Středoškolská odborná činnost 2005/2006

Obor 1 - Matematika a matematická informatika

Genetické programování

Autor:
Jan Tuček
SŠAKHK, Hradecká 1151/9
500 03,Hradec Králové 3, 2. ročník

Konzultant:
Ing.Václav Maixner

Hradec Králové, 2006
Královéhradecký kraj

Rád bych touto cestou poděkoval všem lidem, kteří mě podporovali při vzniku této práce.

Prohlašuji tímto, že jsem soutěžní práci vypracoval samostatně pod vedením Ing. Václava Maixnera a uvedl v seznamu literatury veškerou použitou literaturu a další informační zdroje včetně internetu.

Dne 22. dubna 2006 v Hradci Králové

_____

Podpis autora

Obsah

OBSAH 3

3 ZADÁNÍ 5

4 ÚVOD 5

4.1 Optimalizační algoritmy 5

4.2 Historie EVT 6

4.3 Úvod EVT 6

4.4 Techniky EVT 9
4.4.1 Genetické algoritmy 9
4.4.2 Evoluční strategie 9
4.4.3 SOMA 9

5 GENETICKÉ PROGRAMOVÁNÍ 10

5.1 Účelová funkce 11

5.2 Reprezentace jedinců 11

5.3 Inicializace populace 12

5.4 Rekombinační operátory 12

5.5 Více-typové problémy 13

5.6 Reprezentace jedinců v populaci 13
5.6.1 Kozova reprezentace 13
5.6.2 Gramatická evoluce 13
5.6.3 Kartézské genetické programování 15

6. ADT – ARITY DEFINITION TREE 15

7. GP SDK 18

7.1 Popis struktury GP SDK 18
7.1.1 WGPControl 19
7.1.2 Chromozom 20
7.1.3 Explorer 22

7.2 Určení fitness 22

7.3 Implementace GP SDK 24

8. PROBLÉM MRAVENCE NA STEZCE SANTA FE 25

9. ŘEŠENÍ PROBLÉMU MRAVENCE POMOCÍ GP SDK 27

9. VÝSLEDKY 28

9.1 Test 1 29

9.2 Test 2 30

9.3 Test 3 31

9.4 Test 4 32

9.4 Průměrné hodnoty 33

9.5 Vyhodnocení 33

10. ZÁVĚR 34

LITERATURA 35

POUŽITÉ ZKRATKY 35

3 Zadání

Vytvořit jednoduše použitelné programové jádro (SDK) pro genetické programování.

V průběhu vzniku práce jsem narazil na malé množství publikací v českém jazyce, které jsou sice kvalitní, ale tématem se nezabývají do dostatečné hloubky. Zjistil jsem, že počet lidí, kteří jsou obeznámeni s touto problematikou, je malý, a proto se v první části této práce pokusím vysvětlit základní myšlenky této problematiky.

V druhé části práce se zabýváme samotným programovým jádrem. Je prezentován způsob řešení popisu jedince v populaci i charakteristika způsobu práce s tímto systémem, možnosti rozšiřitelnosti a metodika při implementaci do klientských aplikací. V závěru práce je systém aplikován na problém umělého mravence na stezce "Santa Fe".

4 Úvod

Genetické programování patří do evolučních výpočetních technik (dále jen EVT), a proto jim budu věnovat část této práce. Následující text vychází z pramenů [1],[2],[3].

EVT jsou techniky určené k řešení mnoha různých problémů jako například nalezení optimálních parametrů funkce, nastavení neuronové sítě, hledání předpisu funkce podle známých vstupních a výstupních hodnot, programu jednoduchého stavového automatu apod. Jsou také užívány v mnoha různých oborech od fyziky až po ekonomii.

EVT jsou aplikovány na problémy, kde nelze využít tradiční analytické techniky nebo by bylo prohledávání celé množiny možných řešení časově velmi náročné. I když EVT jsou z velké části založeny na náhodě stejně jako náhodné prohledávání množiny možných řešení, i tak podávají výrazně lepší výsledky. Ne vždy jsou schopné nalézt absolutně nejlepší řešení, ale nalezené možnosti jsou získány v přijatelném čase a jsou alespoň částečným řešením.

Mezi známé a úspěšné aplikace patří návrh motorů pro letadlo Boeing 777. Motory byly navrženy klasickou cestou s důrazem na minimální spotřebu, ale pomocí aplikace genetických algoritmů (součást EVT), které byly použity na nastavení některých parametrů, byla spotřeba snížena o 2,5% oproti původnímu návrhu a finanční úspora činila dva miliony dolarů ročně. Toto využití EVT jasně naznačuje použitelnost těchto technik pro praxi, kde jsou schopny překonávat lidská řešení.

4.1 Optimalizační algoritmy

Optimalizační algoritmy jsou algoritmy určené k nalezení optimálního řešení daného problému, kdy se hledá minimum (maximum) účelové funkce, které říká, jak kvalitní je nalezené řešení.

Optimalizační algoritmy můžeme rozdělit na:

- numerativní – jde o úplné prohledání množiny možných řešení, a proto je nelze používat na složité úlohy z důvodu časové náročnosti; nalezené řešení je globální extrém účelové funkce.
- deterministické – algoritmy postavené pouze na přesných matematických postupech.
- stochastické – metody používající jen náhodu; náhodně se generují řešení a nejlepší z nich je výsledek.
- smíšené – kombinace deterministických a stochastických metod, které podávají velmi dobré výsledy. EVT patří do této kategorie.

EVT můžeme dále dělit podle technik způsobu reprezentace jedinců, podle typu úloh.

4.2 Historie EVT

Po krátkém představení EVT a zařazení mezi optimalizační algoritmy se podíváme, kdy tyto techniky vznikly a kdo jsou jejich tvůrci. Historie EVT je převzata z [1].

První příspěvek do EVT přinesli v polovině 60. let tři studenti z Berlínské univerzity, kteří použili náhodného kombinování existujících převodovek, aby získali konstrukci s lepšími vlastnostmi. Tento mechanismus můžeme považovat za křížení používané v EVT. V druhé polovině 60. let přišel Fogel s evolučním programováním. V jeho práci šlo o evoluci chování konečných automatů.

Další významný zlom přichází v roce 1975, kdy John Holland vydal svou knihu, ve které se snažil použitím algoritmů vysvětlit, proč se liší jedinci téhož druhu. Tím položil základy genetických algoritmů. V této práci pokračoval David Goldberg, který svou práci publikoval v roce 1989. Ten se již systematicky zabýval genetickými algoritmy a ukázal mnoho jejich aplikací.

Poslední autor, kterého zde zmíním je John Koza, kterého považujeme za zakladatele genetického programování, které se liší od genetických algoritmů hlavně reprezentací jedinců a množinou řešitelných úloh. Svoje práce vydal v roce 1992 a 1994. Jedná o novou techniku, jejíž rozvoj začal teprve nedávno. A právě genetické programování bude hlavní náplní této práce.

4.3 Úvod EVT

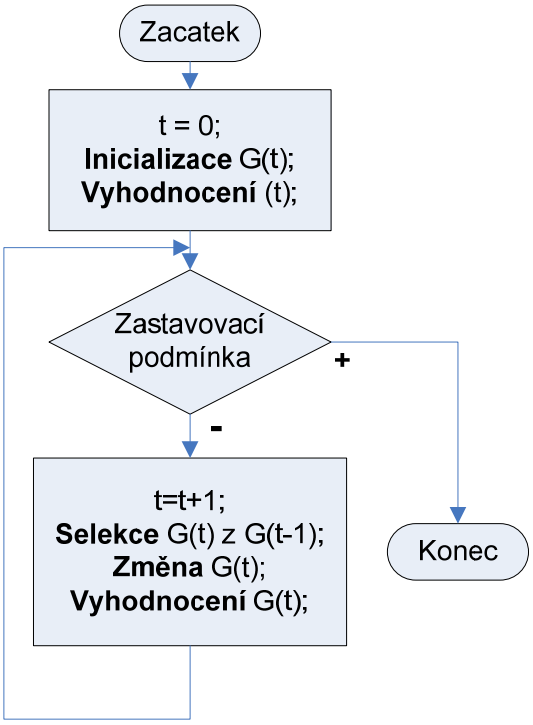
Jak již bylo řečeno, EVT jsou určeny k prohledávání prostoru možných řešení, kombinují deterministické se stochastickými algoritmy a jejich základní principy je inspirovány přírodou.

Většina ostatních optimalizačních algoritmů pracuje jen s jedním řešením, které postupně iteračně zlepšuje. EVT pracuje s množinou možných řešení, které mají různou hodnotu účelové funkce (anglicky označovanou jako ***fitness***), která říká, jak je dané řešení kvalitní. Množinu možných řešení označujeme jako populaci, která obsahuje ***N*** řešení (v literatuře se označuje jako PopSize). Jednotlivá řešení v populaci nazýváme ***jedinec***.

$$G(t) = x_{t,1}, x_{t,2}, ..., x_{t,N}$$

(4.1)

Každá populace prochází v průběhu času ***t*** změnami, které jsou aplikovány na jedince $x_{t,i}$. Populaci v určitém čase ***t*** nazýváme ***generace G(t)***. Každý jedinec $x_{t,i}$ je dán stejnou datovou strukturou. Ta může mít mnoho různých podob od vektoru, který popisuje vstupní parametry funkce, až po stromovou reprezentaci v jazyku LIST. Popisu některých struktur se budeme věnovat v dalších kapitolách.



Obr. 1 Algoritmus EVT

Obecný algoritmus EVT je zobrazen na Obr.1. Takové či velmi podobné schéma používají všechny EVT a je pro ně charakteristické. Tento obecný princip je velmi jednoduchý, většina problémů tkví v reprezentaci jedinců a určení fitness. Je-li problém jednoduchý, většinou je i snadné zapsat tento algoritmus, který obsahuje jednoduchou reprezentaci jedince. Některé typy EVT lze dokonce zapsat v tabulkových procesorech, což svědčí o jejich nenáročnosti. Se zvyšující se složitostí problému rostou i požadavky na reprezentaci jedince a s tím se zvětšují nejen nároky na reprezentaci jedince ale i na celý algoritmus. Samozřejmě že algoritmu s Obr.1 je zachován, ale jednotlivé části jako změna a vyhodnocení se komplikují.

