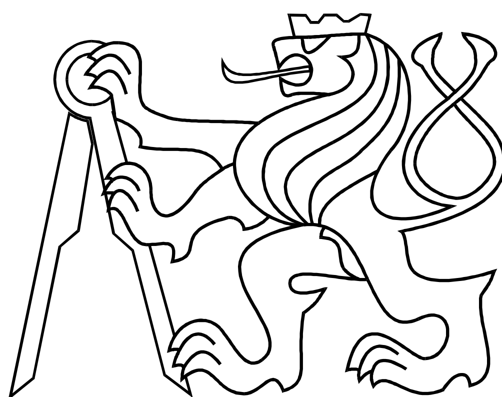
České vysoké učení technické v Praze

fakulta elektrotechnická

Katedra kybernetiky



Diplomová práce

Motivace akcí mobotů

Bc. Jaroslav Kočí

27. leden 2000

Prohlášení

Prohlašuji, že tato práce je duchovním majetkem ČVUT – FEL Praha. Souhlasím, aby s ní bylo naloženo podle potřeb fakulty.

Prohlašuji, že jsem uvedenou práci vypracoval samostatně pod vedením vedoucího diplomové práce, pouze s použitím uvedené literatury.

V Praze dne 27.1.2000 Jaroslav Kočí

Abstract

The aim of this work is improvement of behavior control mobile robots' behavior. This control is based on the principles of ethology with use of evolutionary algorithms. The final result of this study is achievement of the tool for automatic generating of mobile robots' behavior base ("DNA").

This work is interdisciplinary. It deals with both the principles of ethology and evolutionary algorithm problems. The final "simulation" enables generating of behavior base to be suitable for the individual requirements of the concrete problem, which is solved with the help of mobile robots. It means – appropriate motivation of mobile robots' action for relevant task.

The goal of this work is not to design some universal tool of mobile robotics, as somebody could expect, but above all to find out, if such an approach is realizable at all.

Shrnutí

Cílem této práce je zdokonalení chování behaviorálně řízených mobilních robotů. Toto řízení je založené na etologických principech s využitím evolučních algoritmů. V konečném výsledku získání nástroje pro automatickou tvorbu reakční báze („DNA“) mobilních robotů.

Tato práce je svým charakterem interdisciplinární. Pojednává jak o etologických principech, tak i o problematice evolučních algoritmů. Výsledná „simulace“ umožňuje generování takové reakční báze, která bude vyhovovat individuálním požadavkům konkrétního problému, řešeného pomocí mobilních robotů; to znamená vhodně motivovat akce mobilních robotů pro příslušný úkol.

Cílem této práce není konstrukce univerzálního nástroje mobilní robotiky, jak by se mohlo zdát, ale jde především o zjištění, zda-li je takovýto přístup vůbec realizovatelný.

Obsah:

1. ÚVOD 1

2. VÝVOJ MOBILNÍ ROBOTIKY 3

3. STRUČNÝ ÚVOD DO ETOLOGIE A JEJÍCH PRINCIPŮ 6

3.1. Fyziologické mechanismy chování 7
3.1.1. Dědičně koordinované neboli instinktivní chování 7
3.1.2. Vrozený spouštěcí mechanismus (AAM) 8
3.1.3. Komplexní systémy chování 10
Apetenční vyhledávání klidových stavů 10
Hierarchické systémy 10
Mnohonásobně motivované chování 11

3.2. Adaptivní modifikace chování 12
3.2.1. Modifikace 12
3.2.2. Adaptivní modifikace 12
3.2.3. Procesy učení 13

3.3. Společenství 14
3.3.1. Druhy společenství 14
3.3.2. Význam společenstev 15
3.3.3. Spolupráce (kooperace) mobotů 15
Kolaborace 15
Týmová práce 16
Hierarchická spolupráce ve společenství 16

4. EVOLUČNÍ ALGORITMY (EA) 17

4.1. Vývoj evolučních algoritmů 17

4.2. Formulace problému evolučních algoritmů 19

4.3. Datová reprezentace a operátory genetických algoritmů 21

4.4. Genetické operátory genetických algoritmů **22**
4.4.1. Mutace 23
4.4.2. Křížení 23
4.4.3. Operátor inicializace a inverze 24
4.4.4. Operátor selekce, selekční tlak 24
4.4.5. Fitness funkce 26

4.5. Teoretické základy evolučních algoritmů **26**
4.5.1. Schema 27
4.5.2. Teorem schematu 31
4.5.3. Bloková hypotéza 31

4.6. Porovnání klasických prohledávacích algoritmů a EA **32**

5. NÁVRH ŘEŠENÍ PROBLÉMU **34**

5.1. Úkoly pro řešení **34**

5.2. Vlastní řídící systém **35**
5.2.1. Motivace 35
5.2.2. Senzorické vstupy 36
5.2.3. Blok chování 37
5.2.4. Blok akcí 37
5.2.5. Blokové schéma 38

5.3. Hodnotící funkce mobota (Fitness funkce) **38**

5.4. Volba genetické reprezentace **40**
5.4.1. Způsob kódování 40
5.4.2. Význam genu pro mobota 40
5.4.3. Dvojitý genetický kód mobota 41
5.4.4. Hypermutace 42
5.4.5. Funkce priority u genů mobota 42
5.4.6. Chromozom mobota v podobě 2D řetězce 43
5.4.7. Detailní struktura jednoho genu mobota 44

5.5. Vypracovaní GA **45**
5.5.1. Třída genomu 45
5.5.2. Testování Genomu 47
5.5.3. Mutace genomu 48
5.5.4. Křížení genomu 49
5.5.5. Komparace genomů 50
5.5.6. Ohodnocení Genomu 50
5.5.7. Inicializace populace 50
5.5.8. Ohodnocení populace 50
5.5.9. Přepočítání projektu 51
5.5.10. Selekce 51
5.5.11. Genetický algoritmus 53
5.5.12. Ukončení programu 53

5.6. Popis simulátoru **55**

6. ZÁVĚR **58**

7. POUŽITÁ LITERATURA **59**

PŘÍLOHY: **60**

Slovníček základních pojmů evolučních algoritmů **60**

Popis genetických tříd **64**

Jednobodové křížení **79**

1. Úvod

Tato diplomová práce je součástí výzkumu skupiny mobotiky působící na katedře řídící techniky na ČVUT FEL Praha pod vedením Doc. Ing. Pavla Nahodila, CSc.

Výzkum této skupiny je zaměřen na vývoj hardwarových a softwarových prostředků pro autonomní mobilní roboty (moboty). Součástí výzkumu je pokus o začlenění poznatků o mechanismu chování z oblasti biologie, které se zabývají studiem chování živočichů, jako je např. etologie, sociologie a psychologie. Řídící systém mobota založený na těchto principech by umožňovalo rychlou a spontánní reakci systému mobota (instinktivní chování), podobně jak je tomu u živočichů.

Cílem této práce je navržení mechanismu, který bude automaticky vytvářet „reakční báze“ pro mobota, tak aby se mohl v prostředí samostatně pohybovat. Cílem je ověřit vhodnost evolučních algoritmů, kokrétně jejich modifikaci v podobě genetického programování, pro tuto oblast robotiky. Tato metoda, v jiných výpočetních aplikacích úspěšně ověřená, by měla napomoci rozvoji systémům využívající etologických principů aplikovaných na mobilní roboty a tedy včlenit tak do vývoje další mechanizmus, aby bylo možno celý systém zlepšit.

Text diplomové práce je členěn do šesti kapitol. V kapitole 2 je stručně shrnut vývoj mobilní robotiky až po okamžik, respektive projekt, který mi byl inspirací pro téma této diplomové práce a můj záměr aplikovat genetické algoritmy na tuto oblast mobilní robotiky.

Kapitola 3 pojednává o principech etologie, které podstatné pro návrh řešení pomocí automatických spouštěcích mechanizmů, jako je tomu v živé přírodě.

Kapitola 4 se zabývá evolučními algoritmy jako nástroje pro genetické programování. Společně s kapitolou 3 představují zázemí pro celý návrh řešení.

Kapitola 5 představuje popis vlastního návrhu řešené problematiky, stručně stádia jejího jednotlivého vývoje a naleznete zde i stručný popis jednotlivých částí programu a programového prostředí.

Výsledky této práce jsou shrnuty v závěru představovaným kapitolou 6. Dále zde v přílohové části uvádím slovníček některých základních pojmů týkajících se evolučních algoritmů, aby nedošlo k jejich špatné interpretaci. Dále je zde popis hlavních tříd a

jejich funkcí. Přílohou k diplomové práci je i disketa, které obsahuje zkompilovaný program zdrijové kódy a text diplomové práce v elektronické podobě.

2. Vývoj mobilní robotiky

V této kapitole s stručně zmíním o vývoji a současném stavu oblasti robotiky zaměřenou především na mobilní roboty. Při psaní této kapitoly jsem vycházel z literatury [2] a [6].

První *průmyslový robot*, zatím bez zpětné vazby, byl zkonstruován v roce 1962. Od tohoto okamžiku prodělala oblast průmyslových robotů prudký vývoj. V těchto ranných údobí robotiky se roboti používali jako manipulátory, u kterých byla ceněna především jejich mechanická síla a přesnost prováděných úkonů. Tyto roboti měli člověka nahradit v monotónních jednorázových činnostech. Nacházeli se buďto samostatně u výrobní linky nebo po skupinkách, které tvořily tzv. *robotická hnízda*. Úspěšně realizují plně automatizované linky a dokonce celé provozy, kde se člověk vlastního výrobního procesu „vůbec" neúčastní a funguje zde spíše jako dozor nad správným chodem provozu a zasahuje pouze v případě nějaké havárie závažnější povahy, se kterou si řídící systém nedokáže sám poradit.

Po robotech se v dnešní době vyžaduje jejich vitalita a pohyblivost, se kterou by robot vykonával jemu určený okruh činností. Tím, že se robot stal pohyblivým, mobilním robotem (tzv. *mobotem*) se rozšířil obzor pro jejich využití. Jejich mobilita nám umožňuje pružnost vykonávaného programu. Nyní se můžeme setkávat s vozíky pohybujícími se po vytyčené trase nejen na výrobních linkách s výraznou variabilitou výroby (mladoboleslavská Škoda), ale také v takových „netechnických oborech" jakým je zdravotnictví.

Příkladem realizovaného mobilního robota je mobilní autonomní robot *IPAMAR* (IPA Mobile Autonomous Robot) realizovaný firmou IPA ze Stuttgartu v NSR. Tento mobot byl určen pro převážení nákladů po tovární hale a pro tento účel byla pro něj vyvinuta nová řídící architektura nazvaná *integrated sensor action planning* (plánování pomocí integrace senzorů a akcí).

Podstatou tohoto způsobu řízení je, že externí počítač obsahující globální plán prostoru určuje mobotu úkol a prostřednictvím infračervené datové linky mu jej předává. Mobot jej přijme, naplánuje sekvenci akcí vedoucích ke splnění úkolu a pak

začne jednotlivé akce vykonávat. Současně však kontroluje jejich proveditelnost. Zaregistrují-li senzory neočekávanou situaci (např. v cestě je překážka) přeruší vykonávanou činnost a daný stav řeší (např. zastaví před překážkou).

Významnou skupinou zabývající se od 80. let výzkumem mobotů je laboratoř UI při Massachusets Institute of Technology (MIT), v Californii, USA. Zde byly realizovány úspěšné kolové moboty *Allen*, *Herbert*, jejichž úkolem bylo mechanickou rukou sbírat pohozené plechovky od Coca-coly a shromažďovat je na jednom místě. Dále zde bylo vytvořeno několik menších kolových mobotů např. *Tom* a *Jerry* jejichž úkolem bylo pohybovat se rychle v neznámém prostředí bez vytváření map světa, vyhýbat se překážkám a navíc, úlohou Toma bylo sledovat Jerryho.

V poslední době laboratoř zkonstruovala ještě několik šestinohých mobotů tzv. *hexapodů* (např. *GENGHIS*). Všechny zmíněné moboty jsou vybaveny ŘS na bázi *vrstvové architektury*.

Samostatnou oblastí vývoje jsou automatická zařízení používaná v automatických kosmických stanicích určených pro výzkum vesmíru. Ty jsou používány pro získávání údajů o Zemi a dalších planetách - automatické sondy. Jedním z typů těchto automatů jsou také samohybná zařízení určená pro výzkum povrchu planet. Jedním z nejznámějších je *Lunochod*. Dalším úspěšně realizovaným projektem byl průzkum planty Mars pomocí „*Pathfinderu*“. V poslední době se objevují principielně nové přístupy, kterou jsou *fraktální moboti*. Progresivně se rozvíjí výzkum v oblasti autonomních agentů a jejich aplikací. Záměrem je vytvoření kognitivního systému komunikujícího s prostředím. Dalším rozšířeným přístupem jsou práce založené na genetických algoritmech. Výhodou tohoto směru řešení je široký okruh problémů, které jsou algoritmicky vyřešeny.

Ve výzkumu autonomních agentů jsou pro nás velice zajímavé aplikace agentů v oblasti mobotiky. Nejvýznamnější práce v tomto ohledu probíhají v Americe na Massachusetts Institute of Technology (MIT), v Evropě je důležitým centrem Université Libre de Bruxelles (IRIDIA). Ve výzkumu aplikace znalostí etologie genetickými algoritmy je na předním místě School of Cognitive and Computing Sciences na University of Sussex (COGS).

V polední řadě bych nechtěl opomenout významnou práci ve výzkumu mobilní robotiky, kterou provedl (v roce 1997) UK Robotics ve spolupráci s Universitou v Salfordu. Tento projekt se zabýval návrhem řídící struktury MACMR (multiple autonomous co-operant mobile robot). Celý projekt trval přibližně 10 let a byl zaměřen na využití mobilních robotů při rozebírání jaderných elektráren. V řídícím systému robota (robotů) bylo využito behaviorálních principů a principů založených na genetických algotimech a to především v případě konfliktních situací, kdy mohlo dojít k *deathlocku* řídícího systému. Aktivita vzniklá behaviorálním řešením konfliktu je reprezentováno jako kontinuum dvou základních typů opačných chování. Jedním z extrémů je chování, které má být zcela egoistické, kdy se robot řídí pouze chováním prospěšným samotnému robotu (např. uhýbání překážkám a úspora energie). Druhým extrémem je chování, které má být altruistické (skupinka robotů pracující na jednom obecném úkolu). První by způsobil, že by robot zůstal stát v dostatečné vzdálenosti od všech překážek v jeho prostředí (i od ostatních robotů), zatímco druhý typ chování by hnal robota do bezprostřední blízkosti jeho druhů a k týmovému provedení úkolu.. tento výzkum byl zaměřen na vytvoření architektury řídícího systému, která by mohla přizpůsobit takovéto opačné a konfliktní typy chování. V tomto projektu měli dva roboti, *Fred* a *Ginger,* přemisťovat společně trubkový předmět. Jako pomůcka pro nesení předmětu a senzoru pro získávání údajů o poloze druhého robota, zde bylo použito *x-y stolku*, který měly na své „hlavě". Tím se problém lokalizace druhého jedince značně zjednodušil.

