Nastavení optimální trajektorie robota s využitím symbolické regrese

Bc. Jiří Jílek

Diplomová práce
2006

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav řízení procesů
akademický rok: 2005/2006

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU

Jméno a příjmení:	**Bc. Jiří JÍLEK**
Studijní program:	**N 2807 Chemické a procesní inženýrství**
Studijní obor:	**Automatizace a řídicí technika**
Téma práce:	**Nastavení optimální trajektorie robota s využitím symbolické regrese**

Zásady pro vypracování:

1. Seznámení se s problematikou evolučních algoritmů
2. Seznámení se s problematikou symbolické regrese – detailní popis analytického programování
3. Vyberte příklady vhodné k demonstraci využití symbolické regrese pro návrh optimální trajektorie robota
4. Vytvořte simulace a proveďte optimalizace v prostředí Mathematica s využitím Analytického programování
5. Závěrem zhodnoťte výsledky

Rozsah práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. Koza J.R.: Genetic Programming, MIT Press, ISBN 0-262-11189-6, 1998

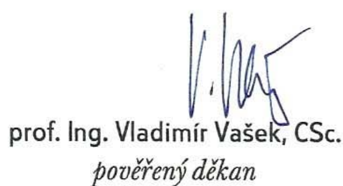
2. Zelinka I.,Oplatkova Z, Nolle L..:Boolean Symmetry Function Synthesis by Means of Arbitrary Evolutionary Algorithms-Comparative Study, International Journal of Simulation Systems, Science and Technology, Volume 6, Number 9, August 2005, pages 44 - 56, ISSN: 1473-8031, online http://ducati.doc.ntu.ac.uk/uksim/journal/Vol-6/No.9/cover.htm, ISSN: 1473-804x

3. Mařík V. a kol., Artificial Intelligence IV., 2004, Academia, Praha, Czech edition

4. Kvasnička V., Pospíchal J., Tiňo P., Evolučné algoritmy, STU Bralislava, 2000, ISBN 80-227-1377-5

Vedoucí diplomové práce:	**Ing. Zuzana Oplatková** Ústav aplikované informatiky
Datum zadání diplomové práce:	**14. února 2006**
Termín odevzdání diplomové práce:	**26. května 2006**

Ve Zlíně dne 25. února 2006



prof. Ing. Vladimír Vašek, CSc.
pověřený děkan

prof. Ing. Petr Dostál, CSc.
ředitel ústavu

ABSTRAKT

Analytické programování (AP) je nová metoda, která využívá ke svému chodu evoluční algoritmy. Cílem této práce je demonstrovat využití analytického programování pro návrh optimální trajektorie robota ze dvou úhlů pohledů, sada pravidel a sekvence přesných příkazů pohybu. K tomuto účelu byla vybrána úloha tzv. umělého mravence, která byla poprvé popsána Kozou [1] a úloha „zahradní sekačky“. Pro simulace byly v AP použity tyto evoluční algoritmy SamoOrganizující se Migrační Algoritmus (SOMA), Diferenciální Evoluce (DE) a Simulované žíhaní (Simulated Annealing - SA).

Klíčová slova: symbolická regrese, analytické programování, evoluční algoritmy, nastavení trajektorie

ABSTRACT

Analytic Programming is a novelty method which uses evolutionary algorithms for its run. The aim of this work is to show use of Analytic Programming for design an optimal trajectory for a robot from two points of view – set of movements rules and exact sequence of movement commands. For this purpose we were chosen task of artificial ant which was proposed by John Koza [1] and task of lawnmower. For simulation 3 evolutionary algorithms were used: SelfOrganizing Migrating Algorithm (SOMA), Differential Evolution (DE) and Simulated Annealing (SA).

Keywords: symbolic regression, Analytic Programming, evolutionary algorithms, setting of robot trajectory

Rád bych tímto poděkoval:

- Ing. Zuzaně Oplatkové vedoucí diplomové práce za její konzultace, cenné rady a za ochotu kdykoliv mi pomoc, když se vyskytl nějaký problém
- Mým rodičům za jejich podporu a trpělivost během mého studia

OBSAH

ÚVOD ... **9**

I TEORETICKÁ ČÁST ... **10**

1 ANALYTICKÉ PROGRAMOVÁNÍ ... **11**

1.1 HILBERTŮV FUNKČNÍ PROSTOR A OBECNÝ PROSTOR FUNKCÍ ... 11

1.2 MANIPULACE S MNOŽINOU DISKRÉTNÍCH HODNOT ... 12

1.2.1 Postup při práci s množinou diskrétních čísel v EA ... 12

1.2.2 Postup při práci s množinou diskrétních čísel v AP ... 13

1.3 ZAJIŠTĚNÍ VYTVOŘENÍ NEPATOLOGICKÝCH FUNKCÍ ... 17

1.4 VYHODNOCENÍ ÚČELOVÉ FUNKCE V AP ... 18

1.5 VERZE AP ... 19

2 GRAMATICKÁ EVOLUCE ... **21**

2.1 REPREZENTACE JEDINCŮ V GE ... 21

2.2 KŘÍŽENÍ V GE ... 24

3 GENETICKÉ PROGRAMOVÁNÍ ... **26**

3.1 REPREZENTACE JEDINCŮ ... 26

3.2 KŘÍŽENÍ V GP ... 27

3.3 MUTACE V GP ... 28

4 EVOLUČNÍ ALGORITMY ... **29**

4.1 SAMOORGANIZUJÍCÍ SE MIGRAČNÍ ALGORITMUS (SOMA) ... 30

4.1.1 Parametry SOMA ... 30

4.1.2 Princip SOMA ... 32

4.1.3 Variace algoritmu SOMA ... 35

4.2 DIFERENCIÁLNÍ EVOLUCE ... 36

4.2.1 Parametry DE ... 36

4.2.2 Princip DE ... 37

4.2.3 Varianty DE ... 39

4.3 SIMULOVANÉ ŽÍHÁNÍ ... 39

II PRAKTICKÁ ČÁST ... **41**

5 DEMONSTRACE VYUŽITÍ AP PRO NÁVRH OPTIMÁLNÍ TRAJEKTORIE ROBOTA ... **42**

5.1 ÚLOHA TZV. UMĚLÉHO MRAVENCE (ARTIFICIAL ANT) ... 43

5.1.1 Stezka „Santa Fe“ ... 43

5.1.2 Množina funkcí použitých k pohybu mravence v daném prostoru ... 44

5.1.3 Ohodnocení účelové funkce ... 46

5.1.4 Použité evoluční algoritmy ... 49

5.1.5 Dosažené výsledky a jejich porovnání ... 51

5.2 ÚLOHA „ZAHRADNÍ SEKAČKY“ (LAWNMOVER) ... 57
5.2.1 Popis trávníku ... 57
5.2.2 Množina funkcí použitých pro pohyb sekačky po trávníku ... 58
5.2.3 Ohodnocení účelové funkce ... 61
5.2.4 Dosažené výsledky ... 61
ZÁVĚR ... **63**
SEZNAM POUŽITÉ LITERATURY ... **64**
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK ... **65**
SEZNAM OBRÁZKŮ ... **68**
SEZNAM TABULEK ... **70**
SEZNAM PŘÍLOH ... **71**

ÚVOD

Lidé celé věky optimalizují vše kolem sebe. Vědci už dlouho zkoumají optimalizaci a zpočátku více využívali analytického řešení problémů. V minulém století během technické revoluce, když začalo být nutné řídit různé složité procesy, tedy i optimalizovat složitější problémy začaly se využívat numerické metody. V této době byl člověk donucen a to nejen kvůli své chybovosti, pomalé reakci ale i ekonomickému faktoru, nahradit v co nejvyšší míře lidskou pracovní sílu stroji nebo roboty ovládanými pomocí počítačů. Tím se docílilo zlepšení kvality výrobků, objemu výroby, snížení ceny a samozřejmě to nejdůležitější zvýšení zisku. Nejen optimální nastavení ať už šířky stěny, tlakové nádoby nebo trajektorie robota, ale i doba za kterou jsme schopni tohoto nastavení dosáhnout dále snižují náklady a zvyšují zisky. A proto se vědci snažili vyvinout nové metody optimalizace.

V letech 1970 – 80 se podařilo vyvinout novou skupinu algoritmů tzv. evoluční algoritmy (EA). Velká výhoda evolučních algoritmů je, že uživateli stačí jen dobře znát svůj problém a být schopen správně ošetřit účelovou funkci. Tyto algoritmy můžou pracovat se všemi typy argumentů (reálné číslo, celé číslo, množina diskrétních hodnot). Pro analytické a číslicové postupy výpočtu optimalizace problémů je využití všech typů argumentů skoro neřešitelný problém.

V roce 2001 vytvořil doc.Ivan Zelinka novou metodu tzv. Analytické Programování (AP), která dovoluje řešit problémy analyticky právě pomocí těchto evolučních algoritmů jako jsou např. SamoOrganizující se Migrační Algoritmus (SOMA), Diferenciální Evoluce (DE), Simulované žíhaní (Simulated Annealing - SA) atd.

Pro zjištění kvalit této nové metody jsem se rozhodl porovnat (při stejném nastavení parametrů) výsledky AP s výsledky, které zveřejnil J.Koza v Genetickém programování (GP), které využívá Genetické algoritmy (GA) [1]. A to na příkladu tzv. umělého mravence a zahradní sekačky.

Obě dvě metody jak AP tak i GP jsou součástí umělé inteligence, mladé vědecké disciplíny s počátky v poslední čtvrtině minulého století.

I. TEORETICKÁ ČÁST

1 ANALYTICKÉ PROGRAMOVÁNÍ

Analytické Programování (AP) je nová metoda symbolické regrese, kterému předchází Gramatická Evoluce (GE) a Genetické programování (GP).

Autor analytického programování vybral jeho jméno, protože jednoduše signalizovalo využití EA pro analytické řešení syntézy (tj. symbolickou regresi). Jak již bylo řečeno je možné využít v AP téměř všechny evoluční algoritmy (a to i experimentálně testované) použitím každého nového algoritmu by vzniklo, dle užívaného přístupu nové jméno např. SOMA programování, DE programování, SA programování atd. Což by jistě bylo matoucí a komplikované.

1.1 Hilbertův funkční prostor a obecný prostor funkcí

Analytické programování bylo inspirováno GP a číslicovými metodami v Hilbertových funkčních prostorech. Princip AP je někde mezi těmito dvěma filozofiemi. Z GP přebírá představu evolučního vytvoření symbolických řešení, zatímco z Hilbertových prostorů přijímá vystavění výsledné funkce prostřednictvím optimalizace (obvykle vytvořeného takovými metodami jako jsou Ritzova nebo Galerkinova) [2].

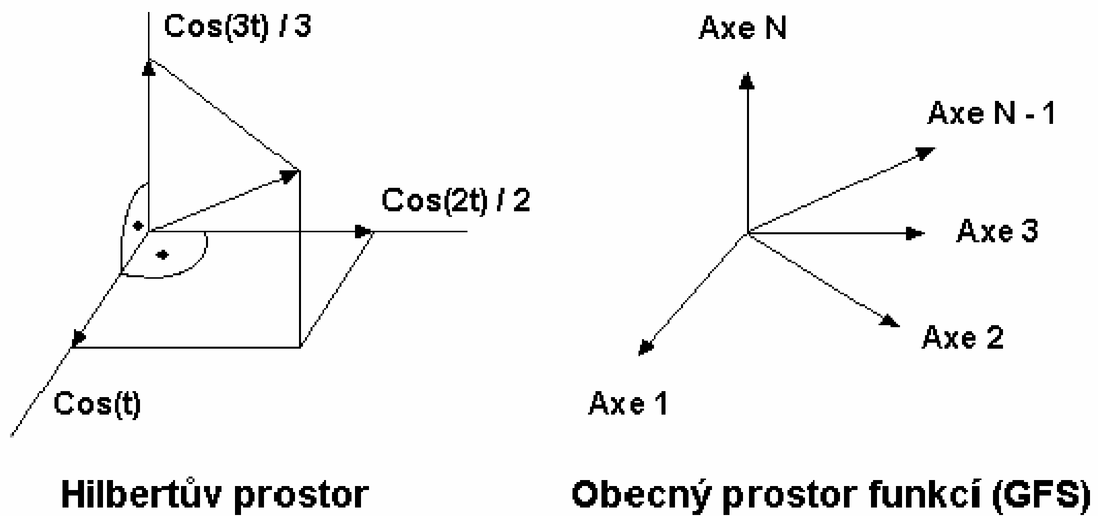
Analytické programování pracuje se složitým funkčním prostorem, který používá zvláštní způsob rozvoje požadovaného řešení a díky svým vlastnostem ho nazýváme obecným prostorem funkcí (GFS převzato z anglického General Function Space). Osy GFS si nejsou navzájem kolmé a nepředstavují jen jedinou funkci jako v případě Hilbertova prostoru, ale každý bod (celé číslo) na ose reprezentuje funkci, operátor nebo konstantu.

GFS je více obecnější než Hilbertův prostor, který je široce využívaný ve fyzice, aplikované matematice atd. Hilbertův prostor, je oproti GFS, definovaný obvykle jako prostor s vlastnostmi:

- Úplný
- Kompaktní
- Orthogonální
- Orthonormální

Tyto požadavky dovolí řešit různé problémy obvykle jednoduchým způsobem. Z geometrického hlediska, může být Hilbertův prostor zobrazen ze vzájemně se sestávajících kolmých os, kde každá z os představuje jednu základní, obvykle periodickou funkci jako je

sinus nebo kosinus (Obr.1). Pozice každého bodu v takovém funkčním prostoru je popsána souřadnicemi všech os. Složení všech složek vektoru (všech os) dává výslednou funkci tedy vektor vycházející z nulových souřadnic k danému bodu v Hilbertově prostoru. Pro řešení daného problému se nejprve vybere vhodný funkční základ, z tohoto funkčního základu se vytvoří funkcionál a pomocí vhodných metod jsou odhadnuty neznámé parametry.



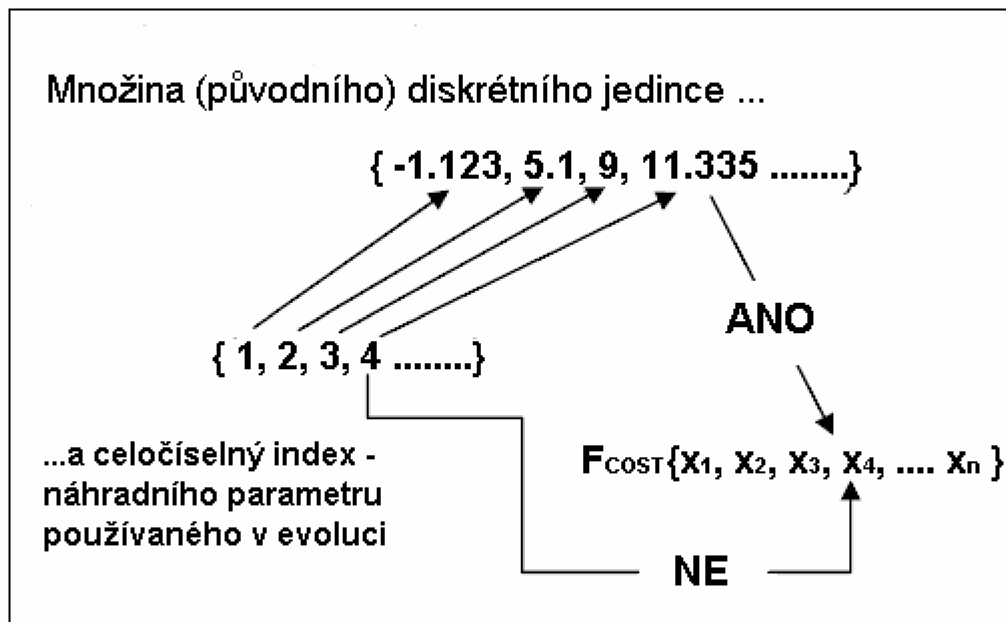
Obr. 1 Vzorové příklady funkčních prostorů

1.2 Manipulace s množinou diskrétních hodnot

Hlavní princip AP je založen na práci s množinou diskrétních hodnot (DSH převzato z anglického - Discrete Set Handling). DSH bylo vyvinuto při práci se SOMA a používá se v optimalizacích evolučními algoritmy (viz níže).

1.2.1 Postup při práci s množinou diskrétních čísel v EA

Při množině diskrétních hodnot jako je např. (-1, -2.5, 20, 3, -5.68, 254.3569…), která představuje parametry jedinců se vytvoří množina celočíselných indexů o stejné velikosti, nabývající hodnot (1, 2, 3, 4, 5, 6 …). Tyto indexy nahradí diskrétní parametry jedince a pracuje se s nimi jako by s normálními celočíselnými parametry, až na to, že při ohodnocení účelové funkce se nedosadí za dané argumenty aktuální celočíselné hodnoty, ale hodnoty z diskrétní množiny, na které tyto celočíselné parametry, coby index, ukazují viz. (Obr.2).



Obr. 2 Manipulace s množinou diskrétních hodnot

Právě díky tomuto principu může být v AP uskutečněno využívání téměř každého evolučního algoritmu.

1.2.2 Postup při práci s množinou diskrétních čísel v AP

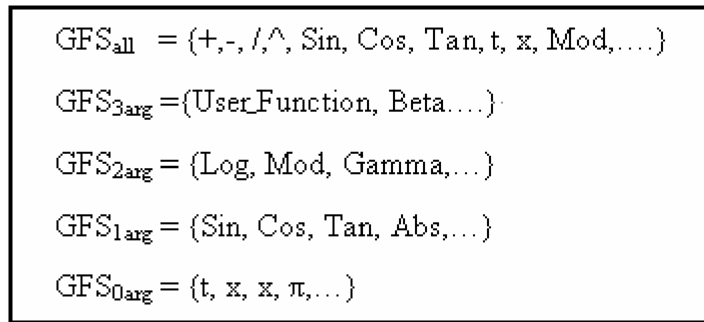
Tak jako GP nebo také GE, tak i AP je založeno na množině matematických objektů, které představují funkce, operátory a tzv. terminály (obvykle konstanty nebo nezávislé proměnné) například:

- funkce: Sin, Tan, And, Or
- operátory: +, -, *, /, dt,…
- terminály: 2.73, 1.75, t, a,…

Všechny tyto matematické objekty tvoří množinu funkcí s různým počtem argumentů nazvanou GFS, ze které AP zkouší syntetizovat vhodné řešení.

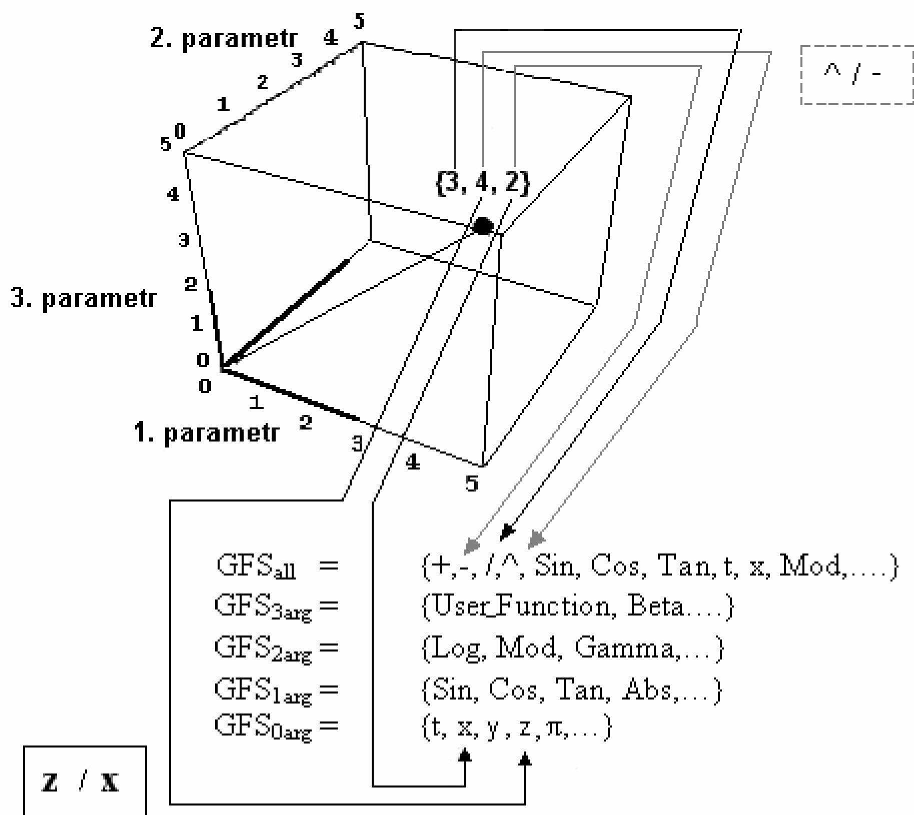
Struktura GFS je tvořena podmnožinami z funkcí podle počtu argumentů viz (Obr.3). Například GFS_{all} je soubor obsahující množinu všech funkcí, operátorů a terminálů, GFS_{3srg} je podmnožina s funkcemi obsahující jen 3 argumenty, GFS_{0arg} obsahuje pouze terminály atd.

Obsah GFS je dán uživatelem.



Obr. 3 Princip struktury GFS

V AP se jedinci skládají z nenumerických výrazů (operátorů, funkcí), které jsou v evolučním procesu reprezentovány jejich celočíselnými indexy. Tento index pak slouží jako ukazovátko do souboru výrazů, který využívá AP k syntéze výsledného cílového programu pro vyhodnocení účelové funkce. Příklad budující funkce z GFS prostřednictvím AP je zobrazen na (Obr. 4). Pro 3-parametrového jedince si můžeme GFS představit jako krychli, kde lemy jsou funkční základny [3].



Obr. 4 Příklad s jedincem o třech parametrech

Složení podmnožin prezentované v GFS je zásadně důležitá pro AP. Užívá se, aby nedošlo k syntéze patologických programů, tj. programů obsahující funkce bez argumentů atd. Vý-

kon AP je samozřejmě lepší, jestliže je výběr funkcí GFS odborně vybrán na základě předchozích zkušeností s řešeným problémem.

Důležitou částí AP je sekvence matematických operací, která je použita pro syntézu programu. Tyto operace jsou užity pro transformaci jedinců populace do vhodného plně funkčního programu. Tato transformace se skládá ze dvou hlavních částí, první částí je DSH a druhá část jsou bezpečnostní procedury, které nedovolí syntetizovat patologické programy.

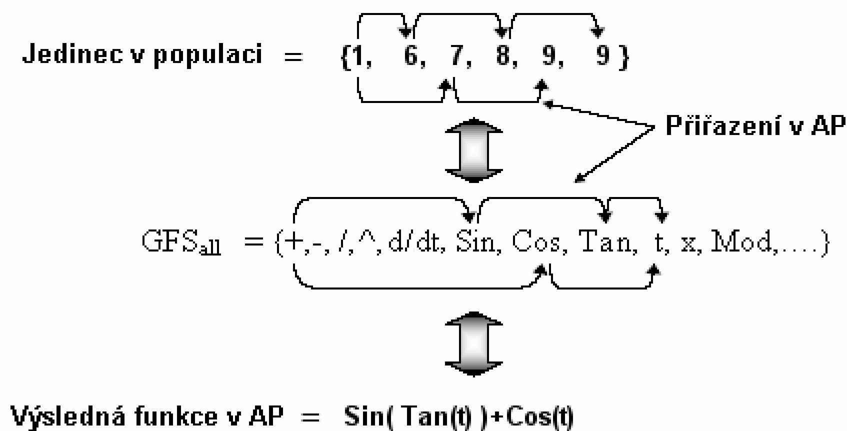
V AP je DSH užito pro transformaci jedince a společně s bezpečnostními procedurami vytváří výše uvedené přepsání, které transformuje libovolného jedince do programu. Jedinci v populaci se skládají z celočíselných parametrů, které ukazují na určitou funkci v GFS.