Samotný algoritmus je založen na tom, že simuluje čas, při kterém se v populaci uplatňuje přirozený výběr, křížení, mutace a další kombinační operátory.

Při inicializaci se vygeneruje počáteční populace, která může vzniknou náhodnou cestou a nebo vytvořením jedinců podle určitých pravidel, vyplývajících ze znalosti řešeného problému. V mnoha případech neznáme nebo nejsme schopni použít znalosti o dané úloze k vytvoření počáteční populace v mnoho případech použijeme náhodné vytváření jedinců v populaci. Při použití této metody bude mít většina jedinců velmi špatné fitness, ale počáteční populace bude různorodá. Pokud použijeme při vytváření populace znalosti o problému, bude mít počáteční populace lepší průměrné fitness, ale kvůli nedostatečné různorodosti populace může dojít

k předčasnému uváznutí v lokálním extrému. Proto by se měly znalosti o daném problému aplikovat s rozmyslem.

Dalším krokem je vyhodnocení. V tomto kroku se určí fitness pro každého jedince. Stanovit hodnotu účelové funkce je mnohdy velmi složitá záležitost a špatně určená hodnota může způsobit neschopnost nalézt správné řešení. Může také nastat situace, kdy nejsme schopni číselnou hodnotu ***fitness*** určit. V takovém případě můžeme použít turnajovou selekci, kdy neurčujeme přesnou hodnotu, ale to, který z jedinců je lepší.

Nyní jsme se již dostali k hlavní části EVT, kde dochází k samotné evoluci jedinců. Ta probíhá v cyklu, ve kterém je stanovena zastavovací podmínka. Tou může být buď nalezení postačujícího řešení nebo vypršení maximálního času pro evoluci. Jedno zopakování odpovídá jedné generaci, a proto je v prvním řádku zapsán přírůstek času.

První významnou částí cyklu je ***selekce***, která slouží k odstranění špatných jedinců z populace a můžeme ji přirovnat k přirozenému výběru v přírodě. Nejdříve se určí kolikrát se daný jedinec objeví v další generaci. Nejlepší jedinec by měl mít největší zastoupení v příští generaci. Průměrný by se měl v příští generaci objevit jednou a nejhorší ani jednou. Další částí selekce je vzorkování, které provede samotný převod populace do další generace. Celý tento proces zlepšuje celkovou fitness populace, ale neslouží k nalezení nového lepšího jedince. Tato problematika je dobře zpracována v publikacích [1], [2], a proto se jí nebudeme více zabývat.

Dalším krokem je ***změna***. V této části již dochází k vytváření nových jedinců použitím kombinačních operátorů. Mezi nejpoužívanější operátory patří křížení a mutace. Jednotlivé techniky EVT používají různé kombinační operátory. I mutace a křížení mohou být jinak implementovány v jednotlivých algoritmech nebo se nemusí používat vůbec. Kombinační operátory jsou pro některé metody charakteristické.

Mutace je náhodná změna jedince. Způsob změny závisí na typu problému a způsobu reprezentace jedince, kde můžeme provádět mutaci na binární, dekadické nebo úplně jiné úrovni, podle způsobu reprezentace jedince.

Křížení je metoda, kdy dochází k výměně části jedinců mezi sebou. Tuto techniku lze přirovnat k páření v přírodě, i zde máme rodiče a potomky. Rodiče nemusí být vždy dva, ale třeba čtyři nebo jakýkoliv počet přesahující dva.

Kombinační operátory jsou aplikovány jen na část populace. Většinou převládá křížení nad mutací.

Závěrečným krokem je určení fitness současné generace, aby mohla být stanovena jak zastavovací podmínka tak i proběhnout selekce v další generaci.

Chybná implementace některé z těchto částí může vést k neschopnosti algoritmu nalézt dobré řešení. Chybná selekce může vyřadit nejlepšího jedince. Chybné kombinační operátory, kdy dochází k příliš velké změně jedince, mohou způsobit, že populace bude jen velmi špatně konvergovat ke správnému řešení.

4.4 Techniky EVT

Nyní již známe základní princip EVT, potřebnou terminologii, a proto se již můžeme podívat na jednotlivé techniky. První budou genetické algoritmy.

4.4.1 Genetické algoritmy

Genetické algoritmy (GA) byly prvně seriózně zkoumanou technikou z EVT. Jsou charakteristické tím, že jsou tvořeny strukturou $\boldsymbol{S}$ ve tvaru řetězce $(s_1,s_2,\ldots,s_L)$ o konečné délce $\boldsymbol{L}$, která se nazývá ***chromozom***. Struktura jedinců je tvořena prvky $\boldsymbol{s_i}$, které nazýváme ***alela*** a její konkrétní hodnotu, nazýváme ***gen*** v souladu s názvoslovím v genetice. Každý element obsahuje hodnotu z konečné abecedy $\boldsymbol{A_i}$. Většinou platí, že všechny prvky struktury jsou vybrány ze společné abecedy $\boldsymbol{Ai = A}$.

Tyto algoritmy mohou nabývat různých podob, od nejjednodušších, kde je abeceda $A \subset \{0,1\}$ až po algoritmy, kde každý prvek chromozomu je tvořen vlastní abecedou. V GA je vidět vysoká inspirace v názvosloví přírodou, ale nemá to nic společného s genovým inženýrstvím či manipulacemi s DNA v buňkách. GA se používají na širokou skupinu optimalizačních úloh od hledání maxima matematické funkce až po problém obchodního cestujícího. Více informací o této problematice lze nalézt např. v [1].

4.4.2 Evoluční strategie

Evoluční strategie (ES) pracuje přímo s reálnými čísly v podobě vektoru, který je mutován pomocí vektoru náhodných čísel.

4.4.3 SOMA

Autorem této metody je Ivan Zelinka, asistent na Institutu informačních technologií Fakulty technologické Univerzity Tomáše Bati ve Zlíně a podrobnosti viz. [3]. SOMA je zkratka slov: „Samo-Organizující se Migrační Algoritmus“. Nejedná se o klasického představitele EVT. Princip lze přirovnat ke smečce inteligentních zvířat (vlků), kteří hledají potravu (globální extrém účelové funkce) v terénu (prostor možných řešení).

5 Genetické programování

V této kapitole nastíníme, co je to genetické programování, jaké jsou jeho cíle a představíme základní způsoby reprezentace jedinců v populaci. Další kapitola již bude o nově vytvořené reprezentaci jedince v populaci.

Genetické programování (GP) je součástí evolučních výpočetních technik. GP se liší od ostatních technik především v cílové množině řešitelných problémů a tomu je přizpůsobena i reprezentace jedinců.

Genetické programování zřejmě představuje nejmladší z evolučních výpočetních technik [2]. Nejvíce se podobá genetickým algoritmům, liší se ale v množině řešitelných úloh, kde GP má za cíl automatickou syntézu programů, algoritmů či rozhodovacích postupů, na rozdíl od genetických algoritmů, kde je cílem nalézt závislé proměnné tak, aby účelová funkce dosáhla svého maxima(minima). Druhým rozdílem ve způsob reprezentace jedinců. Genetické programování používá stromovou strukturu, která může být zakódována do chromozomu nebo se může při evoluci chovat i jako strom. Možné aplikace jsou vypsány v tabulce 1, která je převzata z [2].

	Úloha	**Hledaný algoritmus**	**Vstup**	**Výstup**
1.	Indukce posloupnosti	Analytický předpis	Index	Element posloupnosti
2.	Symbolická regrese množiny dat	Matematický výraz	Nezávislé proměnné	Závislé proměnné
3.	Optimální řízení	Řídící strategie	Stavové proměnné	Akční veličiny
4.	Identifikace a predikce	Matematický model systému	Nezávislé proměnné	Závislé proměnné
5.	Klasifikace	Rozhodovací strom	Hodnoty atributů	Přiřazení do třídy
6.	Učení se cílenému individuálnímu chování	Program popisující chování	Vstupy ze senzorů	Akce organismu
7.	Odvození kolektivního chování	Program popisující chování jedince	Informace o vztahu jedince ke zbytku kolektivu	Akce jedince v kolektivu

Tabulka 1

Jako první příklad aplikace si uvedeme indukci posloupnosti. Je dána částečná posloupnost 1, 4, 9, 16, 25. Již na první pohled je jasné, že se jedná o posloupnost $K(i) = i^2$. Takováto posloupnost je velmi jednoduchá a její nalezení by netrvalo příliš dlouho. Uveďme nyní složitější posloupnost: 0, 1, 3, 6, 10, 15, 21. Na první pohled již není patrný předpis této funkce, který je $K(i)=i*(i-1)/2$, ale pro algoritmus genetického programování, jsou obě úlohy téměř totožné. Řešení úloh, které jsou zde uvedeny, by při větší populaci byly nalezeny již při inicializaci populace nebo v několika málo prvních generacích.

Princip genetického programování je stejný jako u EVT a je zobrazen na Obr. 1. Jsou v nich zachovány všechny části, ale jejich kód je výrazně složitější než u ostatních EVT. Proto se jednotlivým částem budeme podrobněji věnovat.

5.1 Účelová funkce

Vyhodnocení je jednou z nejkomplikovanějších částí genetického programování, protože špatně napsaná účelová funkce může způsobit snížení schopnosti nalézt řešení, jak ukazuje i náš test. U příkladu s indukcí posloupnosti se nám nabízejí hned dvě možnosti jak určit fitness. Můžeme tuto hodnotu stanovit jako počet shod se známou částí posloupnosti nebo jako součet odchylek:

$$^{j}f = \sum_{i=0}^{N} | k_i - {}^{j}K(i) |$$

(5.1)

kde ^{j}f je výsledná hodnota fitness daného řešení, k_i je známá hodnota, $^{j}K(i)$ je známá hodnota hodnoceného výrazu a N je počet známých prvků. Čím menší číslo vrací tato funkce, tím podobnější je řešení hledané posloupnosti. Hledáme tedy minimum funkce a nejlepší možné řešení bude mít hodnotu fitness rovnou nule.