Behaviorálně založené metody, jako je BSA (behavior synthesis architecture) (Brooks 1986) nabízejí přímé mapování senzorických stimulů pro odezvu, ale nejsou schopny adaptace na prostředí, která vyžaduje strukturovanou odpověď (např. únik ze slepé uličky). Tento problém byl vyřešen pomocí Fuzzy řízení, neboť Fuzzy řízení je normálně velmi robustní a může tedy do jisté míry tolerovat degradaci struktury pravidel (Kosko 1992) a šumu na vstupních senzorech. Byly použity trapezoidní funkce, protože se ukázaly jako výpočetně účinné, což bylo důležité pro aplikaci v reálném čase. Byla navržena Fuzzy asociativní paměťová matice (FAM) (Kosko 1992) s nastavováním parametrů pomocí genetických algoritmů. Celá operace návrhu trvala připližně 10-20 hodin, podle toho jak byly voleny parametry pro genetický algoritmus.

3. Stručný úvod do etologie a jejích principů

V této kapitole se zaměřím na poznatky z oblasti biologie, resp. etologie, které motivovaly vývoj robotiky tímto směrem. Podklady pro tuto část jsem čerpal z literatury [1] a [2].

Poznatky z biologie byly podnětem ke vzniku nových řídících systémů pro mobilní roboty. Nepotřebujeme se zde zabývat celou složitostí biologie. Pro nás je důležitá především oblast související s motivací, čili spouštěním a řízením chování, jejíž hlavní myšlenky tvoří základem při vytváření koncepce celého řídícího *systému modelu chování.*

Je potřeba si nadefinovat některé základní vlastnosti živých organismů, které jsou pro ně typické, a kterými se liší od ostatních systémů. *Jedinec (živý organismus)* je hierarchicky uspořádaná, termodynamicky otevřená, autoregulující se nukleoproteinová soustava s vlastnostmi jako je metabolismus, autoreprodukce a nutnost vyvíjet se.

Živé organismy jsou schopny udržovat, popř. zvyšovat svoji vlastní uspořádanost a tím se zcela zásadně liší od neživé přírody, která podléhá fyzikálním zákonům, jako je zákon růstu entropie (neuspořádanosti). Růst organismu tedy není pouze jev kvantitativní, kdy narůstá hmota, ale i kvalitativní, neboť roste složitost vnitřní struktury organismu (viz. semínko - dospělá rostlina). Všechny živé organismy však mají geneticky definovaný životní cyklus, který definuje průběh a délku jednotlivých fází života organismu jako je růst, dospělost, stárnutí a smrt.

Zásadní rozdíl mezi *živými* a *neživými systémy* je tedy v tom, že základní stavební prvky organismu - *buňky* jsou schopny *adaptace* a určité *autonomní činnosti* (např. přerušený nerv doroste opět do původního místa jinou cestou) a jsou nositelé genetické informace, která specifikuje činnost a vlastnosti dané buňky i celého organismu.

Zde leží obrovská propast mezi složitostí a vlastnostmi živých organismů a realizačními možnostmi současné techniky, a proto v technické praxi není snaha živé systémy kopírovat, ale využívat principy, na kterých jsou založeny a ty pak vhodným způsobem technicky realizovat (např. vzor - vážka => technická realizace - vrtulník).

3.1. Fyziologické mechanismy chování

Jeden z hlavních vlivů na chování organismu má jeho metabolismus, resp. soubor metabolických procesů jež jej tvoří.

Metabolické procesy jsou procesy související s přeměnou látek a energií uvnitř organismu a jsou v živé soustavě regulovány *homeostázou* (pomocí zpětné vazby) v závislosti na jejím okamžitém vnitřním stavu a vnějším okolí. Proto se také živé soustavy řadí mezi adaptivní systémy, které reagují na změny okolí změnami svých vnitřních stavů, a to způsobem pro ně výhodným => *účelné chování.*

Homeostáza je regulační okruh v organismu, který se stará o zachování vyváženého vnitřního stavu systému z hlediska energie, tepla, vody, minerálů apod. Základní funkcí organismu je pak udržet tento vyvážený vnitřní stav i celkovou bezpečnost systému, z hlediska vnějších podnětů, prostřednictvím odpovídajících činností, a tím se udržet v „celkové pohodě“ (klidovém stavu) z hlediska vnitřních i vnějších podnětů.

Příčinou určitého typu chování organismu jsou tedy jak *vnitřní,* tak *vnější podněty,* přičemž většinou vnitřní procesy jsou podněty ke spuštění určitého typu chování a vnější podněty pak směrují, ovlivňují způsob realizace tohoto chování.

Jednotlivé typy chování (tzv. *dědičně koordinované pohyby*) *dané vzorcem chování* i *spouštěcí mechanismus* aktivující příslušný typ chování (*vrozený spouštěcí mechanismus* - AAM) jsou zděděné, tedy předávané prostřednictvím genetické informace z generace na generaci. Správné přiřazení jednotlivých typů chování k odpovídajícím podnětovým situacím a jejich zařazení do hierarchie celého systému je v průběhu života organismu vytvářeno a modifikováno na základě jeho iterací s prostředím - *učením.* Schopnost přizpůsobit své chování změnám okolí, *schopnost adaptovat se*, je dána otevřeností, tj. mírou modifikovatelnosti vzorce (programu) chování.

3.1.1. Dědičně koordinované neboli instinktivní chování

Instinktivní chování je podvědomé chování, které se dá vyvolat vnějšími (exogenními) či vnitřními (endogenními) podněty. V typickém případě probíhá podle pevně daného (vrozeného) vzorce, který je tvořen postupně spouštěnými prvky :

Vnitřní vyladění (pud, pocit žízně) => *apetenční chování* (směřuje k aktivnímu vyhledávání podnětů, které uvolní konečné chování pud uspokojující) => podnětová

situace (setkání s *klíčovým podnětem*) => uvolnění spouštěcího mechanismu (neutrální mechanismus, který specifickým způsobem odpovídá na klíčový podnět) => *konečné jednání* (pití) => *uspokojení* (vyhasnutí) pudu (uspokojí svoje potřeby).

V živém organismu se často aktivuje současně několik *funkčních okruhů* (např. potravní + obranný + rozmnožovací), které se vzájemně ovlivňují. Pak nutně dochází k výběru pouze jednoho chování, jež je prováděno, prostřednictvím dočasného potlačení druhých funkčních okruhů. Někdy však dochází k tzv. *naznačenému chování*, kdy jsou prováděny jen jisté prvky chování (např. z útoku zůstává pouze první fáze - hrozba a výpad již není dokončen), nebo k tzv. *přeskokovému chování*, kdy jsou naopak aktivovány dva protichůdné funkční okruhy v přibližně stejné intenzitě (např. útočný a útěkový) výsledné chování pak „osciluje“ mezi oběma typy chování.

Prakticky všechny živé organismy se pohybují. Při tomto pohybu se nutně potřebují jistým způsobem orientovat. Rozlišujeme tyto tři základní typy orientace pohybu:

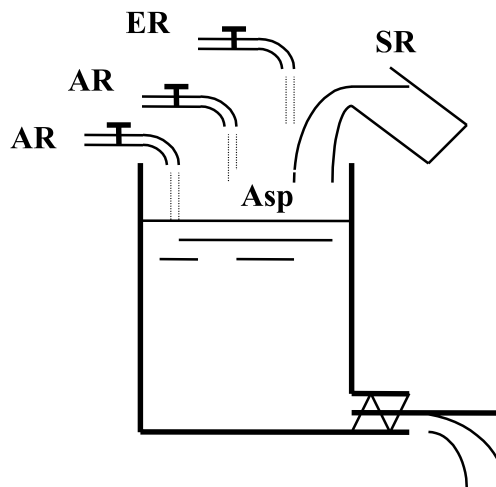
a) *Kineze* - je nejjednodušší forma orientace, kde se pohyb vztahuje pouze k intenzitě podnětu bez ohledu na jeho lokaci (pohybová aktivita klesá v příznivém a klesá v nepříznivém směru - např. pohyb trepky).
b) *Taxe* - jsou usměrněné pohyby probíhající ve směru působení podnětu (pohyb ke světlu proti zemské tíži apod.).
c) *Pilotování* (navigace) - jsou složité formy orientace, kdy se je využíváno:
    1) *pozemních značek* - na krátkou vzdálenost.
    2) *časoprostorového systému* - na dlouhou vzdálenost (založeno na hodnocení azimutu a výšky slunce nad obzorem).

3.1.2. Vrozený spouštěcí mechanismus (AAM)

Vrozený spouštěcí mechanismus, dává zvířeti schopnost rozpoznat relevantní situace v prostředí a uvádět do chodu odpovídající dědičně koordinovaný pohyb. Tento spouštěcí mechanismus reaguje na konfiguraci podnětů, která je též nazývaná klíčový podnět. U vrozených spouštěcích mechanismů může docházet i k jistým typům učení, resp. adaptivní modifikaci chování, zvyšováním selektivity klíčového podnětu, tj. obohacováním podnětové konfigurace o další znaky (podněty).

Základní princip činnosti AAM dobře vystihuje *Lorenzův hydraulický model* (viz.obr.3.1.). Vodní nádrž reprezentuje akumulátor (energie) ze všech podnětů, které

působí na dané AAM. Jejich celkový součet dává aktuální hodnotu tzv. *akčně specifického potenciálu* Asp spouštěcího mechanismu, která odpovídá stavu vnitřní vyladěnosti zvířete - „nadrženosti“. Překročením jisté mezní hodnoty potenciálu - prahu, dojde ke spuštění (otevření tlakového ventilu obr.3.1.) příslušného dědičně koordinovaného chování.



obr. 3.1.: Hydraulický model

ER představuje na obrázku endogenní a automatické buzení, kterým vnitřní procesy v organismu stále působí na hodnotu *Asp*. *AR* označuje další nabíjecí mechanismy, které zvyšují hodnotu *Asp* (např. spouštěcí signál z jiného AAM). *SR* označuje klíčový podnět, který má na zvyšování hodnoty *Asp* dominantní vliv a ve většině případů vede ke spuštění chování daným spouštěcím mechanismem.

AAM je v postatě samostatným funkčním celkem, který je možno zařadit do různých částí hierarchie instinktivního chování. Na jeho základě je pak možno vytvářet popis celého komplexu instinktivního chování organismu (viz hierarchické systémy). Spouštěcí signál z AAM totiž nemusí pouze spouštět jisté chování (cílové nebo apetenční), ale může být sám nabíjecím či klíčovým podnětem (viz AR na obr. 3.1.) v dalším hierarchicky podřízeném AAM. V souvislostí s činností AAM jsou popisované další složité jevy jako unavitelnost specifické aktivity, snížení prahu spouštěcích podnětů či aktivita běhu na prázdno.

3.1.3. Komplexní systémy chování

Apetenční vyhledávání klidových stavů

Apetence je definována jako *stav vrušení* (vybuzení), který trvá tak dlouho, dokud nenastane jistá podnětová situace, označovaná též jako *vyhledávaný podnět* (appeted stimulus). Dojde-li konečně k podnětové situaci, je spuštěno cílové chování (consumatory action), apetenční chování přestává a je vystřídáno stavem relativního klidu.

Apetenční chování je tedy účelné chování směřující k vyhledávání takových podnětových situací, které v konečném důsledku vedou k uspokojení potřeb organismu a tím i vymizení buzení k provádění daného typu chování. Proto jsou tyto typy chování souhrnně označovány jako *apetenční vyhledávání klidových stavů*. Apentence, AAM a instinktivní pohyb patří mezi nejdůležitější faktory určující celkové chování živočichů.

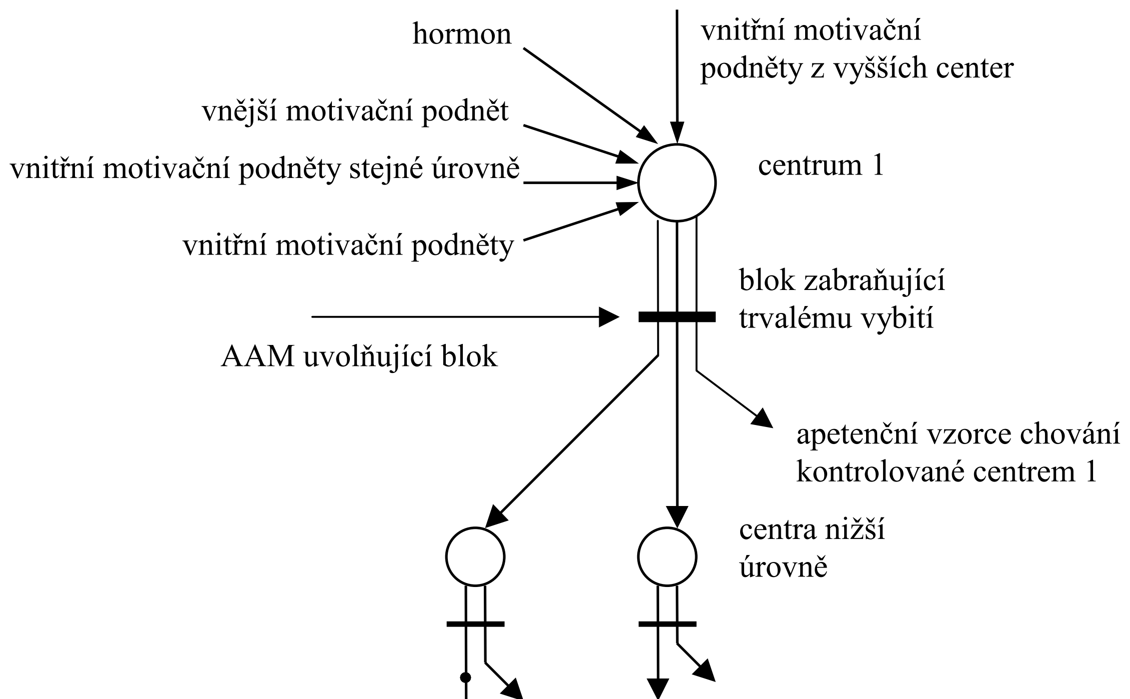
Hierarchické systémy

Z řady pozorování chování živočichů vyplynulo, že určitý spouštěcí mechanismus (AAM) tlumí proces určitého instinktivního pohybu a jiný naopak odtlumuje. Ukázalo se, že na apetenční chování je většinou navázáno na AAM, který po nalezení hledané podnětové situace apetenční chování tlumí, vypíná a spouští jiné (cílové) chování. Celý komplex instinktivního chování je pak vytvořen právě z takovýchto hierarchicky organizovaných elementů.

Princip této *hierarchické organizace* (hierarchie instinktů) znázornil Tinbergen ve svém zjednodušeném diagramu (Obr. 3.2.). Použil zde označení *central excitory mechanism* (CEM) navržené Beachem, které zahrnuje všechny faktory zvyšující pohotovost organismu k určitému typy chování - hormony, vnitřní podněty, nabíjecí podněty, klíčové podněty i stimulující podněty z vyššího centra. Obrázek zachycuje systém CEM-AAM, tedy centrálně podněcující (CEM) a vrozený spouštěcí mechanismus (AAM). Na obrázku je zobrazeno centrum středního stupně, n-tého řádu. Nejvyšší centrum je při tom označeno jako prvního řádu.

Příslušné impulsy různého druhu centrum nabíjejí. Toto centrum přijímá za prvé impulsy z vyššího centra (řádu n-1), které mohou být rozváděny i do dalších center n-tého řádu. Dále pak přijímá impulzy od centra, k němuž je přiřazeno a které automaticky podněcuje samo sebe. Na centrum dále mohou působit hormony a to přímo nebo přes

příslušné automatizační centrum. Posledními typy faktorů působícími na dané centrum pak jsou vnitřní a vnější smyslové podněty.