V případě (Obr.4) je jedinec s indexy {3, 4, 2}vybranými z GFS. Kdyby neexistovala žádná bezpečnostní procedura zamezující tvoření patologických funkcí pak by vznikla funkce ^ / - (označená šedou barvou). Ale výsledná funkce AP je z/x. Následující část bude podrobně popisovat jak se tvoří funkce z jedince s indexy.

V jedinci jsou náhodně vybraná čísla od 1 až po hodnotu danou velikostí souboru všech funkcí definovaných v GFS_{all}.

Proces vytvoření nové funkce je následující:

- První index je nahrazen příslušnou funkcí z GFS_{all}.
- Pak podprogram v AP prověří kolik potřebuje vybraná funkce argumentů a kolik ještě zbylo indexů v jedinci.
- Další výběr funkcí je založen na těchto faktech.



Obr. 5 Příklad ideálního jedince v AP

Příklad na (Obr.5) představuje ideální případ jedince v AP. První index jedince je 1 a ta představuje v GFS_{all} funkci plus. Tato funkce má dva argumenty, proto indexy 6 a7 jsou jejími argumenty (1).

$$6+7 \tag{1}$$

Index 6 je pak nahrazen funkcí sinus a index 7 funkcí cosinus (2).

$$Sin+Cos \tag{2}$$

Sinus a cosinus jsou 1-argumentové funkce. Po indexu 7 následuje index 8, který je nahrazen funkcí tangens, která je taky 1-argumentová a vložen do funkce sinus (3).

$$Sin(Tan)+Cos \tag{3}$$

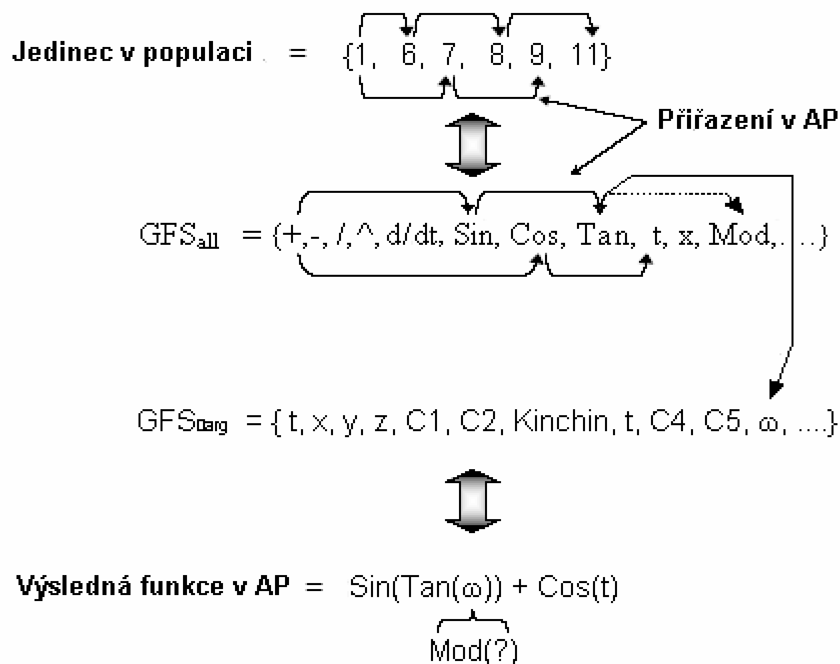
Po indexu 8 následuje index 9, kterému odpovídá proměnná t, která je argumentem funkce cosinus (4).

$$Sin(Tan)+Cos(t) \tag{4}$$

A poslední index je znovu 9, tedy t, které je argumentem tangens. Výsledná funkce AP je pak výraz (5).

$$Sin(Tan(t))+Cos(t) \tag{5}$$

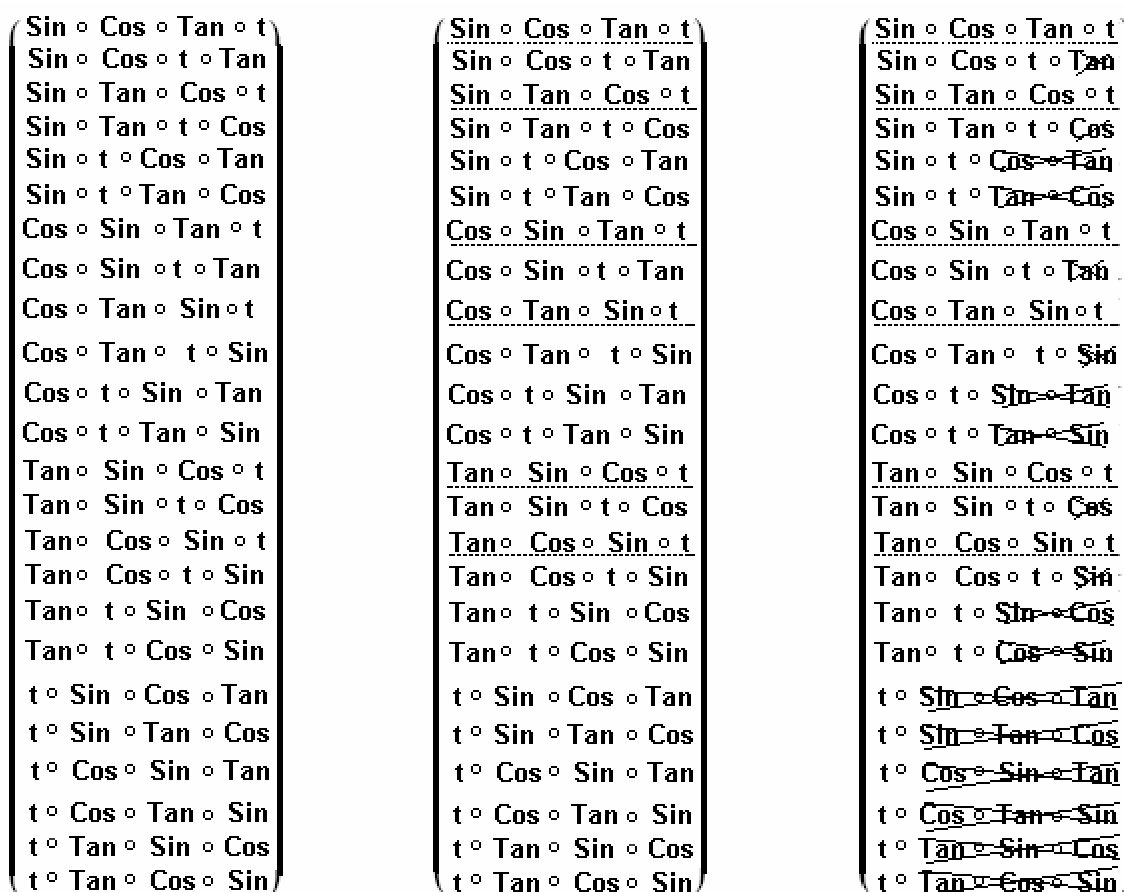
Tento příklad byl ideální, protože volba indexů byla tak dobrá, že nevznikaly žádné patologické funkce. Ale v dalším příkladu na (Obr.6) už musí být využit podprogram AP, aby zamezil vzniku patologických funkcí. Obr.6 je stejný jako Obr.5 s jediným rozdílem, poslední index jedince není 9 ale 11. To znamená že proměnná t je zaměněna za funkci Modulo. Funkce Modulo má dva argumenty, ale jedinec již nemá více argumentů, proto je nutné vybrat nějakou proměnnou z GFS_{0arg}. Jak je vidět na (Obr.6) ze souboru byla vybrána proměnná ω.



Obr. 6 Příklad s využitím podprogramu pro zamezení patologií

1.3 Zajištění vytvoření nepatologických funkcí

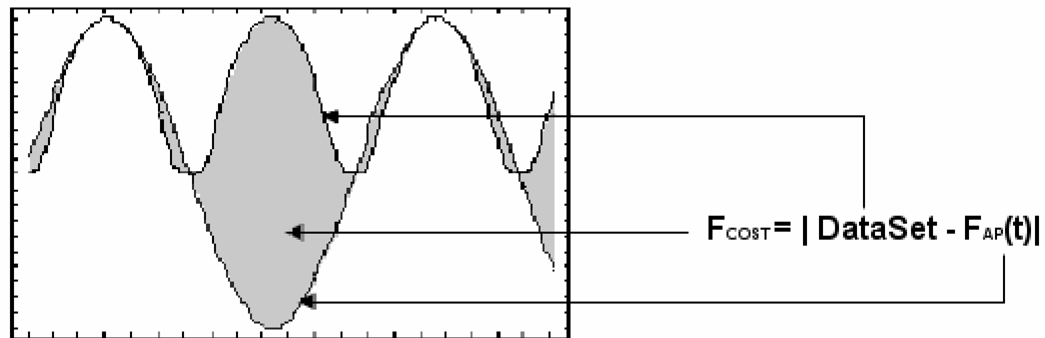
Zajištění vytvoření nepatologických funkcí je důležité. Znamená to, že jsou vytvořeny jen funkce se správným počtem argumentů (viz.výše). Obr.7 demonstruje, jak může vypadat patologická funkce před a po korekci. V obrázku jsou použity jen 4 základní funkce {Sin, Cos, Tan, t}. Tvar všech funkcí (patologických i nepatologických), které mohou být generovány, je dán permutací. V prvním sloupci je zobrazeno n! možných permutací, tedy 24 kombinací 4 základních funkcí. Všechny přirozené nepatologické funkce jsou podtrženy v druhém sloupci. Třetí sloupec zobrazuje patologické funkce po korekci. Patologické části jsou překříženy.



Obr. 7 Schéma korekce patologických funkcí

1.4 Vyhodnocení účelové funkce v AP

Analytické programování se využívá k vytvoření nových funkcí – programů. Může být použito pro nalezení funkcí, které dokáží vhodně prokládat neznámá data. Vhodná účelová funkce pro tyto případy je absolutní hodnota rozdílu mezi originálními daty a právě nalezenou funkcí, jak je zobrazeno na (Obr.8). Cílem AP je minimalizovat tento rozdíl, tedy minimalizovat šedou oblast na Obr. 8. V nejlepším případě by hodnota účelové funkce měla být rovna nule.



Obr. 8 Ohodnocení účelové funkce (Fcost – hodnota účelové funkce, DataSet – množina změřených bodů, Fap – výstup z AP)

1.5 Verze AP

Analytické programování bylo testováno ve třech verzích. Všechny tři verze používají k syntetizaci programu stejnou množinu funkcí, terminálů atd., jako používá Koza in GP [4]. Druhá verze (AP_{meta}, název první verze je AP_{basic}) je upravena ve smyslu odhadu konstanty. Například, Koza používá pro tzv. Sextic úlohu (hledání analytického tvaru funkce dané polynomem 6. řádu) Koza [1] náhodné generování konstant, zatímco AP zde používá jen jednu konstantu pojmenovanou K, která je vložena do generovaného programu v různých místech evolučního procesu, např. (6). Když je program syntetizován, pak všechny K indexovány jako K_1, K_2, … K_n (7) a pak jsou všechny K_n odhadnuty použitím druhého evolučního algoritmu (8). Protože EA podřízený (slave) “pracuje pod“ EA řídící (master), tedy EA_{master} ► program ► indexování K ► EA_{slave} ► odhad K_n) je tato verze pojmenována AP s metaevolucí – AP_{meta}.

$$\frac{x^2 + K}{\pi^K} \tag{6}$$

$$\frac{x^2 + K_1}{\pi^{K_2}} \tag{7}$$

$$\frac{x^2 + 3.56}{\pi^{-229}} \tag{8}$$

Protože je tato metoda časově náročná byla AP_{meta} upravena do třetí verze, která se liší od té druhé v odhadu K. Toho je dosaženo použitím vhodné metody nelineárního proložení

(AP_{nf} non-linear fitting). Tato metoda ukázala nejslibnější výkony u úloh s neznámými konstantami. V této práci je využito první verze AP_{basic} [5].

2 GRAMATICKÁ EVOLUCE

Další služebně starší metodou je tzv. gramatická evoluce [6]. Jde o metodu, která má mnohé společné jak s GA, tak s GP. Ve skutečnosti je to vlastně kombinace obou těchto přístupů. S GP má společný cíl a tím také oblasti použití – je to totiž nástroj pro automatické generování počítačových programů.

2.1 Reprezentace jedinců v GE

Oproti GP je GE obecnější v tom, že je navržena tak, aby byla použitelná pro hledání programů v jakémkoliv jazyce. Přesněji řečeno v jakémkoliv jazyce, který může být popsán bezkontextovou gramatikou, resp. Backusovou-Naurovo formou (BNF). S tím je úzce spojena i reprezentace jedinců. Na rozdíl od stromů požívaných v GP požívá GE binární reprezentaci jedinců. A jako hlavní genetické operátory požívá jednoduché jednobodové křížení a jednoduchou bodovou mutaci, což je obojí vlastní klasickým genetickým algoritmům.

Princip reprezentace s využitím BNF a výhody, které z toho vyplývají, ukážeme na jednoduchém příkladu hledání aritmetických výrazů. Gramatika BNF popisuje jazyk formou produkčních pravidel, ve kterých vystupují terminály, tj. atomické symboly, a neterminály, které jsou dále rozvinuty v jeden nebo více neterminálů a terminálů. Pravidla mají pevnou strukturu, kde na levé straně vystupují jednotlivé neterminály a na pravé straně je rozvoj příslušného neterminálu pomocí neterminálů a terminálů. Každý neterminál může mít více alternativních pravidel pro expandování..

neterminály		*rozvoj*	*index*
expr	::=	op expr expr	(0)
		var	(1)
op	::=	+	(0’)
		-	(1’)
		*	(2’)
		/	(3’)
var	::	X	(0’’)
		Y	(1’’)

Obr. 9 GE gramatika příkladu

V našem příkladu uvažujeme výrazy, ve kterých se mohou vyskytovat operace{+, -, *, /} a proměnné X a Y. Dohromady tvoří množinu terminálů T ={+, -, *, /, X, Y}. Množina neterminálů obsahuje symboly F={expr, op, var}. gramatika generující výrazy je zobrazena na (Obr.9), expr je startovací symbol.

Tato gramatika se nyní použije pro dekódování lineárního chromozomu. Ten má v GE takovou funkci, že reprezentuje posloupnost pravidel tak, jak budou postupně aplikována během generování programu. V GE mají chromozomy proměnnou délku. Celkový řetězec je formálně členěn na osmibitové podřetězce kódující čísla 0-255, které se nazývají kodony. Tyto kodony jsou postupně čteny od začátku chromozomu a na základě jejích hodnoty je použito odpovídající pravidlo pro rozvinutí aktuálního nejlevějšího terminálu. Vzhledem k tomu, že hodnota kodonu je ve většině případů větší než počet pravidel použitelných pro jednotlivé neterminály, je číslo pravidla, které se pro rozvinutí neterminálu použije stanoveno pomocí výrazu (9).

Pravidlo = kodon mod počet_pravidel_pro_daný_neterminál (9)

Uvažujme například, že máme rozvinout neterminál op, pro který si můžeme vybrat ze čtyř možných pravidel, a dále, že poslední přečtený kodon má hodnotu 17. V tomto případu by tedy byl neterminál op nahrazen podle pravidla (1′) z tabulky 2 znaménkem mínus, neboť dělením 17 modulo 4 získáme hodnotu 1.

Čtení chromozomu pokračuje zleva doprava, dokud nedojde k jedné z následujících situací:

- Program je hotov, tzn. že všechny neterminály už byly přepsány na terminály.
- Ještě je třeba rozvinout jeden nebo více neterminálů, ale už by byl přečten poslední kodon chromozomu.

V prvním případě, pokud ještě zbývají v chromozomu nějaké nepoužité kodony, se jedná o tzv. přespecifikovaný chromozon. Tuto situaci nijak speciálně ošetřovat nemusíme, prostě zbytek chromozomu ignorujeme. Ve druhém případě, ale musíme nějak dokončit odvození programu, jinak nebudeme schopni ohodnotit daného jedince. V takovém případě si můžeme pomoci například tak, že chybějící kodony budeme číst znovu od začátku chromo-

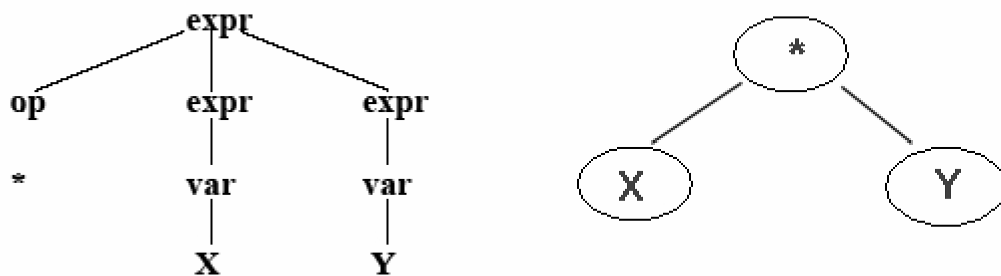
zomu. Aby nedošlo k nekonečnému, nebo příliš dlouhému cyklení používá se omezení počtu průchodů chromozomem.

Celý proces sestavení programu pomocí naší gramatiky se ukážeme na chromozomu viz. (Tab.1).

Tab. 1 Ukázkový chromozom

Chromozom	**Binární řetězec**	**Celé číslo**	**BNF index**
Codon1	00101000	40	(0)
Codon2	11000011	162	(2´)
Codon3	00001100	67	(1)
Codon4	10100010	12	(0´´)
Codon5	01111101	125	(1)
Codon6	11100111	231	(1´)
Codon7	10010010	146	Nepoužitý kodon
Codon8	10001011	139	Nepoužitý kodon

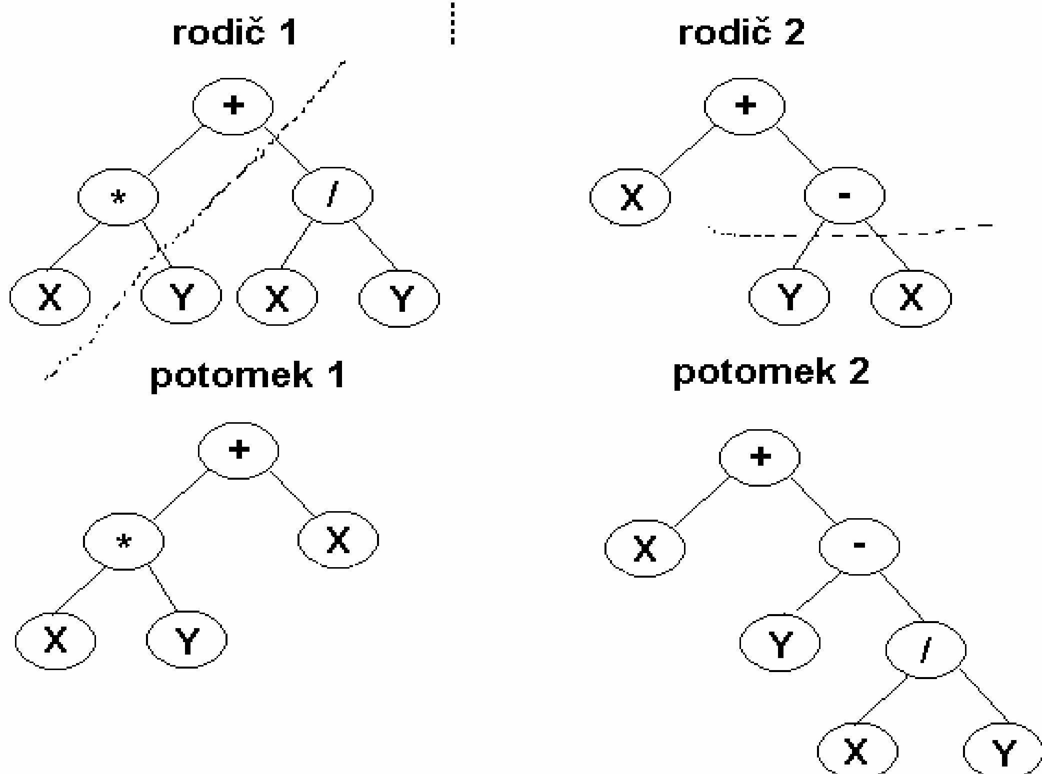
Generování programu začíná rozvinutím symbolu expr, pro který máme dvě pravidla. Protože první kodon chromozomu má hodnotu 40, bude se expr přepisovat pomocí pravidla (0) na op expr expr. Následuje rozvinutí symbolu op. Ze čtyř možných pravidel vybereme pomocí kodonu 162 pravidlo (2´) a op přepíšeme na „*“. Pokračujeme rozvinutím levého ze dvou zbývajících symbolů expr. Ze dvou možných pravidel vybereme pomocí kodonu 67 pravidlo (1) a expr přepíšeme na var. Následující kodon 12 určuje, že symbol var bude přepsán podle pravidla (0´´) na X. Druhý symbol expr se opět přepíše na var podle pravidla (1), což je dáno kodonem 125. A konečně var bude přepsáno podle pravidla (1´) na Y. Tím je program hotov, přičemž v chromozomu zůstaly ještě dva nepoužité kodony viz. (Obr.10).



Obr. 10 Derivační strom a odpovídající program

2.2 Křížení v GE

Standardně se v gramatické evoluci používá jednobodové křížení, které je aplikováno na chromozomy. Podle efektu, který má tento operátor na křížené stromové struktury rodičovských jedinců se také nazývá vlnové křížení (ripple crossover). Křížení funguje tak, že si rodičovské řetězce vzájemně prohodí sekvence bitů za náhodně zvoleným bodem křížení. Tato jednoduchá operace má však poměrně velký dopad na nově vygenerované jedince. Zde je jednoduchý příklad viz. (Obr. 11) křížení těchto dvou chromozomů (pro jednoduchost jsou chromozomy zapsány přímo jako postoupnosti celočíselných kodonů) v místě



vyznačeného dělící čarou.

Obr. 11 Křížení znázorněné přímo na programech

V místě křížení se chromozomy rozdělily na dvě části: první reprezentuje kostru nového jedince a druhá větve, které byly rodičovskému stromu odříznuty. Křížení se dokončí tak, že kostra nového jedince bude doplněna uzly a podstromy, které budou generovány použitím kodonů ze zbývající části druhého rodiče. I když se tedy jedná jen o jednobodové křížení, je jeho efekt (pokud jde o vzájemné obměny genetického materiálu) výrazně vyšší než u jednoduchého prohazování podstromů obvyklého u GP (popsáno v následující kapitole). V průměru si jedinci při křížení vymění polovinu genetického materiálu bez ohledu na jejich velikost. Gramatická evoluce byla testována na mnoha úlohách a mnoha případech předčila klasické GP.