Tento algoritmus generuje velmi složitá řešení, která mnohdy popisují jednu a tu samou funkci, protože výraz ***K(i)=i*i*i*i*** je stejný z hlediska odchylky jako $K(i)=i^4$. Tato vlastnost je způsobena tím, že algoritmus nerozlišuje jednoduší a složitější řešení a neumí říct, který ze dvou zápisů je esteticky hezčí. Složitost řešení můžeme upravovat různými způsoby jako ***editací***, což je jeden ze sekundárních kombinačních operátorů, o kterém bude řeč později a nebo ovlivněním účelové funkce tak, aby preferovala kratší řešení. Můžeme ji upravit třeba takto:

$$^{j}f = \sum_{i=0}^{N} | k_i - {}^{j}K(i) | + k * len(K())$$

(5.2)

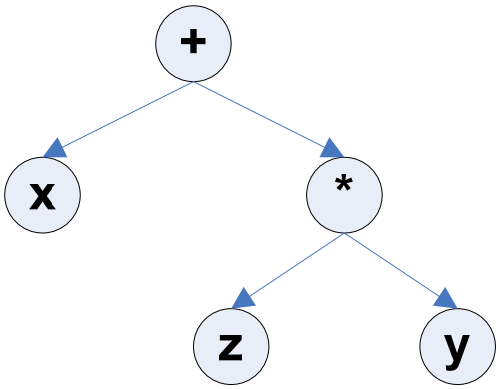
kde ***len(K())*** je funkce, která určuje délku daného řešení a ***k*** určuje důležitost této hodnoty, neboli jaký důraz přikládáme délce řešení. Toto číslo by nemělo být konstantou, ale mělo by se dynamicky měnit úměrně k průměrné hodnotě fitness všech jedinců tak, jak je definována v (5.1).

V genetickém programování se také požívá takzvaná turnajová selekce, kde není třeba určit přesnou hodnotu účelové funkce, ale jenom to, který ze dvou jedinců je lepší.

5.2 Reprezentace jedinců

Genetické programování má za cíl hledat programy, algoritmy, předpisy funkcí apod. Proto jsou na reprezentaci jedinců kladeny velké nároky. Nejznámější techniky jsou gramatická evoluce a Kozova reprezentace. Nejčastějším způsobem je stromová struktura, která je tvořena uzly, ve kterých jsou funkce a listy, kde se nachází konstanty a proměnné. Jednoduchý strom je

znázorněn na Obr. 2 a obsahuje předpis funkce ***f=x+z*y***. Struktury jsou tvořeny množinou ***T terminálních symbolů***, představujících listy stromu a množinu ***neterminálních symbolů F***, které jsou uzly stromu. V případě zobrazeném na Obr. 2 jsou terminály ***T={x,y,z}*** a neterminálních ***F={*,+}***. Tyto množiny se mohou zjednodušit do společné množiny ***C ={x,y,z,*,+}***, která by měla být adekvátní k řešené úloze.



Obr. 2 Ukázka stromu

5.3 Inicializace populace

Inicializací počáteční populace se rozumí vytvoření stromové struktury jednotlivých jedinců v populaci. Špatně vygenerovaná populace může způsobit rychlejší uváznutí v lokálním extrému.

Mnoho jedinců v počáteční populaci je velmi špatných a při použití čistě náhodné tvorby budou stromy velmi hluboké a nesmyslné. Z toho důvodu se omezuje maximální hloubka generovaných stromů.

Pro generování populace se používá metoda lineárně půl na půl. O této metodě a problematice generování stromů se lze dočíst např. v [2], kde je tato problematika velmi dobře zpracována.

5.4 Rekombinační operátory

Křížení a mutace jsou primární rekombinační operátory v genetickém programování. Hlavní je křížení, do kterého stejně jako v GA vstupují dva rodičové a vystupují dva potomci. Implementace je následující: zvolí se libovolný uzel nebo list u obou rodičů. Následovně se celá větev, která je za bodem mutace, vymění, a tak vzniknou dva potomci.

Mutace probíhá jen u jednoho jedince, kde se zvolí náhodně místo mutace a pod tímto bodem se znovu vygeneruje větev.

Genetické programování obsahuje ještě sekundární rekombinační operátory, kterými jsou ***permutace, editace, zapouzdření a decimování.***

Permutace je proces, při kterém se vybere určitý uzel a jeho vstupy se prohodí. Počet způsobů prohození větve odpovídá faktoriálu počtu vstupů permutované funkce.

Editace je proces, který se používá buď pro kosmetickou úpravu výstupu nebo pro zjednodušování řešení při evoluci. Jejím cílem je podle známých pravidel zjednodušit daný

program. Tato operace výrazně zjednodušuje řešení, ale také může způsobit snížení rozmanitosti populace.

Zapouzdřením rozumíme to, že zvolená větev je označena za funkci z množiny neterminálních symbolů a již se nemění.

Sekundární rekombinační operátory nejsou pro samotnou evoluci důležité, ale mohou evoluci urychlit či přispět k nalezení globálního extrému. Problematika rekombinačních operátorů je dobře popsána v [2].

5.5 Více-typové problémy

Můžeme se také setkat s problémy, které neobsahují jen jeden datový typ (jako třeba celá čísla), ale i více typů najedou třeba (logická hodnota a reálné číslo). V této situaci nastává problém, aby při křížení nedošlo k přesunutí větve vracející určitý typ pod uzel, který požaduje typ jiný. Například v programu, kde by byly jak celočíselné hodnoty tak i logické (pravada, nepravda). V takovémto případě může klidně dojít k přesunu logické proměnné pod funkci násobení, která požaduje celočíselnou hodnotu.

Řešení takového problému je několik: buď aplikace metod jako ADT nebo gramatická evoluce nebo při použití klasické Kozovy reprezentace provádět typovou kontrolu, kde se vyberou jen uzly, které mají stejný typ výstupu, a u nich se provede křížení. Toto řešení znamená zbytečné režijní výdaje na typovou kontrolu.

Více informací o této problematice lze nalézt [2].

5.6 Reprezentace jedinců v populaci

Následující podkapitoly jsou věnovány nejznámějším reprezentacím jedinců v populaci. Mezi ně patří Kozova reprezentace, gramatická evoluce a kartézské genetické programování.

5.6.1 Kozova reprezentace

Koza je zakladatelem genetického programování a je nejblíže obecné teorii. Jedinec se i během evoluce chová jako strom. Jeho implementace byla v jazyku List a trpí problémy u více-typových úloh.

5.6.2 Gramatická evoluce

Jedná se o metodu, která již nepatří čistě do genetického programování, ale i do genetických algoritmů. Cílem této metody je hledat programy přičemž jedinci jsou uchováváni jako lineární řetězec.

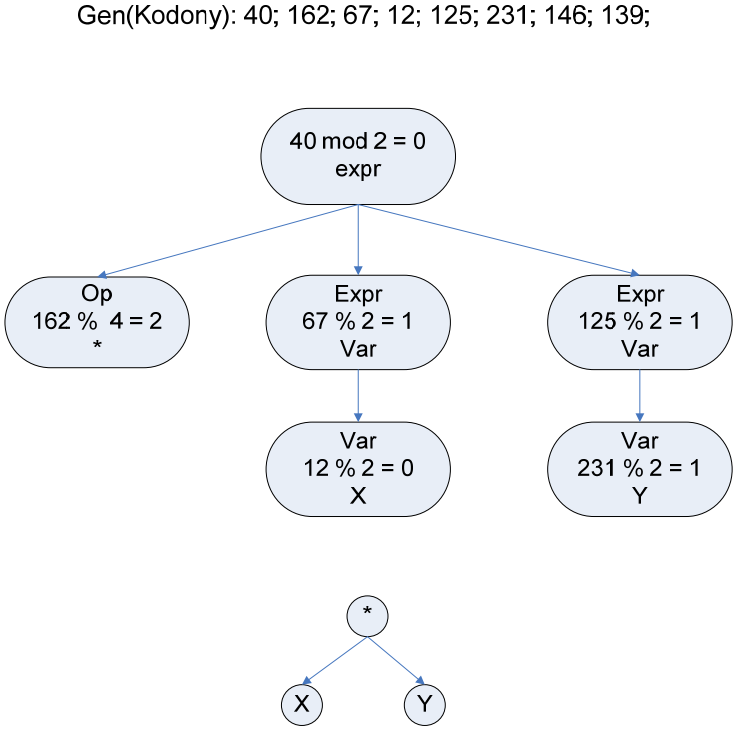
V této reprezentaci je jedinec popsán bezkontextovou gramatikou, ve které vystupují terminály a neterminály, které jsou rozvíjeny pomocí pravidel v jeden nebo více terminálů a neterminálů. Každý neterminál může mít více alternativních pravidel podle kterých se bude rozvíjet. Také je nutné určit, které pravidlo rozvíjí kořen jedince. Uvažujme jedince, který může být tvořen množinou funkcí {+,-,*,/} a proměnnými X,Y. Dále existují pravidla, kde na pravé straně je rozvíjený symbol a nalevo pravidla, podle kterých se může rozvíjet.

expr	::= op expr expr	(0)
	| var	(1)
op	::= +	(0‘)
	| -	(1‘)
	| *	(2‘)
	| /	(3‘)
var	::= X	(0‘‘)
	| Y	(1‘‘)

Tato gramatika se následovně použije pro dekódování lineárního chromozomu, který je potom dělen po osmi bytech, které nazýváme ***kodony*** a určují, jaká pravidlo se použije. Kodon se na dané pravidlo převede podle následujícího vztahu:

$$pravidlo = kodon \bmod počet\_pravidel\_pro\_daný\_neterminál \tag{5.3}$$

Protože kodon může nabývat hodnot 0-255 a převede se tím to na číslo pravidla, které bude použito k rozvoji. Celý proces je znázorněn na následujícím schématu:



Obr. 3 Převod chromozomu na strom

Převod probíhá následovně. Jako kořen je zvoleno pravidlo ***expr.*** Při rozvoji tohoto pravidla byla vybrána první možnost a to rozvoj na op expr expr. Pokračuje se rozvojem uzlu, který je nejvíce nalevo, pro něj dojde k výběru operátoru pro násobení, která se již dále nerozvíjí. Algoritmus se vrátí o úroveň výš a vybere další uzel, který se bude rozvíjet.