Obr. 3.2.: Hierarchický model AAM podle Tinbergena

Pokud není vrozený spouštěcí mechanismus drážděn (levá šipka AAM na obr. 3.2.) nemohou se žádné impulsy vybíjet. V případě, že nastane spouštěcí podnětová situace (klíčový podnět), je blok odstraněn a impulsy odtékají do nižších center, popř. přímo spouští určité apetenční, či cílové chování. Spouštěcí, uvolňující podnět může přicházet z center vyšší či stejné úrovně.

Takto (hierarchicky) vytvářené „*etologické*“ sítě v podstatě popisují procesy, k nimž dochází v nervové soustavě organismů v souvislosti s instinktivním a apetenčním chováním. Jako takové mají tedy blízko k architektuře a vlastnostem neuronových sítí tvořících centrální nervové soustavy živočichů.

Mnohonásobně motivované chování

Velmi často vzniká v jednom okamžiku motivace k několika typům chování, tzv. *mnohonásobná motivace*, a jelikož může být spuštěn vždy jen jeden typ chování, musí dojít jistým způsobem k výběru jednoho z nich.

Nejjednodušší formou vyřešení problému napětí mezi dvěmi současné aktivovanými podnětovými zdroji je *superpozice*. Přitom se můžeme setkat jak s efektem převrstvení ve smyslu součtu, tak ve smyslu rozdílu působení obou podnětů. Například člověk v konfliktu mezi dvěma motivacemi působí „napjatě“. To je způsobeno superpozicí jednotlivých aktivovaných pohybů.

Dalším formou řešení takové situace je *vzájemné tlumení* a *alternování* jednoho typu chování jiným. Případů, kdy aktivování jednoho chování způsobuje úplné tlumení jiného chování, je známo jen několik. Nejznámějším případem je útěk. Kdyby například zajíc prchající před vlkem běžel přes pole s lákavým jetelem jen o něco pomaleji než po jiném pozemku, bylo by takové chování z hlediska zachování druhu velmi neúčelné.

U člověka však tento zcela tlumící účinek bohužel způsobuje velmi neúčelné potlačení vyšších funkcí učení a inteligence (např. ohlupující účinek paniky). Výsledkem tlumícího vlivu může být také tzv. *náznakové chování*, kdy z původního chování zůstává jen první fáze (např. při útoku je provedena jen hrozba a není dokončen výpad, jak už bylo zmiňováno výše).

Dostane-li se motivace dvou typů pohybu do vzájemného konfliktu, nastupuje v mnoha případech zcela nesmyslně třetí typ, patřící ke zcela jinému systému chování. Tento jev se nazývá *přeskokové chování*.

3.2. Adaptivní modifikace chování

3.2.1. Modifikace

Modifikace je každá *trvalá změna*, která je vyvolaná v organismu během jeho individuálního života vnějšími vlivy. Stejně jako mutace je i modifikace neorientovaná změna a nemusí tedy znamenat adaptaci na vnější vlivy.

3.2.2. Adaptivní modifikace

Adaptivní modifikace je vždy uskutečnění fylogeneticky vzniklého, v genotypu (genech) zabudovaného programu, který je v každém jedinci připraven, jako adaptace na určitou očekávanou proměnlivost odpovídajícího životního prostoru. Takovéto genetické dispozice, které počítají s proměnlivostí prostředí, označuje E. Mayer jako otevřený program. Funkce fyziologického mechanismu je základem, jak pro získávání nové v genomu neobsažené informace, tak v uchování té staré, a tím také hromadění trvalé adaptibility jedince (embriogeneze, učení).

3.2.3. Procesy učení

Procesy učení patří mezi adaptační modifikace fyziologického mechanismu, který se projevuje jako chování zvířat a lidí. Liší se však od embriogeneze, resp. embriologické indukce, kde se modifikuje přímo genetická informace, v tom, že učení je reverzibilní - většina naučeného může být zapomenuta.

Z hlediska způsobu získání naučené informace můžeme rozlišit učení:

a) *Obligatorní učení* (nucené) - většinou navazuje na vrozené, geneticky programované chování. Příkladem může být podmiňování či vtištění.
b) *Fakultativní učení* (příležitostné) - může vyplynout z hravého či zvídavého (exploračního) chování.

S učením velmi úzce také souvisí otázka existence *krátkodobé* a *dlouhodobé paměti,* protože učení je uložení zkušeností z prožitých situací a událostí a jejich zhodnocení. Tato informace není v organismu apriori zděděná. Co ovšem musí mít organismus geneticky předáno, je *učitel - arbiter,* jenž posuzuje úspěšnost chování jako odpovědi na určitý podnět či kombinaci podnětů.

Učení tohoto typu, kdy se vytváří relace mezi spouštěcím podnětem a adekvátní akcí má *asociativní charakter*. Takto asociovaný podnět může být podmíněný i nepodmíněný (často je to kombinace obou). ***Asociace*** vzniká opakováním dané konfigurace podnětů v relativně krátkém čase před příslušným typem chování, nebo podmíněným podnětem - tzv. *podmiňování*.

Učení může být *typu S*, kdy dochází k selekci podnětu spouštějícího daný typ chování, nebo *typu R*, kdy se vybírá pro daný podnět vhodné chování (tedy jedno z kolekce použitelných typů chování, jež má organismus k dispozici) - *selekce chování, instrumentální učení.*

Jsou známé také typy učení bez asociace jako *facilitace, senzitizace* či *habituace*, které jsou převážně založeny na korekci aktivačních hodnot spouštěcího mechanismu a síle vlivu jednotlivých podnětů v závislosti na změně jejich významu.

Proces *facilitace* lze pozorovat u různých centrálních nervových funkcí, dochází při něm patrně ke změnám v synapsích a způsobuje zlepšování kvality provedení jistých úkonů jejich opakováním, tzv. *učení opakováním*.

Podobné procesy, které se však odehrávají na senzorické, nikoli na motorické straně centrální nervové soustavy, se nazývají *senzitizace*. Při spouštění určitého typu

chování nejprve klesá prahová hodnota klíčových podnětů, což vede k tomu, že zvíře je uvedeno do stavu zvýšené „pozornosti“. Je to však jev pouze *dočasný* a na rozdíl od facilitace, po určité době opět vymizí.

Proces učení nazývaný jako *habituace* (přivykání) má charakter desenzitování vůči působení daného podnětu, jehož účinek se oproti počáteční reakci zmenšuje.

3.3. Společenství

V této práci se mám mimo jiné zabývat kooperací mobotů v mobotím společenstí, proto bych rád v této kapitole uvedl některé vlastnosti, které se týkají tohoto tématu. Při psaní této kapitoly jsem vycházel z literatury [3].

3.3.1. Druhy společenství

Jaký je význam společenstev, obecně společností chcete-li, je vhodné studovat na nám známém objektu, tj. na přírodě zejména na zvířatech a lidech.

V přírodě existují dva odlišné druhy společenstev:

- neorganizovaná
- organizovaná (hierarchická)

Neorganizovaná společenství jsou dočasná seskupení různých jedinců, u kterých neexistuje dělba práce. Tímto pojmem lze popsat společenství prvoků u společného zdroje potravy, hromadné tahy ryb nebo nákupní horečku v obchodním domě. To znamená, že v takovém společenství existuje každý jedinec bez cíleného vztahu k ostatním.

Organizovaná společenství jsou vyšší formou organizace jedinců. Předpokladem je totiž vytvoření určité struktury (hierarchie) jednotlivých členů společnosti. Takováto společenství jsou charakteristická tím, že jedinec závisí (i existencionálně) na druhých. Většinou dochází u jedinců v takovéto společnosti k vysoké specializaci na jednu konkrétní činnost. Uvnitř této společnosti jsou vytvořeny velice detailní sociální vztahy. Vznik takovéhoto společenství je podmíněn existencí určitého stupně komunikace. Tj. na sdělování dojmů, potřeb apod. navzájem. Bez komunikace si totiž není možno vysvětlit vznik sociálních struktur.

Příkladem takové vysoce hierarchicky organizované skupiny jsou mravenci. Existuje u nich totiž vysoká specializovanost (válečníci, dělnice, královny apod.). Tito

jedinci nemohou většinou dělat žádnou jinou činnost než svou vlastní. Dalším příkladem je společenstvo včel a mnoho dalších převážně hmyzích společenstev.

3.3.2. Význam společenstev

Kromě již zmiňovaných výhodách společenstev, je například to, že v jeden časový okamžik jedinci stejného druhu potkávají. Proto může snáze docházet k rozmnožování (uchovávání rodu). Je jasné, že četnost množení stoupá s počtem jedinců ve skupině a tím i možnost vzniku genetické odchylky, která může znamenat vývojový skok vpřed.

3.3.3. Spolupráce (kooperace) mobotů

Spoluprací rozumíme společné řešení daného úkolu nebo problému. Pro tento způsob řešení (spoluprací) je nutná komunikace. Komunikací se zde nebudu zabývat, protože je to velice rozsáhlé téma a není nezbytně nutné pro řešení tohoto problému.

Spolupráci mobotů vyžadují různé typy úkolů, např. tam, kde:

- jeden mobot nemůže úkol zvládnout, protože jeho vyřešení by vyžadovalo přítomnost mobota na více místech současně (např. stavba složité kostrukce, umisťování dlouhých nosníků atd.)
- řešení úkolu vyžaduje spolupráci jinak specializovaných mobotů (např. bagrování a odvoz zeminy).
- pro úspěšné vyřešení úkolu je nutno vyřešit mnoho různých podúkolů, pak se využitím více mobotů výrazně snižuje čas potřebný ke splnění tohoto úkolu (např. shromažďování geologických vzorků).

Podle těchto typů úloh lze typy spolupráce rozdělit do tří kategorií:

Kolaborace

Kolaborace je spolupráce více mobotů na jednom společném úkolu, kde je vyřešení úkolu cílem všech zúčastněných a kde neexistuje organizační člen. Příkladem může být obrana společného zdroje energie (potravy), nikdo nevelí (nemusí velet), všichni v podstatě bojují o přežití.

Využití takovéto formy spolupráce pro mobilní robotiku je možno v případě existence relativně vysoce automomních na sobě nezávislých jedinců (mobotů).

Týmová práce

Týmovou prací můžeme vyřešit problémy, u kterých požadujeme koordinaci spolupracujících mobotů. V takovém případě se jeden mobot (obvykle „nejchytřejší“) stane organizátorem práce druhých mobotů (zadává jim úkoly, popř. kontroluje jejich činnost). Tento „vedoucí“ může být zvolený námi, nebo vnitřní volbou společenství.

Zde je jasně vidět, že týmová práce je již velmi náročná na komunikaci (více než kolaborace).

Hierarchická spolupráce ve společenství

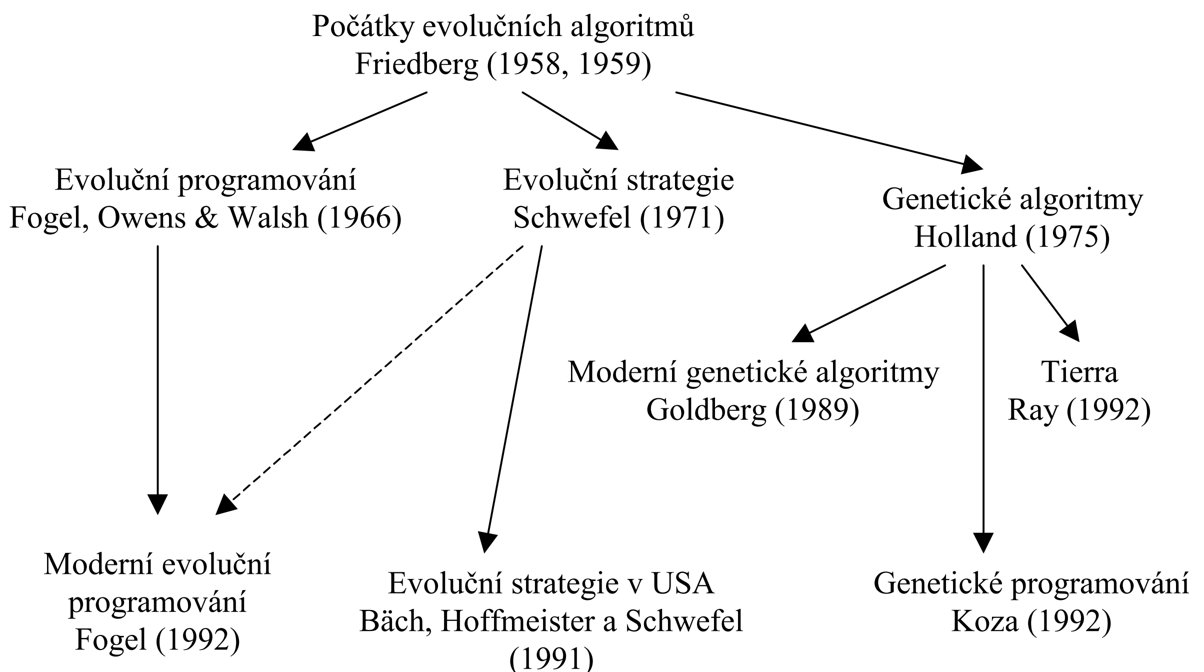
Tento typ organizace je nejvyšším možným typem. Zde má každý jedinec přesně stanovený úkol a „společenské“ zařazení. Tato organizace je nejvhodnější pro specializaci na jeden druh činnosti, takže pak v podstatě není potřeba univerzálních strojů, ale množství jednoduchých mobotů (po konstrukční stránce těla).

4. Evoluční algoritmy (EA)

Tato kapitola pojednává o evolučních algoritmech a jejich principech. Jedná se o základní přehled, jak a proč evoluční algoritmy fungují. Toto minimum z této oblasti slouží společně s kapitolou zabývající se etologií jako podklad pro pochopení problematiky vlastního řešení. Čerpal jsem převážně z poznatků ze seminářů Doc. Macha (TU Košíce), a z literatury [4].