3 GENETICKÉ PROGRAMOVÁNÍ

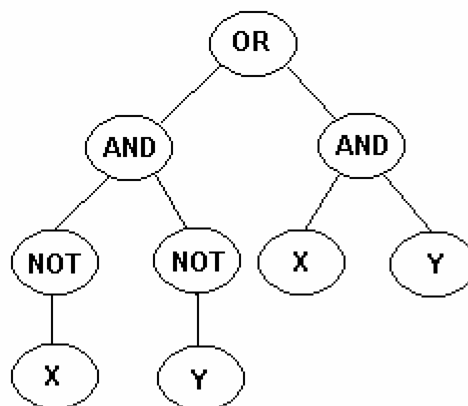
Genetické programování představuje nejstarší, tedy i první metodu symbolické regrese [6]. GP vychází v principu z genetických algoritmů, od nichž se liší ve dvou směrech:

- Účelem, pro nějž se používají. Na rozdíl od GA, jejichž základním aplikačním polem je vyhledávání hodnot nezávislých proměnných, pro něž jistá účelová funkce nabývá svého optima, je genetické programování určeno pro automatickou syntézu programů, algoritmů či rozhodovacích postupů.
- Reprezentací jedinců představujících potenciální řešení problému, který chceme automaticky hledaným postupem řešit. Na rozdíl od GA, které reprezentují jedince pomocí chromozomů ve tvaru lineárních řetězců. GP používá na místě chromozomů nejčastěji stromové struktury.

3.1 Reprezentace jedinců

Hledaným řešením v GP jsou výrazy, programy či algoritmy. Ty lze obvykle zapsat ve tvaru vhodně interpretovaných stromových struktur. Příklad vyjádření logické funkce (9) proměnných X a Y je na (Obr.12)

$$\left(\overline{X} \wedge \overline{Y}\right) \vee \left(X \wedge Y\right) \tag{9}$$



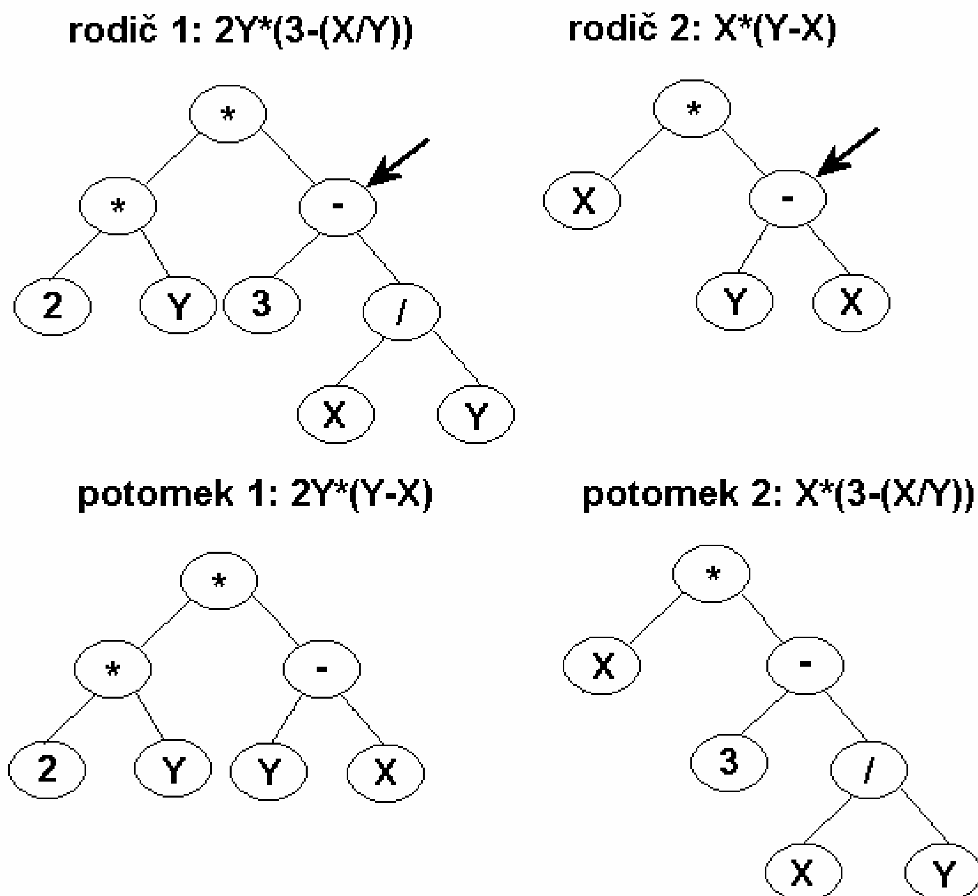
Obr. 12 Reprezentace logického výrazu(9) stromovou strukturou

Takový strom je konstruován ze dvou množin symbolů, z množiny T terminálních symbolů představujících ve stromu listové uzly a z množiny F neterminálních symbolů (funkcí)

užitých ve stromu jako vnitřní (nelistové) uzly. Z hlediska modelovaného výrazu zobrazují neterminálové prvky funkce či operace, terminální symboly pak vstupní proměnné, konstanty nebo výstupy senzorů. Terminály mohou být i funkce bez argumentů mající nějaký vedlejší efekt.

3.2 Křížení v GP

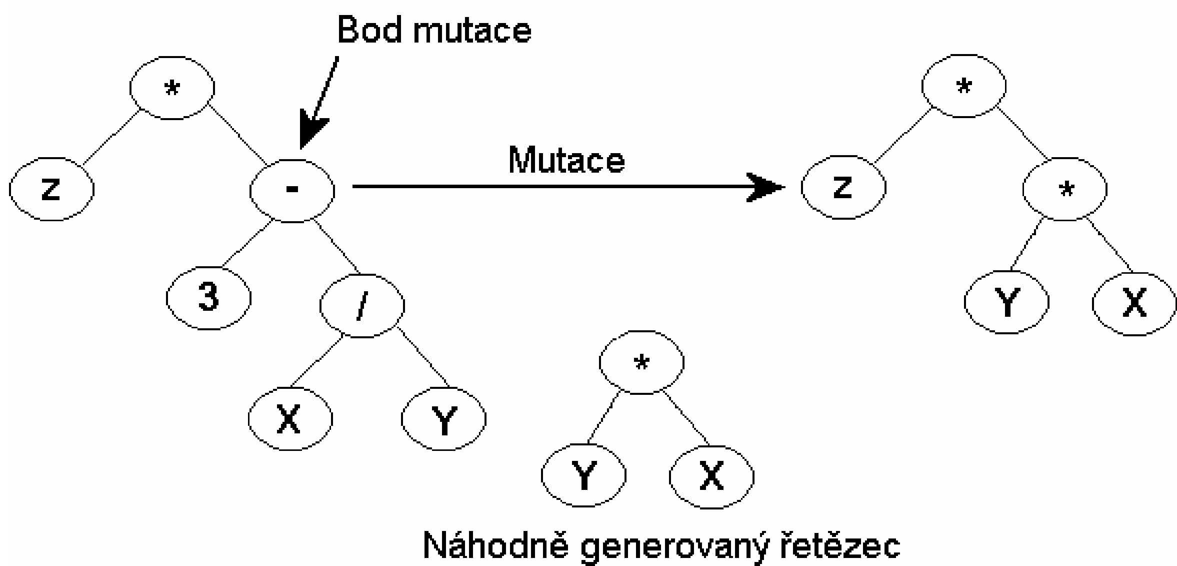
Podobně jako u GA využívá GP jako základní rekombinační operátor křížení. Do této operace opět vstupují dva jedinci a výsledkem je jeden nebo dva potomci. Standardně se postupuje tak, že v každém z rodičovských chromozomů se náhodně vybere po jednom uzlu a podstromy vycházející z těchto uzlů se vzájemně zamění. Celý postup názorně ilustruje (Obr.13), v němž náhodně vybrané uzly v rodičovských chromozomech jsou vyznačeny šipkou.



Obr. 13 Křížení stromových struktur

3.3 Mutace v GP

V GP představuje primární operaci i mutace. Ta probíhá zpravidla tak, že se ve zvoleném jedinci náhodně vybere jeden uzel a podstrom z tohoto uzlu vycházející se zcela nahradí nově vygenerovaným stromem, tedy způsobem podobným tvorbě stromů v počáteční populaci. Příklad mutace ilustruje (Obr.14), kde náhodně vybraný uzel původního jedince je opět vyznačen šipkou.



Obr. 14 Mutace stromových struktur

Genetické programování je pojmenované tímto jménem, protože je založeno na genetickém algoritmu, který je vhodný pro svou strukturu chromozomu.V GP není možno využívat jiné evoluční algoritmy.

4 EVOLUČNÍ ALGORITMY

Většina problémů inženýrské praxe může být definována jako optimalizační problém např. nalezení optimální trajektorie robota, optimální tloušťky stěny, tlakové nádoby, optimální nastavení parametrů regulátoru, optimální nastavení spotřeby paliva u automobilů atd. Příkladů lze nalézt nespočetně. Tyto řešené problémy převedeme na matematický problém daný vhodným funkčním přepisem, jehož optimalizace vede k nalezení argumentů účelové funkce. Výsledné nejlepší řešení je dáno globálním extrémem (obvykle minimem) ohodnocení účelové funkce.

Řešení takových problémů obvykle vyžaduje práci s argumenty optimalizovaných funkcí, přičemž definiční obor těchto argumentů může být různorodého charakteru jako např. obor celočíselný, reálný, komplexní, diskrétní apod. Navíc se může stát (případ od případu), že pro určité subintervaly z povoleného intervalu hodnot může příslušný argument optimalizované funkce nabývat různých typů hodnot (opět celočíselný, reálný, komplexní, diskrétní apod.). Navíc v rámci optimalizace mohou být uplatněny různé penalizace a omezení nejen na dané argumenty, ale také na funkční hodnotu optimalizované funkce. Řešení takového optimalizačního problému analytickou cestou je mnohdy možné, nicméně značně komplikované a zdlouhavé.

Pro úspěšné řešení takových problémů byla v posledních dvou desetiletích vyvinuta množina velmi výkonných algoritmů, které umožňují řešit velmi složité problémy efektivním způsobem. Tato třída algoritmů má svůj specifický název a to „evoluční algoritmy“. Tyto algoritmy jsou schopny řešit velni složité problémy tak elegantně, že se staly velmi oblíbené a používané v mnoha inženýrských oborech. Jejich výhody vedly k tomu, že se staly i nedílnou součástí výše popsaného Analytického Programování. Výhodou AP, v porovnání s GP nebo GE, je, že si uživatel na řešení svého problému může zvolit libovolný algoritmus na základě svých zkušeností s jeho výkonem.

Typickým rysem pro evoluční algoritmy je, že pracují s tzv. populacemi možných řešení, kterým se říká jedinci. Tito jedinci navzájem ovlivňují svou kvalitu na základě určitých evolučních principů v cyklech, které obvykle nesou jméno „Generace“. Cílem celého evolučního procesu je nalézt nejlepší řešení [3]. Další kapitoly stručně popisují evoluční algoritmy použité v této práci, tedy SamoOrganizující se Migrační Algoritmus (SOMA), Diferenciální Evoluce (DE), Simulované žíhaní (Simulated Annealing – SA).

4.1 SamoOrganizující se Migrační Algoritmus (SOMA)

SOMA je algoritmus existující od roku 1999. Vzhledem k tomu, že pracuje s populacemi podobně jako např. genetické algoritmy a výsledek po jednom evolučním cyklu (migračním kole) je totožný s genetickými algoritmy, lze jej řadit např. mezi evoluční algoritmy navzdory faktu, že během jeho průběhu nejsou vytvářeni noví potomci, jak je tomu u jiných EA. Mnohem přesnější je však řazení mezi algoritmy memetické [7].

Původní myšlenka, která vedla k jeho vytvoření, spočívá v napodobení chování skupiny inteligentních jedinců, kteří kooperují při řešení společního problému jako např. hledání zdroje potravy apod.

Tento algoritmus, který pracuje stejně jako ostatní evoluční algoritmy s populací jedinců, byl vyvinut na principech, které lze odpozorovat v přírodě a kterými se v sociálně-biologickém prostředí řídí inteligentní jedinci, jenž kooperují na řešení společního úkolu. Na rozdíl od ostatních evolučních algoritmů, v něm totiž neprobíhá tvorba nových řešení (jedinců, potomků) filozofií křížení rodičů, ale je založena na kooperativním prohledávání (migraci) prostoru možných řešení daného problému.

Jako příklad může sloužit chování smečky lovících vlků, včelího úlu, termitích kolonií. U těchto příkladů je společným úkolem např. hledání potravy, v rámci níž jedinci spolupracují, ale i, byť nevědomky soutěží. Ve fázi spolupráce si navzájem jednotliví jedinci sdělují jakou kvalitu hledaného momentálně nalezli a na základě toho se snaží přizpůsobovat své chování. Ve fázi soutěžení se každý jedinec snaží vyhrát nad ostatními – snaží se nalézt nejlepší zdroj potravy. Ostatní opustí své nalezené zdroje potravy a migrují směrem k jedinci s nejlepším zdrojem potravy a během této migrace se snaží nalézt ještě lepší zdroj. To se opakuje dokud se všichni nesejdou u nejvydatnějšího zdroje potravy. Na tomto silně zjednodušeném principu funguje algoritmus SOMA.

4.1.1 Parametry SOMA

Běh algoritmu SOMA, stejně jako ostatní evoluční algoritmy, je ovlivňován speciální množinou parametrů, které se dělí na dva druhy a to na parametry řídící a ukončovací. Řídící parametry jsou ty, které maní vliv na kvalitu běhu algoritmu (z hlediska hodnoty účelové funkce) a ukončovací jsou ty, které za předem nadefinovaných podmínek běh algoritmu ukončují. Všechny tyto parametry jsou voleny uživatelem.

Tab. 2 Význam parametrů SOMA

Parametr	**Doporučený rozsah**	**Poznámka**
PathLength	<1.1, 1.5>	Řídící parametr
Step	<0.11, PathLength >	Řídící parametr
PRT	<0, 1>	Řídící parametr
Dim	Dáno problémem	Počet argumentů účelové funkce
PopSize	<10, definuje uživatel>	Řídící parametr
Migrace	<10, definuje uživatel>	Ukončovací parametr
AcceptedError	< ± , definuje uživatel>	Ukončovací parametr

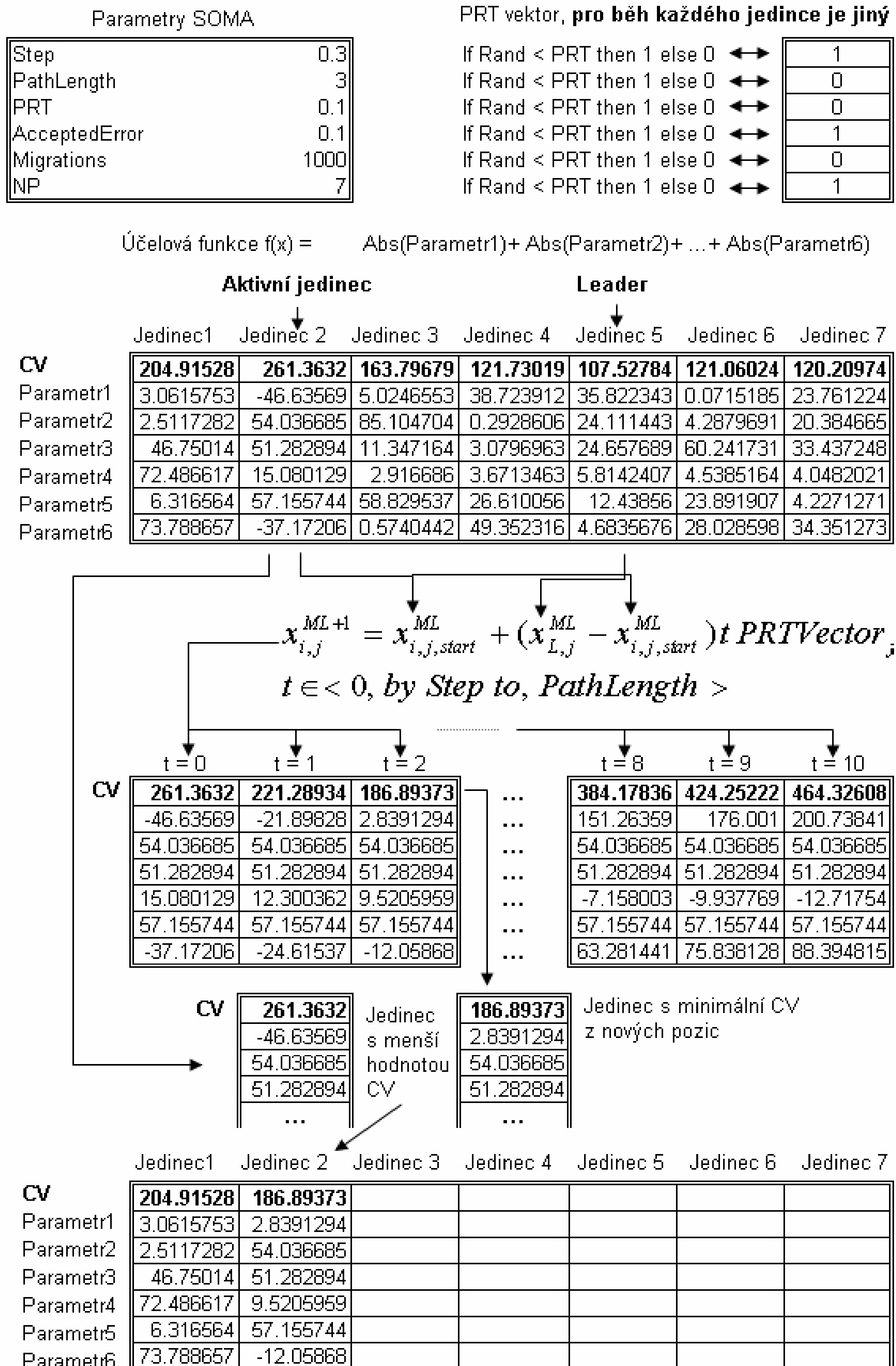
- **PathLength** ∈**<**1.1, 3**>.**Tento parametr určuje jak daleko se aktivní jedinec zastaví od tzv. vedoucího jedince. Při PathLength = 1 se aktivní jedinec zastaví na pozici vedoucího jedince, při PathLength = 2 za ním ve stejné vzdálenosti z jaké startoval, atd. Jestliže je PathLength < 1 pak dojde k tomu, že se zastaví před vedoucím jedincem, což povede k degeneraci migračního procesu, tzn. Že budou nalezeny jen lokální extrémy. Z tohoto důvodu je doporučeno mít PathLength > 1.
- **Step**∈**<**0.11, PathLength**>.** Parametr Step určuje „zrnitost“, s jakou bude mapována cesta aktivního jedince. V případě jednoduché unimodální účelové funkce (konvexní, pár lokálních extrémů, atd.), je možné použít vekou hodnotu Step pro urychlení chodu algoritmu. Jestliže není známo ani přibližně jaká geometrie reprezentuje účelovou funkci, pak je doporučeno nastavit Step na nízkou hodnotu. Prostor možných řešení pak bude prohledán podrobněji, čímž se zvýší pravděpodobnost nalezení globálního extrému. Je rovněž důležité nastavit Step tak, aby vzdálenost mezi vedoucím aktivním jedincem nebyla celočíselným násobkem parametru Step. Pokud by se tak stalo, diverzibilita populace by klesla, protože každý jedinec by mohl rychleji skončit v lokálním extrému. Proto step o velikosti 0.11 je lepší než 0.1.
- **PRT**∈**<**0, 1**>.** PRT znamená perturbaci. Podle tohoto řídícího parametru se tvoří perturbační vektor (PRTVector), který ovlivňuje to, zda se aktivní jedinec bude pohybovat přímo k vedoucímu jedinci či ne. Je to jeden z nejdůležitějších parametrů s nejvyšší citlivostí. Jeho optimální hodnota je okolo 0.1. Pokud hodnota PRT narůstá, pak konvergence SOMA k lokálním extrémům silně vzrůstá. V případě nízkodimenziální funkce a vysokého počtu jedinců je možné nastavit PRT na 0.7-1.0. Jestliže je PRT =1 pak stochastická složka chování SOMA zaniká a algoritmus se

chová jen a pouze podle deterministických pravidel, což znamená, že v případě multimodální účelové funkce se hodí pouze k lokální optimalizaci.

- **Dim** je parametr, který udává počet optimalizovaných proměnných daného problému neboli počet argumentů účelové funkce. Je dán samotným problémem a může být změněn pouze tehdy, když je předefinován celý problém
- **PopSize** ∈ <10, dáno uživatelem>**.** Tento řídící parametr určuje kolik jedinců bude tvořit populaci. Obecně může být nastaven v rozsahu 0.2 až 0.5 parametru D v případě, že tento je vysoký. Například když účelová funkce má 100 argumentů, pak se populace může skládat z cca 30–50 jedinců.
- **Migrace** ∈ <10, dáno uživatelem>**.** Tento parametr je ekvivalentem parametru Generace z jiných evolučních algoritmů. V podstavě udává, kolikrát se populace jedinců přeorganizuje – obrodí. Termín Migrace byl zvolen na základě filozofie SOMA algoritmu. Je to ukončovací parametr.
- **AcceptedError** ∈ <± libovolný, dáno uživatelem>**.** Ukončovací parametr AcceptedError definuje, jaký maximální rozdíl mezi nejhorším a nejlepším jedincem v aktuální populaci je povolen. Jestliže je rozdíl menší nežli AcceptedError pak je běh algoritmu ukončen. Je doporučeno používat malé hodnoty jako např. AcceptedError =1. Pokud je tento parametr nastaven na příliš vysokou hodnotu, tak se algoritmus zastaví dříve, nežli populace stačí lokalizovat globální extrém. Pokud se nastaví na malou či nulovou hodnotu, tak se nestane nic horšího než to, že se algoritmus zastaví až po vyčerpaní všech migračních kol.

4.1.2 Princip SOMA

Princip SOMA algoritmu je graficky znázorněn na (Obr.15). Každý jedinec je ohodnocen účelovou funkcí a je zvolen Leader(jedinec s nejlepší hodnotou účelové funkce) pro následující migrační kolo. V tomto okamžiku se začnou ostatní jedinci pohybovat směrem k Leaderovi pomocí skoků, jejichž velikost je dána parametrem Step. Po každém skoku si každý jedinec na takto získané pozici přepočítá svou hodnotu účelové funkce a pokud je lepší nežli předchozí, tak si ji zapamatuje. Pohyb jedince pokračuje tak dlouho, dokud není dosaženo pozice, jež je dána parametrem PathLength. Každá nová pozice je vypočítána rovnicemi (10,11), kde „r“ a „r_0“ reprezentují jedince.