Při dekódování mohou nastat dvě možnosti: buď je strom příliš dlouhý, což se stalo i v našem případě (tuto možnost nemusíme nijak řešit), a nebo může dojít k tomu, že bude nedostatek kodonů pro rozvoj pravidla. V tom případě se můžeme opět vrátit k prvnímu kodonu a opakovat průchod chromozomem. Musíme přitom kontrolovat maximální počet opakování chromozomu aby nedošlo k zacyklení.

Tato metoda vyniká především bezproblémovým křížením a mutací, kde nás nemusí díky rozvoji pomocí pravidel, zajímat obsah jedince. Tato technika nemá potíže s více typovými problémy.

Mutace probíhá změnou jednoho kodonu. Při křížení si zvolíme bod křížení v lineárním chromozomu a všechna data za tímto bodem se prohodí. Tím dochází nejen ke změně větve, která je vybrána pro křížení, ale k úplné destrukci části stromu, která se rozvíjí za tímto bodem.

Tato reprezentace má ještě tu vlastnost, že pro zjištění informace o libovolném bodu chromozomu, musíme provést rozvoj tohoto stromu k danému bodu.

Jedná se zřejmě o jednu z nejlepší dosud známých metod genetického programování pro více typové problémy díky jednoduché aplikaci primárních rekombinačních operátorů.

5.6.3 Kartézské genetické programování

Kartézské genetické programování se již výrazně liší od klasického genetického programování. Jedinec již není tvořen stromovou strukturu ale "sítí". To znamená, že uzel nižší úrovně může být připojen na více elementů ve vyšší úrovni.

Chromozom je tvořen lineární posloupností genů, která reprezentuje jednotlivé elementy v řešení. V každém genu jsou obsaženy jeho vstupy, které mohou ukazovat jen na uzly, které jsou v chromozomu na nižší úrovni. Oddělenou částí je potom množina, která obsahuje indexy elementů, které tvoří výstupy. Tato metoda je schopna pracovat s úlohami, které mají více výstupů, což je vhodné při automatickém návrhu logických obvodů.

O kartézském genetickém programování se podrobněji zmiňují publikace [5] a [6].

6. ADT – Arity Definition Tree

V této kapitole je představena nová reprezentace jedince, kterou jsem vytvořil. Dále práce bude pokračovat představením programového jádra GP SDK, ve kterém je tato reprezentace implementována.

Při tvorbě programového jádra jsme byli postaven před problém, jakou reprezentaci jedince zvolit. Měli jsme na výběr z těch neznámějších, a to gramatická evoluce, Kozovy stromové reprezentace a kartézského genetického programování. Jako první jsme vyloučili kartézské genetické programování, protože není dostatečně obecné a lze ho aplikovat jen na omezenou množinu úloh. Proto by tato metoda nebyla vhodná při implementaci v programovém

jádře, které je navrhováno s ohledem na velkou množinu řešitelných úloh. Obě zbylé metody jsou již dostatečně obecné, ale každá z nich trpí určitými problémy. Kozova reprezentace má potíže s více-typovými problémy a gramatická evoluce má nesnáze s křížením, kde dochází k velmi výrazným změnám jedince.

Z tohoto důvodu jsme vytvořili novou reprezentaci jedinců, která kombinuje vlastnosti obou dvou. Z Kozovi reprezentace přebírá vlastnosti stromu a z gramatické evoluce základní myšlenky reprezentace a schopnost jednoduše řešit více typovéproblémy.

Stromová struktura je uchovávána v podobě lineárního chromozomu. Tento chromozom je tvořen dvěma logicky oddělenými množinami dat. První množina obsahuje strukturu stromu a druhá funkci (konstantu, vstup) daného uzlu (listu). V současné podobě má chromozom lineární délku, ale program v něm uchovávaný je limitován maximální délkou, takže počet uzlů a listů může být jeden až maximální délka chromozomu.

První množina obsahující informace o struktuře stromu se nazývá ***Tree*** a skupina data, která obsahuje funkce uzlů, se nazývá ***Func***.

Inovátorský algoritmus mapování struktury stromu dal název i této metodě - Arity Definition Tree(ADT). Přeložit do češtiny můžeme jako arita[1] definující strom. Právě arita funkce hraje v ADT rozhodující roli, neboť se účastní procesu vytvoření stromové struktury i určení významu daného uzlu nebo listu . Celý tento proces popisuje obr.4.

Princip převodu množiny Tree na strom je velmi jednoduchý. Chromozom obsahuje posloupnost čísel určující aritu daného uzlu. Algoritmus převodu je následující. Vezme se první číslo a tím je určena arita prvního uzlu. Ten následovně rozvíjíme tak, že vezmeme další číslo z řady a určíme aritu elementu, který je pod prvním uzlem. Tento proces dále pokračuje ve vnořování a ve chvíli, když narazíme na terminál, se vnořování zastaví a vrátí se o úroveň výš. Takto rozvineme celou větev a poté se opět vrátíme k prvnímu uzlu a rozvineme další větev podle následujících prvků množiny ***Tree***. Proces je graficky znázorněn v kroku jedna.

Dalším krokem je určení významů jednotlivých elementů stromu. Informace o tom, která funkce má být použita v daném uzlu nebo listu, je obsažena v posloupnosti čísel označované jako ***Func***. Samotný proces je následující, každé funkci či vstupu f_i stromu je přiřazen F_i prvek množiny ***Func***. Pořadí prvků stromu vyplívá ze způsobu vzniku stromu. Kořen stromu má index jedna. Následující uzel má hodnotu dva. Takto se dále vnořujeme do větší hloubky stromu a přičítáme index až narazíme na list, a potom se vrátíme o stupeň výš a pokračujeme v přidělování čísel až určíme index všem prvkům stromu. Dále máme tabulku funkcí a vstupů roztříděnou podle arity. Samotné číslo funkce v daném uzlu je dáno následujícím vztahem (6.1), kde číslo funkce zjistíme jako zbytek po dělení F_i počtem funkcí, které mají aritu jako uzel, pro který určujeme jeho funkci. Tento proces je znázorněn v kroku dva. V závěrečné etapě se určí z těchto hodnot, která funkce jim odpovídá.

Popis této metody může vypadat zdlouhavě, ale jedná se o velmi jednoduchou věc, kterou lze dobře implementovat pomocí rekurze.

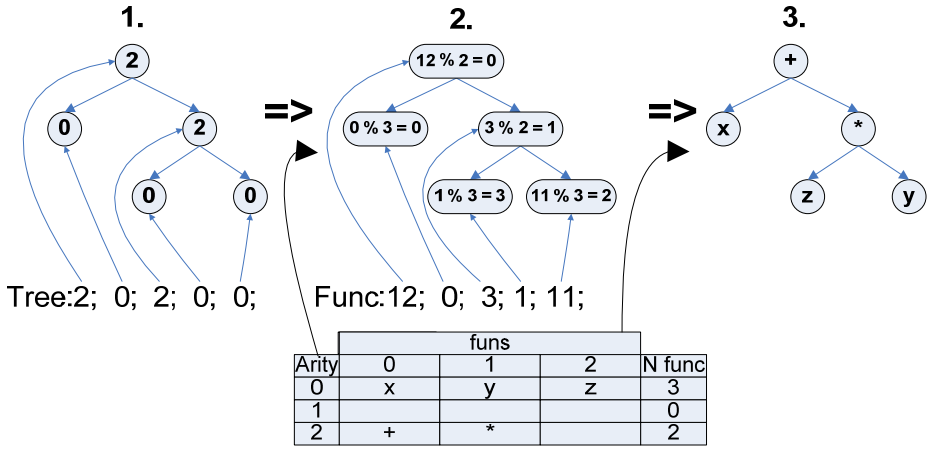
Další vlastností je, že v každém okamžiku jsme schopni určit aritu dané funkce, kde daná větev chromozomu začíná a kde končí.

Nyní je načase objasnit, jak si ADT poradí s více-typovými problémy. Základní princip je zachován, ale funkce a vstupy jsou rozděleny do oddělených tabulek(množin) podle typu jejich výstupu. Při určování významu elementu stromu se postupuje následovně. Funkce, která je v kořenu stromu, je vybrána z množiny(tabulky) funkcí, které mají stejný návratový typ jako je požadovaný návratový typ programu. Výběr funkcí a vstupů pod tímto uzlem je dán vstupními

[1] arita je počet argumentů dané funkce

typy tohoto uzlu. Například kdyby první uzel měl na vstupu celé číslo a logickou hodnotu, tak první uzel pod tímto uzlem je vybrán z dané množin funkcí, které vrací celé číslo a bude přiřazen pod vstup, který požaduje celé číslo a úplně stejně to provedeme i s logickou hodnotou. Tímto způsobem můžeme postupně rozvíjet strom více-typových programů až k listům.

$$f_i = F_i \bmod množství\_funkcí\_o\_dané\_aritě \tag{6.1}$$



Obr. 4 Proces převodu chromozomu na program

Počet prvků stromu je dán rovnicí (6.2), kde ***len*** je délka jedince a a_i je prvek z množiny Tree.

$$\sum_{i=1}^{len} (a_i - 1) = -1 \tag{6.2}$$

Pro určení délky větve, které začíná v libovolném bodě chromozomu, kde ***n*** je bod na kterém začíná větev, lze analogicky psát:

$$\sum_{i=n}^{len-n} (a_i - 1) = -1 \tag{6.3}$$

Pro maximální počet P_{max} prvků stromu o dané hloubce h_{max}, kde In_{max} je maximální počet vstupů(výstupů) uzlu, lze odvodit vztah:

$$P_{\max} = \sum_{i=0}^{h_{\max}} In_{\max}^{i} \tag{6.4}$$

ADT se v jedno typových úlohách chová jako Kozova reprezentace, ale při více-typových úlohách si zachovává jednoduchost bez nutnosti typové kontroly při křížení jedinců. V současné době probíhá testování této reprezentace a dosavadní výsledky ukazují, že tato metoda je schopna řešit základní úlohy. To jsou základní informace o ADT a nyní se již můžeme pustit do popisu GP SDK.