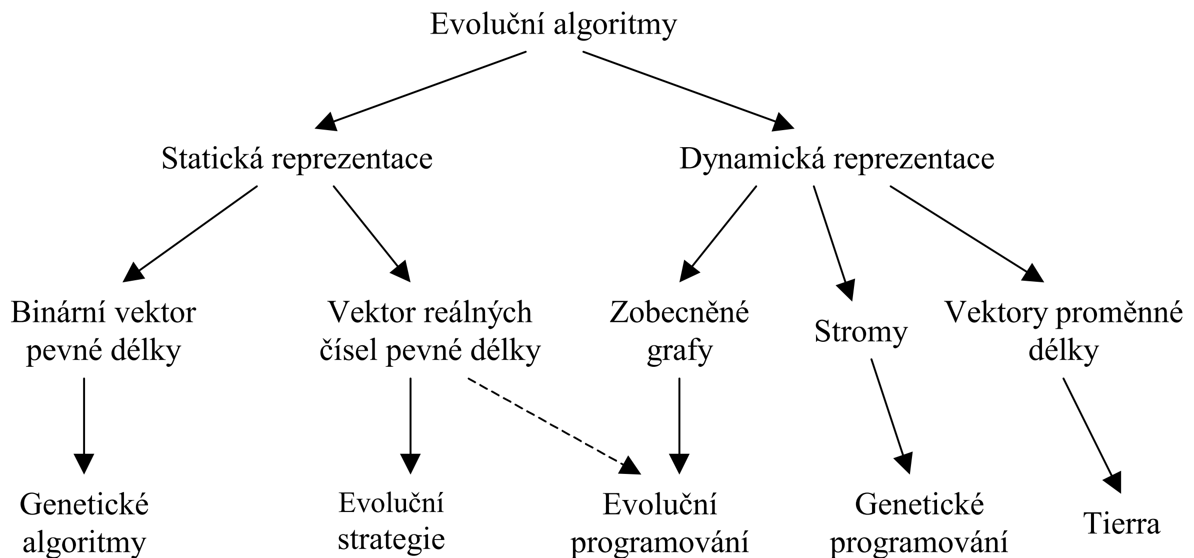
4.1. Vývoj evolučních algoritmů

Evolučními algoritmy (EA – Evolutionary Algorithms) jsou souhrnně nazývány algoritmy, které pro řešení problémů využívají některého z mechanizmů evoluce. V současné době existuje celá řada mechanizmů s využitím evoluce. Mezi nejdůležitější patří genetické algoritmy, evoluční programování, evoluční strategie, klasifikační systémy a genetické programování. Vývoj evolučních algoritmů je pro názornost zobrazen na obr. 4.1. a vývoj používaných datových struktur na obr. 4.2. Společným rysem všech evolučních algoritmů a jejich modifikací je simulace přírodního procesu evoluce jedinců pomocí procesů *selekce*, *mutace* a *reprodukce* (vysvětleno dále v textu).



Obr. 4.1.: Evoluce evolučních algoritmů

Tento „přírodní výběr“ je ovlivňován výkonností jedince (respektive jeho adaptivity) v kontextu s prostředím, ve kterém je jedinec definován.



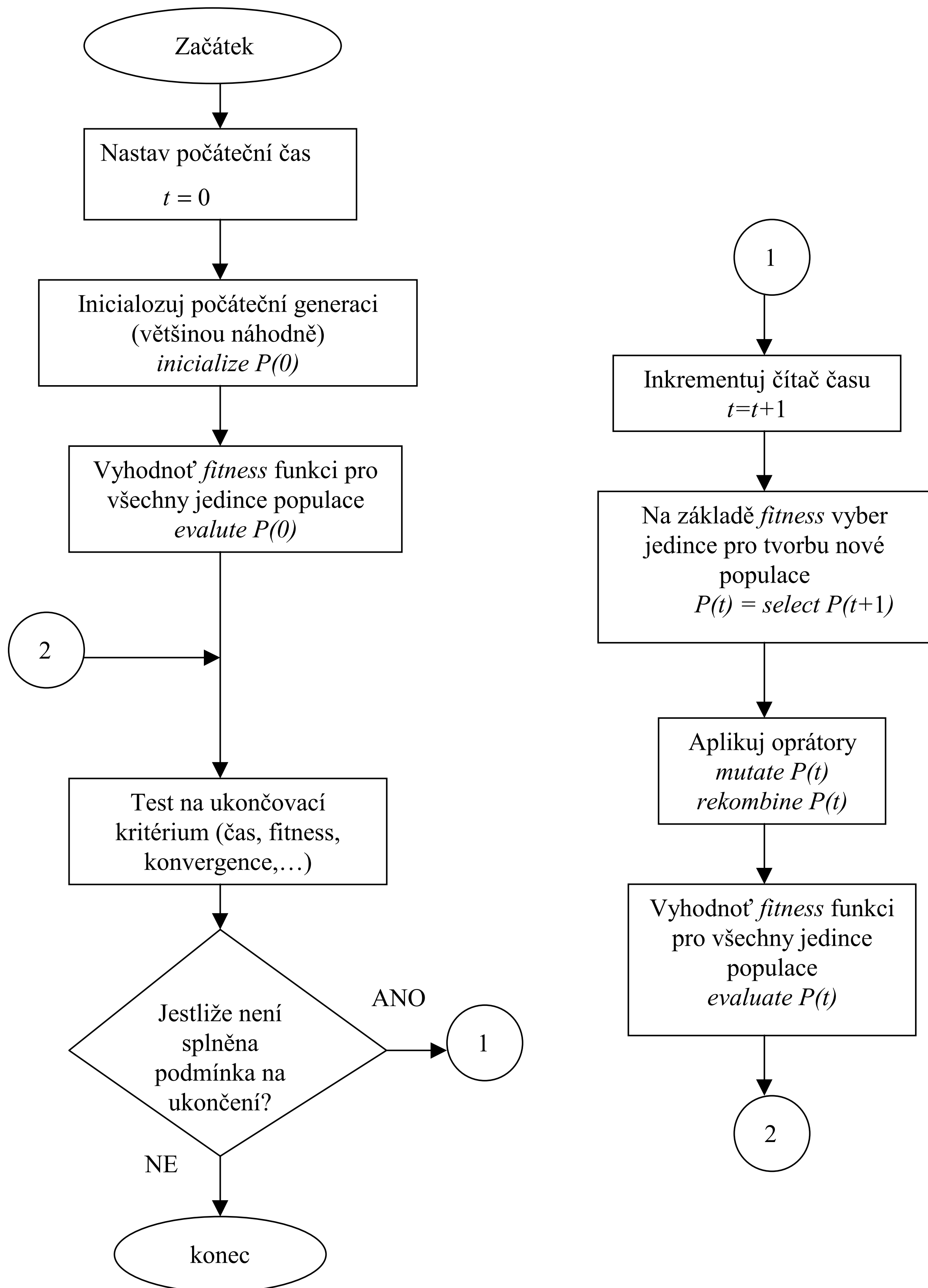
Obr. 4.2.: Vývoj datové reprezentace evolučních algoritmů

Evoluční algoritmy pracují s populací jedinců (jedinec je představován datovou strukturou), na kterou jsou aplikovány genetické operátory (genetic operators) jako je například mutace. Aplikací genetických operátorů na populaci dochází k její evoluci. Každý jedinec zaujímá bod ve stavovém prostoru a tím tedy představuje potencionální řešení problému. Proto je každému jedinci přiřazeno hodnotící číslo, které představuje míru výkonnosti či vhodnosti pro dané prostředí (fitness). Jedná se vlastně o míru kvality řešení zpracovávaného problému. Reprodukce se zaměřuje na jedince s nejlepší jedince v populaci, dochází tedy k vyčerpávání (exploitation) informace (fitness). Rekombinace a mutace jedince perturbuje, poskytuje tedy obecnou heuristiku pro prohledávání (exploration) stavového prostoru. Přestože jsou z biologického hlediska tyto algoritmy silně zjednodušeny, jsou dostatečně komplexní, aby poskytly robustní a výkonný adaptivní prohledávací mechanizmus.

4.2. Formulace problému evolučních algoritmů

Evoluční algoritmy jsou paralelní pravděpodobnostní algoritmy spravující populaci jedinců $P(t)=\{x_1^t,\ldots,x_n^t\}$, v čase t. Každý jedinec reprezentovaný datovou strukturou S představuje potencionální řešení daného problému. Každé řešení x_i^t je ohodnoceno mírou (fitness, objective function), což je zobrazení $f: x_i^t \rightarrow \Re$, kde $\Re$ je množina reálných čísel. Nová populace (iterace $t+1$) je formována operátorem selekce, který je závislý na fitness funkci. Někteří jedinci takto vytvořené populace jsou ještě transformováni reprodukčními operátory, představujícími obecnou heuristiku prohledávání (unárním operátorem typu mutace, $m_i : S \rightarrow S$ a operacemi vyššího řádu typu křížení, $c_j = S \times \ldots \times S \rightarrow S$. Po několika iteracích dochází většinou ke konvergenci. Nejlepší jedinec (nejvyšší hodnota udávaná fitness funkcí) pak představuje řešení blížící se optimu.

Kostru evolučních algoritmů lze popsat následujícím algoritmem (viz vývojový diagram na:obr.4.3.):

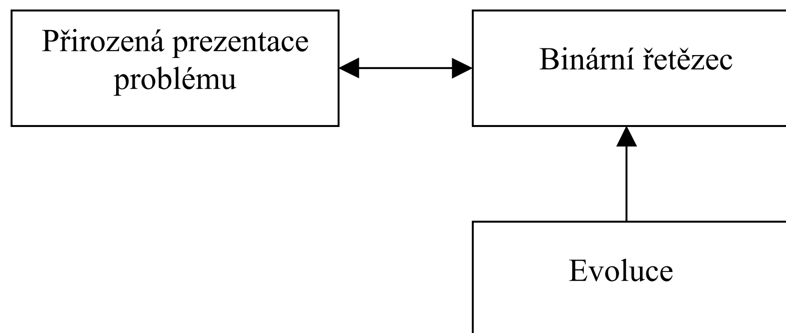


Obr. 4.3.: Vývojový diagram evolučního algoritmu

Tento algoritmus bývá často označován jako jednoduchý evoluční nebo genetický algoritmus (simple genetic algorithm).

4.3. Datová reprezentace a operátory genetických algoritmů

Datovou strukturou konvenčních genetických algoritmů je binární řetězec pevné délky, který tvoří slovo pevné délky s prvky abecedy $\{0, 1\}$. Obecně tato abeceda může být víceprvková (viz obr. 4.4.)



Obr. 4.4.: Interpretace obecnou abecedou

Reprezentace problému pomocí binárního řetězce vyžaduje vytvoření jednoznačného kódování při překladu z přirozené reprezentace problému na binární řetězec a naopak. Například máme-li najít optimální hodnotu parametru x na spojitém intervalu $\langle -1,2 \rangle$ s přesností na šest desetinných míst, je třeba diskretizovat parametr x na minimálně $3 \cdot 10^6$ úrovní. Parametr tedy bude reprezentován 22 bity, protže platí: $2^{21} < 3 \cdot 10^6 < 2^{22}$. Výsledný tvar dekodéru pro $x \in \Re$ pak bude: $x = -1.0 + \left(\langle b_{21} b_{20} \dots b_0 \rangle_2\right)_{10} \cdot \frac{3}{2^{22} - 1}$.

Z tohoto je zřejmé, že se data v přirozené reprezentaci musí diskretizovat na tolik úrovní, aby byla zaručena potřebná přesnost. Je zřejmé, že počet úrovní diskretizace je nutné volit tak, aby odpovídal mocnině čísla 2. Toto lze bez potíží zaručit při diskretizaci spojitých veličin, protože v případě potřeby můžeme zvětšit přesnost reprezentace. Problém ovšem může nastat v případě, kde parametr v přirozené reprezentaci nabývá konečného počtu hodnot, který mocnině dvou neodpovídá. V takovém případě není kódování jednoznačné, a protože jsou evoluční algoritmy doménově nezávislé (pracují nad datovou reprezentací bez znalosti významu obsahu),

může binární řetězec nabývat hodnot. Tento problém lze například řešit zavedením penalizační (pokutové) funkce, kdy dochází k jistému snížení výsledku fitness funkce. Takový jedinec je pak eliminován pomocí operátoru selekce. Dalším způsobem může být opravný algoritmus, kde například nahrazením nepřípustného jedince jedincem jiným, náhodně vygenerovaným, nebo také lze genotyp nepřípustného jedince převést na fenotyp a ten zpátky transformovat na genotyp, který již je přípustný. Tato transformace je příliš algoritmicky nákladná a vzájemná zobrazení nemusí být prostá. Nabízí se další možnost, která ovšem vyžaduje upuštění od striktně binárního kódování a zavedením hybridního kódování a patřičně (problémově) upravených genetických operátorů tak, aby při jejich použití byla minimalizována šance na vznik nepřípustného řešení. Zejména posledního způsobu se v součastné době hojně využívá, protože vede k dobrým výsledkům. Data jsou reprezentována přímo v přirozeném formátu (čísla s pohyblivou desetinou čárkou), nebo pomocí složitějších struktur jako jsou stromy, grafy (neuronové sítě), seznamy, vektory, atd. Často se též používá proměnná délka genomu. Tyto modifikace sovisí s dalšími typy evolučních algoritmů, jako je například genetické programování a hlavně nalezení vhodných operátorů pro každou jednotlivou úlohu je poměrně náročné. V případech, kdy kódování vyžaduje více parametrů (například vícerozměrné optimalizační úlohy), jsou jednotlivé parametry kódovány odděleně do samostatných binárních řetězců. Výsledný vektor pak vznikne jejich vzájemným zřetězením. Pomocí experimentů bylo prokázáno, že pro takto kódované parametry není potřeba upravovat genetické operátory.

V případě použití binárních řetězců je možné použít Grayův kód. Ukazuje se, že mutace takto kódovaných řetězců přináší lepší výsledky. Patrně to souvisí s tím, že mutace neposune jedince skokově jinam do stavového prostoru, nýbrž jeho posun je více „lineárnější“.

4.4. Genetické operátory genetických algoritmů

Genetické operátory jsou v podstatě funkce, které pracují buďto nad celou polulací jedinců, nebo přímo s konkrétními jedinci. Genetické algoritmy využívají standardně pět genetických operátorů. Jsou to:

- mutace
- křížení
- selekce

- inicializace
- inverze

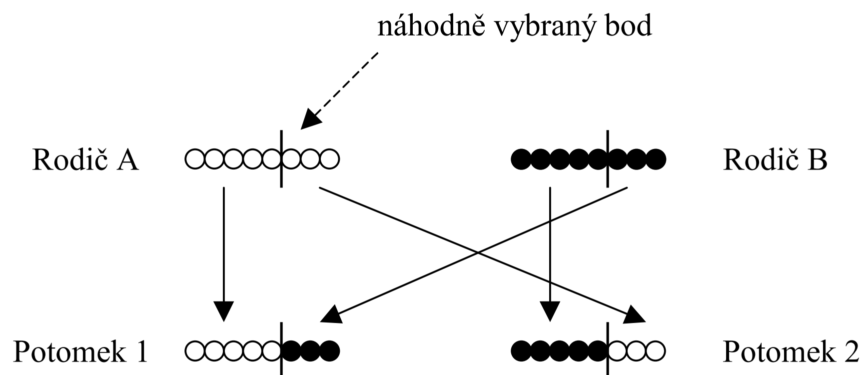
4.4.1. Mutace

Mutace je jedním z operátorů (a u genetických algoritmů jediným), který vytváří rozmanitost populace. Mutace spočívá v inverzi každého bitu řetězce s předem určenou pravděpodobností p_m,. Jinými slovy, invertuje se pouze ten bit řetězce, kterému pravděpodobnostní funkce přikáže inverzi (vlastní mutaci).

Takovýmto zavedením umělých poruch zabraňuje předčasné konvergenci a nutí algoritmus k prohledání větší části stavového prostoru. Tím zabraňuje trvalé ztrátě informace nějaké *allly* (bitu řetězce). V poslední době se objevují experimentálně potvrzené názory, které tvrdí, že aplikace pouze mutace, bez použití operátoru křížení, může produkovat stabilní algoritmus.

4.4.2. Křížení

Obecně křížení může být jedno nebo více bodové, případně uniformní.