Obr. 15 Princip SOMA – převzato [7]

Po ukončení běhu se jedinec vrací na pozici, kde byla nalezena nejlepší hodnota účelové funkce během jeho cesty. To má za následek ten efekt, že po skončení aktuálního migračního kola jsou všichni jedinci, mimo Leadera, přemístěni.

$$\vec{r} = \vec{r}_0 + \vec{m}.t.PRTVector \quad (10)$$

Podrobněji:

$$x_{i,j}^{ML+1} = x_{i,j,start}^{ML} + \left(x_{L,j}^{ML} - x_{i,j,start}^{ML}\right)t.PRTVector_j \quad (11)$$

Kde:

$x_{i,j}^{ML+1}$ - hodnota i-jedince j-parametru, v kroku t v dalším migračním kole

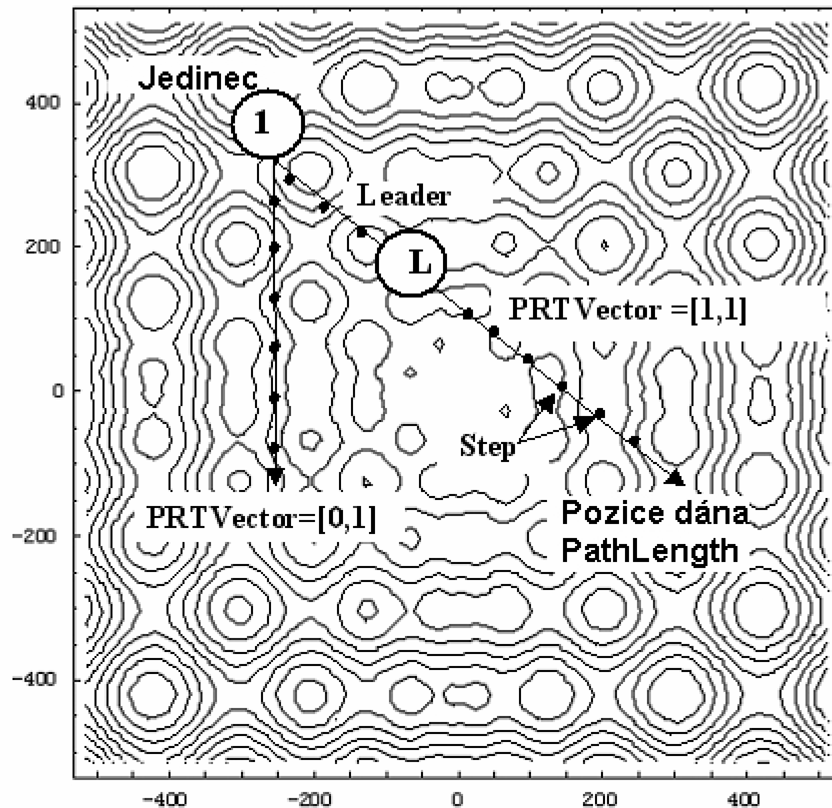
$x_{i,j,START}^{ML}$ - hodnota i-jedince j-parametru, počáteční pozice v aktuálním migrační kole

$x_{L,j}^{ML}$ - hodnota Leaderova j-parametru v aktuálním migračním kole

t - hodnota parametru Step

$PRTVector_j$ - je vektor nul a jedniček závislý na PRT. Jestliže je náhodné číslo z intervalu <0, 1> menší než je hodnota PRT, pak se uloží do PRTVectoru 1, jinak 0.

Jak je z rovnice vidět, tak PRTVector ovlivňuje pohyb jedince tak, že pokud má všechny prvky rovny 1, tak aktivní jedinec putuje rovnou podle směrového vektoru k Leaderovi. Pokud však má některé prvky rovny 0, pak je druhý člen v obou rovnicích anulován a daná souřadnice se nemění – je „zmražená“. Díky tomu se jedinec pohybuje v N-k rozměrném prostoru (Obr.16).



Obr. 16 „PRTVector“v grafickém detailu

4.1.3 Variace algoritmu SOMA

- **AllToOne** (všichni k jednomu). Tato variace byla popsána výše a je možné ji považovat ze základní variaci algoritmu SOMA.
- **AllToAll** (všichni ke všem). Zde neexistuje leader. Každý jedinec migruje ke všem ostatním.
- **AllToOneRand** (všichni k náhodnému). Každý jedinec migruje jen k jednomu náhodně zvolenému jedinci.
- **AllToAllAdaptive** (adaptivně všichni ke všem). Tato variace je založená na strategii AllToAll s rozdílem, že aktuálně migrující jedinec se nepřesouvá do nové pozice až po migraci směrem ke všem jedincům, ale bezprostředně po dokončení své migrace ke každém z jedinců a v migraci k dalším jedincům pokračuje už z této nové polohy

4.2 Diferenciální evoluce

Diferenciální evoluce je další z poměrně nových typů evolučních algoritmů (od r. 1995). Jeho schéma je dost podobné GA, s nimiž má několik společných rysů, jako je např. tvorba potomků nebo používání tzv. generací. Tento algoritmus má ty zvláštnosti, že ke tvorbě nového jedince je potřeba celkem 4 rodičů, kteří vytvoří potomka, jenž nakonec soupeří o místo v nové populaci. V procesu tvorby nového potomka je uplatněna náhoda pomocí křížící konstanty CR a pomocí tzv. šumového vektoru. Díky tomu je diferenciální evoluce poměrně robustní algoritmus s poměrně vysokou diverzibilitou.

4.2.1 Parametry DE

Činnost a kvalita diferenciální evoluce je ovlivněna jejími řídícími parametry stejně jako u ostatních evolučních algoritmů. Jejich označení a význam je viz. (Tab.3).

Tab. 3 Parametry DE

Parametr	Doporučený rozsah	Poznámka
CR	<0, 1>	Řídící parametr
NP	<2D, 100D >	Řídící parametr
F	<0, 2>	Řídící parametr
Generations	Uživatel	Ukončovací parametr
Dim	Dáno problémem	Počet argumentů účelové funkce

- **CR** $\in$<0, 1>. Jde o tzv. práh křížení. V případě , že se jedná o funkci, která je separabilní, pak je doporučeno nastavit tento parametr na hodnoty blízké 0. V opačném případě jsou výhodné hodnoty, které se blíží 1. V případě, že se CR nastaví na 0 dojde k tomu, že se mutace nedostane do zkušebního jedince, který díky tomu bude čistou kopií aktuálního (čtvrtého rodiče). Vývoj evoluce se pak zastaví. V případě, že CR bude nastavena na 1, bude zkušební jedince tvořen pouze ze tří náhodně vybraných rodičů – jedinců z populace a diferenciální evoluce se bude spíše podobat náhodnému hledání nežli evolučnímu algoritmu (všichni tři jedinci, byť evolučně vytvoření, jsou náhodně vybráni bez ohledu na jejich kvalitu). Je proto vhodné, by CR nikdy nenabývalo těchto hodnot.
- **NP**$\in$<2D, 100D>. Jedná se o parametr udávající velikost populace. Hodnoty tohoto parametru jsou získány pouze zkušeností s tímto algoritmem. Jediné, co se dá

s určitostí říci je, že NP by neměl být menší než 4. to je minimální velikost populace, při které diferenciální evoluce ještě pracuje.

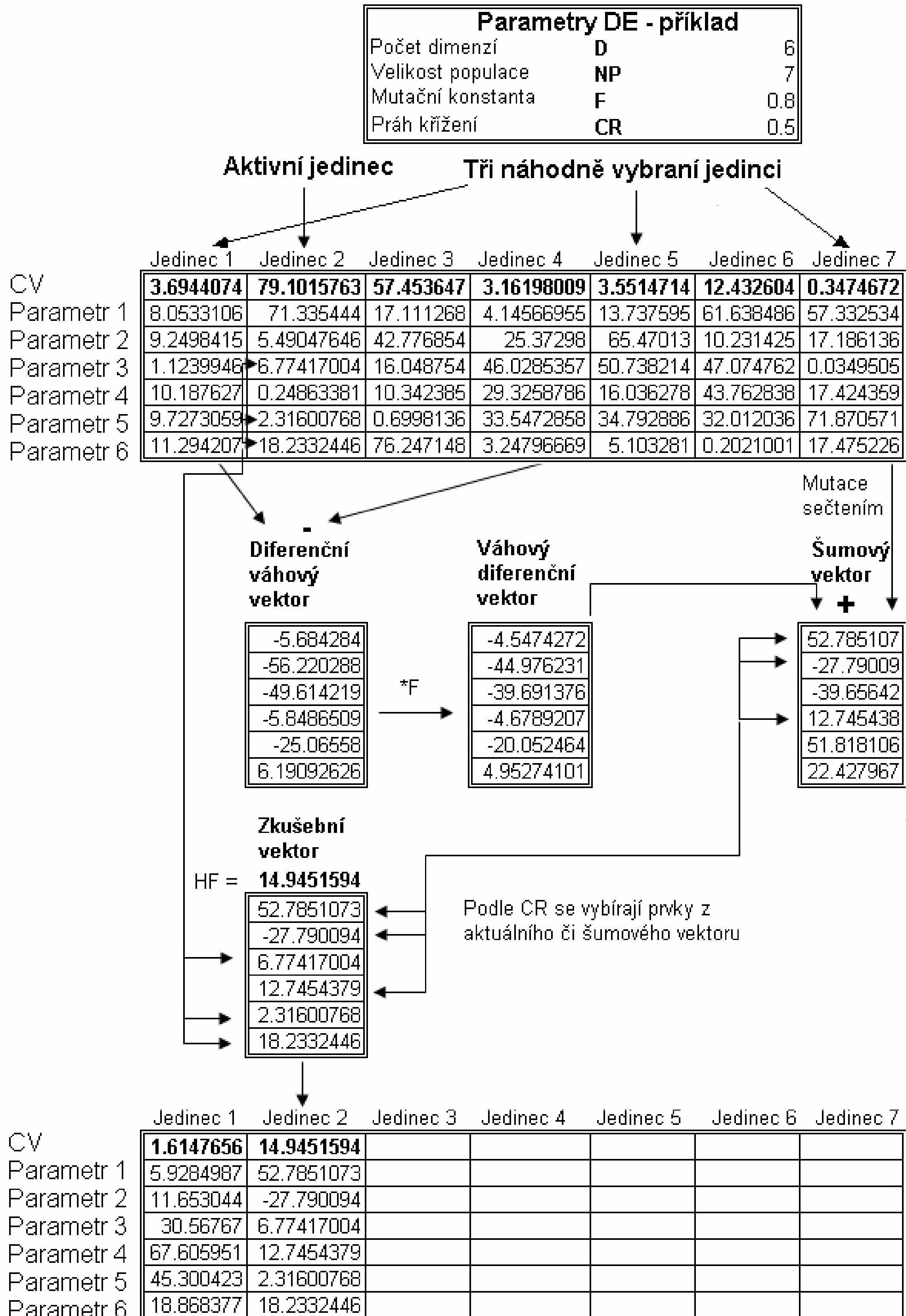
- **F** ∈ <0, 2>. Mutační konstanta je poslední řídící parametr diferenciální evoluce
- **Generations** > 0. Udává počet evolučních cyklů, tzv. generací, během nichž se celá populace vyvíjí.
- **Dim** – dimenze problému. Jde o počet argumentů účelové funkce. Parametr „d“ je tedy dán řešeným problémem a lze ho změnit pouze reformací problému.

4.2.2 Princip DE

Cílem DE je v cyklech zvaných „generace“ vyšlechtit co nejlepší populaci jedinců ve smyslu hodnot účelové funkce, jenž je spojena s každým jedincem. Během každé generace se provádí tyto kroky:

Nejprve se stanoví parametry (F, CR, NP, Generations, Dim). Poté je nutné nadefinovat prototyp jedince (Specimen) – tj. z jakých typů čísel se budou skládat jedinci, např. reálných, celočíselných atd. Po sestavení Specimen se vytvoří populace vygenerováním množiny jedinců podle prototypového vektoru. U každého jedince se musí počítat s jedním prvkem navíc a tím je hodnota účelové funkce. Během generace se provádí ještě cyklus, který zabezpečuje postupné evoluční šlechtění každého jedince z populace viz. (Obr.17).

V tomto cyklu se postupně vybírá jeden jedinec (aktivní jedince, cílový vektor) za druhým až do konce populace (tím je ukončena jedna generace) a pro každého z nich je proveden evoluční cyklus, v němž je prováděna mutace a křížení, tj. náhodně se zvolí tři další různé vektory (jedinci) z populace. První dva se od sebe odečtou získá se tzv. diferenční vektor. Ten se vynásobí mutační konstantou F, která jej tím pádem změní , a získá se váhový diferenční vektor. Ten se přičte k třetímu náhodně vybranému vektoru a získá se tzv. šumový vektor. Poté si připraví tzv. „zkušební vektor“ a z cílového a šumového vektoru se bere postupně jeden prvek za druhým (první z obou, druhý z obou…) a pro takto vybranou každou dvojici se generuje náhodné číslo v rozsahu 0 – 1 a porovnává se s konstantou CR. Pokud je toto číslo menší než CR, pak se do příslušné pozice ve zkušebním vektoru umístí prvek z vektoru šumového a v opačném případě z vektoru cílového.



Obr. 17 Princip DE – převzato z [7]

Tak se získá zkušební vektor, jehož hodnota účelové funkce se porovná s hodnotou účelové funkce cílového vektoru. Na pozici cílového vektoru v nové populaci je vybrán ten vektor – jedinec, který má hodnotu účelové funkce lepší [7].

Diferenciální evoluce je ukončena pouze tehdy, provede-li se uživatelem zadaný počet generací. Jiný ukončovací parametr tento algoritmus nemá.

4.2.3 Varianty DE

V praxi existuje 10 možných variant DE, princip je stejný, liší se pouze ve způsobu výpočtu šumového vektoru [7].

4.3 Simulované žíhání

Metoda simulovaného řihaní patří mezi stochastické optimalizační algoritmy, které, jak už naznačuje jejich název, mají základ ve fyzice (na rozdíl od jiných stochastických optimalizačních algoritmů, které mají svůj základ většinou v biologii).

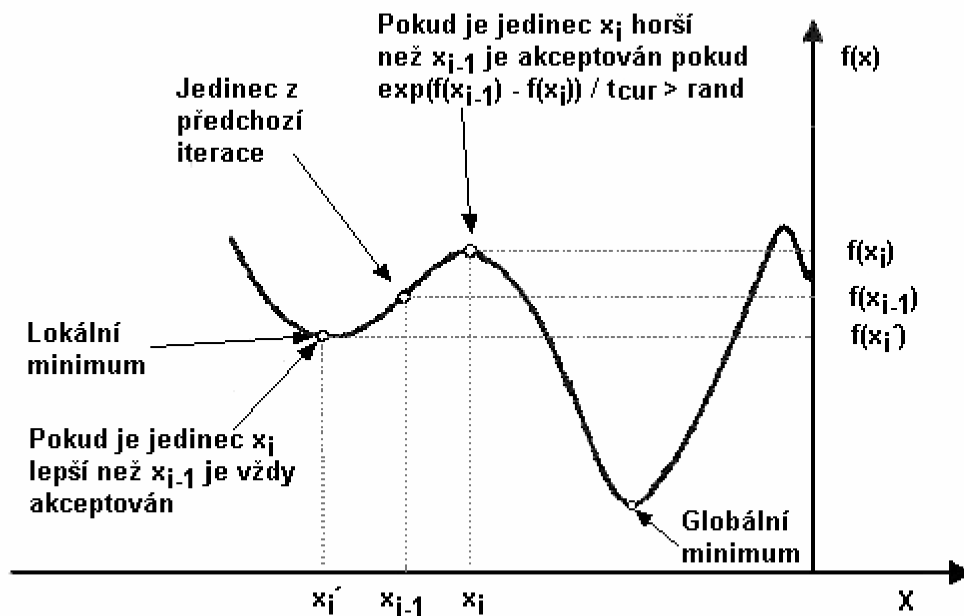
Název metody je odvozen z představy simulování fyzikálních procesů probíhajících při odstraňování defektů krystalové mřížky kovu, které se projevují pnutím v materiálu. Při žíhání se kov zahřeje na tak vysokou teplotu, při které atomy v krystalové mřížce mohou překonat lokální energetické hladiny, a defekty krystalové mřížky mají velkou pravděpodobnost zániku. Po dosažení tohoto stavu se kov pomalu ochlazuje (žíhá), a tím se atomy dostanou do rovnovážných poloh s nejmenší energií. Při konečné teplotě žíhání (podstatně nižší, než byla počáteční) jsou všechny atomy kovu v rovnovážných polohách a těleso neobsahuje žádné vnitřní defekty ani pnutí.

V simulovaném žíhání je krystal reprezentován jedincem x. Ke každému jedinci může být přiřazena funkční hodnota $f(x)$, která představuje energii krystalu. Analogií minimalizace energie krystalu je v metodě simulovaného žíhání minimalizace funkce $f(x)$ a pomalé ochlazováni představují postupné iterace. V každé iteraci je původní jedinec x_{i-1} nahrazen novým, náhodně vygenerovaným jedincem xi.

Pravděpodobnost nahrazení (12), kde t_{cur} je parametr vyjadřující teplotu v daném kroku.

$$P(x_{i-1} \rightarrow x_i) = \left\{ 1, \quad e\frac{f(x_{i-1}) - f(x_i)}{t_{cur}} \right\}, \qquad (12)$$

Jestliže jedinec x_i má menší, nebo stejnou funkční hodnotu jako původní jedinec x_{i-1}, je automaticky akceptován do další iterace. V opačném případě je pravděpodobnost akceptování jedince *xi* menší než jednotková, ale i v tomto případě má nový jedinec šanci postoupit do další iterace (Obr.18), kde *rand* je náhodně, s normálním rozložením, vygenerovaná konstanta [8].



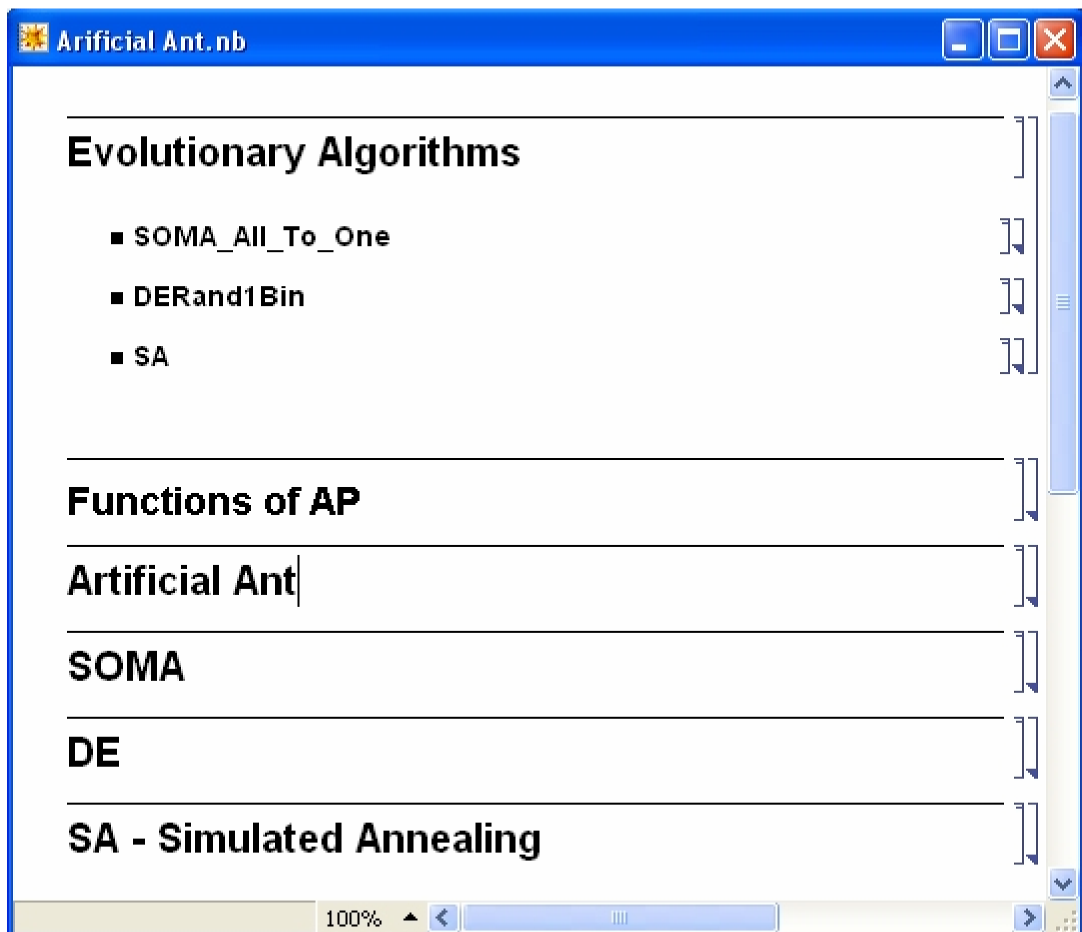
Obr. 18 Grafické znázornění postupu jedince

II. PRAKTICKÁ ČÁST

5 DEMONSTRACE VYUŽITÍ AP PRO NÁVRH OPTIMÁLNÍ TRAJEKTORIE ROBOTA

Obě dvě úlohy viz. níže (úloha „umělého mravence“ a „zahradní sekačka“) použité pro demonstraci AP jsou naprogramovány v programu Mathematica od společnosti Wolfram (www.wolfram.com).

Program Mathematica pracuje s soubory, které se nazývají notebooky. Notebook obsahuje buňky do kterých se zapisují příkazy. Každá funkce musí být v notebooku nadefinována před svým spuštěním. Na (Obr.19) je ukázka notebooku pro úlohu umělého mravence, který obsahuje šest základních buněk.



Obr. 19 Notebook pro úlohu umělého mravence

První sekce pojmenovaná Evolutionary Algorithms obsahuje buňky, v nichž jsou naprogramované EA (SOMA, DE, SA), které AP využívá pro svoji činnost. Druhá sekce pod názvem Functions of AP obsahuje funkce AP. Ve třetí sekci je naprogramována úloha umělého mravence. A v posledních třech sekcích jsou buňky, které umožňují spouštět dané

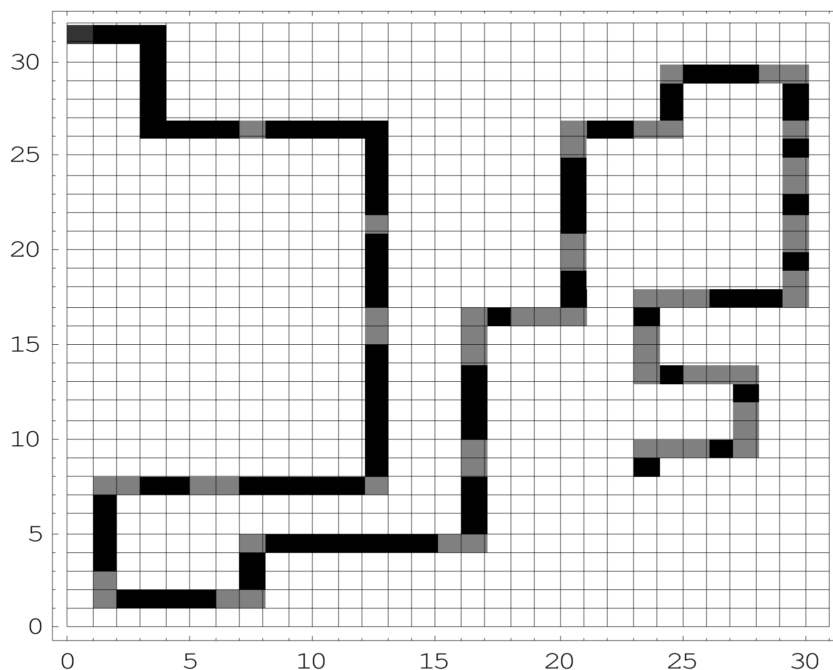
EA pod AP. První dvě sekce byly převzaty i s původními anglickými názvy funkcí a proměnných a doplněny o zbylé sekce ve stejném stylu, pro možnost použití tohoto notebooku na mezinárodních konferencích.

5.1 Úloha tzv. umělého mravence (Artificial Ant)

Úloha tzv. umělého mravence byla vybrána z [1], kde Koza řeší stejnou úlohu pomocí GP. Cílem této úlohy je, aby mravenec prošel definovanou trať (tzv. stezku Santa Fe) a snědl všechny návnady, které jsou na ni rozmístěny.