7. GP SDK

Hlavní cílem této práce je vytvořit programové jádro pro genetické programování, pro které se v anglické literatuře používá výraz Software Development Kit(SDK). Jedná se komplexní vývojový balík obsahující knihovny, dokumentaci a ukázkové aplikace. SDK využívají další vývojáři, kteří použijí již hotové knihovny. Tento přístup zkracuje a zlevňuje vývoj jejich aplikací. SDK jsem nazval GP SDK(Genetic Programming SDK), které je volně ke stažení na www.gpsdk.wz.cz.

Hlavními rysy jsou jednoduchá použitelnost, obecnost, rozšiřitelnost a implementace v jazyce C#. Jediným problémem, který musí řešit klientská aplikace je určení fitness jedince.

Implementace v jazyce C# znamená, že tento produkt může využít vývojář libovolného z jazyků .NET a proto ho má možnost použít široká programátorská komunita v mnoha různých programovacích jazycích.

7.1 Popis struktury GP SDK

Základ tvoří namespace ***GP***. Ten obsahuje tři třídy, které jsou na sobě přímo závislé, protože populace jedinců ve ***WGPControl*** je tvořena polem tříd ***Chromozom*** a při určování fitness je volán delegát, který má jako vstupní parametr typ ***Explorer***.

Název	Popis
WGPControl	Stará se o inicializaci populace a evoluci.
Chromozom	Obsahuje kód jedince a základní metody pro práci s jedincem.
Explorer	Pomocná třída při určování fitness.

Model je založen na oddělení jednotlivých částí algoritmu, aby při případném rozšiřování stačilo přeimplementovat jen jednu logickou část systému, a to buď jedince, nebo správce populace. To se provede děděním. Tím je dosažena jednoduchá rozšiřitelnost, kterou můžeme použít při určité znalosti o dané úloze, kde vývojář získává možnost aplikovat danou znalost

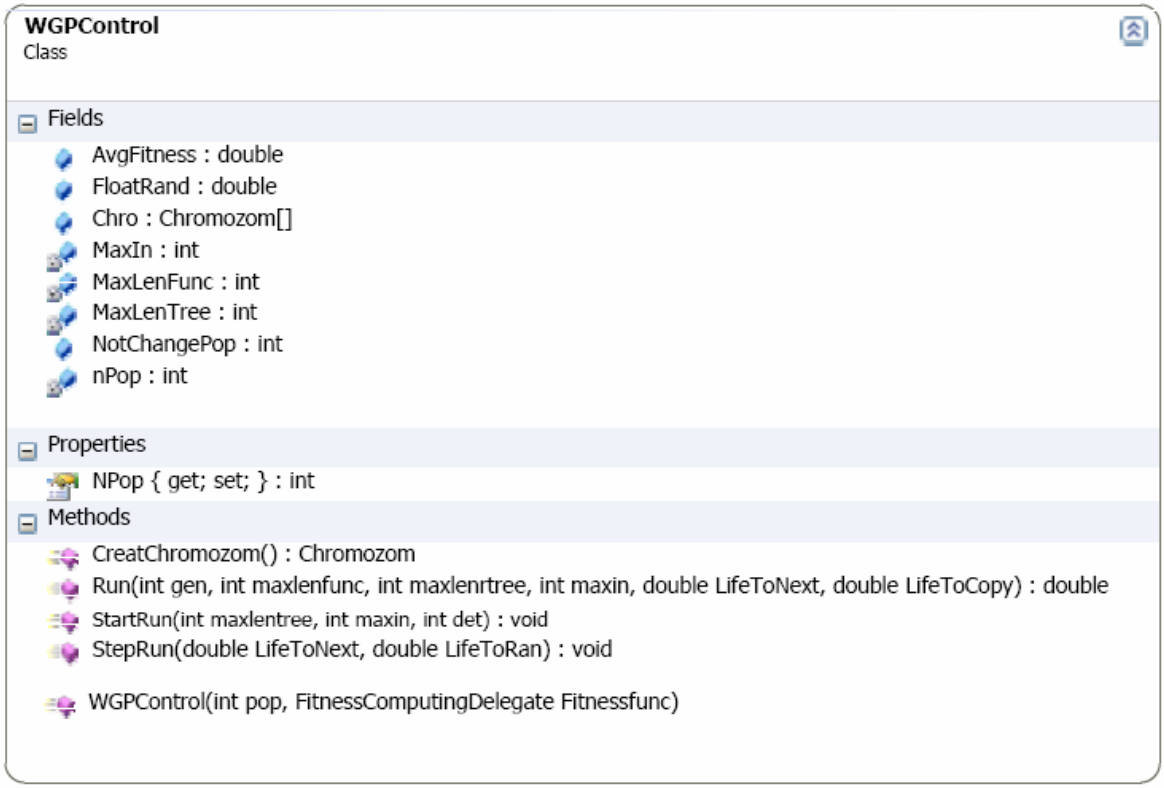
přímo do algoritmu a tím zvýšit schopnost nalézt globální extrém. Přitom ale nemusí znovu programovat celý algoritmus, stačí mu jen přeimplementovat část, o které má danou znalost.

7.1.1 WGPControl

Tato třída je správcem populace a stará se o inicializaci a evoluci. Klíčovou úlohou třídy je vytvoření další generace, kde se uskutečňuje selekce a aplikuje rekombinačních operátory, které provádí voláním metod jedinců.

Populace je tvořena polem typu ***Chromozom***. Každý element je jedinec, který umí na sebe používat jednotlivé rekombinační operátory.

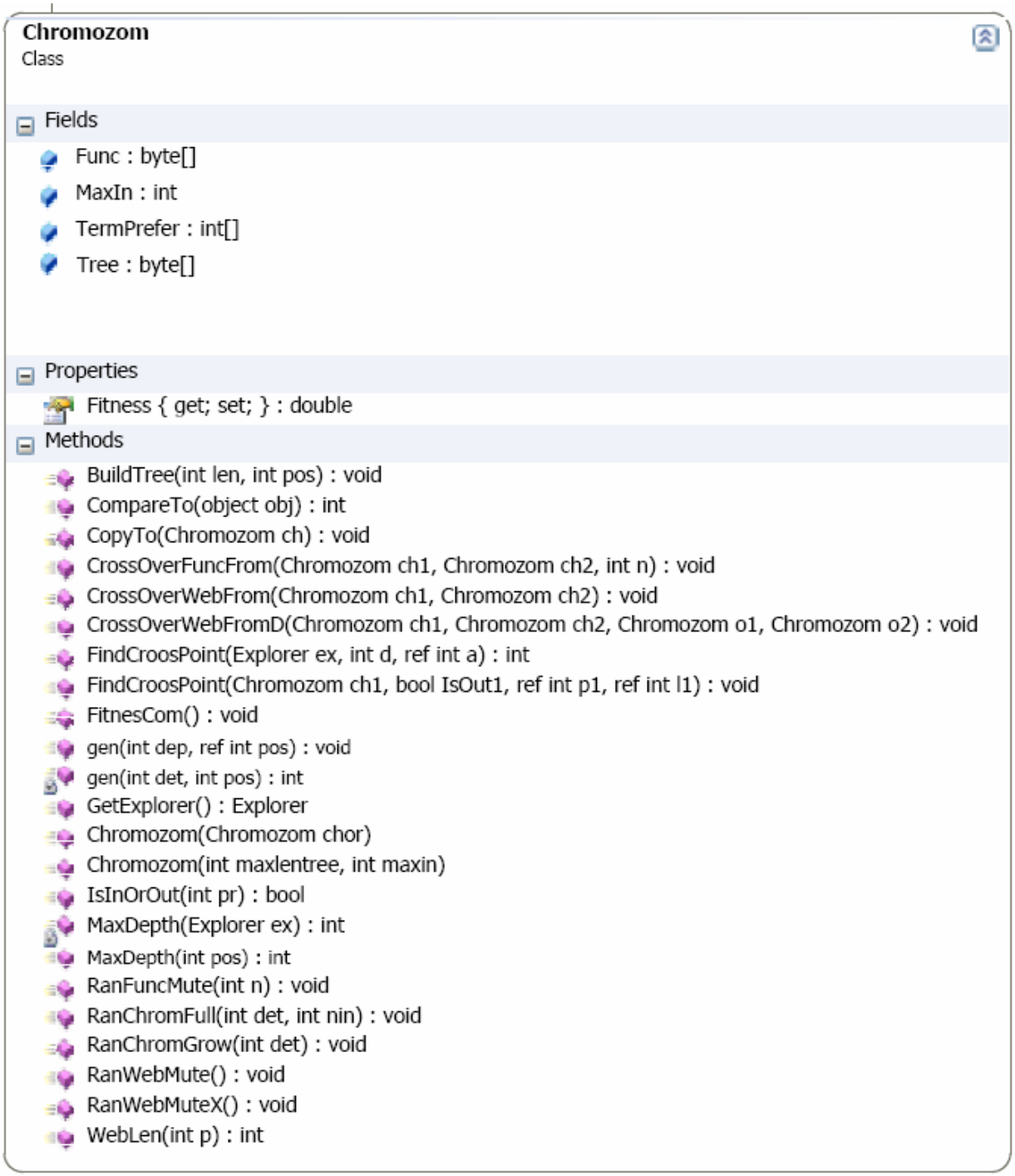
Při vytváření objektu je zadána velikost populace a také je předán delegát na funkci schopnou určit fitness jedince. Dalším krokem je zavolání metody ***StartRun***, která se stará o vytvoření počáteční populace. Zde je potřeba určit další informace o jedincích jako je největší hloubka generovaného stromu, maximální počet vstupů (výstupů) uzlu a elementů stromu. Po vygenerování již stačí jen zavolat metodu ***StepRun***, která vytvoří novou generaci. Tato metoda v první fázi provede selekci jedinců a poté na ně aplikuje mutaci a křížení v poměru, který je dán vstupními parametry. Ty vyjadřují, jak velká část populace vznikne použitím daného rekombinačního operátoru. Pro parametry 0,8 pro křížení a 0,1 pro mutaci vznikne 80% jedinců křížením a 10% mutací a 10% bude zachováno z minulé generace. Při selekci se používá lineární transformace ohodnocení s dynamickým parametrem ***s***. V závěru této metody je populace setříděna podle kvality jedinců od nejlepších po nejhorší.