Obr. 4.5.: Jednobodové křížení binárních řetězců

V případě jednobodového křížení se vytvoří dva potomci ze dvou rodičů. Chromozomy (binární řetězce) potomků obsahují části chromozomů obou rodičů. Viz obr. 4.5. Postup je následující. U dvou rodičů vybraných pro křížení se náhodně určí bod „řezu“, ve kterém se bude řetězec rozdělovat (u obou rodičů stejné místo řezu) a po zkopírovaní částí řetězců, jak můžeme vidět na obr. 4.5., dojde k vytvoření potomků.

Algoritmus dvoubodového křížení vypadá obdobně. Potomek je pak složen ze tří částí rodičů (pozn.: rodiče jsou stále dva). Tento způsob křížení bývá často považován za lepší než jednobodové, protože je u něj nižší pravděpodobnost zničení schématu.

Uniformní křížení probíhá bit po bitu s tím, že určení ze kterého rodiče bit bude se určuje náhodně.

4.4.3. Operátor inicializace a inverze

Operátor inicializace náhodně nastavuje jednotlivé bity binárního řetězce. Operátor inverze provádí binární inverzi řetězce.

4.4.4. Operátor selekce, selekční tlak

Operátor selekce se snaží napodobit proces přírodní selekce. Vybírá jedince na základě schopnosti adaptace (hodnoty fitness funkce pro daného jedince). Takto vybraní jedinci přežijí a budou produkovat potomstvo další generace a tím šířit svůj genetický materiál. Tato počítačová selekce, na rozdíl od té přírodní, není „absolutní“. Myšleno tak, že není dána nějaká minimální hodnota fitness, kterou daný jedinec musí mít, aby přežil v prostředí, ale je vztažena k fitness všech jedinců celé populace. Větší šance, pro šíření genetického materiálu, je nejlepším jedincům (jedinci s největšími hodnotami fitness v populaci) zaručena tím, že jsou do nové populace častěji vybíráni pro křížení (s pravděpodobností závislou na jejich fitness).

Operátor selekce v sobe zahrnuje dva protichůdné požadavky:

- požadavek na co nejrychlejší konvergenci a nalezení řešeni
- a požadavek na prohledání co největšího podprostoru stavového prostoru úlohy a tím minimalizaci rizika na uváznutí v lokálním extrému.

Řízení rychlosti konvergence se označuje termínem selekční tlak (selective pressure). Zvyšováním selekčního tlaku dochází ke velmi rychlé konvergenci a k většímu vyčerpání informace z již prohledaných bodů stavového prostoru. Výsledná populace se tím stává homogenní díky eliminaci slabších jedinců.

Při použití algoritmů selekce, které neumožňují selekční tlak řídit, může také dojít k předčasné konvergenci. Tato vlastnost je vysvětlována malým počtem jedinců v populaci a tím, že tyto algoritmy jsou pravděpodobnostní. Předpokládá se totiž, že počet jedinců v nové populaci bude $c(x_i, t+1) = p_i \cdot velikost_populace$, kde hodnota $p_i \approx f(x_i)$ představuje selekční tlak a c představuje koeficient růstu ($c > 1$) či poklesu

($c < 1$) počtu jedinců v následující populaci. Toto platí pouze za předpokladu, že se velikost populace blíží k nekonečnu ($velikost_populace \to \infty$). Snížení selekčního tlaku způsobuje prohledání větší části stavového prostoru a tedy větší rozmanitost populace. Při návrhu evolučních algoritmů je třeba správně vyvážit sílu selekčního tlaku s podmínkou pro ukončení běhu programu (ukončení vyhledávání řešení v podobě nejlepšího jedince).

Hledáním vhodného kompromisu mezi sílou selekčního tlaku a rozmanitostí populace vzniklo několik modelů selekce.

Klasickým modelem, který neumožňuje řízení poměru selekčního tlaku a rozmanitosti populace, je algoritmus *roulette-wheel*, u kterého je selekce přímo úměrná s fitness funkcí. Mějme tedy populaci s jedinci (genomy) $g_0 \dots g_{velikost_populace-1}$, algoritmus *roulette-wheel* pak lze popsat:

1) spočítej hodnotu fitness celé populace F

$$F = \sum_0^{velikost_populace} f(g_i)$$

2) spočítej pravděpodobnost přežití jednotlivých jedinců

$$p_i = \frac{f(g_i)}{F}$$

3) spočítej kumulativní pravděpodobnosti

$$q_i = p_0 + p_1 + \dots + p_i$$

4) vlastní selekce

for (i = 0; i < *velikost_populace*; ++i)

{

if (q_{i-1} < (generuj_náhodné_číslo_v_intervalu <0, 1>) <= g_i)

select g_i

}

Příkladem algoritmu, který oproti předešlému umí řídit selekční tlak, je algoritmus *tournament selection*:

```
for (i = 0; i < velikost_populace; ++i)
{
  náhodně vyber k jedinců (typicky bývá k = 2)
  a nejlepšího z nich vyber do další generace
}
```

Jednou z dalších možností je například *remainder stochastic sampling*, která pracuje tak, že vezme celočíselnou část podílu fitness jedince s průměrnou fitness populace. Tato hodnota pak určuje kolikrát bude jedinec zkopírován do nové populace. Zbytek po dělení udává pravděpodobnost další kopie.

4.4.5. Fitness funkce

Hodnotící funkcí, která ohodnocuje jednotlivé body stavového prostoru (jedince), je fitness funkce. Fitness funkce je závislá na řešeném problému a je obvykle součástí popisu problému. Pro některé problémy může být určení fitness funkce značně obtížné.

Neexistuje obecný návod jakým způsobem navrhnout fitness funkci. Vhodnou volbu fitness funkce lze přirovnat například k návrhu expertních systémů. Každý typ problému má jiná specifika. Za příklad určení fitness funkce může posloužit několik ukázek. V úloze optimalizace je fitness funkcí například množství spotřeby paliva, v rozpoznávání je to počet rozpoznaných objektů a při strategických hrách jednoduše skóre. Často bývá hodnota fitness funkce určena jako $\sum_i \left| hodnota_i - referenční_hodnota_i \right|$. Experimentálně bylo prokázáno, že určení fitness jako funkce celé generace, $f(x_i) = F(x_i, x_{j1}, \ldots, x_{jvelikost_populace-1})$, kde $j_k = 1 \ldots velikost_populace$ a $j_k \neq i$, může vést k robustnějšímu algoritmu.

V některých případech není nutné, aby byla fitness funkce úplně vyhodnocena, pak stačí částečné vyhodnocení nebo aproximace, protože jedním z požadavků na fitness funkci je rychlost jejího vyhodnocení.

4.5. Teoretické základy evolučních algoritmů

V tomto odstavci budou stručně popsány teoretické základy, které evoluční algoritmy využívají (tzv. teorem schematu). Tyto principy by měli vysvětlit, proč vlastně

genetické algoritmy fungují. Důkaz je založen na předpokladu reprezentace dat pomocí binárních řetězců, na dvoubodovém křížení, pojmu schema a na fitness typu proporcionální selekce (roulette-wheel).

4.5.1. Schema

Schema vzniká rozšířením základní abecedy binárních řetězců {0, 1} o „don't care" symbol „* ". Na pozici „* " se může vyskytovat jak znak „0", tak znak „1", Schema pak reprezentuje všechny řetězce, které jsou shodné ve všech pozicích vyjma „* ". Například schematu (10*1) odpovídají dva řetězce, (1001) a (1011). Na schema se lze dívat také jako na podprostor prohledávaného stavového prostoru.

Zavedením schématu můžeme porovnávat vlastnosti bodů (resp. kombinace parametrů) s nadprůměrnou hodnotou fitness funkce a také nám poskytuje do jisté míry návod jak hledat další body s nadprůměrnou hodnotou fitness funkce. Tato vlastnost je velice užitečná, protože v nelineárním prostoru není možné zjistit směr, kterým se má optimalizace (hledání maxima fitness funkce ve stavovém prostoru) dále ubírat. Další vlastností schematu je možnost ohodnotit schema průměrnou hodnotou fitness funkce všech řetězců, které do tohoto schematu náleží a schemata navzájem porovnávat. Schema samo o sobe poskytuje i vysvětlení, proč je lepší než ostatní schemata pomocí kombinací jednotlivých bitů. Například schema (1**) lze vysvětlit některou z kombinací bitů: (100), (101), (110) nebo (111). Optimální hodnota je vyhledávána kombinováním krátkých nadprůměrných schemat (building blocks). Toto si ukážeme dále v textu.

Abychom mohli popisovat vlastnosti jednotlivých schemat, musíme si zavést ještě dva pojmy:

- *Řád schematu S*, označený $o(S)$, představuje počet pevných („non-don't care") pozic ve schematu.
- *Definující délka schematu S*, značená $\delta(S)$, která představuje vzdálenost mezi první a poslední pevnou pozicí v řetězci.

S využitím těchto pojmů můžeme stanovit definovat:

- počet jedinců ve schematu (*L* označuje délku řetězce) jako: $2^{L-o(S)}$
- počet schemat pro daný řád: $\binom{L}{o(S)} \cdot 2^{o(S)}$
- a z toho celkový počet schemat: 2^L

Pro následující odvození můžeme, bez újmy na obecnosti, uvažovat pouze problém maximalizace. Platí totiž, že $\min\{f(x)\} = \max\{g(x)\} = \max\{-f(x)\}$. Dále budeme předpokládat, že $f(x)$ je fitness funkcí a že $f(x) > 0$.

Jak bylo již dříve v textu uvedeno, je evoluční proces simulován opakováním následující sekvence, dokud není splněna podmínka pro ukončení programu (např. konvergence hodnoty fitness funkce):

$t = t + 1$

vyber $P(t)$ z $P(t-1)$

rekombinuj $P(t)$

vyhodnoť $P(t)$,

kde *t* číslo iterace (číslo generace), $P(t)$ je populace v čase *t*. Dále označíme velikost populace: *velikost_populace*, délku řetězce: *L*, počet řetězců odpovídajících schematu *S*: $\xi(S,t)$ a hodnotu fitness funkce v čase *t* (která je definována jako průměrná hodnota fitness funkce všech řetězců v populaci odpovídajících schematu *S*): $eval(S,t)$. Předpokládejme, že v čase *t* v dané populaci *p* řetězců $\{g_{i_1}, g_{i_2}, \ldots, g_{i_p}\}$ odpovídá schematu *S*. Potom platí

$$eval(S,t) = \sum_{j=1}^{p} \frac{eval(g_{i_j})}{p}.$$

Během kroku selekce je ze staré populace vybírána „prostřední“ populace, na niž budou aplikovány operátory rekombinace a mutace. Tak vznikne populace nová se stejným počtem řetězců. Každý řetězec ze staré populace může být do „prostřední“ populace nulakrát, jedenkrát nebo vícekrát v závislosti na své hodnotě fitness funkce. Pravděpodobnost výběru řetězce g_i do nové populace je

$$p_i = \frac{eval(g_i)}{F(t)},$$

kde

$$F(t)=\sum_{i=1}^{i=velikos_populace} eval(g_i)$$

je hodnota fitness v čase *t*.

Po kroku selekce bude schematu *S* odpovídat $\xi(S,t+1)$ řetězců, přičemž pravděpodobnost, že v kroku selekce bude vybrán řetězec odpovídající schematu *S* je $p_i=\frac{eval(S,t)}{F(t)}$. Je tedy zřejmé, že

$$\xi(S,t+1)=\xi(S,t)\cdot velikost_populace\cdot\frac{eval(S,t)}{F(t)}.$$

Pokud označíme průměrnou hodnotu fitness funkce populace jako $\overline{F(t)}=\frac{F(t)}{velikost_populace}$, můžeme psát:

$$\xi(S,t+1)=\xi(S,t)\cdot\frac{eval(S,t)}{\overline{F(t)}}.$$

Počet řetězců v populaci tedy roste s velikostí poměru fitness daného schematu a průměrné fitness populace, proto se zvětšuje počet řetězců odpovídajících nadprůměrným schematům, tedy těm, pro které platí nerovnost $\frac{eval(S,t)}{\overline{F(d)}}>1$. Tato nerovnice se nazývá *nerovnice reproduktivního růstu schematu*.

Z hlediska časového horizontu *t* iterací a za předpokladu nadprůměrnosti daného schematu o konstantní hodnotu *e*, tj. $eval(S,t)=\overline{F(t)}\cdot(1+e)$, můžeme psát:

$$\xi(S,t)=\xi(S,0)\cdot(1+e)^t.$$

Z toho je patrné, že počet řetězců ve schematu exponenciálně roste.

Dalším krokem cyklu evolučního algoritmu je rekombinace, která vytváří nové jedince z „prostřední" populace. Proces rekombinace je tvořen aplikací dvou genetických operátorů a to operátorů *křížení* a *mutace*. Proto tedy budeme zjišťovat vliv těchto dvou operátorů na počet schemat v populaci za jednu iteraci. Uvažujme zatím pouze binární operátor jednobodového křížení. Ten náhodně vybere pozice *x*, $1\le x<L$. Vzniknou dva nové řetězce n_1 a n_2. Řetězec n_1 bude složen z bitů $1\ldots x$ řetězce s_1

(rodiče A) a z bitů $x+1\ldots L$ řetězce s_2 (rodiče B), řetězec n_2 bude začínat bity $1\ldots x$ řetězce s_2 a končit bity $x+1\ldots L$ řetězce s_1.

Pokud operaci křížení budeme provádět místo na řetězcích na schematech, je zřejmé, že může dojít ke zničení schematu. Pravděpodobnosti zničení schematu je závislá na definující délce schematu. Budeme-li brát v úvahu i to, že pozici, ve které má dojít ke křížení, vybíráme z $L-1$ bodů, je pravděpodobnost zničení schematu $p_d = \frac{\delta(S)}{L-1}$ a tím pádem je pravděpodobnost zachování schematu $p_s = 1 - p_d = 1 - \frac{\delta(S)}{L-1}$. Musíme uvažovat ještě dvě následující skutečnosti a to:

- křížení je podmíněno pravděpodobností p_c a
- schema může přežít i při výběru místa křížení mezi pevnými pozicemi ve schematu (např. křížení dvou stejných řetězců),

pak dostaneme výslednou pravděpodobnost přežití:

$$p_s \geq 1 - p_c \cdot \frac{\delta(S)}{L-1}.$$

Nyní dáme dohromady operátory selekce a křížení, dostaneme tak nový vztah pro schema reproduktivního růstu:

$$\xi(S,t+1) \geq \xi(S,t) \cdot \frac{eval(S,t)}{\overline{F(t)}} \cdot \left[1 - p_c \cdot \frac{\delta(S)}{L-1}\right].$$

Z této rovnice je zřejmé, že se počet nadprůměrně krátkých schemat bude během jednotlivých iterací exponenciálně zvětšovat.

Aby byl výpočet růstu reprodukčního schematu kompletní, zbývá už jen zahrnout operátor mutace, který je rovněž aplikován na schema během každé iterace. Mutace náhodně invertuje bity řetězce s pravděpodobností p_m. Pokud má schema přežít mutaci, musí zůstat nezměněny všechny pevné pozice. Vzhledem k tomu, že počet pevných pozic je dám řádem schematu, je pravděpodobnost přežití:

$$p_s(S) = (1 - p_m)^{o(S)}.$$

Pravděpodobnost mutace je zpravidla volena podstatně menší než 1 ($p_m \langle\langle 1$), pokud není přímo $p_m = 0$. Pak můžeme předchozí vztah přepsat na:

$$p_s \approx 1 - o(S) \cdot p_m.$$

Zahrnutím konečně všech tří operátorů (selekce, křížení a mutace) dostáváme konečnou podobu schematu reproduktivního růstu:

$$\xi(S,t+1) \geq \xi(S,t) \cdot \frac{eval(S,t)}{\overline{F(t)}} \cdot \left[1 - p_c \cdot \frac{\delta(S)}{L-1} - o(S) \cdot p_m\right],$$

kde jsme člen $\left(o(S) \cdot p_m \cdot p_c \cdot \frac{\delta(S)}{L-1}\right) \to 0$ zanedbali.