5.1.1 Stezka „Santa Fe“

Mravenec se pohybuje v prostoru definovaném formou pravoúhlé mřížky o rozměru 32x31 políček s počátkem na pozici (1, 32) natočen čelem k východu. V tomto prostoru je nepravidelně vyznačena stezka „Santa Fe“ pomocí 89 návnad určených pro mravence. Pro zajištění, aby se mravenec nedostal mimo tento vymezený prostor, je životní prostor mravence chápán jako „koule“, tj. např. vykročení mimo pole směrem dolů znamená vstup do pole opět shora (vertikální pozice se počítá jako mod 32 a horizontální jako mod 31).



Obr. 20 Stezka Santa Fe

Černá políčka na (Obr.20) představují návnadu pro mravence. Bílá představují prázdná políčka bez návnady. A šedá políčka jsou v podstatě stejná jako bílá, jen s tím rozdílem, že

reprezentují překážku (políčka bez návnady na cestě), pro větší zřetelnost vyznačení stezky byla pro ně použita šedá barva. Pokud by na stezce nebyly tyto mezery, mravenec by šel přímo po cestě ve stylu start – cíl. Když by měl návnadu před sebou, tak by šel o políčko v před a snědl by ji. Když ne tak by se otáčel dokola a zkoumal, kde je a v tomto cyklu by pokračoval dokud by nesnědl poslední návnadu. Ale v skutečném světě roboti mají překážky bránící v jejich pohybu. Proto i mravenec se musí vyrovnat s těmito nástrahami:

- Změna směru (pozice [4, 32] Obr.20)
- Jednoduchá mezera (pozice [8, 27])
- Dvojnásobná mezera(pozice [13, 16], [13, 17])
- Jednoduchá mezera v rohu (pozice [13, 8])
- Dvojnásobná mezera v rohu, tzv. krátký tah koněm (pozice[1, 8], [2, 8])
- Trojnásobná mezera, tzv. dlouhý tah koněm (pozice [17, 15], [17, 16], [17, 17])

5.1.2 Množina funkcí použitých k pohybu mravence v daném prostoru

Mravenec má pouze jednoduchý senzor, kterým dokáže poznat, zda na políčku bezprostředně před ním kousek potravy je, nebo není. Kromě toho dokáže udělat následující jednoduché akce:

- Krok v před a sníst potravu(je-li tam)
- „vlevo bok“
- „vpravo bok“

Množina terminálů je tedy dána předpokládanými schopnostmi mravence (jsou to vlastně příkazy pro motorickou sekci umělého živočicha), které umožňují pohyb mravence. Všechny tyto terminály jsou 0–argumentové:

$$GFS_{0arg} = \{Left, Right, Move\} \quad (13)$$

Kde:

- **Left** = je terminál pro otočení mravence vlevo
- **Right** = je terminál pro otočení mravence vpravo

- **Move** = je terminál pro posunutí mravence o jedno políčko vpřed ve směru ke kterému je natočen.

Na (Obr.21) je ukázka zápisu terminálu Move v Mathematice (pro kratší zápis výstupního programu je název MoveDirect zkrácen na Move, toto platí i pro ostatní názvy funkcí a terminálů). Kde vstupy jsou pozice mravence (osaxx, osayy) a směr, ke kterému je mravenec čelem (WhereIsFace je pomocná proměnná, do které se ukládá aktuální směr natočení anglickými zkratkami světových stran). Poté program zkoumá, jestli se mravenec při svém posunu nedostane mimo vymezený prostor:

- pokud ano, funkce modulo jej zas vrátí zpět do vymezeného prostoru
- pokud ne, tak se mravenec posune o jedno políčko vpřed

Výsledkem je, že se nová pozice mravence uloží. Pokud obsahovala návnadu, tak se sníží hodnota účelové funkce zavoláním funkce FoodFunction (popsáno níže) a terminál vrátí hodnotu své nově nabité pozice.

```
MoveDirect[{osaxx_, osayy_}, WhereIsFace_] :=
 Module[{}, osaxx1 = osaxx; osayy1 = osayy;
  If[WhereIsFace === {Ea}, osayy1 = If[Mod[osayy + 1, 31] === 0, 31, Mod[osayy + 1, 31]];,
   If[WhereIsFace === {We}, osayy1 = If[Mod[osayy - 1, 31] === 0, 31, Mod[osayy - 1, 31]];,
    If[WhereIsFace === {No}, osaxx1 = If[Mod[osaxx + 1, 32] === 0, 32, Mod[osaxx + 1, 32]];,
     osaxx1 = If[Mod[osaxx - 1, 32] === 0, 32, Mod[osaxx - 1, 32]];]]];
  AppendTo[PositionList, {osaxx1, osayy1}]; AppendTo[PositionListAll, {osaxx1, osayy1}];
  Pozice = {osaxx1, osayy1}; Return[{{osaxx1, osayy1}, FoodFunction[Pozice]}]]
```

Obr. 21 Terminál Move naprogramovaný v buňce programu Mathematica

Aby bylo možno zpětně realizovat průchod mravence po stezce „Santa Fe", jsou všechny pozice mravence při průchodu stezkou ukládány do souboru pod názvem PositionList. Pokud bychom, ale chtěli simulovat chování mravence při průchodu stezkou, byl by tento soubor nedostačující. Pro tyto účely slouží soubor pod názvem PositionListAll, ve kterém se ukládají nejen pozice, ale i všechna natočení, které mravenec při svém průchodu udělal.

Pro úspěšné splnění požadovaného úkolu je představený soubor terminálů nedostatečný. Je nezbytné použít i funkce, proto využíváme 2 a 3-argumentové funkce:

$$GFS_{2arg} = \{IfFoodAhead, Prog2\} \tag{14}$$

$$GFS_{3arg} = \{Prog3\} \tag{15}$$

Kde:

- **IfFoodAhead =** je rozhodovací funkce, mravenec kontroluje pole před sebou a pokud je tam návnada, tak vykoná argument na pozici pravda (true), jinak vykoná argument na pozici nepravda (false).
- **Prog2,Prog3** = jsou v principu totožné funkce, které se liší jen počtem argumentů. Prog2 je 2-argumentová funkce a Prog3 je 3-argumentová. Slouží k provedení dvou (nebo tří) příkazů najednou, v podstatě se jedná o rozvětvení programu (Obr.22).

```
Program2[fce__] := List[fce]
Program3[fce__] := List[fce]
```

Obr. 22 Zápis funkce Prog2,Prog3 v Mathematice

Tyto dvě funkce byly definované již Kozou, v AP jsou však nezbytné kvůli struktuře generujícího programu. Ze zápisu na Obr.22, je zřejmé že obě tyto funkce neslouží k pohybu mravence, ale jejich význam spočívá v rozvětvení výsledného funkcionálu.

5.1.3 Ohodnocení účelové funkce

Cílem mravence je sníst všechnu potravu na stezce, na které je 89 návnad v rozumném počtu kroků. To znamená, že mravenec má k dispozici omezený počet kroků, konkrétně to je 600 kroků, potom končí. Mravenec se chová tak, že cyklicky opakuje „svůj program“, dokud nevyužije počet zadaných kroků, nebo nepozře všech 89 návnad. Omezení na počet kroků se zavedlo ze dvou důvodů:

- K zamezení nekonečnému zacyklení (mravenec se točí na místě)
- Vyloučení těch řešení, která reprezentují náhodné nebo systematické prohledávání všech 992 políček

Kvalita řešení je dána počtem kroků potřebných k spořádání jistého množství návnad (v našem případě tedy 89). Tato hodnota představuje tzv. hrubé ohodnocení (raw fitness). Hodnota účelové funkce (16) je počítána z rozdílu tohoto hrubého ohodnocení a počtu snězených návnad, které snědl mravenec během aktuálně vygenerované stezky viz (Obr.23).

$$CV = 89 - NumberFood \tag{16}$$

NumberFood – počet snězených návnad během jedné vygenerované stezky

Jediný zájem je, aby mravenec našel ve vymezeném počtu kroků všechnu potravu a je jedno, jestli ji najde za 150 kroků nebo za 600 kroků. Nehledáme toho nejrychlejšího mravence.

```
■ Cost Function

NumberOfSteps = 600.

600.

CostValue[fce_] :=
 Module[{cv1, cv22, radim},
  RetFce = fce[[1]];
  funkce1 = RetFce /. {Hold[Program2] → Prog2,
     Hold[Program3] → Prog3,
     Hold[MoveDirect[Pozice, WhereIsFace]] → Move,
     Hold[TurnRight[WhereIsFace]] → Right,
     Hold[TurnLeft[WhereIsFace]] → Left,
     Hold[IFA] → IfFoodAhead};
  While[((Length[PositionListAll] ≤ NumberOfSteps)),
   AppendTo[vysl, ReleaseHold[RetFce]];
   If[NumberFood == 0, Break[]]];
  cv2 = NumberFood;
  Return[cv2]]
```

Obr. 23 Ukázka ošetření ohodnocení účelové funkce v Mathematice

Slovní popis ohodnocení CV je následující. Vstupem do funkce je náhodně pomocí AP vygenerovaný jedinec, později je to jedinec šlechtěný pomocí některého evolučního algoritmu. Názvy funkcí a terminálů jedince se převedou na zjednodušený slovní popis (v obrázku vyznačeno červenou barvou), aniž by se program spustil.

Potom se spustí cyklus While, který spustí program (mravenec se začne pohybovat dle předpisu po vymezeném prostoru) a ten probíhá dokud počet kroků nepřesáhne mez danou hodnotou NumberOfSteps nebo dokud účelové funkce nedosáhne 0. Paralelně se při každém kroku mravence spouští pomocná funkce FoodFunction viz (Obr.24) která zjišťuje, jestli mravenec nenarazil na potravu. Jestliže ano sníží hodnotu NumberFood , jinak ji ponechá. Hodnota NumberOfFood je na počátku rovna 89.

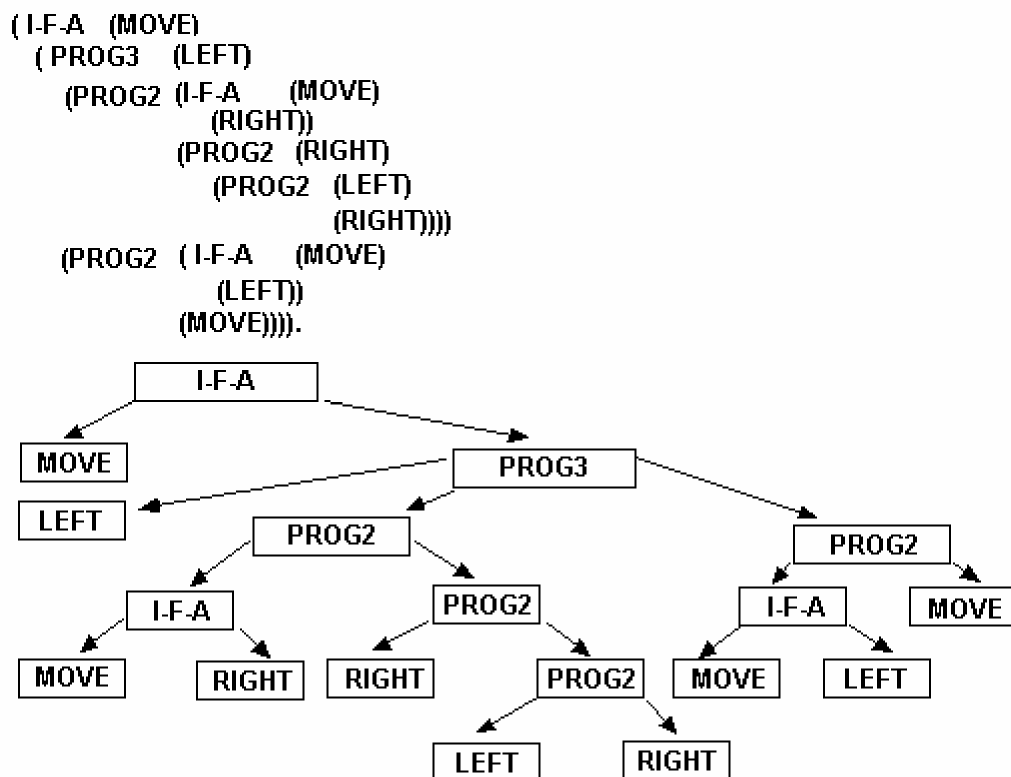
```
NumberFood = Count[Flatten[SantaFe2], 1]

89

FoodFunction[Pozice_] :=
 If[SantaFe2[[Pozice[[1]], Pozice[[2]]]] === 1,
  SantaFe2[[Pozice[[1]], Pozice[[2]]]] = 0;
  NumberFood = NumberFood - 1, NumberFood = NumberFood]
```

Obr. 24 Pomocná funkce FoodFunction

Koza ve své knize [4] uvádí jako postačující interval mravence 400 kroků. A tvrdí, že v programu viz. (Obr.25) spotřebuje pro snězení všech 89 návnad méně než 400 kroků.



Obr. 25 Stromová struktura Kozova programu pro optimální procházení mravence stezkou „Santa Fe“

Ale Mařík ve své knize [6] píše, že lze prokázat pomocí jednoduché simulace, že jich potřebuje 545 a i po vypuštění dvou zbytečných operací (Prog2(Left,Right)), by bylo potřeba 404 kroků. Tato simulace byla odzkoušena (viz. příloha PI) a zjistilo se, že mravenec potřebuje dokonce 551 kroků pro snězení všech 89 návnad při průchodu stezkou „Santa Fe“ pomocí tohoto programu. Slovní popis programu zní takto. Pokud mravenec na políčku

před sebou vidí návnadu, tak se posune vpřed a návnadu sní. Jinak se otočí doleva a podívá se, je-li tam něco k jídlu. Pokud ano, tak opět postoupí vpřed. Pokud¨ne, tak se otočí dvakrát doprava, takže se dívá vpravo od původního směru. Pokud vidí návnadu, tak udělá krok vpřed. Pokud ani v jednom směru nebyla potrava, tak se otočí do původního směru a postoupí o krok vpřed. A cyklicky opakuje program.

5.1.4 Použité evoluční algoritmy

Úloha se testovala pomocí SOMA, DE a SA algoritmu. Pro SOMA byla zvolena varianta ALLToOne a pro DE varianta DE/rand/1/bin, která je považována za nejlepší verzi diferenciální evoluce. Simulace byly časově náročné průměrně zabraly 1 – 3 dny na počítači: Intel Pentium 4 CPU 3.00 GHz , 1GB RAM. Díky tomu, že byly k dispozici tři počítače a nejrychlejší simulace trvaly jen několik hodin, bylo možno provést dohromady 125 simulací. Pokud by každá z simulací trvala tři dny a byl by použit jen jeden počítač, zabralo by testování 375 dní. Aby se mohly porovnat výsledky simulací těchto evolučních algoritmů, byly parametry nastaveny tak, aby počet ohodnocení účelové funkce byl pro všechny algoritmy stejný.

Nastavení parametrů SOMA je zobrazeno v (Tab. 4). Kde počet ohodnocení účelové funkce je roven:

$$((PopSize-1)*PathLength*Migrations)/Step = 135682 \quad (17)$$

Tab. 4 Nastavení parametrů SOMA

Parametr	Hodnota
PathLength	3
Step	0.22
PRT	0.21
PopSize	200
Migrations	50
MinDiv	- 0.1
Individual Length	50

- **Individual Length** tento parametr je stejný pro všechny algoritmy a udává nám maximální povolenou délku jedince (maximální počet do sebe vnořených funkcí a terminálů). Jeho hodnota byla nastavená na 50, aby nevznikaly zbytečně velké programy.

Nastavení parametrů DE je zobrazeno v (Tab.5). Počet ohodnocení účelové funkce je roven:

$$NP*Generations = 140\ 000 \quad (18)$$

Tab. 5 Nastavení parametrů DE

Parametr	**Hodnota**
NP	200
F	0.8
CR	0.2
Generation	700
Individual Length	50

Nastavení parametrů SA viz. (Tab.6), kde počet ohodnocení účelové funkce je hodnota:

$$MaxIter*MaxIterTemp = 136\ 618 \quad (19)$$

Tab. 6 Nastavení parametrů SA

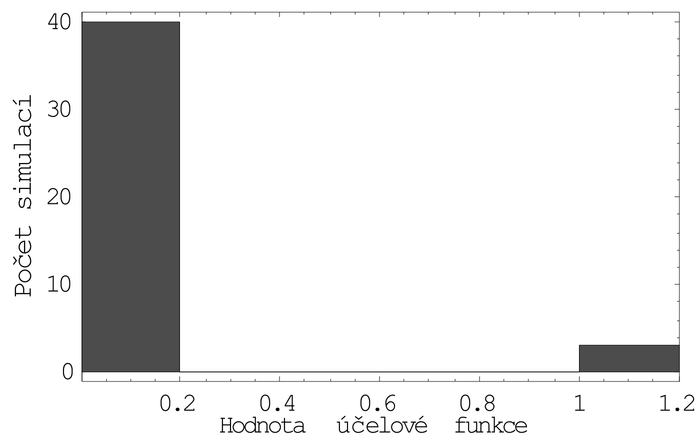
Parametr	**Hodnota**
T	10 0000
T_{min}	0.000 01
α	0.986
MaxIter	1 500
MaxIterTemp	93
Individual Length	50

5.1.5 Dosažené výsledky a jejich porovnání

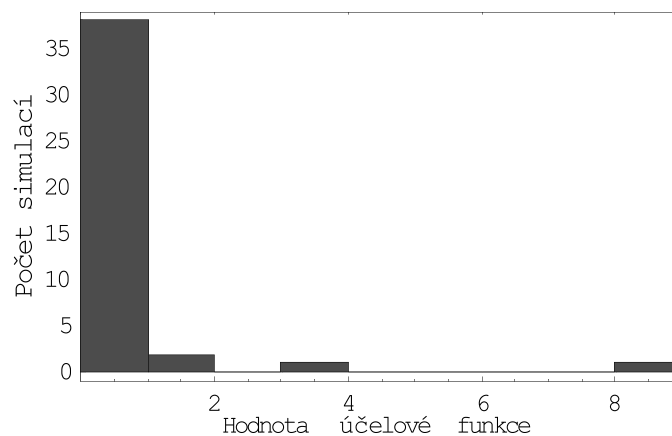
Pro SOMA algoritmus bylo provedeno 43 simulací z kterých bylo 40 úspěšných, DE simulací bylo 42 z toho 38 úspěšných a SA 40 z toho 14 úspěšných, viz. (Tab. 7). Již z těchto výsledků je zřejmé, že SA algoritmus není tak výkonný jako SOMA nebo DE. Na (Obr.26 - 28) jsou graficky znázorněny histogramy úspěšnosti simulací, tj. počet simulací odpovídající hodnotě účelové funkce.

Tab. 7 Přehled simulací

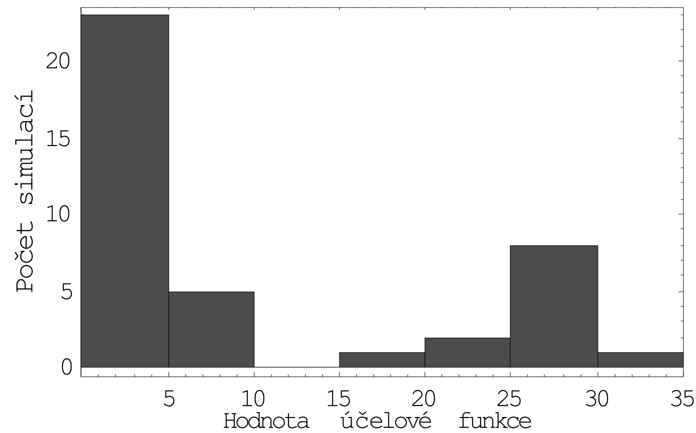
	Použité algoritmy		
	SOMA	DE	SA
Počet simulací	43	42	40
Úspěšné	40	38	14
Neúspěšné	3	4	26



Obr. 26 Histogram úspěšnosti SOMA



Obr. 27 Histogram úspěšnosti DE



Obr. 28 Histogram úspěšnosti SA

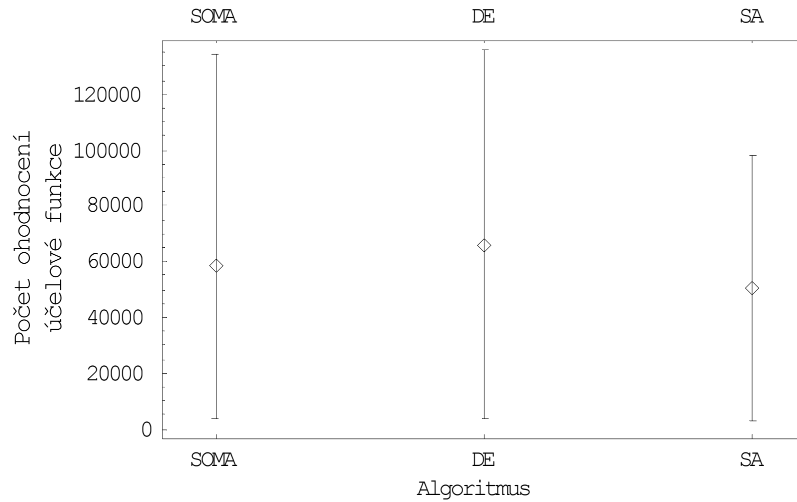
U SOMA algoritmu všechny 3 neúspěšné simulace skončili na hodnotě účelové funkce 1 (mravenec snědl všechno kromě 1návnady). Nejhůře dopadla SA, u které byl nejhorší výsledky na hodnotě 30.

Jednou z možností, jak porovnat výsledky jednotlivých algoritmů, je porovnat počty ohodnocení účelové funkce. Toto porovnání se provádělo jen pro úspěšné simulace. Nejnižší hodnota 2697 byla nalezena u SA algoritmu a nejhůře z tohoto porovnání vyšel DE algoritmus s hodnotou 4030 viz. (Tab.8).

Tab. 8 Počet ohodnocení účelové funkce pro SOMA, DE, SA

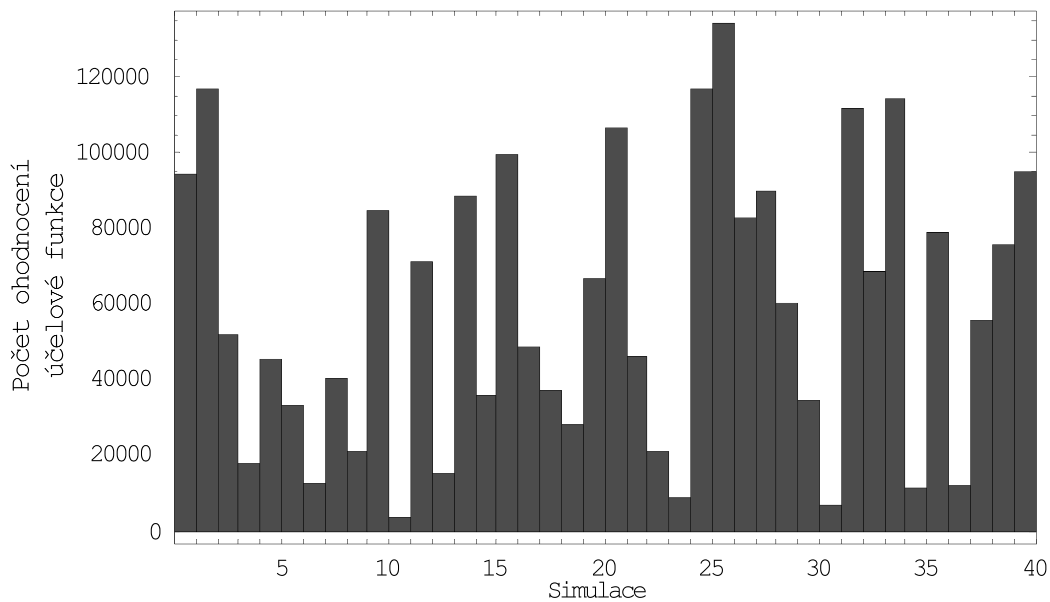
	Počet ohodnocení účelové funkce		
	SOMA	DE	SA
Minimum	3 396	4 030	2 697
Maximum	134 114	136 011	98 241
Průměr	58 559	65 684	50 142

Hodnoty z Tab.8 jsou pro lepší názornost graficky zobrazeny na (Obr.29), průměrné hodnoty představují kosočtverce.