Obr. 5 WGPControl – metody a datové složky

7.1.2 Chromozom

Tato třída reprezentuje jedince v populaci a implementuje metody pro práci s ním. Implementuje metodu pro vygenerování jedince, provádění rekombinačních operátorů důležitou součástí je vlastnost ***Fitness***, která určuje kvalitu daného řešení. Hodnota nula při tom určuje nejlepší řešení.



Obr. 6 Chromozom

Další zajímavostí je pole ***TermPrefer***, které určuje preferenci uzlů s určitou aritou při generování stromů. To může výrazně urychlit nalezení dobrého řešení. I když nastavíme preferenci nula, uzel s danou aritou se může ve stromě vyskytnout.

V současné době obsahuje tento objekt některé metody, se kterými se nepočítá v dalších verzích.

7.1.3 Explorer

Tato třída je využívána při určování hodnoty fitness a slouží k jednoduchému procházení stromu daného jedince. Odkaz na tento objekt získá uživatelsky definovaná metoda, která se stará o určování fitness, na níž byl při inicializaci ***WGPControl*** předán delegát.

Nejednoduší způsob procházení stromu jedince je pomocí rekurze. Tomu je přizpůsobena daná třída, kde každá instance reprezentuje určitou pozici ve stromu a je schopna vytvořit instance na objekty, které reprezentují uzly pod tímto bodem.

Obr. 7 Explerer

Ke všem třídám je vytvořena XML dokumentace ve zdrojovém kódu, kde lze nalézt podrobnosti o jednotlivých metodách.

Zde prezentovaná verze GP SDK není finální verzí, je potřeba dodělat kompletní infrastrukturu třídy a odstranit metody, které jsou v tomto produktu již jen z historických důvodů.

7.2 Určení fitness

Jediným velkým problémem, který musí řešit klientská aplikace GP SDK, je nutnost vytvořit hodnotící funkci (fitness), která říká jak je dané řešení dobré. Tuto problematiku nelze nijak obejít, protože každý řešený problém má tuto funkci jinou. Ale GP SDK dává velmi dobrý základ pro implementaci, protože se nemusíte starat o to, jak jedince dekódovat, ale o co nejlepší určení fitness.

Celý proces určování fitness začíná již při vytvoření třídy ***WGPControl***, kdy musíme předat konstruktoru delegáta na hodnotící metodu. Ta vrací hodnotu fitness a jejím parametrem je instance třídy ***Explorer***, která nám celý problém velmi zjednoduší.
Metoda určující fitness by měla mít singularitu danou následujícím delegátem.

```
public delegate double FitnessComputingDelegate(Explorer ex);
```

A samotná funkce by mohla vypadat následovně:

```
public double UrciFitness(Explorer ex)
{
      double fitness;
      //určení fitness
      return fitness;
}
```

Uvést přesný návod, jak napsat samotné tělo metody nelze, ale obecně doporučujeme, aby k tomu byly použity minimálně dvě metody. Metoda, která je předána delegátem, by se neměla starat o samotné provádění a analýzu programu uloženého ve stromu, ale nastavit počáteční hodnoty a zavolat metodu, která analýzu provede. Mnohdy totiž musíme spouštět program ve stromu vícekrát. Například když hledáme předpis matematické funkce, známe několik různých vstupních parametrů a k nim výsledek. Potom musíme provést výpočty se všemi známými vstupními daty.

Celá metoda pro hledání předpisu matematické funkce by mohla vypadat následovně:

```
public double UrciFitness(Explorer ex)
{
      double fitness = 0;
      for (int i = 1; i <=leninput; i+=2)// opakuj výpočet pro známe data
      {
            ex = ex.ToRoot();//nastav první uzel stromu
            x = input[i-1];//načti hodnotu proměnné x
            y = input[i];// načteni proměnné y
            fitness += Math.Abs((output[i] - Vypocti(ex)));//urceni odchylky
            //vypoctene a skutecne hodnoty
      }


      return fitness;
}
```

Tento kód vždy načte vstupní hodnoty funkce do globálních proměnných a poté zavolá funkci, která vypočte hodnotu výrazu obsaženého ve stromu a určí rozdíl mezi vypočtenou a požadovanou hodnotou. A fitness jedince je suma těchto odchylek.

Funkce ***Vypocti()*** by mohla obsahovat třeba následující kód:

```
public double Vypocti(Explorer ex)
{
      int c = ex.GetNFunc();//zjištění počtu vstupů daného uzlu(funkce)

      if (c <2)//nula vstupů – promená, jeden vstup se přeskočí a bere se jako
      //proměnná
      {
      if (ex.GetFunc() % 2 == 0)//určení proměnné
                  return x;//vrať x
```

```
            else
                  return y;//vrať y
      }

      if (c == 2)// dva vstupy - vyber funkce
            {
                if (ex.GetFunc() % 4 == 0)//vyber funkce scitani
                    return nic(ex.GetNextChildren()) +
                        nic(ex.GetNextChildren());
                if (ex.GetFunc() % 4 == 1)//děleni
                    return nic(ex.GetNextChildren()) /
                        nic(ex.GetNextChildren());
                if (ex.GetFunc() % 4 == 2)//násobení
                    return nic(ex.GetNextChildren()) *
                        nic(ex.GetNextChildren());
                if (ex.GetFunc() % 4 == 3)//odčítání
                    return nic(ex.GetNextChildren()) -
                        nic(ex.GetNextChildren());
            }

      return 0;//kdy se program dostane sem jedná se o chybu.
}
```

Metoda se stará o samotný výpočet funkce s danými parametry. V první fázi určí aritu, tím se zjistí, zda se jedná o funkci nebo proměnou. Protože v tomto případě neexistuje funkce s jedním vstupem, tak ji prohlásíme za proměnou. Pokud se arita rovná nule, vybere se proměnná. Pokud se arita rovná dvěma, potom se vybere funkce, která rekurzivně zavolá sama sebe, ale jako parametr použije element stromu o jednu úroveň níž a poté se hodnota parametru funkce vrátí jako návratová hodnota zavolané metody.

To je stručný popis toho, jak by mohla vypadat hodnotící funkce pro hledání matematického předpisu.

7.3 Implementace GP SDK

V následujícím textu bude popsán způsob, jak používat GP SDK. Vytvoření fitness funkce jsme se již zabývali, a proto ji budeme považovat za hotovou metodu.

V první fázi musíme vytvořit třídu ***WGPControl***, kterou zajistíme následujícím kódem, kde určujeme velikost populace na 600 jedinců a předáváme odkaz na hodnotící funkci:

```
WGPControl control;
control = new WGPControl(600, UrciFitness);
```

Poté musíme inicializovat populaci metodou ***StartRun***, jejíž parametry jsou maximální počet elementů stromu, nejvyšší počet vstupů (výstupů) uzlu a maximální délka jedince v inicializované populaci:

```
control.StartRun(64,3,3);
```

Tím je dokončena inicializace a nyní již přistoupíme k samotné evoluci. Ta probíhá v jednotlivých generacích, které provedeme zavoláním metody ***StepRun()***, jejíž parametry určují počet jedinců, kteří vzniknou mutací, křížením a kteří budou zachováni z minulé generace.

```
control.StepRunSelekce(0.8, 0.1);
```

Fitness nejlepšího jedince získáte následujícím způsobem.

double TopFitness = `control.Chro[0].Fitness;`

Odkaz na objekt ***Explorer*** nejlepšího jedince, který může být použit při jeho prezentaci je potom:

Explorer `ex = control.Chro[0].GetExplorer();`

Samozřejmě že GP SDK má mnohem více možností. Ty jsou popsány v dokumentaci k tomuto SDK a bylo by zbytečné je zde vypisovat.

V této kapitole jsme si ukázali, jak jednoduchým nástrojem je GP SDK, ale i přesto řeší složitou problematiku genetického programování. V současnosti stále ještě probíhá vývoj tohoto produktu, z čehož plyne, že v něm může dojít ještě ke změnám. Až časem a testováním dalšími vývojáři se ukáže kvalita či nedokonalost GP SDK.

8. Problém mravence na stezce Santa Fe

Za testovací úlohou pro GP SDK potažmo ADT, které je jeho součástí, jsme zvolili mravence na stezce Santa Fe. Při jeho řešení se testuje většina aspektů algoritmu genetického programování.