Tato rovnice nám tedy udává očekávaný počet řetězců odpovídajících schematu *S* za jeden krok iterace. Jak již bylo výše uvedeno, je počet řetězců vyhovujících nadprůměrně krátkým schematům se exponenciálně zvětšuje. Výsledek rovnice schematu reproduktivního růstu lze shrnout do dvou vět známých jako *teorem schematu* a *bloková hypotéza.*

4.5.2. Teorem schematu

Počet krátkých nadprůměrných schemat nízkého řádu se během jednotlivých generací SGA (Simple Genetic Algorithm) exponenciálně zvětšuje.

4.5.3. Bloková hypotéza

Genetické algoritmy hledají body, které se blíží optimu, srovnáváním krátkých nadprůměrných schemat nízkého řádu nazývaných stavební bloky (building blocks).

Z předcházející úvahy o teoremu schematu je zřejmé, že jsme se dopustili několika zjednodušení, které užitečnost teoremu schematu snižují. Tím, že v úvahách nefigurují nerovnosti v jednotlivých vztazích, bereme v úvahu v podstatě jen limitní případy a díky pravděpodobnostem i s jistým rozptylem.. Tím dochází ke ztrátě, případně ke zkreslení informace. Pokud bychom se pokusili tento nedostatek eliminovat například predikováním počtu řetězců v jednotlivých schematech přes několik iterací (generací), mohli bychom dospět k naprosto zavádějícím údajům. Jinou problematickou úvahou by byl pokus o odhad vývoje (v čase $t+1,\ldots,t+n$) schemat založený na průměrné hodnotě fitness funkce celé populace v čase *t*. Je jasné, že tento odhad má smysl v první, maximálně v druhé, generaci.

4.6. Porovnání klasických prohledávacích algoritmů a EA

Největší rozdíly mezi klasickými prohledávacími metodami a evolučními algoritmy lze znázornit následující tabulkou:

Klasické algoritmy	*Evoluční algoritmy*
deterministické	nedeterministické
komplexní	nekomplexní
zpracovávají pouze jeden bod najednou	paralelní
většinou lokální prohledávání	globální prohledávání
doménově závislé	doménově nezávislé

Tab.č.4.1.: Porovnání klasických prohledávacích algoritmů a EA

Přestože se zdá, že principy, na kterých jsou evoluční algoritmy založeny, se diametrálně liší od principů klasických prohledávacích algoritmů. Existují studie, které dokazují, že evoluční algoritmy jsou variací informovaného prohledávacího algoritmu s několika novými vlastnostmi. Odpovídající analogii v názvosloví klasických prohledávacích algoritmů a evolučních algoritmů ukazuje následující tabulka:

Klasické algoritmy	*Evoluční algoritmy*
uzel	jedinec
seznam expandovaných uzlů	populace
reprezentace řešení	chromozom
část řešení	gen
operátory expanze	křížení, mutace
strategie expanze	selekce

Tab.č.4.2.: Analogie terminologie mezi klasickými prohledávacími algoritmy a EA

Evoluční algoritmy oproti klasickým prohledávacím algoritmům mají především tyto přednosti:

- podstatně menší nebezpečí uváznutí algoritmu v lokálním extrému (některé z klasických algoritmů se tomuto riziku dovedou do jisté míry vyhnout, například simulované žíhání),
- kombinací řízené a stochastické strategie je možnost vhodně volit poměr strategií selekčním tlakem (řízená část strategie se především projevuje využíváním informací o předchozím hledání obsažených v předešlých generacích),
- přirozená možnost paralelního zpracování (nejčastěji rozdělením populace na subpopulace a vzájemnou migrací),
- robustnost a rychlost
- univerzálnost (pomocí evolučních algoritmů je možné řešit širokou oblast problémů a to i v silně nelineárních stavových prostorech vysokých dimenzí),
- k hodnocení kvality řešení a k následnému rozhodování o dalším postupu se využívá jednoduchá míra (fitness funkce), proto není potřeba hledat složité diferenciální rovnice, jako například u gradientních metod.

Snad jedinou nevýhodou evolučních algoritmů je zřejmě skutečnost, že pro správný běh programu je třeba správně nastavit různé parametry, jako je například pravděpodobnost mutace, křížení, atd. Tyto parametry je velmi obtížné odhadnout a zjišťují se nejčastěji empiricky.

5. Návrh řešení problému

V této kapitole bych se chtěl zabývat jednotlivými kroku mého postupu vlastního návrhu. Postupně bych zde chtěl popsat, jednotlivé fáze, které mě vedli ke konečné volbě mého řešení problému mobilních robotů a to, skloubení evolučních algoritmů s etologickými principy, které byli stručně rozebrány v předešlých kapitolách.

5.1. Úkoly pro řešení

Pro potřeby návaznosti projektů v naší mobotické skupině mi byly zadány následující úkoly:

- **Naučit mobota nenarážet do předmětů**

Senzorické vstupy:

Detekce objektů pomocí IR čidel a taktilních nárazníků.

Motivační vstupy:

Nenarážej - tato hodnota vyjadřuje touhu nenarážet, resp. kvantifikuje náraz. Jak je zřejmé z její definice je diskrétní.
0 pokud nenarazil
1 pokud narazil

Putuj - tato motivace je v systému zavedena jako popud k pohybu, respektive základní vlastností mobota bude pohyb. Pak totiž může teprve dojít k učení se nenarážet, budu-li se pohybovat.

Motivace, kterými má být mobot naučen se vyhýbat předmětů jsou vlastně parametry vícekriteriálního problému a jsou tedy zahrnuty do hodnotící fitness kunkce (viz dále v textu).

- **Nalézt danou definovanou oblast**

Senzorické vstupy:

Detekce objektů pomocí IR čidel, taktilních nárazníků případně sonaru.

Motivační vstupy:

Nalezni cíl - tato hodnota vyjadřuje touhu dosáhnout cílové oblasti.

Putuj - tato motivace je v systému zavedena jako popud k pohybu (jako u předešlého.

5.2. Vlastní řídící systém

Tato podkapitola se bude zabývat definicí vlastního řídícího systému. Z důvodu návaznosti zde budu vycházet hlavně z prací Petruse [2] a Maixnera [3].

5.2.1. Motivace

Jak již bylo poznamenáno, měl by tento ŘS sloužit i pro výzkum motivací. Definujme si tedy, co jsou vlastně motivace.

Motivace je vnitřní puzení jedince k nějaké dané činnosti. Toto puzení je v konečném efektu uspokojeno buď danou cílovou činností, nebo uspokojením nějaké vnitřní hodnoty (veličiny). Motivaci je možno si v jistém smyslu představit i jako puzení k nějaké činnosti. Například hlad (nedostatek energie) je jednoznačnou motivací pro jeho uspokojení. V tomto případě je motivace projevem vnitřní hodnoty hladu, která přesáhla nějakou definovanou hranici. Avšak je možno definovat i další motivace, například nutkání být někde, jehož hodnota je nulová právě v místě, kde chceme být.

Je nasnadě, že motivace, resp. její vnitřní hodnota je v čase proměnná. Například hlad nám stoupá, endogenní podněty rostou, dokud nejsou uspokojeny vnější činností apod. Pro možnost zpracování problému pomocí evolučních algoritmů bude motivace pouze diskrétní (0 – on / 1 – off) a míru její důležitosti bude vyjádřena její prioritou (viz níže struktura kódování genu).

Dovolím si malé odbočení. Právě výše zmiňované apetenční chování je vyhledávání takové konfigurace vnějších podmínek, která vede k uspokojení takové motivace.

Je však také pravdou, že jedinec nebude vědět, jaká je správná konfigurace vnějších podmínek. Proto je nutné vytvářet mechanismus, jenž by mohl vést aktivně k nalézání dvojic motivace/pud – uspokojení pud). V etologii se tomuto chování říká

explorativní. To znamená, že organismus testuje, zda daná vnější (potažmo i vnitřní) konfigurace uspokojuje nějakou motivaci, případně zda nějaká motivace nesnižuje svou vnitřní hodnotu.

Dovoluji si také poznamenat, že explorativní chování může mít za následek určení množiny uspokojující pudy jako posloupnosti jednotlivých chování. Tato definice by pak vedla na odlišnou strukturu ŘS, než je uvedeno pro účely tohoto textu.

Dalším důsledkem takto definovaných motivací je ta skutečnost, že inteligence mobota bude závislá na senzorech (rozpoznávací úrovních senzorů). Těžko může vzniknout vazba mezi objektem a chováním, pokud není senzorický systém schopen tento objekt rozlišit od ostatních objektů. Tím se však již dostáváme do oblasti rozpoznávání. Jak již bylo zmíněno, etologie zahrnuje i tuto oblast, ale tato oblast je již nad rámec této práce. Dá se však říci, že čím dokonalejší senzory, tím je možno vytvořit specifičtější, potažmo inteligentnější chování.

5.2.2. Senzorické vstupy

Pro potřeby tohoto textu nebudeme výrazněji rozlišovat mezi přímým senzorickým vstupem a již zpracovanou hodnotou, resp. čitím (viz. [2] a [3]). Proto nám vstupy budou představovat informaci, která je nutná pro řešení dané úlohy.

Vstupy jsou zvoleny v co nejjednodušší formě, tj. jsou neparametrické, udávají nám pouze hodnotu zda je daný senzor aktivní či neaktivní. Toto omezení je zvoleno s ohledem na co nejjednodušší zápis struktury chování pro potřeby GA. Pak je totiž možné tvořit řetězce chování, aniž bychom hodnotili kvalitativní senzorickou informací. Tato informace je však pro řešení složitějších úloh nutná a proto je potřeba ji zadat způsobem uvedeným na následujícím příkladu.

Představme si, že potřebujeme reagovat na překážku. Tuto jsme schopni identifikovat od vzdálenosti 2 m až po její přímý kontakt s tělem robota.

V klasickém případě bychom zavedli senzor překážka, jehož hodnota by definovala vzdálenost senzoru od robota. Hodnota větší než dva by pak znamenala nepřítomnost překážky.

Definujeme senzor **Dotýká se** pro vzdálenost = 0,

Je blízko pro vzdálenost $0 \div 0.5$ m

Je daleko pro vzdálenost $0.5 \div 2$ m

Skrze takto definované senzory jsem schopni přenést kvalitativní informaci do jednoduché struktury ŘS.

Tato struktura senzorů je velmi blízká fuzzy množinám. Je tedy možno některé závěry fuzzy logiky použít i pro potřeby tuto oblast robotiky. V dalším textu však této vlastnosti nevyužiji.

5.2.3. Blok chování

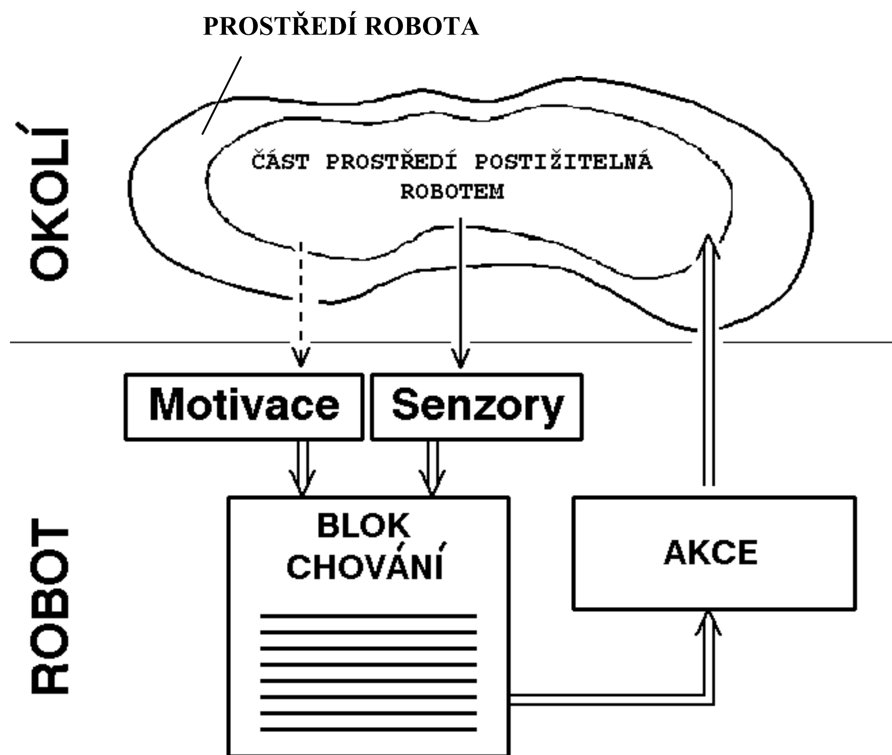
Tento blok je vlastně základní znalostní bází. Je v něm uloženo, jak má systém reagovat na vzniklou vnější událost. Svým členěním a významem je tato báze blízká reakční bázi.

V bloku chování je uložen řetězec chromozomu (výsledek po aplikaci GA, viz dále v textu), kde jednotlivé geny představují v podstatě vylepšenou reakční bázi. Každý gen totiž obsahuje vektor vstupních senzorů (vnímajících okolí), vektor motivací (vnitřní stav mobota) příslušnou akci její prioritu. Celý chromozom funguje tak, že se vezme aktuální stav senzorů a vnitřní stav mobota a ten se porovná s informacemi obsaženými v genech (s reakční bází). Vybere se gen (vektor reakční báze, dále už jen gen) nejpodobnější vstupnímu vektoru (senzory a motivace) a ten se předá k dalšímu zpracování (vykonání blokem akcí). V případě více stejně podobných genů (lišících se ve stejném počtu bitů) se přihlíží na hodnotu priority. Z vektoru genu se tedy přečte akce, která se má vykonat, a ta se postoupí *bloku akcí.*

5.2.4. Blok akcí

Tento blok má za úkol realizovat zvolené akce z bloku chování. Dostane vektor akcí k provedení a postará se o předání povelu příslušným akčním prvkům. Další funkcí je, či může být realizace a kontrola složených akcí. Pak blok akcí spouští jednotlivé primitiva (nejjednodušší, dále nerozložitelné akce).