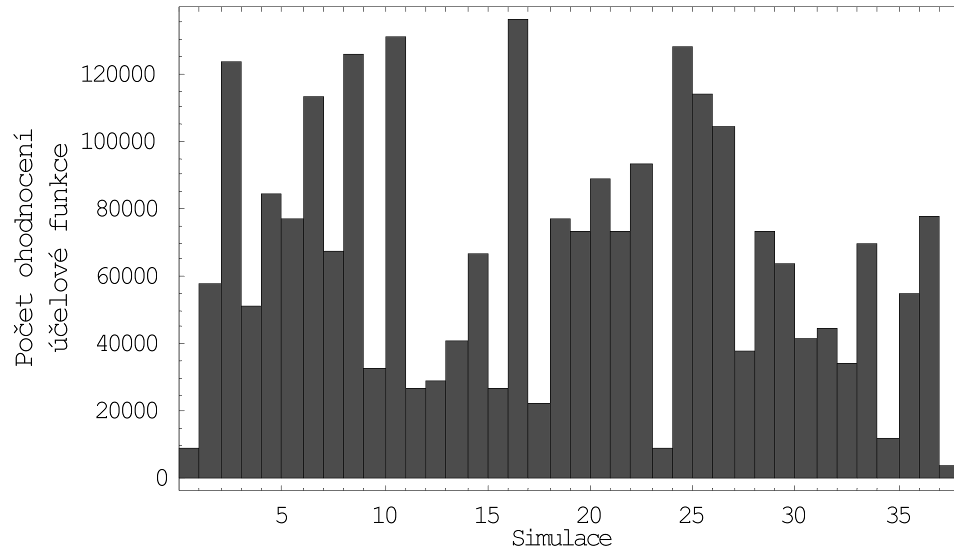


Obr. 29 Grafická reprezentace minimální, maximální a průměrné hodnoty SOMA, DE a SA algoritmu

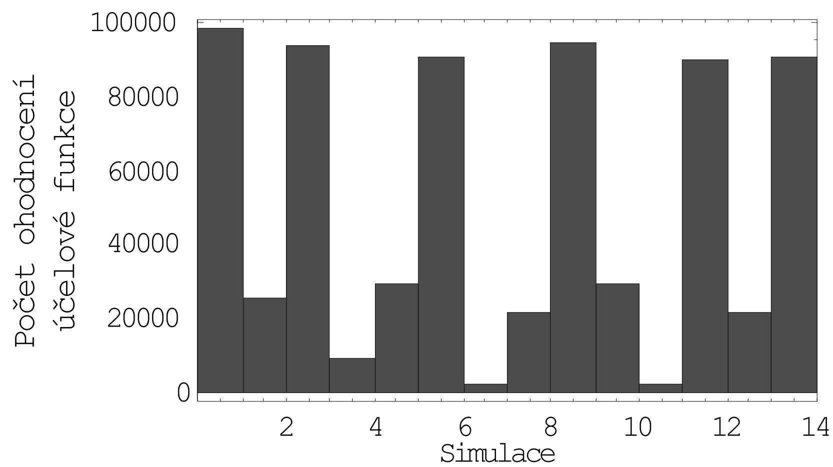
Dalším ukazatelem je zobrazení počtu ohodnocení účelové funkce pro každou úspěšně nalezenou simulaci (Obr. 30 – 32).



Obr. 30 Histogram SOMA



Obr. 31 Histogram DE



Obr. 32 Histogram SA

Dále můžeme výsledky porovnat pomocí počtu příkazů mravence a pomocí počtu kroků nutných k snězení všech návnad na stezce viz. (Tab. 9 - 10).

Tab. 9 Počet příkazů mravence

	Počet příkazů		
	SOMA	DE	SA
Minimum	11	11	15
Maximum	50	50	50
Průměr	31	31	26

Tab. 10 Počet kroků mravence

	Počet kroků		
	SOMA	DE	SA
Minimum	396	387	406
Maximum	606	604	605
Průměr	549	540	535

Z Tab.9 je zřejmé, že minimální počet příkazů 11 je stejný pro SOMA a DE, ale to neznamená, že oba vygenerované programy mají stejné pořadí vnořených funkcí a stejný počet kroků. Programy jsou vykresleny na (Obr.33). Počet kroků při průchodu mravence stezkou pomocí SOMA je roven hodnotě 594, DE 599.

▪ **SOMA - 11**

```
Prog3[Move, Right, IfFoodAhead[Move, Prog3[Left, Left, IfFoodAhead[Move, Right]]]]
```

pocet kroku 594

▪ **DE - 11**

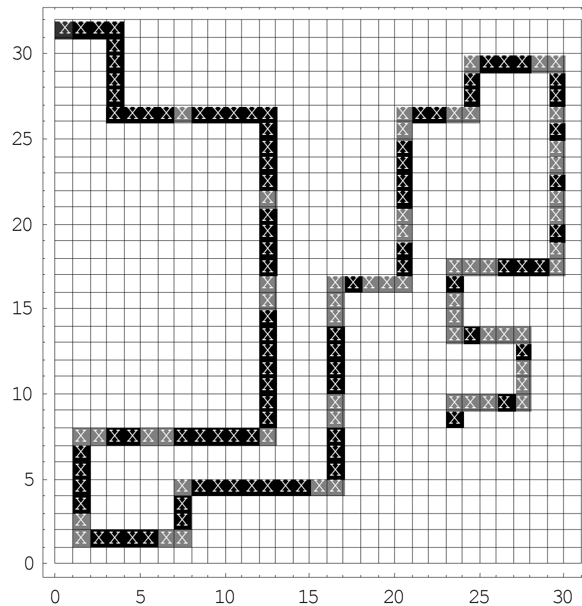
```
Prog3[Right, IfFoodAhead[Move, Prog3[Right, Right, IfFoodAhead[Move, Right]]], Move]
```

pocet kroku 599

Obr. 33 Programy o počtu 11 příkazů vygenerovaný pomocí SOMA a DE

Simulace ukázaly, že nejnižší počet příkazů mravence neznamená, že mravenec vykoná nejnižší počet kroků. A naopak malý počet kroků není přímo úměrný malému počtu příkazů. Seřazení výsledků dle počtu kroků a počtu příkazů pro jednotlivé algoritmy najdete v příloze PII-IV.

Z pohledu počtu kroků byla minimální hodnota 387 nalezena DE algoritmem. Program tohoto nejrychlejšího mravence obsahuje 49 příkazů (viz. příloha PV). Na (Obr. 34) je důkaz, že cesta, kterou prochází nejrychlejší mravenec je shodná s stezkou „Santa Fe" viz. (Obr.20). Bíle „X" představují políčka, která mravenec navštívil. Všechny pohyby mravence při průchodu stezkou (i s otáčením) si můžete prohlédnout v příloze PV.



Obr. 34 Nejrychlejší mravenec kopírující stezku „Santa Fe"

Výše uvedené výsledky dokázaly, že analytické programování dokáže pomocí SOMA, DE a také SA algoritmu (jehož výkon sice nedosahoval kvalit SOMA a DE v počtu úspěšných nalezených simulací) řešit i poměrně náročné úlohy. Během proběhlých simulací se podařilo najít program, který dokáže během 387 kroků provést mravence stezkou „Santa Fe". Přitom Kozův optimální program potřeboval k projití stezky 551 kroků.

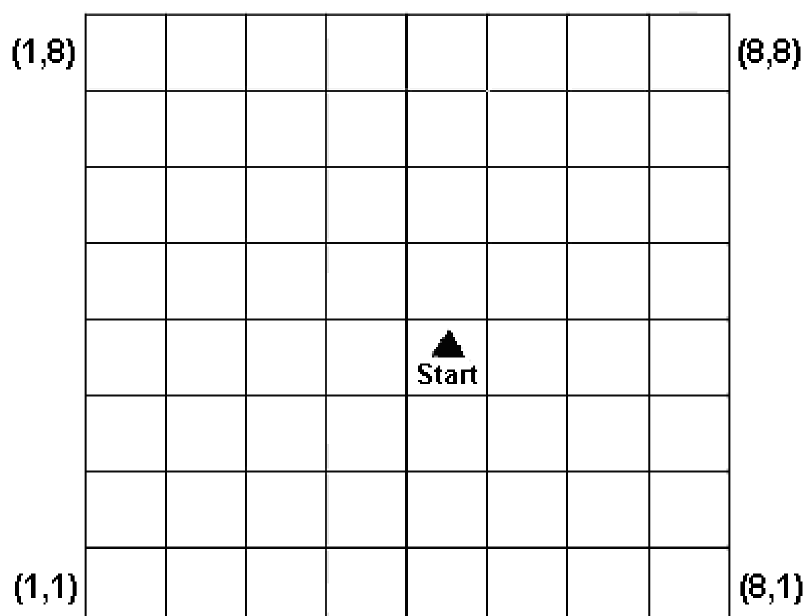
5.2 Úloha „zahradní sekačky" (Lawnmover)

Umělý mravenec procházel stezku „Santa Fe" pomocí určité sady pravidel. V druhé úloze se tzv. „zahradní sekačka" pohybuje po trávníku dle sekvence přesných příkazů to nám umožňuje získat i jiný pohled na danou problematiku. Cílem této úlohy je tedy najít program pro ovládání pohybu sekačky, která má za úkol posekat všechnu trávu na zahradě. Tato úloha byla vybrána z knihy J. Kozy [10].

5.2.1 Popis trávníku

Sekačka operuje na ploše, kterou představuje mřížka 8x8 oddělených čtvercových ploch trávníku, která má zpočátku trávu na všech 64 čtvercích.

Každý čtverec trávníku je jedinečně identifikován pomocí celočíselného vektoru, kde hodnota (1,1) představuje počátek, který je umístěn v levém dolním rohu, s tím že se počet čtverců zvyšuje směrem doprava nebo směrem nahoru a pravý horní roh představuje hodnotu (8,8). Koza ve svém originále požívá souřadnicový systém s počátkem (0,0) v levém horním rohu a koncem (7,7) v pravém dolním rohu, v prostředí programu Mathematica je vhodnější výše popsaný systém. Trávník neobsahuje žádné překážky (Obr.35).



Obr. 35 Zobrazení trávníku a startovací pozice sekačky

Sekačka startuje z pozice (5, 4) natočena směrem k severu. K zajištění, aby se sekačka nedostala mimo vymezený prostor trávníku je prostor chápán (stejně jako u mravence) jako „koule“ tj. např. při vykročení mimo pole směrem dolů znamená vstup do pole opět shora (pozice se počítá jako mod 8). Stav sekačky je dán její pozicí na trávníku a směrem ke kterému je natočena čelem např. stav sekačky ve startovní pozici je ((5, 4), No).

5.2.2 Množina funkcí použitých pro pohyb sekačky po trávníku

Při pohybu sekačky po trávníku (ať už prostřednictvím sekání nebo skoku) sekačka kosí všechnu trávu na jednotlivých políčkách přes které přejíždí. Sekačka nemá žádný senzor a k pohybu po ploše trávníku používá následující akce:

- „otočení vlevo“
- „sekání“
- „skok“

Množina terminálů pro řešení této úlohy se skládá z tří 0-argumentových operací (otáčející sekačku vlevo, sekací operace a z operace generující náhodné vektory).

$$GFS_{0arg}= \{Left, Mow, R_{v8}\} \quad (17)$$

Kde:

- **Left** – představuje terminál pro otočení sekačky doleva, terminál vrací hodnotu pozice sekačky. Ukázka programu viz. (Obr. 36)

```
TurnLeft[WhereIsF_] := Module[{}, lengthPositionList = Length[PositionList];
  {osaxx1, osayy1} = {osaxx, osayy} = PositionList[[lengthPositionList]];
  FacePoc = Flatten[Position[FaceSet, WhereIsF[[1]]]][[1]];
  If[FacePoc <= 1, FacePoc = 4, FacePoc--];
  WhFace = {FaceSet[[FacePoc]]}; WhereIsFace = WhFace;
  NumberOfTurnLeft++;
  AppendTo[PositionListAll, WhFace];
  Return[{osaxx1, osayy1}]]
```

Obr. 36 Terminál otáčející sekačku vlevo

Vstupem této operace je směr natočení sekačky. Jakmile se sekačka otočí doleva uloží změnu do proměnné WhereIsFace , zvýší hodnotu NumberOfTurnLeft (v proměnné se ukládá počet natočení sekačky) a vrátí hodnotu své pozice kterou, si uložila v proměnných

(osaxx1,osayy1), tím že našla poslední uloženou pozici v PositionList (plní stejnou funkci jako u umělého mravence.)

- **Mow** - představuje terminál posunující sekačku vpřed ve směru natočení a sekající trávu na políčku na které sekačka přijíždí, také vrací hodnotu pozice sekačky
- **R_{v8}** - představuje terminál náhodně generující hodnoty pozic na trávníku, tedy 64 možných vektorů v rozsahu (1, 1) až (8, 8), vrací hodnotu náhodně vygenerovaného vektoru

Tabulka obsahující všech 64 možných pozic trávníku pod názvem TabulRV8 je zobrazena v prostředí Mathematica viz. (Obr.37)

```
TabulRV8 = {{1, 1}, {1, 2}, {1, 3}, {1, 4}, {1, 5}, {1, 6}, {1, 7}, {1, 8}{2, 1}{2, 2},
    {2, 3}, {2, 4}, {2, 5}, {2, 6}, {2, 7}, {2, 8}, {3, 1}, {3, 2}, {3, 3}, {3, 4},
    {3, 5}, {3, 6}, {3, 7}, {3, 8}, {4, 1}, {4, 2}, {4, 3}, {4, 4}, {4, 5}, {4, 6},
    {4, 7}, {4, 8}, {5, 1}, {5, 2}, {5, 3}, {5, 4}, {5, 5}, {5, 6}, {5, 7}, {5, 8},
    {6, 1}, {6, 2}, {6, 3}, {6, 4}, {6, 5}, {6, 6}, {6, 7}, {6, 8}, {7, 1}, {7, 2},
    {7, 3}, {7, 4}, {7, 5}, {7, 6}, {7, 7}, {7, 8}, {8, 1}, {8, 2}, {8, 3}, {8, 4},
    {8, 5}, {8, 6}, {8, 7}, {8, 8}}
```

Obr. 37 Tabulka možných pozic

Množina funkcí se skládá z 1-argumentové funkce Frog:

$$GFS_{1arg} = \{Frog\} \quad (18)$$

Kde:

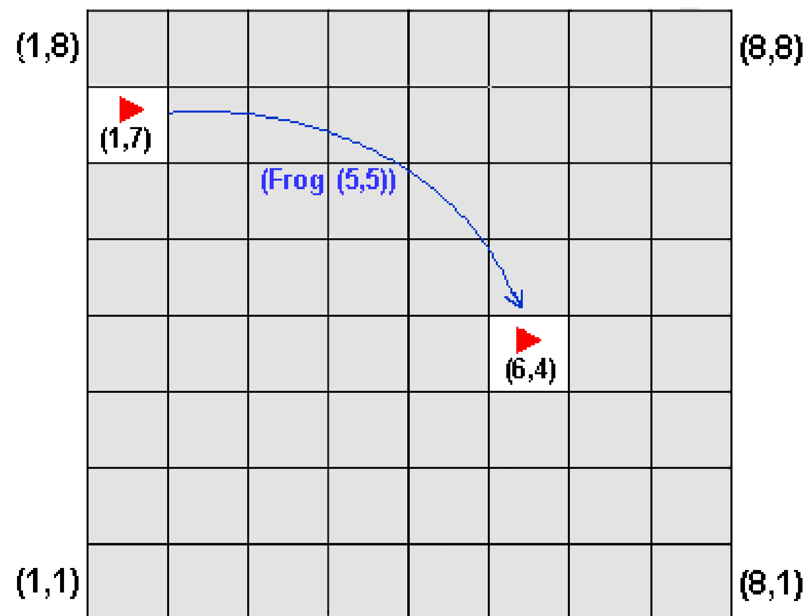
- **Frog** - 1-argumentová funkce způsobující skok na políčko trávníku, které je rovno součtu aktuální pozice sekačky s argumentem funkce Frog, natočení sekačky se během této operace nemění. Ukázka v prostředí Mathematica viz. (Obr. 38).

```
frog[{fosay_, fosax_}] := Module[{},
   osaxx1 = fosax; osayy1 = fosay; lengthPositionList = Length[PositionList];
   {x, y} = PositionList[[lengthPositionList]];
   fosax1 = If[Mod[Abs[x - osaxx1], 8] === 0, 8, Mod[Abs[x - osaxx1], 8]];
   fosay1 = If[Mod[Abs[y + osayy1], 8] === 0, 8, Mod[Abs[y + osayy1], 8]];
   NumberOfMowAndFrog++; AppendTo[PositionList, {fosax1, fosay1}];
   AppendTo[PositionListAll, {fosax1, fosay1}];
   Pozice = {fosax1, fosay1}; MowFunction[Pozice];
   Return[{osaxx1, osayy1}]];
```

Obr. 38 Funkce Frog

Zajímavostí této funkce je že nevrací hodnotu nové pozice sekačky, ale hodnotu vstupního argumentu.

Příklad funkce Frog je zobrazen na (Obr.39) pokud je sekačka na pozici (1,7) a natočena směrem k východu, (Frog (5, 5)) skočí se sekačkou na pozici (6,4) sekačka je stále natočena k východu. Přitom pozice (6,4) je posečena, funkce vrací hodnotu argumentu v našem příkladě tedy (5,5). Šedá políčka představují trávník, bílá posečená místa.



Obr. 39 Ukázka skoku pomocí funkce Frog

Další funkce, které sekačka využívá jsou 2- argumentové funkce V8A a Prog2:

$$GFS_{2arg} = \{V8A, Prog2\} \quad (19)$$

Kde:

- **V8A** - je dodatečná funkce sloužící k rozvoji programu.Vrací hodnotu, která je rovna vektorovému součtu jejích vstupů po aplikaci modulo 8 (8x8 políček), např. (V8A (1, 2) (3, 7)) vrací hodnotu (4, 1).
- **Prog2** - nezbytná funkce pro strukturu generovaného programu. Slouží k provedení dvou příkazů najednou, v podstatě se jedná o rozvětvení programu.

5.2.3 Ohodnocení účelové funkce

Cílem sekačky je posekat všech 64 políček trávníku pomocí vygenerovaného programu. Tak jako mravenec, má i sekačka omezený počet kroků po trávníku, může vykonat buď:

- 200 sekacích operací (Mow) a skoků (Frog) nebo
- 100 otoční vlevo (Left)

Po dosažení těchto hodnot je pohyb sekačky po trávníku ukončen. Počet všech políček trávníku, tedy hodnota 64 představuje hrubé ohodnocení. Hodnota účelové funkce (20) je počítána z rozdílu tohoto hrubého ohodnocení a počtu posečených políček trávníku, které sekačka posekala během aktuálně vygenerovaného programu.

$$CV = 64 - NumberOfGrass \tag{20}$$

NumberOfGrass – počet posečených políček trávníku během aktuálně vygenerovaného programu

5.2.4 Dosažené výsledky

Úloha se testovala pomocí SOMA algoritmu, pro který byla zvolena varianta ALLToOne. Simulace byly stejně časově náročné jako simulace mravence, zabraly tedy 1 – 3 dny na počítači: Intel Pentium 4 CPU 3.00 GHz , 1GB RAM.

Nastavení parametrů SOMA je zobrazeno v (Tab. 11), počet ohodnocení účelové funkce je roven:

$$((PopSize-1)*PathLength*Migrations)/Step = 326182 \tag{21}$$

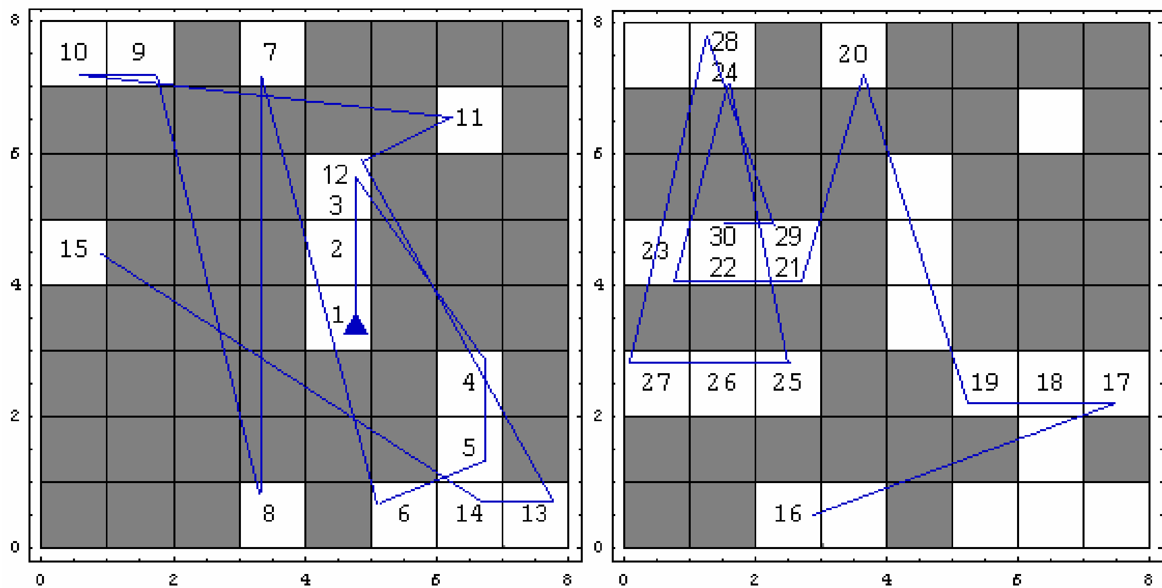
Tab. 11 Nastavení parametrů SOMA

Parametr	Hodnota
PathLength	3
Step	0.22
PRT	0.21
PopSize	300
Migrations	80
MinDiv	- 0.1
Individual Length	500

Vzhledem k časové náročnosti předešlé úlohy bylo experimentálně provedeno jen několik simulací, přičemž se našly dva pozitivní výsledky.

První výsledek byl nalezen při počtu ohodnocení účelové funkce 92 701 podklady (viz. příloha PVI).

Druhý výsledek potřebovala k nalezení 185 101 ohodnocení účelové funkce. Na (Obr. 40) je graficky znázorněno prvních 30 kroků sekačky po trávníku. V každé části trávníku vždy po 15 krocích s tím, že kroky jsou očíslovány od jednička do třicítky. Bílé políčka bez čísel v druhé polovině obrázku jsou místa, která sekačka „pokosila“ během prvních 15 kroků. Z obrázku je patrné že sekačka nemá žádný senzor (přejíždí místa na kterých již kosila), co patrné není - natočení sekačky, najdete i s programem v příloze PVII.