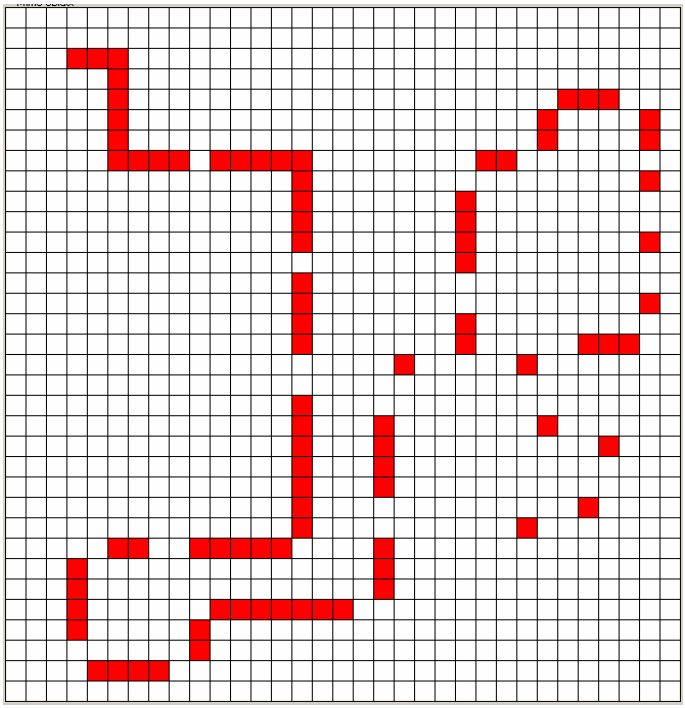
Cílem této úlohy je nalézt program virtuálního mravence, který sní, co nejvíce potravy v daném čase. Jeho program může být tvořen jen následujícími jednoduchými instrukcemi:

1. Krok vpřed – GO();
2. Zatočit doleva – Left();
3. Zatočit doprava – Right();
4. Rozhodovací struktura if_food
5. dva příkazy – {}
6. tři příkazy – {}

První tři příkazy slouží k pohybu mravence a odečítají čas. Čtvrtá rozhodovací struktura určuje, co má mravenec dělat, když před ním potrava je nebo není. Poslední dvě struktury slouží k větvení programu mravence.

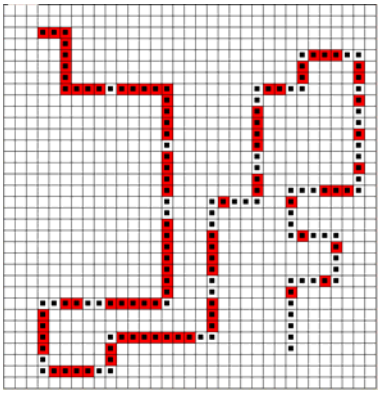
Mravenec má jen 400 kroků, aby snědl co nejvíce potravy. Stezka Santa Fe je vidět na Obr. 8. Na ní je celkem 89 kusů potravy vyznačených červenou tečkou. Překážky v podobě přerušení potravy na této stezce lze rozdělit do několika typů:

1. Zatáčka doprava a doleva
2. mezera
3. dvojitá mezera
4. mezera v zatáčce
5. dvojitá mezera v zatáčce
6. trojitá mezera v zatáčce



Obr. 8 Stezka Santa Fe

Na tyto překážky sice existuje obecné řešení, ale to potřebuje ke snědení všech návnad více jak 400 kroků. Proto nejlepší možné řešení není obecný program, který dokáže překonat zde popsané překážky, ale takový, který to stihne v dané čase. Možné řešení může vypadat třeba následovně.



Obr. 9 Cesta mravence

Jedná se o jedno z nalezených řešení pomocí GP SDK a program nalezeného mravence vypadal takto:

```
{
   {
      if_food
      {
         {
            Go();
            Go();
         }
      }
      else
      {
         Left();
      }
      Go();
}
   {
      if_food
      {
         Go();
      }
      else
      {
         {
            Left();
            if_food
            {
               {
                  Go();
                  Go();
               }
            }
            else
            {
               Right();
            }
         }
      }
      Right();
   }
}
```

Za povšimnutí stojí, že pokud se před mravencem nachází potrava, většinou urazí okamžitě dva kroky. Plyne to z toho, že většinou jsou za sebou více jak dvě potravy.

9. Řešení problému mravence pomocí GP SDK

Pro řešení této úlohy byla vytvořena klientská aplikace GP SDK, která může pracovat ve dvou režimech. První režim není určen pro komplexní testování, ale pro ukázku vývoje řešení během evoluce, při které probíhá záznam dosažených výsledků. Druhá část aplikace ukládá data do csv souborů, které lze načíst z většiny tabulkových procesorů jako je např. Excel.

Protože ve všech EVT hraje významnou roli náhoda, i dva testy se stejnými řídícími parametry dopadnou jinak. Buď dojde k nalezení stejně kvalitního řešení v různém čase nebo jedno může uváznout v lokálním extrému a druhé řešení přitom může být globální extrémem. Proto se musí k hodnocení výsledků přistupovat statisticky.

Také je potřeba zjistit, při jakých řídících parametrech dosahuje GP SDK nejlepších výsledků, a proto se musí provést testy s co největším množstvím nastavení.

Další věcí kterou je třeba studovat, je dopad způsobu určování fitness na kvalitu výsledků, které při evoluci získáme. Tím určíme, zda skutečně závisí na kvalitním určení hodnotící funkce.

Testy z těchto důvodů probíhaly následně. Bylo vytvořeno několik různých hodnotících funkcí. Pro každou z nich proběhl kompletní test, který zahrnoval různé nastavení řídících parametrů pro evoluci. Parametry byly nastavovány tak, aby zahrnuly všechny možné kombinace. Jednotlivé hodnoty se zvyšovaly o jednu desetinu. Počet opakování se stejným parametrem byl stanoven na dvacet a řešení se vyvíjela v 200 generacích.

Jednotlivé hodnotící funkce se lišily ve způsobu prezentace světa okolo mravence a množstvím dat, která se započítávaly do fitness. Prostředí se lišily v reakci na to, co se stane, když mravenec narazí na kraj mapy, buď se jedná o kulatý svět, kde se mravenec objeví na druhé straně mapy nebo placatý, kde po překročení hranice se nic neděje a mravenec je v prostoru mimo ni, ve kterém se může pohybovat a opět se vrátit zpět na plochu s jídlem.

Dále také můžeme do fitness započítávat počet kroků, které urazil od snědení poslední návnady, kdy čím více kroků urazil, tím je daný jedinec lepší. Toto hodnocení je ale sekundární, a proto není jeho váha větší než snědení jedné potravy.

Kombinace hodnotících funkcí jsou následující:

1. svět je kulatý a do fitness se započítává počet volných kroků.
2. svět je kulatý a nezapočítává se do fittnes počet volných kroků.
3. svět je rovina a do fitness se nezapočítává počet volných kroků
4. svět je rovina a do fitness se započítává počet volných kroků

Provádění programu virtuálního mravence probíhá podobně jako u matematického výrazu. Vždy, když dojde program na konec je znovu spuštěn, dokud nedojde k vyčerpání všech kroků. Přitom se počítá celková suma snědených návnad.

9. Výsledky

V minulé kapitole jsme si řekli, z čeho se test skládá a nyní si ukážeme, jak tento test dopadl. Kapitola je rozčleněna do šesti částí, kde první čtyři obsahují výsledky pro jednotlivé způsoby ohodnocení jedince a polední podkapitola obsahuje průměrné hodnoty.

V každém testu bude nejdříve tabulka obsahující průměrnou hodnotu fitness všech běhů se stejnými řídícími parametry a druhá tabulka obsahuje v procentech vyjádřený počet jedinců, kteří byli schopni nalézt všech 89 návnad.

9.1 Test 1

V tomto testu je „kulatý svět“ a do fitness se nezapočítává počet volných kroků.

Křížení / mutace	0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
0,1	38,15	27,2	23,35	24,45	25,7	21,55	20,5	23,9	23,45	**17,45**
0,2	30,9	31,7	23,15	21,4	26,95	**18,4**	21,65	**10,55**	**19,8**	
0,3	29,05	20,7	24,2	22,5	**16,45**	**18,4**	21,95	**15,4**		
0,4	28,4	20,8	26,5	21,45	21,8	**15,65**	**16,65**			
0,5	32,7	**19,35**	21,75	**19,15**	**14**	**15,7**				
0,6	22,65	22,95	20,6	**13,6**	20,4					
0,7	20,75	**11,95**	**19,6**	***9,75***						
0,8	22,35	20,45	**18,95**							
0,9	22,85	**11**								
1	**18**									

Tabulka 1 Průměrná hodnota fitness

Křížení / mutace	0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
0,1	0%	5%	5%	5%	0%	0%	5%	0%	10%	10%
0,2	0%	0%	10%	0%	10%	10%	0%	25%	10%	
0,3	0%	10%	10%	10%	15%	15%	15%	10%		
0,4	5%	10%	0%	5%	10%	15%	15%			
0,5	0%	5%	10%	15%	20%	15%				
0,6	10%	10%	20%	35%	15%					
0,7	10%	50%	10%	20%						
0,8	10%	15%	15%							
0,9	10%	30%								
1	20%									

Tabulka 2 Procentuelní schopnost nalézt globální řešení

9.2 Test 2

V tomto testu je „kulatý svět“ a do fitness se započítává počet volných kroků.

Křížení / mutace	0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
0,1	36,84	34,25	31,20	25,50	24,30	20,35	23,75	**16,00**	**19,29**	**17,94**
0,2	31,30	31,79	**18,75**	22,90	22,35	**17,55**	20,10	**16,90**	**17,40**	
0,3	30,25	27,35	20,50	20,29	**19,15**	21,50	**12,50**	***10,80***		
0,4	22,40	21,39	25,05	**17,10**	**19,34**	**17,45**	21,39			
0,5	29,04	22,99	**19,30**	**17,55**	**13,04**	**17,90**				
0,6	24,24	22,00	**13,69**	**18,64**	**14,49**					
0,7	23,75	21,00	21,39	**13,85**						
0,8	**19,75**	**18,50**	21,84							
0,9	**19,49**	**15,00**								
1	**16,35**									

Tabulka 3 Průměrná hodnota fitness

Křížení / mutace	0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
0,1	0%	5%	0%	0%	5%	10%	5%	15%	5%	15%
0,2	0%	0%	20%	15%	10%	25%	10%	5%	20%	
0,3	0%	15%	15%	5%	15%	5%	20%	25%		
0,4	5%	15%	0%	15%	20%	20%	0%			
0,5	5%	0%	0%	25%	30%	0%				
0,6	10%	15%	40%	15%	0%					
0,7	5%	5%	20%	0%						
0,8	15%	20%	0%							
0,9	25%	25%								
1	25%									

Tabulka 4 Procentuelní schopnost nalézt globální řešení

9.3 Test 3

V tomto testu je rovina a do fitness se nezapočítává počet volných kroků.

