Python Standard Library, 2024. Dostupné z: <https://docs.python.org/3/library/csv.html> [cit. 2025-05-22].
33. PYTHON SOFTWARE FOUNDATION. uuid – UUID objects according to RFC 4122. Online. Python Standard Library, 2024. Dostupné z: <https://docs.python.org/3/library/uuid.html> [cit. 2025-05-22].
34. PYTHON SOFTWARE FOUNDATION. os – Miscellaneous operating system interfaces. Online. Python Standard Library, 2024. Dostupné z: <https://docs.python.org/3/library/os.html> [cit. 2025-05-22].
35. PYTHON SOFTWARE FOUNDATION. operator – Standard operators as functions. Online. Python Standard Library, 2024. Dostupné z: <https://docs.python.org/3/library/operator.html> [cit. 2025-05-22].
36. FORTIN, Félix-Antoine; RAINVILLE, François-Michel de; GARDNER, Marc-André; PARIZEAU, Marc a GAGNÉ, Christian. DEAP: Evolutionary Algorithms Made Easy. Online. Journal of Machine Learning Research. 2012, roč. 13, s. 2171 – 2175. ISSN 1532-4435. Dostupné z: <https://jmlr.org/papers/v13/fortin12a.html> [cit. 2025-05-22].
37. LUKE, Sean. Essentials of Metaheuristics – A Set of Lectures. 3. vyd. Fairfax, VA: Lulu, 2020. ISBN 978-1-300-65497-2. Dostupné také z: <http://cs.gmu.edu/~sean/book/metaheuristics/> [cit. 2025-05-22].
38. LANGDON, William B. a POLI, Riccardo. Foundations of Genetic Programming. Genetic Programming Series. Berlin: Springer, 2002. ISBN 978-3-642-08002-1. Dostupné také z: <https://doi.org/10.1007/978-3-662-05094-1> [cit. 2025-05-23].
39. DEAP Development Team. deap/examples – Sample Implementations of Evolutionary Algorithms in Python. Online. GitHub Repository. 2024. Dostupné z: <https://github.com/DEAP/deap/tree/master/examples> [cit. 2025-05-23].
40. O’NEILL, Michael et al. PonyGE2: Grammatical Evolution in Python. Online. GitHub Repository. 2024. Dostupné z: <https://github.com/PonyGE/PonyGE2> [cit. 2025-05-23].
41. DE LIMA, Allan; CARVALHO, Samuel et al. GRAPE: Grammatical Algorithms in Python for Evolution. Online. GitHub Repository. 2024. Dostupné z: <https://github.com/bdsul/grape> [cit. 2025-05-23].
42. PONYGE2 Development Team. PonyGE2 Grammar Examples – BNF súbory pre gramatickú evolúciu. Online. GitHub Repository. 2024. Dostupné z: <https://github.com/PonyGE/PonyGE2/tree/master/grammars> [cit. 2025-05-23].
43. HANSEN, Nikolaus a OSTERMEIER, Andreas. Completely Derandomized Self-Adaptation in Evolution Strategies. Online. Evolutionary Computation. 2001, roč. 9, č. 2, s. 159-195. ISSN 1063-6560. Dostupné z: https://doi.org/10.1162/106365601750190398 [cit. 2025-05-26].
44. FRIEDMAN, Milton. The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance. Online. Journal of the American Statistical Association. 1937, roč. 32, č. 200, s. 675-701. ISSN 0162-1459. Dostupné z: <https://doi.org/10.1080/01621459.1937.10503522> [cit. 2025-05-26].
45. CONOVER, W. J. *Practical nonparametric statistics*. 3rd ed. Wiley series in probability and statistics. Applied probability and statistics. New York: John Wiley, 1999. ISBN 0-471-16068-7.
46. FRIEDMAN, Milton. A Comparison of Alternative Tests of Significance for the Problem of $m$ Rankings. Online. The Annals of Mathematical Statistics. 1940, roč. 11, č. 1, s. 86-92. ISSN 0003-4851. Dostupné z: <https://doi.org/10.1214/aoms/1177731944> [cit. 2025-05-26].
47. DEMŠAR, Janez. Statistical comparisons of classifiers over multiple data sets. Journal of Machine Learning Research. 2006, 7, 1-30. ISSN 1532-4435. Dostupné z: <https://jmlr.org/papers/v7/demsar06a.html> [cit. 2025-05-26].

Seznam obrázků

[Obrázok 1 Geometrická reprezentácia vývoja jedincom v priestore 19](#_Toc191573999)

Seznam tabulek

[Tabuľka 1 Porovnanie základných evolučných algoritmov 49](#_Toc191574006)

Seznam použitých symbolů a zkratek

A Význam první zkratky

B Význam druhé zkratky

C Význam třetí zkratky

Seznam příloh

Příloha P I: Název první přílohy

Příloha P I: Název první přílohy

Obsah první přílohy.