Obr. 40 Grafické znázornění prvních 30 kroků sekačky po trávníku

ZÁVĚR

Tato studie dokazuje že analytické programování je nejen vhodné pro matematickou regresi, ale také pro nastavení optimální trajektorie umělého mravence nebo sekačky, kteří mohou být nahrazeni opravdovými roboty v reálném světě např. pro průmyslové využití.

Ve srovnání s genetickým programováním, dle výsledků uvedených výše může AP tento druh problémů řešit v kratších časových intervalech, díky nižšímu počtu ohodnocení účelové funkce. Časová náročnost by mohla být snížena také pomocí paralelizace procesu, které využíval Koza, který měl aktivně zapojeny stovky počítačů pro výpočet GP [10].

Cílem této práce ale není porovnat, jestli je AP horší nebo lepší jak GP (nebo GE), ale dokázat, že je to plnohodnotný nástroj symbolické regrese podporovaný různými evolučními algoritmy a že symbolická regrese pomocí AP dokáže řešit také případy u kterých jsou lingvistické podmínky jako jsou například příkazy pro pohyb umělého mravence nebo skutečného robota.

Pro úlohu umělého mravence se provedly simulace použitím těchto evolučních algoritmů SOMA, DE a SA a nalezené výsledky byly uspořádány v tabulkách a obrázcích, z kterých lze usoudit že SOMA a DE algoritmus dosahovaly přibližně stejných výsledku a nalezly mnohem více pozitivních výsledků oproti SA algoritmu. U sekačky se nalezly dva pozitivní výsledky pomocí SOMA algoritmu.

Úloha sekačky byla dodatečně vybrána, aby návrh trajektorií byl veden ze dvou úhlů pohledů. Sekvencí přesných příkazů pohybu sekačky po trávníku a sadou pravidel, podle nichž mravenec prochází stezku „Santa Fe“.

SEZNAM POUŽITÉ LITERATURY

[1] Koza J. R., Genetic Programming, MIT Press, 1998, ISBN 0-262-11189-6

[2] Rektorys K., Variational methods in Engineering Problems and Problems of Mathematical Physics, Czech edition, 1999, ISBN 80-200-0714-8

[3] Oplatková. Z.: Analytic Programming, diplomová práce, UTB Zlín, 2003, 74 p.

[4] Koza J.R.,Bennet F.H., Andre D., Keane M. 1999, Genetic Programming III, Morgan Kaufnamm pub., ISBN 1-55860-543-6, 1999

[5] Zelinka I., Oplatkova Z, Nolle L., Boolean Symmetry Function Synthesis by Means of Arbitrary Evolutionary Algorithms-Comparative Study, International Journal of Simulation Systems, Science and Technology, Volume 6, Number 9, August 2005, pages 44 - 56, ISSN: 1473-8031, online http://ducati.doc.ntu.ac.uk/uksim/journal/Vol-6/No.9/cover.htm, ISSN: 1473-804x

[6] Mařík V., Štěpánková O., Lažanský J. 2004, Umělá inteligence IV, Academia, ISBN 80-200-0496-3, 1993

[7] Zelinka I., Umělá inteligence v problémech globální optimalizace, BEN, Praha, 2002, ISBN 80-7300-069-5

[8] Kvasnička V., Pospíchal J., Tiňo P. 2000, Evolučné algoritmy, STU Bratislava, ISBN 85-246-2000, 2000

[9] Koza J. R., Genetic Programming II: Automatic Discovery of Reusable Programs, Cambridge, MIT Press, 1994, ISBN 02-621-1189-6

[10] Koza J. R., Keane M. A., Streeter M. J. 2003: Evolving Inventions, ScientificAmerican, February 2003, p. 40-47, ISSN 0036-8733, (online www.sciam.com)

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

$x_{i,j}^{ML+1}$	j-tý prvek i-tého jedince v migračním kole ML+1
$x_{i,j,START}^{ML}$	j-tý prvek i-tého jedince v migračním kole ML ve startovní pozici
$x_{L,j}^{ML}$	j-tý prvek vedoucího jedince (Leadera) v migračním kole ML
AcceptedError	Ukončovací parametr SOMA, definuje, jaký maximální rozdíl mezi nejhorším a nejlepším jedincem v aktuální populaci je povolen
AllToAll	Strategie SOMA, každý jedinec migruje ke všem ostatním
AllToAllAdapti-ve	Strategie SOMA, pozměněná All to All tak, že všichni jedinci prohle-dávaií směrem ke všem adaptivním způsobem
AllToOne	Strategie SOMA, základní variace SOMA – všichni jedinci migrují směrem k Leaderovi
AllToOneRand	Strategie SOMA, kdy náhodně vybraní jedinci prohledávají směrem k Leaderovi
AP	Analytické programování
AP_{basic}	První verze AP
AP_{meta}	Druhá verze AP
BNF	Backusova-Naurova forma, bezkontextová gramatika
CR	Řídící parametr DE; tzv. práh křížení
DataSet	Množina změřených bodů
DE	Diferenciální evoluce
Dim	Udává počet parametrů účelové funkce
DSH	Manipulace s diskrétními hodnotami
EA	Evoluční algoritmy
F	Řídící parametr DE; mutační konstanta
$F_{AP(t)}$	Výstup z AP

F_{cost}	Hodnota účelové funkce
Frog	Funkce způsobující skok na políčko trávníku, které je rovno součtu aktuální pozice sekačky s argumentem funkce Frog
GE	Gramatická evoluce
Generations	Ukončovací parametr DE; udává počet evolučních cyklů
GFS	Obecný prostor funkcí
GFS_{all}	Soubor všech funkcí
GFS_{3arg}	Podmnožina s funkcemi obsahující jen 3 argumenty
GFS_{2arg}	Podmnožina s funkcemi obsahující jen 2 argumenty
GFS_{1arg}	Podmnožina s funkcemi obsahující jen 1 argumenty
GFS0arg	Podmnožina terminálů
GP	Genetické programování.
IfFoodAhead	Rozhodovací funkce, mravenec kontroluje pole před sebou a pokud je tam návnada tak vykoná argument na pozici pravda (true), jinak vykoná argument na pozici nepravda (false).
Individua Length	Maximální povolená délka jedince, nastavuje se u všech algoritmů
Leader	Jedinec s nejlepším ohodnocením účelové funkce
Left	Terminál, který otáčí mravence (sekačku) vlevo
Migrace	Ukončovací parametr SOMA; udává, kolikrát se populace jedinců přeorganizuje – obrodí
Move	Terminál posouvající mravence o jedno políčko ve směru ke kterému je natočen
Mow	Terminál posunující sekačku vpřed ve směru natočení a sekající trávu na políčku na které sekačka přijíždí
NP	Řídící parametr DE; udává velikost populace
PathLength	Řídící parametr SOMA; udává jak daleko se aktivní jedinec zastaví od vedoucího jedince

PopSize	Udává počet jedinců v populaci
Prog2	Funkce slouží k provedení dvou příkazů najednou, v podstatě se jedná o rozvětvení programu, používá se u obou úloh
Prog3	Funkce slouží k provedení tří příkazů najednou, používá se jen u sekačky
PRT	Řídící parametr SOMA; dle kterého se vytváří perturbační vektor, který ovlivňuje pohyb jedince
$PRTVector_j$	Vektor jedniček a nul ovlivňující pohyb jedince v SOMA
SA	Simulované žíhání
„Santa Fe“	Stezka, kterou prochází mravenec
Sextic	Hledání analytického tvaru funkce dané polynomem 6. řádu
SOMA	Samoorganizující se algoritmus.
SP	Bezpečnostní procedury AP
Step	Řídící parametr SOMA; udává velikost kroku jedince během prohledávání
V8A	Funkce sloužící k rozvoji programu u sekačky.Vrací hodnotu, která je rovna vektorovému součtu jejích vstupů (po aplikaci modulo 8)
WhereIsFace	Funkce, ve které se ukládá na točení mravence (sekačky)

SEZNAM OBRÁZKŮ

Obr. 1 Vzorové příklady funkčních prostorů 12
Obr. 2 Manipulace s množinou diskrétních hodnot 13
Obr. 3 Princip struktury GFS 14
Obr. 4 Příklad s jedincem o třech parametrech 14
Obr. 5 Příklad ideálního jedince v AP 16
Obr. 6 Příklad s využitím podprogramu pro zamezení patologií 17
Obr. 7 Schéma korekce patologických funkcí 18
Obr. 8 Ohodnocení účelové funkce (Fcost – hodnota účelové funkce, DataSet – množina změřených bodů, Fap – výstup z AP) 19
Obr. 9 GE gramatika příkladu 21
Obr. 10 Derivační strom a odpovídající program 24
Obr. 11 Křížení znázorněné přímo na programech 25
Obr. 12 Reprezentace logického výrazu(9) stromovou strukturou 26
Obr. 13 Křížení stromových struktur 27
Obr. 14 Mutace stromových struktur 28
Obr. 15 Princip SOMA – převzato [7] 33
Obr. 16 „PRTVector“v grafickém detailu 35
Obr. 17 Princip DE – převzato z [7] 38
Obr. 18 Grafické znázornění postupu jedince 40
Obr. 19 Notebook pro úlohu umělého mravence 42
Obr. 20 Stezka Santa Fe 43
Obr. 21 Terminál Move naprogramovaný v buňce programu Mathematica 45
Obr. 22 Zápis funkce Prog2,Prog3 v Mathematice 46
Obr. 23 Ukázka ošetření ohodnocení účelové funkce v Mathematice 47
Obr. 24 Pomocná funkce FoodFunction 48
Obr. 25 Stromová struktura Kozova programu pro optimální procházení mravence stezkou „Santa Fe“ 48
Obr. 26 Histogram úspěšnosti SOMA 51
Obr. 27 Histogram úspěšnosti DE 51
Obr. 28 Histogram úspěšnosti SA 52

Obr. 29 Grafická reprezentace minimální, maximální a průměrné hodnoty SOMA, DE a SA algoritmu 53
Obr. 30 Histogram SOMA 53
Obr. 31 Histogram DE 54
Obr. 32 Histogram SA 54
Obr. 33 Programy o počtu 11 příkazů vygenerovaný pomocí SOMA a DE 55
Obr. 34 Nejrychlejší mravenec kopírující stezku „Santa Fe“ 56
Obr. 35 Zobrazení trávníku a startovací pozice sekačky 57
Obr. 36 Terminál otáčející sekačku vlevo 58
Obr. 37 Tabulka možných pozic 59
Obr. 38 Funkce Frog 59
Obr. 39 Ukázka skoku pomocí funkce Frog 60
Obr. 40 Grafické znázornění prvních 30 kroků sekačky po trávníku 62

SEZNAM TABULEK

Tab. 1 Ukázkový chromozom 23
Tab. 2 Význam parametrů SOMA 31
Tab. 3 Parametry DE 36
Tab. 4 Nastavení parametrů SOMA 49
Tab. 5 Nastavení parametrů DE 50
Tab. 6 Nastavení parametrů SA 50
Tab. 7 Přehled simulací 51
Tab. 8 Počet ohodnocení účelové funkce pro SOMA, DE, SA 52
Tab. 9 Počet příkazů mravence 54
Tab. 10 Počet kroků mravence 55
Tab. 11 Nastavení parametrů SOMA 61

SEZNAM PŘÍLOH

PI Kozův optimální program se slovním popisem cesty

PII Seřazení dle počtu kroků a počtu příkazů mravence pro SOMA algoritmus

PIII Seřazení dle počtu kroků a počtu příkazů mravence pro DE algoritmus

PIV Seřazení dle počtu kroků a počtu příkazů mravence pro SA algoritmus

PV Ukázka programu nejrychlejšího mravence se slovním popisem cesty

PVI Program první sekačky se slovním popisem cesty

PVII Program druhé sekačky se slovním popisem cesty

PŘÍLOHA P I: KOZŮV OPTIMÁLNÍ PROGRAM SE SLOVNÍM POPISEM CESTY

```
IfFoodAhead[Move,
  Prog3[Left, Prog2[IfFoodAhead[Move, Right], Prog2[Right, Prog2[Left, Right]]],
   Prog2[IfFoodAhead[Move, Left], Move]]]
```

```
{{1, 32}, {2, 32}, {3, 32}, {4, 32}, {No}, {Ea}, {So}, {Ea}, {So}, {4, 31}, {4, 30}, {4, 29}, {4, 28},
 {4, 27}, {Ea}, {5, 27}, {So}, {Ea}, {So}, {Ea}, {6, 27}, {7, 27}, {No}, {Ea}, {So}, {Ea},
 {So}, {Ea}, {8, 27}, {9, 27}, {10, 27}, {11, 27}, {12, 27}, {13, 27}, {No}, {Ea}, {So}, {Ea},
 {So}, {13, 26}, {13, 25}, {13, 24}, {13, 23}, {Ea}, {So}, {We}, {So}, {We}, {So}, {13, 22},
 {13, 21}, {13, 20}, {13, 19}, {13, 18}, {Ea}, {So}, {We}, {So}, {We}, {So}, {13, 17}, {Ea},
 {So}, {We}, {So}, {We}, {So}, {13, 16}, {13, 15}, {13, 14}, {13, 13}, {13, 12}, {13, 11},
 {13, 10}, {13, 9}, {Ea}, {So}, {We}, {So}, {We}, {So}, {13, 8}, {Ea}, {So}, {We}, {So}, {We},
 {12, 8}, {11, 8}, {10, 8}, {9, 8}, {8, 8}, {So}, {We}, {No}, {We}, {No}, {We}, {7, 8}, {So},
 {We}, {No}, {We}, {No}, {We}, {6, 8}, {5, 8}, {4, 8}, {So}, {We}, {No}, {We}, {No}, {We},
 {3, 8}, {So}, {We}, {No}, {We}, {No}, {We}, {2, 8}, {So}, {2, 7}, {We}, {So}, {We}, {So},
 {2, 6}, {2, 5}, {2, 4}, {Ea}, {So}, {We}, {So}, {We}, {So}, {2, 3}, {Ea}, {So}, {We}, {So},
 {We}, {So}, {2, 2}, {Ea}, {3, 2}, {So}, {Ea}, {So}, {Ea}, {4, 2}, {5, 2}, {6, 2}, {No}, {Ea},
 {So}, {Ea}, {So}, {Ea}, {7, 2}, {No}, {Ea}, {So}, {Ea}, {So}, {Ea}, {8, 2}, {No}, {8, 3}, {Ea},
 {No}, {Ea}, {No}, {8, 4}, {We}, {No}, {Ea}, {No}, {Ea}, {No}, {8, 5}, {We}, {No}, {Ea}, {No},
 {Ea}, {9, 5}, {10, 5}, {11, 5}, {12, 5}, {13, 5}, {14, 5}, {15, 5}, {No}, {Ea}, {So}, {Ea},
 {So}, {Ea}, {16, 5}, {No}, {Ea}, {So}, {Ea}, {So}, {Ea}, {17, 5}, {No}, {17, 6}, {Ea}, {No},
 {Ea}, {No}, {17, 7}, {17, 8}, {We}, {No}, {Ea}, {No}, {Ea}, {No}, {17, 9}, {We}, {No}, {Ea},
 {No}, {Ea}, {No}, {17, 10}, {17, 11}, {17, 12}, {17, 13}, {17, 14}, {We}, {No}, {Ea}, {No},
 {Ea}, {No}, {17, 15}, {We}, {No}, {Ea}, {No}, {Ea}, {No}, {17, 16}, {We}, {No}, {Ea}, {No},
 {Ea}, {No}, {17, 17}, {We}, {No}, {Ea}, {No}, {Ea}, {18, 17}, {19, 17}, {No}, {Ea}, {So}, {Ea},
 {So}, {Ea}, {20, 17}, {No}, {Ea}, {So}, {Ea}, {So}, {Ea}, {21, 17}, {No}, {21, 18}, {Ea}, {No},
 {Ea}, {No}, {21, 19}, {We}, {No}, {Ea}, {No}, {Ea}, {No}, {21, 20}, {We}, {No}, {Ea}, {No},
 {Ea}, {No}, {21, 21}, {21, 22}, {21, 23}, {21, 24}, {21, 25}, {We}, {No}, {Ea}, {No}, {Ea},
 {No}, {21, 26}, {We}, {No}, {Ea}, {No}, {Ea}, {No}, {21, 27}, {We}, {No}, {Ea}, {No}, {Ea},
 {22, 27}, {23, 27}, {No}, {Ea}, {So}, {Ea}, {So}, {Ea}, {24, 27}, {No}, {Ea}, {So}, {Ea}, {So},
 {Ea}, {25, 27}, {No}, {25, 28}, {Ea}, {No}, {Ea}, {No}, {25, 29}, {We}, {No}, {Ea}, {No},
 {Ea}, {No}, {25, 30}, {We}, {No}, {Ea}, {No}, {Ea}, {26, 30}, {27, 30}, {28, 30}, {No}, {Ea},
 {So}, {Ea}, {So}, {Ea}, {29, 30}, {No}, {Ea}, {So}, {Ea}, {So}, {Ea}, {30, 30}, {No}, {Ea},
 {So}, {Ea}, {So}, {30, 29}, {30, 28}, {Ea}, {So}, {We}, {So}, {We}, {So}, {30, 27}, {30, 26},
 {Ea}, {So}, {We}, {So}, {We}, {So}, {30, 25}, {Ea}, {So}, {We}, {So}, {We}, {So}, {30, 24},
 {30, 23}, {Ea}, {So}, {We}, {So}, {We}, {So}, {30, 22}, {Ea}, {So}, {We}, {So}, {We}, {So},
 {30, 21}, {30, 20}, {Ea}, {So}, {We}, {So}, {We}, {So}, {30, 19}, {Ea}, {So}, {We}, {So}, {We},
 {So}, {30, 18}, {Ea}, {So}, {We}, {So}, {We}, {29, 18}, {28, 18}, {27, 18}, {So}, {We}, {No},
 {We}, {No}, {We}, {26, 18}, {So}, {We}, {No}, {We}, {No}, {We}, {25, 18}, {So}, {We}, {No},
 {We}, {No}, {We}, {24, 18}, {So}, {24, 17}, {We}, {So}, {We}, {So}, {24, 16}, {Ea}, {So}, {We},
 {So}, {We}, {So}, {24, 15}, {Ea}, {So}, {We}, {So}, {We}, {So}, {24, 14}, {Ea}, {25, 14},
 {So}, {Ea}, {So}, {Ea}, {26, 14}, {No}, {Ea}, {So}, {Ea}, {So}, {Ea}, {27, 14}, {No}, {Ea},
 {So}, {Ea}, {So}, {Ea}, {28, 14}, {No}, {Ea}, {So}, {Ea}, {So}, {28, 13}, {28, 12}, {Ea},
 {So}, {We}, {So}, {We}, {So}, {28, 11}, {Ea}, {So}, {We}, {So}, {We}, {So}, {28, 10}, {Ea},
 {So}, {We}, {So}, {We}, {27, 10}, {26, 10}, {So}, {We}, {No}, {We}, {No}, {We}, {25, 10},
 {So}, {We}, {No}, {We}, {No}, {We}, {24, 10}, {So}, {24, 9}, {We}, {So}, {We}, {So}, {24, 8}}
```

PŘÍLOHA P II: SEŘAZENÍ POČTU KROKŮ A POČTU PŘÍKAZŮ MRAVENCE PRO SOMA ALGORITMUS

Seřazení podle počtu kroků		Seřazení podle počtu příkazů	
396	49	11	594
409	21	11	596
409	23	14	568
421	37	14	594
456	50	15	577
489	17	16	544
521	50	16	590
532	50	16	594
533	20	16	606
533	27	17	489
540	27	17	544
542	27	17	583
544	16	18	576
544	17	20	533
548	30	21	409
548	50	21	589
551	43	23	409
551	50	24	559
559	24	26	583
562	50	27	533
568	14	27	540
572	34	27	542
574	27	27	574
576	18	30	548
577	15	34	572
581	50	37	421
583	17	43	551
583	26	47	603
589	21	49	396
590	16	49	596
594	11	49	604
594	14	49	606
594	16	50	456
596	11	50	521
596	49	50	532
601	50	50	548
603	47	50	551
604	49	50	562
606	16	50	581
606	49	50	601

PŘÍLOHA P III: SEŘAZENÍ POČTU KROKŮ A POČTU PŘÍKAZŮ MRAVENCE PRO DE ALGORITMUS

Seřazení podle počtu kroků		Seřazení podle počtu příkazů	
387	49	11	599
390	50	13	564
409	18	14	542
409	18	14	577
409	50	14	581
475	16	14	581
496	50	15	583
509	21	15	594
516	46	16	475
517	49	16	533
519	49	18	409
525	38	18	409
533	16	18	568
533	20	19	584
542	14	19	604
550	20	20	533
551	50	20	550
557	31	21	509
564	13	22	581
568	18	23	596
572	50	31	557
573	49	38	525
577	14	46	516
581	14	47	581
581	14	49	387
581	22	49	517
581	47	49	519
583	15	49	573
584	19	49	589
588	50	49	595
589	49	49	597
594	15	50	390
595	49	50	409
595	50	50	496
596	23	50	551
597	49	50	572
599	11	50	588
604	19	50	595

PŘÍLOHA P IV: SEŘAZENÍ POČTU KROKŮ A POČTU PŘÍKAZŮ MRAVENCE PRO SA ALGORITMUS

Seřazení podle počtu kroků		Seřazení podle počtu příkazů	
406	25	15	577
406	25	16	592
409	23	16	605
503	22	17	592
503	22	19	537
537	19	22	503
577	15	22	503
577	49	23	409
592	16	25	406
592	17	25	406
592	50	34	594
594	34	34	594
594	34	49	577
605	16	50	592