Křížení / mutace	0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
0,1	31,15	22,45	22,2	20,65	21,95	**16,5**	**18,1**	**18,2**	**12,3**	**18,35**
0,2	25,4	**15,3**	22,85	**17,55**	**18,6**	**17**	**14,8**	**16,6**	**16,45**	
0,3	25,4	**17,9**	**11,25**	23,1	**15,5**	**14,6**	**12,3**	**17,05**		
0,4	**16,4**	20,35	**13,1**	**18,7**	**13,9**	**13,85**	**13,4**			
0,5	**10**	**18,15**	**14,3**	**11,55**	**11,75**	**12,15**				
0,6	**19,6**	**12,95**	***6,4***	**12,2**	**8,3**					
0,7	**16,55**	**16,95**	**9,8**	**11,4**						
0,8	**8,8**	**9,9**	**15,95**							
0,9	**12,5**	**10,15**								
1	**14,4**									

Tabulka 5 Průměrná hodnota fitness

Křížení / mutace	0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
0,1	0%	5%	15%	5%	10%	15%	15%	5%	25%	0%
0,2	0%	20%	10%	20%	10%	15%	5%	10%	10%	
0,3	5%	10%	40%	5%	15%	5%	30%	5%		
0,4	10%	10%	40%	20%	25%	20%	20%			
0,5	25%	10%	10%	25%	30%	15%				
0,6	15%	30%	45%	30%	30%					
0,7	30%	15%	25%	20%						
0,8	35%	30%	25%							
0,9	10%	40%								
1	25%									

Tabulka 6 Procentuelní schopnost nalézt globální řešení

9.4 Test 4

V tomto testu je rovina a do fitness se nezapočítává počet volných kroků.

Křížení / mutace	0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
0,1	36,39	22,09	**18,94**	24,14	**17,39**	**15,59**	**17,59**	**19,49**	**18,64**	**18,59**
0,2	22,94	23,44	**18,29**	**19,39**	**17,89**	**19,49**	**13,59**	**12,94**	**13,54**	
0,3	21,34	**17,09**	**19,69**	**17,74**	**17,74**	**14,44**	**19,59**	**11,9**		
0,4	21,64	**15,29**	**13,75**	**18,94**	**19**	**11,44**	**13,1**			
0,5	**15,89**	**18,14**	**13,49**	**12,59**	**16,84**	**10,74**				
0,6	**17,19**	**16,59**	**10,49**	**14,24**	**14,14**					
0,7	**15,09**	**14,44**	***7,794***	**16,39**						
0,8	**15,79**	**11,59**	**11,04**							
0,9	**11,14**	**10,7**								
1	**14,14**									

Tabulka 7 Průměrná hodnota fitness

Křížení / mutace	0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
0,1	0%	5%	10%	5%	5%	15%	15%	20%	15%	15%
0,2	0%	0%	15%	25%	5%	15%	30%	30%	25%	
0,3	10%	20%	15%	20%	15%	25%	15%	15%		
0,4	15%	5%	10%	25%	5%	20%	40%			
0,5	20%	25%	30%	20%	10%	30%				
0,6	15%	20%	35%	25%	20%					
0,7	15%	25%	40%	20%						
0,8	15%	40%	30%							
0,9	20%	40%								
1	25%									

Tabulka 8 Procentuelní schopnost nalézt globální řešení

9.4 Průměrné hodnoty

Další dvě tabulky obsahují průměrné hodnoty všech testů.

Křížení / mutace	0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
0,1	35,63	26,50	23,92	23,68	22,33	**18,50**	**19,99**	**19,40**	**18,42**	**18,08**
0,2	27,63	25,56	20,76	20,31	21,45	**18,11**	**17,53**	**14,25**	**16,80**	
0,3	26,51	20,76	**18,91**	20,91	**17,21**	**17,24**	**16,58**	**13,79**		
0,4	22,21	**19,46**	**19,60**	**19,05**	**18,51**	**14,60**	**16,13**			
0,5	21,91	**19,66**	**17,21**	**15,21**	**13,91**	**14,12**				
0,6	20,92	**18,62**	**12,80**	**14,67**	**14,33**					
0,7	**19,03**	**16,09**	**14,65**	**12,85**						
0,8	**16,67**	**15,11**	**16,95**							
0,9	**16,50**	**11,72**								
1	**15,72**									

Tabulka 9 Průměrná hodnota fitness

Křížení / mutace	0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
0,1	0%	5%	8%	4%	5%	10%	10%	10%	14%	10%
0,2	0%	5%	14%	15%	9%	16%	11%	18%	16%	
0,3	4%	14%	20%	10%	15%	13%	20%	14%		
0,4	9%	10%	13%	16%	15%	19%	25%			
0,5	13%	10%	13%	21%	23%	20%				
0,6	13%	19%	35%	26%	22%					
0,7	15%	24%	24%	20%						
0,8	19%	26%	23%							
0,9	16%	34%								
1	24%									

Tabulka 10 Procentuelní schopnost nalézt globální řešení

Poslední tabulka, kterou zde uvedeme, obsahuje průměrné hodnoty z tabulek Průměrná hodnota fitness.

Test	
1	21,25
2	21,07
3	15,94
4	16,58
Průměr	18,71

Tabulka 11 Průměrné hodnoty tabulek

9.5 Vyhodnocení

Test jasně ukazuje závislost výpočtu fitness na schopnosti algoritmu hledat kvalitní řešení. To je velmi dobře patrné na testech, ve kterých se liší způsob reprezentace okolí virtuálního

mravence. U započítávání počtu volných kroků do fitness již jasně nelze říct, zda dochází k ovlivnění výsledků.

Za vhodné řídící parametry lze považovat takové, jejichž součet přesáhne 0,6 a nejlepších výsledků na této úloze lze dosáhnout s hodnotami v rozsahu 0,6-0,7 pro křížení a 0,2 pro mutaci.

Při žádných řídících parametrech nebylo dosaženo stoprocentní úspěšnosti, ale vždy bylo nalezeno částečné řešení daného problému. Za důvod této neschopnosti můžeme považovat neoptimalizovanost pro danou úlohu, protože se jedná o obecný algoritmus pro genetické programování. Výsledky ale výrazně nezaostávají za jinými řešeními, které jsou optimalizované pro daný úkol. Tento problém by mohl být vyřešen lepší selekcí populace či dalšími inovacemi.

Jedním z problémů tohoto testu je malé množství běhů evoluce se stejnými řídícími parametry, které pravděpodobně způsobují některé anomálie. Proto bude potřeba provést další testy, kde počet dat, ze kterých se počítá průměr, bude minimálně zdvojnásoben, aby se anomálie potvrdily či vyvrátily.

Další důležitou věcí, kterou nelze žádným syntetickým testem určit je jednoduchost a použitelnost GP SDK. Tuto vlastnost mohou posoudit jen další vývojáři, kteří použijí námi navržený produkt.

10. Závěr

V první části práce byly představeny evoluční výpočetní techniky. Bylo pojednáno o jejich zařazení do optimalizačních algoritmů a o jejich základních rysech. Další text pokračoval představením genetického programování, které slouží k automatickému vytváření programů. V úvodní části byly představeny techniky, které nejsou běžně známé široké veřejnosti.

V další etapě jsme se již zabývali vlastním přínosem k této problematice, představili jsme nový způsob reprezentace jedince v populaci, která se stal základem GP SDK. V závěru byl proveden test na úloze mravence na stezce Santa Fe.

GP SDK není v současné době plně dokončeno, ale je již použitelné. Dále se musí pracovat na zlepšení selekce populace a dalších inovacích. Také je potřeba, aby ostatní vývojáři vyzkoušeli tento produkt a sdělili své námitky a připomínky, které by pomohly zdokonalit GP SDK.

Až další čas a vývoj ukáže kvalitu či nedostatky zde představených technologií.

Literatura

[1] Mařík M. a kol.,Umělá inteligence(3), dotisk 2004, Praha: Academia 2001, ISBN 80-200-0472-6, Kapitola 3, Evoluční výpočetní techniky, st. 117-160.

[2] Mařík M. a kol.,Umělá inteligence(4), 1.vyd., Praha: Academia 2003, ISBN 80-200-1044-0, Kapitola 5, Genetické programování a vybrané problémy evolučních výpočtů, st. 128-170.

[3] Ivan Zelinka, Umělá inteligence v problémech globální optimalizace, 1 vyd., Praha: Ben 2002, ISBN 80-7300-069-5

[4] Ivan Zelinka, Umělá inteligence hrozba nebo naděje?, 1 vyd., Praha: Ben 2003, ISBN 80-7300068-7

[5] Zdeněk Vašíček a Ing. Lukáš Sekanina, Ph.D., Evoluční návrh kombinačních obvodů[online], Brno:Elektrorevue, Dostupné z http://www.elektrorevue.cz/clanky/04043/.en.windows-1250

[6] J. F. Miller, P. Thomson, Cartesian Genetic Programming [online], Edinburgh 2000. Dostupné z http://www.elec.york.ac.uk/intsys/users/jfm7/cgp-eurogp2000.pdf

Použité zkratky

EVT	evoluční výpočetní techniky
GA	genetické algoritmy
GP	genetické programování
SDK	Software Development Kit
GP SDK	Genetic programming SDK