5.2.5. Blokové schéma



Obr. 5.1.: Blokové schéma ŘS mobota

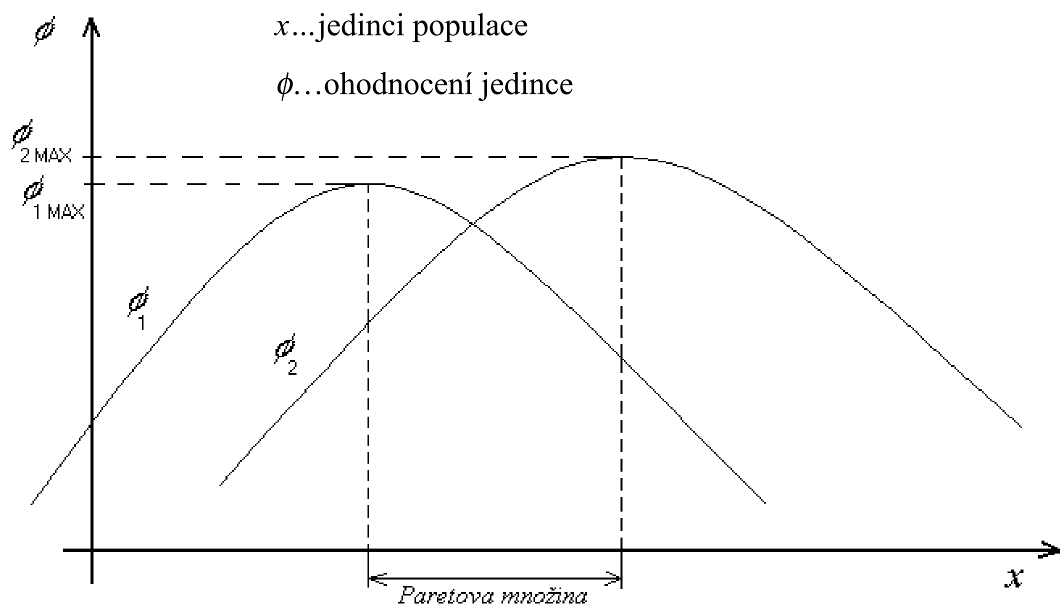
Toto schéma popisuje vzájemnou vazbu všech výše jmenovaných struktur i s ohledem na jejich vztah k okolí mobota.

5.3. Hodnotící funkce mobota (Fitness funkce)

Pro aplikaci EA, resp. pro potřeby porovnávání úspěšnosti jednotlivých sad chování v bloku chování je nutné mít jisté ohodnocení. Toto ohodnocení by v sobě mělo obsahovat jak hodnocení dosavadní činnosti robota, tak i hodnocení jeho schopností.

Klasické systémy, tj. systémy neodvozené z etologických, mají takové funkce většinou odvozeny od přesného sledování cíle, resp. postupu plnění cíle. Tato metoda je ovšem vhodná pouze pro roboty řešící jen několik snadno specifikovatelných úloh. Pro roboty typu R.U.R. je však vytvoření takového kritéria touto cestou skoro neřešitelné. Nevíme totiž přesně dopředu, co bude daný robot dělat, s jakými se setká překážkami apod. Dokonce nevíme, jak přesně hodnotit, když je daný robot výtečný v jedné činnosti, ale další činnost prostě provádět neumí. Je takový robot dobrý, či špatný?

Na tuto otázku v evolučních algoritmech odpovídá hodnotící funkce (fitness funkce), jejíž nalezení není triviální. Tato funkce hodnotí úspěšnost jedince podle schopnosti přežít, jinými slovy, testuje vhodnost jeho chromozomu. V tomto případě se dokonce jedné o vícekriteriální problém, kde jeden jedinec perfektně splňující jeden požadavek, nemusí umět plnit jiný a naopak. Touto problematikou se zabývá teorie *Paretových množin,* kde se jedná o nalezení vhodného kompromisu mezi jednotlivými kritérii. Ideální jedinec by měl mít všechna kritéria maximální, ale to většinou není možné. Vystoupí-li jedno kritérium jiná naopak může poklesnout (viz obr. 5.2.).



Obr. 5.2.: Paretova množina pro dvě hodnotící kritéria

Pro řešení našeho problému postačíme s podmínkou, že suma všech kritérií bude udávat úspěšnost jedince (jedno z možných řešení vícekriteriálních problémů). Tím dostaneme jedince stejně úspěšné, ale každý z nich bude mít jiné predispozice k plnění úkolů.

Nalezení hodnotící funkce není triviální, jak již jsem uvedl. Abych mohl správně ohodnotit vhodnost jedince, musím jej touto funkcí otestovat. Pro tento účel slouží simulace reálného prostředí, kde se mobot určitý čas (počet cyklů) nechá pohybovat, nejlépe tak dlouho, aby se stačili projevit všechny vlastnosti (geny) jedince a tak je

ohodnotit. Tedy simulátor prostředí mobota nemá jiný význam, než jak funkce ohodnocení kvalit mobota (fitness funkce).

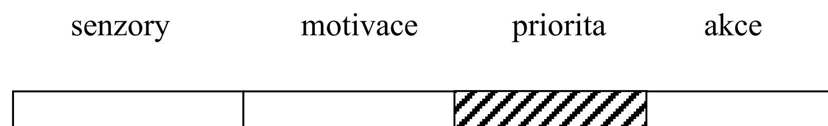
5.4. Volba genetické reprezentace

5.4.1. Způsob kódování

Z důvodů jednoduššího kódování jednoho genu jsem se rozhodl pro volbu binárního řetězce. Vedl mě k tomu především ten fakt, že pro evoluční operátor *mutace* je tento způsob výhodnější. Když totiž dochází k výběru pozice v genu, kterou je třeba zmutovat nastává v případě nebinárního kódování problém na jakou hodnotu se má hodnota na té pozici obsažená zmutovat. Toto by zaneslo do celého algoritmu zbytečně navíc další náhodný, či jinak řešený (např. empiricky), prvek, který by celý problém příliš zesložitil a také zpomalil. Z tohoto důvodu se *binární kódování* jeví jako vhodné.

5.4.2. Význam genu pro mobota

Po zvážení možné kompatibility na výzkumu ve skupině mobilní robotiky jsem se rozhodl gen mobilního robota koncipovat jako jednu instrukci reakční báze. Tento gen (potažmo vektor) by měl pak následující podobu (viz obr. 5.3.):



Obr. 5.3.: Grafická podoba jednoho genu mobota

Jednotlivé části genu mají následující význam. Část *senzory* představuje oblast pro binární reprezentaci stavu senzorů z vnějšího prostředí (taktilní nárazníky, IR čidla,…), každému bitu bude pevně přidělen příslušní senzor a bity 0 a 1 v podstatě představují stav ON/OFF (detailnější popis viz dále v textu). Blok *motivace* je v podstatě totéž jako blok pro senzory, akorát s tím rozdílem, že se jedná o vnitřní senzory mobilního robota (jedince). Tyto vnitřní senzory mohou signalizovat stavy jako je například hlad (pokles energie).

Další částí genu (vektoru) je *priorita*, která má z pohledu genetického programování trochu odlišný význam. Představuje totiž prioritu akce (práh spouštění AAM v etologii), proto se jeho hodnota bude nutně v průběhu „života“ jedince měnit, což není zrovna typické pro genetické programování. Takto začleněný faktor priority byl v konečném řešení pozměněn, aby se daly použít klasické operátory evolučních algoritmů (viz dále v textu), pro tento okamžik slouží pouze pro lepší ilustraci celého problému.

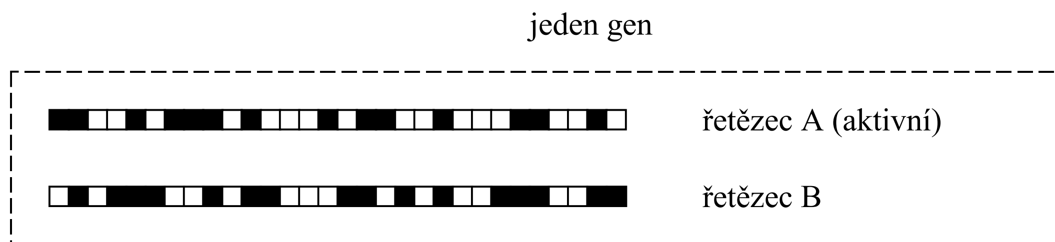
Konečně v blok *akce* je binárně zakódováno, jaké akční prvky mají být spuštěny a jak mají být spuštěny (například levý motor na 50% výkonu). Takto vytvořená gen tvoří akční primitivum mobota.

Celou genetickou výbavu mobota tvoří binární řetězec sestavený za sebou jednotlivé geny. Toto byl jeden z prvních návrhů, který přinesl několik poznatků. Jedním z nich byl ten fakt, že při ponechání dynamické délky řetězce se tento „DNA“ kód neúměrně prodlužoval. Toto se dalo snadno odstranit zavedením penalizační funkce, která pokutovala jedince za příliš dlouhou délku jeho binárního řetězce. Jiný problém byl v tom, že kód byl špatně přístupný pro proces vyhledávání vhodného genu pro aktuální situaci. Navíc vznikali jedinci, kteří měli naopak málo genů (instrukcí reakční báze) a proto šli špatně ohodnocovat fitness funkcí. Proto jsem od binárního řetězce s dynamicky se měnící délkou opustil a zavedl raději pevnou délku tohoto řetězce.

5.4.3. Dvojitý genetický kód mobota

Při výpočtech, založených na evolučních algoritmech, dochází k předčasné konvergenci schématu (nalezení lokálního extrému ve stavovém prostoru, který se od globálního extrému příliš liší a tedy není vhodný). Tím, že populace již zkonvergovala došlo ke ztrátě genetického materiálu. Jinými slovy, jedinci již nejsou rozmanitě rozptýleni po „celém“ stavovém prostoru, nýbrž se nacházejí v blízkém okolí lokálního extrému do něhož zkonvergovali. Proto mě napadlo pro obnovu genetického materiálu, pro takovouto situaci, použít dvojitý genetický kód jako je tomu běžně v přírodě. Každý gen bude tedy kódován dvakrát s tím, že vhodost jedince (hodnota fitness funkce) bude počítána pouze z jednoho (aktivního) řetězce genu. Operátory křížení a mutace budou samozřejmě aplikovány na oba dva řetězce stejným způsobem. V případě, že dojde k předčasné konvergenci, prostým prohození obou řetězců se obnoví „pestrost“

genetického materiálu a algoritmus se již nemusí starat o hledání nového genetického materiálu.



Obr.5.4.: Gen mobota s dvojitým genetickým kódem

Toto je jeden z možných způsobů. Později v literatuře jsem se dozvěděl, že tuto metodu již použil Bagly, Holstein a Brindl již přibližně před 15 lety. Tato metoda se nazývá *multiploidní dědičnost*. Nicméně se tato metoda ukázala být nevhodná pro svoji paměťovou náročnost a pro zpomalení celého běhu programu, protože většina operací musela být prováděna nad dvojnásobným objemem dat.

Pro bezpečný běh programu jsem proto použil jiný způsob a tím je *hypermutace*.

5.4.4. Hypermutace

Hypermutace je metoda, která nahrazuje předešlou metodu multiploidní dědičnosti. Jede o to, že v případě předčasné konvergence se získá nový genetický materiál obnovou starého (z aktuální populace) tím, že se koeficient, představující pravděpodobnost mutace bitů v binárním řetězci, extrémně zvýší na 30 ÷ 50 % (ze svých původních řádově setin procenta). Takto sice ztratíme více času pro obnovu genetického fondu, ale celkově se celý algoritmus urychlí skoro na dvojnásobek.

5.4.5. Funkce priority u genů mobota

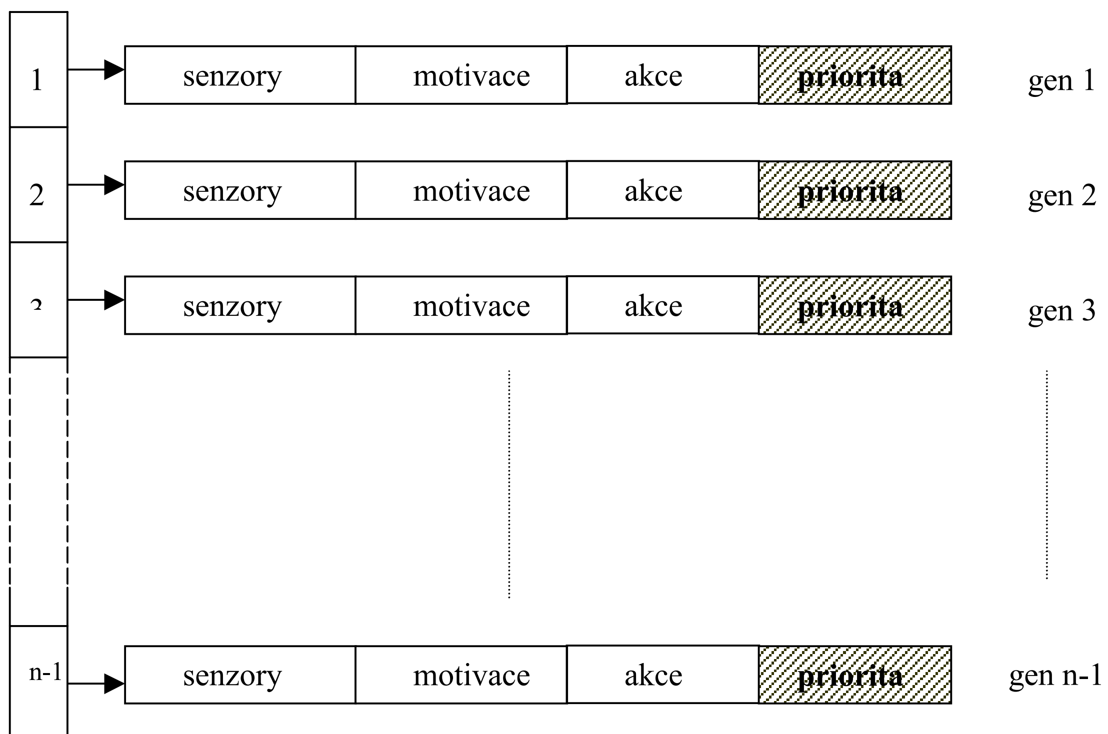
Priorita genu mobota má význam v případě, že se nalezne porovnáváním senzorické a motivační části vektoru genu více než jeden genový vektor jedince (mobota). V případě takovéto shody pak rozhoduje priorita jednotlivých genů. Po „narození“ jedince mají všechny geny stejnou prioritu (v případě shody i v prioritách se rozhoduje náhodným výběrem). V průběhu života jedince v prostředí se tyto váhy (priority) mění v závislosti na tom, zda-li byl či nebyl gen použit. V případě jeho použití se jeho hodnota (priorita má stejný význam jako práh spouštění AAM, viz kapitola o etologii) sníží a úměrně tomu se zvýší hodnota všech ostatních prahů (priorit u ostatních

genů). Tím dochází k jevu, který nastavuje jednotlivé hodnoty priorit genů jedince a tak jedinec získává jisté „zkušenosti“ v prostředí.

Protože při procesu křížení a mutace dochází ke změnám v genetickém kódu mobota (popisu jedince) není vhodné hodnoty priorit předávat potomkům. Vznikají tak dvě formy informace přesně jako je tomu v přírodě. Na jedné straně genetická výbava jedince (v našem případě reakční báze) a informace představující zkušenosti nashromážděné v průběhu existence jedince.

Informace o zkušenostech jedince se totiž ukládají formou těchto priorit a proto by došlo k předání informace o prostředí na modifikovaný gen. Proto jsou tyto dvě formy informace odděleny.