PŘÍLOHA P V: UKÁZKA PROGRAMU NEJRYCHLEJŠÍHO MRAVENCE SE SLOVNÍM POPISEM CESTY

```
IfFoodAhead[IfFoodAhead[Move,
  IfFoodAhead[Move, IfFoodAhead[Left, Move]]], Prog3[Right,
  IfFoodAhead[Move, Prog3[IfFoodAhead[Move, Right], Right, IfFoodAhead[
     Move, IfFoodAhead[Prog2[IfFoodAhead[Right, IfFoodAhead[IfFoodAhead[
          Prog2[Move, Prog3[IfFoodAhead[Prog3[Right, Left, Left], Right],
            IfFoodAhead[Move, Right], Prog2[Left, Left]]], Move],
         Prog2[Move, Move]]], Prog2[Right, Move]], Right]]]], Move]]
```

```
{{1, 32}, {2, 32}, {3, 32}, {4, 32}, {So}, {4, 31}, {4, 30}, {4, 29}, {4, 28}, {4, 27},
 {We}, {No}, {Ea}, {5, 27}, {6, 27}, {7, 27}, {So}, {We}, {No}, {Ea}, {8, 27}, {9, 27},
 {10, 27}, {11, 27}, {12, 27}, {13, 27}, {So}, {13, 26}, {13, 25}, {13, 24}, {13, 23},
 {We}, {No}, {Ea}, {So}, {13, 22}, {13, 21}, {13, 20}, {13, 19}, {13, 18}, {We}, {No},
 {Ea}, {So}, {13, 17}, {We}, {No}, {Ea}, {So}, {13, 16}, {13, 15}, {13, 14}, {13, 13},
 {13, 12}, {13, 11}, {13, 10}, {13, 9}, {We}, {No}, {Ea}, {So}, {13, 8}, {We}, {12, 8},
 {11, 8}, {10, 8}, {9, 8}, {8, 8}, {No}, {Ea}, {So}, {We}, {7, 8}, {No}, {Ea}, {So},
 {We}, {6, 8}, {5, 8}, {4, 8}, {No}, {Ea}, {So}, {We}, {3, 8}, {No}, {Ea}, {So}, {We},|
 {2, 8}, {No}, {Ea}, {So}, {2, 7}, {2, 6}, {2, 5}, {2, 4}, {We}, {No}, {Ea}, {So}, {2, 3},
 {We}, {No}, {Ea}, {So}, {2, 2}, {We}, {No}, {Ea}, {3, 2}, {4, 2}, {5, 2}, {6, 2}, {So},
 {We}, {No}, {Ea}, {7, 2}, {So}, {We}, {No}, {Ea}, {8, 2}, {So}, {We}, {No}, {8, 3},
 {8, 4}, {Ea}, {So}, {We}, {No}, {8, 5}, {Ea}, {9, 5}, {10, 5}, {11, 5}, {12, 5}, {13, 5},
 {14, 5}, {15, 5}, {So}, {We}, {No}, {Ea}, {16, 5}, {So}, {We}, {No}, {Ea}, {17, 5},
 {So}, {We}, {No}, {17, 6}, {17, 7}, {17, 8}, {Ea}, {So}, {We}, {No}, {17, 9}, {Ea},
 {So}, {We}, {No}, {17, 10}, {17, 11}, {17, 12}, {17, 13}, {17, 14}, {Ea}, {So}, {We},
 {No}, {17, 15}, {Ea}, {So}, {We}, {No}, {17, 16}, {Ea}, {So}, {We}, {No}, {17, 17},
 {Ea}, {18, 17}, {19, 17}, {So}, {We}, {No}, {Ea}, {20, 17}, {So}, {We}, {No}, {Ea},
 {21, 17}, {So}, {We}, {No}, {21, 18}, {21, 19}, {Ea}, {So}, {We}, {No}, {21, 20}, {Ea},
 {So}, {We}, {No}, {21, 21}, {21, 22}, {21, 23}, {21, 24}, {21, 25}, {Ea}, {So}, {We},
 {No}, {21, 26}, {Ea}, {So}, {We}, {No}, {21, 27}, {Ea}, {22, 27}, {23, 27}, {So}, {We},
 {No}, {Ea}, {24, 27}, {So}, {We}, {No}, {Ea}, {25, 27}, {So}, {We}, {No}, {25, 28},
 {25, 29}, {Ea}, {So}, {We}, {No}, {25, 30}, {Ea}, {26, 30}, {27, 30}, {28, 30}, {So},
 {We}, {No}, {Ea}, {29, 30}, {So}, {We}, {No}, {Ea}, {30, 30}, {So}, {30, 29}, {30, 28},
 {We}, {No}, {Ea}, {So}, {30, 27}, {30, 26}, {We}, {No}, {Ea}, {So}, {30, 25}, {We},
 {No}, {Ea}, {So}, {30, 24}, {30, 23}, {We}, {No}, {Ea}, {So}, {30, 22}, {We}, {No},
 {Ea}, {So}, {30, 21}, {30, 20}, {We}, {No}, {Ea}, {So}, {30, 19}, {We}, {No}, {Ea},
 {So}, {30, 18}, {We}, {29, 18}, {28, 18}, {27, 18}, {No}, {Ea}, {So}, {We}, {26, 18},
 {No}, {Ea}, {So}, {We}, {25, 18}, {No}, {Ea}, {So}, {We}, {24, 18}, {No}, {Ea},
 {So}, {24, 17}, {24, 16}, {We}, {No}, {Ea}, {So}, {24, 15}, {We}, {No}, {Ea}, {So},
 {24, 14}, {We}, {No}, {Ea}, {25, 14}, {26, 14}, {So}, {We}, {No}, {Ea}, {27, 14},
 {So}, {We}, {No}, {Ea}, {28, 14}, {So}, {28, 13}, {28, 12}, {We}, {No}, {Ea}, {So},
 {28, 11}, {We}, {No}, {Ea}, {So}, {28, 10}, {We}, {27, 10}, {26, 10}, {No}, {Ea}, {So},
 {We}, {25, 10}, {No}, {Ea}, {So}, {We}, {24, 10}, {No}, {Ea}, {So}, {24, 9}, {24, 8}}
```

PŘÍLOHA P VI: PROGRAM PRVNÍ SEKAČKY SE SLOVNÍM POPISEM CESTY

```
Prog2[Prog2[Frog[Prog2[V8A[
     V8A[Prog2[Prog2[Frog[{1, 4}], Prog2[Prog2[Frog[V8A[Left, Mow]], Prog2[{2, 1}, Mow]], Mow]],
       Prog2[Left, Prog2[Prog2[Prog2[V8A[
            Frog[Prog2[V8A[Prog2[Prog2[Prog2[Frog[{3, 1}], Prog2[{1, 7}, Frog[V8A[Left, Mow]]]],
                 V8A[Left, Prog2[V8A[Left, Left], Frog[V8A[Mow, Frog[Left]]]]]], Left], Prog2[
                 Left, Frog[Mow]]], Mow]], Left], Left], Frog[Frog[Prog2[Mow, Mow]]]], Mow]]],
      Prog2[Prog2[Prog2[V8A[Prog2[V8A[V8A[Frog[Frog[Frog[Prog2[Mow, Prog2[Prog2[
                    Prog2[Mow, Left], Mow], Prog2[V8A[Prog2[{5, 3}, Prog2[Prog2[V8A[Left, {2, 1}],
                        Mow], Left]], Prog2[Frog[Prog2[Mow, Left]], {6, 8}]], {6, 3}]]]]]]],
             V8A[{5, 3}, Prog2[Mow, Mow]]], Prog2[{8, 1}, V8A[V8A[Left, Frog[
                V8A[Frog[V8A[{2, 7}, Frog[V8A[Prog2[Mow, Mow], Frog[Mow]]]]], Prog2[
                  Frog[V8A[Frog[V8A[Frog[Left], Left]], Left]], Prog2[Frog[{5, 4}], Mow]]]]],
              Frog[Prog2[Frog[Frog[Left]], Prog2[{2, 5}, Frog[V8A[Prog2[Prog2[
                     {3, 3}, Left], V8A[Prog2[{6, 4}, Prog2[Frog[V8A[Frog[Prog2[
                           Prog2[V8A[Frog[Prog2[Prog2[Frog[Mow], Left], {8, 6}]], Left], V8A[
                             Frog[Left], Prog2[V8A[Prog2[V8A[Mow, Mow], V8A[Mow, Left]], Left],
                              Prog2[V8A[Frog[Mow], Mow], Prog2[Frog[Mow], {7, 8}]]]]], Left]],
                       V8A[Prog2[Prog2[Prog2[Mow, Left], Left], Prog2[Left, Mow]],
                        V8A[Left, Mow]]]], Left]], Left]], Mow]]]]]]]], {7, 1}], {6, 6}],
         Prog2[{4, 6}, Prog2[Prog2[V8A[Left, V8A[Frog[Prog2[Left, Left]], V8A[Frog[
                Prog2[Prog2[Prog2[Prog2[Prog2[V8A[V8A[Frog[Left], Prog2[V8A[Left, Mow],
                        Frog[V8A[Frog[{3, 5}], {3, 2}]]]], Frog[Left]], Frog[Mow]],
                    Frog[V8A[Left, Frog[Prog2[Frog[Mow], Prog2[Left, Frog[Frog[Left]]]]]]]],
                   Prog2[Mow, {6, 6}]], Left], V8A[Frog[{7, 5}], Mow]]], Prog2[Frog[Left],
                Frog[Prog2[Frog[Prog2[Frog[Mow], Prog2[{2, 3}, Prog2[Left, {3, 2}]]]],
                  V8A[Mow, Prog2[Mow, Prog2[Frog[Prog2[Prog2[V8A[Mow, Frog[Left]], Mow], Prog2[
                        V8A[{5, 4}, Frog[Frog[Prog2[Frog[Mow], Left]]]], Prog2[Left, Mow]]]],
                     Frog[{1, 6}]]]]]]]]]], Frog[Prog2[Prog2[V8A[Left, V8A[{8, 5},
                 Prog2[Frog[Frog[Prog2[V8A[V8A[Left, V8A[Left, Frog[V8A[Mow, Mow]]]],
                      Prog2[Prog2[{7, 3}, Prog2[{6, 4}, V8A[Left, Mow]]], Left]], Left]]],
                  {5, 8}]]], Mow], {5, 2}]]], Prog2[{6, 4}, Left]]]],
        V8A[V8A[V8A[Prog2[Prog2[Prog2[{4, 1}, Frog[Left]], Prog2[Frog[Frog[
                Frog[Frog[Prog2[Frog[Mow], {8, 8}]]]]], Frog[V8A[Left, Mow]]]], {3, 5}],
           Prog2[Prog2[V8A[Frog[Prog2[Prog2[Frog[Prog2[{1, 7}, Frog[Left]]],
                 Prog2[V8A[V8A[{1, 5}, Prog2[{2, 6}, Prog2[Mow, Mow]]], Prog2[{6, 7}, {8, 7}]],
                  V8A[V8A[Mow, Mow], Mow]]], V8A[Prog2[Mow, V8A[Prog2[Prog2[Left, {6, 5}],
                    Prog2[V8A[V8A[V8A[Mow, Prog2[V8A[{7, 7}, Mow], Prog2[Frog[Frog[Frog[Mow]]],
                          Prog2[Frog[Mow], V8A[V8A[Frog[V8A[Mow, Left]], Prog2[Frog[{4, 3}],
                              Frog[Frog[Frog[Left]]]]]], {1, 5}]]]]]], V8A[Mow, Frog[
                         V8A[Prog2[Prog2[Frog[Prog2[Frog[Frog[V8A[{6, 2}, Mow]]], V8A[
                               V8A[{4, 1}, Left], Mow]]], {4, 2}], Prog2[Left, {8, 7}]], Mow]]]],
                      Frog[Mow]], V8A[{7, 3}, Prog2[{5, 2}, V8A[Prog2[V8A[Frog[Frog[Prog2[
                             Prog2[Mow, Mow], Prog2[Left, V8A[Left, Prog2[{4, 3}, {7, 1}]]]]]],
                         Frog[{2, 5}]], Left], Left]]]]], {7, 3}]], Mow]],
              Mow], V8A[Mow, Mow]], Frog[Mow]], Frog[Mow]], {8, 3}]], Left]],
     Prog2[Left, Mow]], Mow]], Prog2[Frog[Frog[Frog[
      {2,
       7}]]], Mow]], Frog[{4, 5}]]
```

```
{{5, 4}, {1, 3}, {We}, {8, 3}, {1, 3}, {8, 3}, {7, 3}, {So}, {8, 8}, {Ea}, {1, 8}, {2, 8}, {No}, {We},
 {So}, {2, 7}, {Ea}, {4, 8}, {8, 2}, {No}, {We}, {7, 2}, {6, 8}, {5, 8}, {6, 3}, {So}, {Ea}, {7, 3},
 {8, 3}, {7, 8}, {6, 5}, {7, 5}, {8, 5}, {1, 5}, {No}, {1, 6}, {We}, {8, 6}, {So}, {8, 5}, {Ea},
 {8, 8}, {8, 3}, {8, 2}, {8, 3}, {1, 3}, {2, 3}, {No}, {2, 4}, {2, 5}, {2, 6}, {4, 8}, {8, 6}, {3, 2},
 {We}, {6, 8}, {So}, {7, 6}, {Ea}, {7, 2}, {3, 3}, {4, 3}, {7, 1}, {No}, {6, 8}, {5, 7}, {We}, {4, 7},
 {8, 8}, {So}, {4, 1}, {Ea}, {No}, {8, 8}, {8, 1}, {8, 2}, {8, 3}, {We}, {So}, {8, 2}, {8, 8}, {8, 7},
 {8, 6}, {8, 8}, {Ea}, {8, 8}, {1, 8}, {No}, {We}, {So}, {1, 7}, {Ea}, {2, 7}, {6, 1}, {No}, {We},
 {5, 1}, {5, 3}, {4, 2}, {So}, {Ea}, {No}, {8, 8}, {We}, {8, 8}, {So}, {8, 7}, {5, 4}, {4, 2}, {Ea},
 {8, 8}, {1, 8}, {2, 8}, {No}, {2, 1}, {4, 8}, {We}, {So}, {8, 8}, {4, 8}, {6, 7}, {2, 6}, {2, 5},
 {Ea}, {7, 2}, {8, 2}, {4, 1}, {No}, {8, 8}, {8, 1}, {8, 8}, {We}, {8, 7}, {7, 7}, {6, 7}, {5, 7},
 {So}, {2, 8}, {2, 7}, {2, 6}, {4, 8}, {Ea}, {6, 2}, {8, 4}, {No}, {8, 5}, {2, 1}, {8, 8}, {8, 7},
 {We}, {So}, {Ea}, {1, 7}, {2, 7}, {5, 1}, {No}, {5, 2}, {We}, {So}, {3, 1}, {1, 2}, {1, 1}, {4, 1},
 {Ea}, {No}, {8, 8}, {8, 1}, {8, 8}, {8, 7}, {8, 6}, {8, 5}, {8, 4}, {We}, {7, 4}, {6, 4}, {So},
 {4, 8}, {3, 7}, {3, 6}, {3, 5}, {3, 4}, {3, 3}, {3, 2}, {3, 1}, {Ea}, {4, 1}, {5, 1}, {6, 1}, {4, 8},
 {2, 7}, {8, 6}, {1, 6}, {2, 8}, {3, 8}, {No}, {1, 8}, {4, 4}, {We}, {8, 8}, {4, 4}, {8, 8}, {7, 8},
 {6, 8}, {6, 2}, {6, 4}, {So}, {6, 3}, {6, 3}, {Ea}, {7, 3}, {6, 2}, {7, 2}, {6, 8}, {7, 8}, {8, 8},
 {No}, {We}, {7, 8}, {6, 8}, {3, 6}, {So}, {Ea}, {4, 6}, {3, 5}, {4, 5}, {5, 5}, {6, 5}, {7, 5},
 {6, 8}, {7, 8}, {6, 8}, {No}, {We}, {5, 8}, {4, 8}, {1, 4}, {8, 2}, {7, 8}, {6, 6}, {5, 6}, {2, 2}}
```

PŘÍLOHA P VII: PROGRAM DRUHÉ SEKAČKY SE SLOVNÍM POPISEM CESTY

```
Prog2[
 Prog2[Frog[Prog2[Frog[V8A[Mow, Mow]], Prog2[Prog2[Prog2[Frog[V8A[Frog[Prog2[Left, Prog2[Frog[
            Prog2[V8A[Prog2[Frog[Prog2[Left, Mow]], Left], Frog[Frog[Prog2[Frog[Prog2[Frog[
                    Left], Prog2[Mow, Left]]], Mow]]]], Frog[Frog[V8A[Frog[{5, 3}], Mow]]]]],
           Frog[Prog2[Frog[Prog2[Mow, Frog[Mow]]], Prog2[{5, 6}, Prog2[Prog2[Mow,
               V8A[Frog[Frog[Mow]], Mow]], Frog[Frog[V8A[Frog[Frog[Mow]], Mow]]]]]]]]],
         Frog[V8A[Frog[Prog2[Prog2[Prog2[Frog[V8A[{6, 6}, {2, 1}]], Frog[Mow]],
             Frog[Prog2[V8A[Prog2[Frog[Prog2[Prog2[Frog[{6, 6}], Frog[Prog2[Frog[Frog[Frog[
                         Prog2[V8A[V8A[Prog2[Frog[{6, 7}], Mow], Mow], Prog2[Left, Prog2[Frog[
                             {8, 8}], Left]]], Prog2[Prog2[Left, Mow], Prog2[Left, Mow]]]]]],
                      Frog[Left]]]], Prog2[Prog2[{6, 7}, Frog[Mow]], Prog2[Mow,
                     V8A[Frog[Frog[Left]], V8A[Prog2[V8A[Mow, Prog2[{8, 1}, Mow]],
                        V8A[Frog[Frog[Prog2[Frog[Prog2[Prog2[Prog2[Prog2[Prog2[V8A[
                                   Prog2[Left, Mow], Frog[V8A[Mow, {4, 7}]]], Mow], Mow], Left],
                               Prog2[Frog[V8A[Left, Mow]], Mow]], Left]], Prog2[Mow, Left]]]],
                         Frog[Left]]], V8A[V8A[Frog[{2, 3}], Prog2[Mow, {6, 1}]], Prog2[
                         V8A[Frog[Frog[Mow]], Prog2[V8A[Left, Left], Frog[Left]]], Mow]]]]]]],
                 Frog[Prog2[Frog[Left], Left]]], Frog[Frog[V8A[{3, 7}, Frog[{7, 1}]]]]], Mow]]],
            Prog2[Prog2[V8A[Frog[Frog[Prog2[V8A[Prog2[V8A[Frog[Mow], Left], Mow],
                   Prog2[Frog[Left], Mow]], Left]]], Prog2[Prog2[V8A[Mow, V8A[{7, 3},
                   Prog2[Frog[V8A[{7, 4}, Mow]], Frog[Prog2[Mow, {8, 6}]]]]], Mow], Mow]],
              V8A[Left, Frog[V8A[Mow, V8A[Prog2[Mow, Left], {5, 4}]]]]], Mow]]],
          Prog2[Prog2[Frog[Prog2[Mow, Mow]], Left], Frog[Prog2[Prog2[Prog2[
               Prog2[Frog[V8A[Mow, Frog[Left]]], Left],
               Prog2[Prog2[Prog2[Frog[V8A[Left, Prog2[Mow, Frog[Frog[V8A[Left, Mow]]]]]],
                  Prog2[Prog2[Mow, V8A[V8A[Prog2[Prog2[Left, Mow], Mow],
                      Prog2[Frog[Frog[Prog2[Mow, Frog[V8A[Prog2[Frog[V8A[{8, 1}, Mow]], Mow],
                            Frog[Prog2[Frog[Left], Frog[Frog[V8A[Mow, Mow]]]]]]]]], Mow]],
                     V8A[Mow, Left]]], {3, 2}]], V8A[V8A[V8A[V8A[Prog2[Prog2[
                       Frog[Mow], V8A[Prog2[Prog2[V8A[Prog2[Frog[{7, 4}], Mow],
                           Prog2[Frog[Prog2[Frog[Mow], V8A[{1, 6}, Mow]]], {8, 3}]], {4, 8}],
                         Frog[Mow]], Prog2[Mow, Left]]], Left], Prog2[Frog[Mow], Mow]],
                    Frog[V8A[V8A[Frog[Mow], Left], Left]]], Frog[Mow]], Frog[Prog2[Prog2[
                     Prog2[Mow, Prog2[Frog[Mow], Frog[Frog[Mow]]]], Prog2[Frog[Mow], Prog2[
                       Prog2[Frog[Prog2[Prog2[Frog[Prog2[Left, {8, 7}]], Prog2[Mow, {1, 1}]],
                          Left]], Frog[Frog[{1, 3}]]], {3, 7}]]], Mow]]]], Frog[{2, 1}]]], Frog[
               {2, 6}]], Prog2[Frog[Left], V8A[Mow, Frog[Left]]]]]]]]], {4, 4}],
      V8A[Left, Prog2[Mow, Prog2[Prog2[V8A[Prog2[Mow, Prog2[Frog[Prog2[Prog2[
               V8A[Prog2[Mow, {1, 8}], {3, 2}], Mow], Mow]], {4, 6}]], {4, 1}],
          Frog[Frog[Frog[Prog2[Left, {2, 4}]]]]], Mow]]], Mow]]], {4, 5}], Frog[{5, 3}]]
```

```
{{5, 4}, {5, 5}, {5, 6}, {7, 3}, {We}, {So}, {7, 2}, {6, 1}, {Ea}, {No}, {4, 8}, {4, 1}, {We},
 {2, 8}, {1, 8}, {7, 7}, {5, 6}, {8, 1}, {7, 1}, {1, 5}, {3, 1}, {8, 3}, {7, 3}, {6, 3}, {4, 8},
 {3, 5}, {2, 5}, {1, 5}, {2, 8}, {3, 3}, {2, 3}, {1, 3}, {2, 8}, {3, 5}, {2, 5}, {5, 3}, {8, 5},
 {7, 2}, {6, 1}, {5, 7}, {4, 7}, {8, 8}, {6, 2}, {5, 4}, {4, 4}, {3, 4}, {So}, {3, 4}, {Ea}, {No},
 {3, 5}, {We}, {2, 5}, {7, 1}, {4, 5}, {1, 1}, {So}, {2, 8}, {7, 2}, {7, 1}, {6, 8}, {6, 7}, {Ea},
 {4, 8}, {2, 1}, {3, 1}, {4, 1}, {No}, {4, 2}, {4, 3}, {7, 4}, {7, 5}, {7, 6}, {We}, {So}, {7, 5},
 {5, 2}, {5, 1}, {Ea}, {4, 7}, {5, 7}, {No}, {4, 1}, {3, 7}, {We}, {6, 8}, {1, 6}, {8, 6}, {7, 6},
 {6, 8}, {5, 2}, {So}, {Ea}, {No}, {2, 8}, {2, 1}, {8, 5}, {We}, {8, 8}, {So}, {8, 3}, {1, 4},
 {1, 2}, {1, 8}, {1, 7}, {7, 1}, {7, 8}, {6, 8}, {Ea}, {7, 8}, {No}, {6, 8}, {6, 1}, {We}, {2, 7},
 {6, 1}, {5, 1}, {4, 1}, {4, 7}, {3, 7}, {6, 8}, {5, 8}, {4, 8}, {So}, {4, 7}, {4, 6}, {Ea}, {8, 4},
 {1, 4}, {8, 4}, {1, 4}, {2, 4}, {3, 8}, {No}, {3, 1}, {We}, {6, 8}, {4, 6}, {So}, {Ea}, {5, 6},
 {No}, {5, 7}, {7, 2}, {1, 3}, {2, 1}, {2, 2}, {We}, {1, 2}, {8, 2}, {7, 2}, {6, 2}, {5, 8}, {4, 8},
 {So}, {8, 8}, {8, 7}, {8, 6}, {8, 1}, {8, 4}, {4, 4}, {7, 2}, {6, 8}, {5, 6}, {5, 5}, {5, 4}, {Ea},
 {6, 4}, {4, 8}, {8, 1}, {1, 1}, {2, 1}, {4, 8}, {5, 8}, {7, 7}, {8, 7}, {8, 8}, {1, 8}, {No}, {We},
 {8, 8}, {8, 8}, {7, 8}, {6, 8}, {4, 8}, {So}, {Ea}, {2, 8}, {3, 8}, {6, 8}, {7, 8}, {8, 8}, {8, 8},
 {1, 8}, {2, 8}, {3, 8}, {4, 8}, {8, 8}, {No}, {8, 8}, {8, 1}, {We}, {8, 7}, {3, 6}, {6, 5}, {5, 5},
 {4, 3}, {5, 1}, {3, 1}, {So}, {6, 8}, {6, 7}, {Ea}, {4, 8}, {2, 6}, {2, 2}, {1, 5}, {No}, {1, 6},
 {1, 7}, {1, 8}, {1, 1}, {1, 2}, {4, 1}, {We}, {8, 8}, {4, 7}, {8, 6}, {7, 6}, {6, 6}, {8, 3}, {3, 2}}
```