5.4.6. Chromozom mobota v podobě 2D řetězce



Obr. 5.5.: Chromozom mobota v podobě 2D řetězce

Z důvodů jednodušší práce s genetickým materiálem (konkrétně s křížením) a také snadnějšího vyhledávání příslušného genu v řetězci jsem se rozhodl sekvenční binární kód řetězce, kde následuje jeden gen za druhým, nahradit tuto strukturu dvourozměrným řetězcem. Jedná se v podstatě o matici, jejíž řádky jsou tvořeny jednotlivými geny (viz obr. 5.5.). Na tento chromozóm je aplikován genetický algoritmus tak, že v jednotlivých

genech dochází pouze k mutacím (podle teorie GA je mutace postačujícím nástrojem pro nalezení řešení) a ke křížení dochází mezi celými geny. Tedy jedinci si mezi sebou vyměňují celé řetězce genů.

5.4.7. Detailní struktura jednoho genu mobota

Jak jsem již uvedl výše, skládá se gen z několika částí (viz obr. 5.3.) a to ze senzorické části, motivační části a části akcí. Prioritu ponechám zatím stranou.

Senzorická část vypadá následovně:

0.-23. bit	taktilní čidla nárazu
24.-39. bit	IR čidla nárazu (nastavená na vzdálenost 10 cm)
39.-219. bit	data ze sonaru (pro náš případ nepřipojené)
220. bit	mobot vidí před sebou jídlo
221. bit	mobot vidí před sebou cíl
222. bit	mobot vidí před sebou kolegu

(ostatní bity do počtu 256 jsou určeny jako rezerva)

Část motivů:

0. bit	nenarážej (taktilní čidla)
1. bit	nenarážej (IR čidla)
2. bit	putuj
3. bit	hledej cíl
4. bit	hledej zdroje (jídlo)

Část akcí:

8.-10. bit	8 úrovní pravý motor
11.-13. bit	8 úrovní levý motor

(Části motivů a akcí tvoří dohromady jedno 256 bitové slovo, zbytek je opět rezerva.)

5.5. Vypracovaní GA

Tato část popisuje stručně jednotlivé složky knihovny genetických algoritmů, jejich definice a popisy jednotlivých genetických operátorů.

5.5.1. Třída genomu

Zde je vytvořeno jádro třídy genomu jako multipotomka ze třídy základního genomu a samotného datového typu:

```
// Definice třídy objektu genomu, vložené statické deklarace definované pro nastavené
// hodnocení, inicializaci, mutaci, srovnání metod této třídy genomu
class MyGenome : public MyObject, public GAGenome {
public: GADefineIdentity("MyGenome", 201);
   static void Init(GAGenome&);
   static int Mutate(GAGenome&, float);
   static float Compare(const GAGenome&, const GAGenome&);
   static float Evaluate(GAGenome&);
   static int Cross(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
public:
   MyGenome() : GAGenome(Init, Mutate, Compare) {
     evaluator(Evaluate);
     crossover(Cross);
    }
    MyGenome(const MyGenome& orig) {copy(orig); }
     virtual ~MyGenome() {}
 MyGenome& operator=(const GAGenome& orig){ if(&orig != this) copy(orig); return *this;}
 virtual GAGenome* clone(CloneMethod) const {return new MyGenome(*this);}
 virtual void copy(const GAGenome& orig) {
         GAGenome::copy(orig);  // this copies all of the base genome parts
         if(&orig == this) return;
           const GA2DBinaryStringGenome* c =
             DYN_CAST(const GA2DBinaryStringGenome*, &orig);
           if(c) {
             GAGenome::copy(*c);
             GABinaryString::copy(*c);
             nx = c->nx; ny = c->ny;
             minX = c->minX; minY = c->minY;
             maxX = c->maxX; maxY = c->maxY;
           }
};
void
MyGenome::Init(GAGenome&){
         GA2DBinaryStringGenome &child=DYN_CAST(GA2DBinaryStringGenome &, c);
           child.resize(GAGenome::ANY_SIZE, GAGenome::ANY_SIZE);
           child.set(0, 0, child.width(), child.height());
         // vlastní inicializace
         }
```

```
int
MyGenome::Mutate(GAGenome&, float){
        // mutace
        GA2DBinaryStringGenome &child=DYN_CAST(GA2DBinaryStringGenome &, c);
          register int n, m, i, j;
          if(pmut <= 0.0) return(0);

          float nMut = pmut * STA_CAST(float, child.size());
          if(nMut < 1.0){          // testuje každý bit
            nMut = 0;
            for(i=child.width()-1; i>=0; i--){
              for(j=child.height()-1; j>=0; j--){
                  if(GAFlipCoin(pmut)){
                    child.gene(i, j, ((child.gene(i,j) == 0) ? 1 : 0));
                    nMut++;
                  }
              }
            }
          }
          else{                          // provádí pouze u vybraného bitu
            for(n=0; n<nMut; n++){
              m = GARandomInt(0, child.size()-1);
              i = m % child.width();
              j = m / child.width();
              child.gene(i, j, ((child.gene(i,j) == 0) ? 1 : 0));
            }
          }
          return(STA_CAST(int,nMut));
        }
};

float
MyGenome::Compare(const GAGenome&, const GAGenome&){
        // porovnávání
        const GA2DBinaryStringGenome &sis=
            DYN_CAST(const GA2DBinaryStringGenome &, a);
          const GA2DBinaryStringGenome &bro=
            DYN_CAST(const GA2DBinaryStringGenome &, b);
          if(sis.size() != bro.size()) return -1;
          if(sis.size() == 0) return 0;
          float count = 0.0;
          for(int i=sis.width()-1; i>=0; i--)
            for(int j=sis.height()-1; j>=0; j--)
              count += ((sis.gene(i,j) == bro.gene(i,j)) ? 0 : 1);
          return count/sis.size();
        }

float
MyGenome::Evaluate(GAGenome&){
        // ohodnocení
        if(_evaluated == gaFalse || flag == gaTrue){
            GAGenome *This = (GAGenome*)this;
            if(eval){ This->_neval++; This->_score = (*eval)(*This); }
            This->_evaluated = gaTrue;
          }
          return _score;
        }
```

```
int
MyGenome::OnePointCross(const GAGenome& mom, const GAGenome& dad,
          GAGenome* sis, GAGenome* bro){
          // křížení – viz dodatek, pro hlavní text je příliš dlouhá
          }
```

Genetické porovnávání (komparace) není nutně to samé jako operátory boolovského typu (== a !=). Genetický komparátor vrací 0 jestliže dva jedinci jsou shodní, –1 jestliže je komparace z nějakých příčin neúspěšná a reálné číslo větší než 0, které ukazuje stupeň odlišnosti dvou jedinců, pokud nejsou shodní.

Toto je možný základ genotypu nebo fenotypu. Boolenovská komparace, v ostatních případech, udává pouze fakt, zda-li jsou dva jedinci identičtí. Ve více případech, boolenovská komparace může jednoduše volat genetické porovnávání, ale v některých případech to není žádoucí (komparace boolenovského typu jsou volány podstatně častěji, než genetické komparátory, specielně nepotřebujeme-li pracovat s mírou neuspořádanosti populace).

Pro práci s vlastnostmi s knihovnou genetických operátorů musí být dodrženy následující zápisy:

```
MyGenome( -defoltní_argumenty_pro_genetický_konstruktor )
MyGenome(const MyGenome&)
virtual GAGenome* clone(GAGenome::CloneMethod) const
```

Pro přidání důležitých datových proměnných, musí být tyto definice:

```
virtual ~MyGenome()
virtual copy(const GAGenome&)
virtual int equal(const GAGenome&) const
```

Pro povolení zápisu a čtení těchto dat, použít následující definice:

```
virtual int read(istream&)
virtual int write(ostream&) const
```

V případě získání odlišného genomu se nesmí zapomenout *_evaluated flag* pro indikaci, kde se nachází místo genomu, kde tedy došlo ke změně fitness funkce. Tento *flag* se předá do *gaFalse.*

5.5.2. Testování Genomu

K otestování funkce navrženého genomu jsem použil následující algoritmus (viz níže). Tento test je zde pro případ, že by bylo potřeba, z jakéhokoli důvodu, změnit strukturu genomu. Otestuje se tak jeho správnost a tím i potvrzení toho, že bude program správně pracovat. Použití pro odlaďování.

```
int TestGenome() {
  MyGenome genome;    // test na nastaveni konstruktoru (jestliže existuje)
  cout << "genom po vytvoření:\n" << genome << endl;

  genome.initialize(); // test na inicializaci
  cout << "genom po inicializaci\n" << genome << endl;

  genome.mutate();     // test na mutaci
  cout << "genom po mutaci:\n" << genome << endl;

  MyGenome* a = new MyGenome(genome);   // test kopírování konstruktorů
  MyGenome* b = new MyGenome(genome);

  MyGenome* c = genome.clone(GAGenome::CONTENTS);
  cout << "obsah klonu:\n" << *c << "\n";
  MyGenome* d = genome.clone(GAGenome::ATTRIBUTES);
  cout << "atributy klonu:\n" << *d << "\n";

  a->initialize();
  b->initialize();
  cout << "rodice:\n" << *a << "\n" << *b << "\n";

  MyGenome::DefaultCrossover(*a, *b, c, d);   // test na 2 potomky křížením
  cout << "potomci po krizeni:\n" << *c << "\n" << *d << "\n";
  MyGenome::DefaultCrossover(*a, *b, c, 0);   // test na 1 potomka křížením
  cout << " potomek po krizeni:\n" << *c << "\n";
  a->compare(*b);      // test na komparaci

  delete a;
  delete b;
  delete c;
  delete d;

  return 0;
}
```

5.5.3. Mutace genomu

Funkce mutace genomu požaduje dva argumenty:

- genom před mutací
- pravděpodobnost mutace

Přesný význam pravděpodobnosti mutace je uveden v části zabývající se samotným operátorem mutace. Pravděpodobnost mutace lze zvýšit v průběhu výpočtu například z důvodu uváznutí v lokálním extrému (předčasné konvergence). Extrémním zvýšením této hodnoty dojde k obnovení genetického materiálu (rozmístění populace po větší oblasti stavového prostoru) a tím vyváznutím z takovéhoto lokálního extrému.

Spousta genetických algoritmů používají metodu mutace s každým vytvořením nového jedince.

Zde je implementace operátoru mutace pro třídu GA1DbinaryString. Jedná se zde o *flip mutator*, který počítá pravděpodobnost mutace každého bitu řetězce zvlášť.

```
int
GA1DBinStrFlipMutator(GAGenome & c, float pmut)
{
  GA1DBinaryStringGenome &child=(GA1DBinaryStringGenome &)c;
  if(pmut <= 0.0) return(0);

  int nMut=0;
  for(int i=child.length()-1; i>=0; i--){
    if(GAFlipCoin(pmut)){
      child.gene(i, ((child.gene(i) == 0) ? 1 : 0));
      nMut++;
    }
  }
  return nMut;
}
```

5.5.4. Křížení genomu

Význam operátoru křížení je podrobně popsán v kapitole zabývající se evolučními algoritmy. Zde je použito jednobodové křížení s možností jednoho nebo dvou potomků s pevnou délkou řetězce (genomu). Pro jednoduchost pouze jednorozměrný binární řetězec.

```
int
SinglePointCrossover(const GAGenome& p1, const GAGenome& p2, GAGenome* c1,
GAGenome* c2){
  GA1DBinaryStringGenome &mom=(GA1DBinaryStringGenome &)p1;
  GA1DBinaryStringGenome &dad=(GA1DBinaryStringGenome &)p2;

  int n=0;
  unsigned int site = GARandomInt(0, mom.length());
  unsigned int len = mom.length() - site;

  if(c1){
    GA1DBinaryStringGenome &sis=(GA1DBinaryStringGenome &)*c1;
    sis.copy(mom, 0, 0, site);
    sis.copy(dad, site, site, len);
    n++;
  }
  if(c2){
    GA1DBinaryStringGenome &bro=(GA1DBinaryStringGenome &)*c2;
    bro.copy(dad, 0, 0, site);
    bro.copy(mom, site, site, len);
    n++;
  }

  return n;
}
```

5.5.5. Komparace genomů

V této komparační metodě jsem použil *diversity calculations* (výpočet odchylky – různorodost). Funkce vrací reálné číslo 0 (pro naprostou shodu) a číslo větší než nula, kde číslo je přímo úměrné míře odlišnosti dvou jedinců. Hodnotu –1 vrací funkce v případě, že z nějakého důvodu nemohlo dojít k porovnání (výskyt chyby, například řetězce jedinců mají různou délku).

```
float
GA1DBinStrComparator(const GAGenome& a, const GAGenome& b){
  GA1DBinaryStringGenome &sis=(GA1DBinaryStringGenome &)a;
  GA1DBinaryStringGenome &bro=(GA1DBinaryStringGenome &)b;
  if(sis.length() != bro.length()) return -1;
  float count = 0.0;
  for(int i=sis.length()-1; i>=0; i--)
    count += ((sis.gene(i) == bro.gene(i)) ? 0 : 1);
  return count/sis.length();
}
```

5.5.6. Ohodnocení Genomu (Genome Evaluation)

Argumentem této funkce je jeden genom (jeden řetězec). Funkce vrací hodnotu, které představuje míru úspěšnosti jedince.

```
float
Objective(GAGenome& g){
  GARealGenome& genome = (GARealGenome &)g;
  return MAX_GOOD - g.penaly;
}
```

5.5.7. Inicializace populace

Tato metoda se volá při inicializaci celé populace. Zde je její implementace:

```
void
PopInitializer(GAPopulation & p){
  for(int i=0; i<p.size(); i++)
    p.individual(i).initialize();
}
```

5.5.8. Ohodnocení populace (Population Evaluation)

Tato metoda ohodnocuje celou populaci jedinců. Je zde proto, abychom se nemuseli dožadovat vyhodnocení každého jedince samostatně na jeho ohodnocení.

```
void
PopEvaluator(GAPopulation & p){
  for(int i=0; i<p.size(); i++)
    p.individual(i).evaluate();
}
```

5.5.9. Přepočítání projektu (Scaling Scheme)

Funkce transformuje „surové“ ohodnocení (Objective scores) na ohodnocení fitness funkce (fitness scores).

Třída GAScalingScheme je čistě virtuální (abstract) třídou. Pro vytvoření nevirtuální třídy je nezbytná definice funkcí *clone* a *eveluate*. Zde je její implementace:

```
class SigmaTruncationScaling : public GAScalingScheme {
public:
  GADefineIdentity("SigmaTruncationScaling", 286);

  SigmaTruncationScaling(float m=gaDefSigmaTruncationMultiplier) : c(m) {}
  SigmaTruncationScaling(const SigmaTruncationScaling & arg){copy(arg);}
  SigmaTruncationScaling & operator=(const GAScalingScheme & arg)
    { copy(arg); return *this; }
  virtual ~SigmaTruncationScaling() {}
  virtual GAScalingScheme * clone() const
    { return new SigmaTruncationScaling(*this); }
  virtual void evaluate(const GAPopulation & p);
  virtual void copy(const GAScalingScheme & arg){
    if(&arg != this && sameClass(arg)){
      GAScalingScheme::copy(arg);
      c=((SigmaTruncationScaling&)arg).c;
    }
  }
  float multiplier(float fm) { return c=fm; }
  float multiplier() const { return c; }
protected:
  float c;                              // std multiplikativní odchylka
};


void
SigmaTruncationScaling::evaluate(const GAPopulation & p) {
  float f;
  for(int i=0; i<p.size(); i++){
    f = p.individual(i).score() - p.ave() + c * p.dev();
    if(f < 0) f = 0;
    p.individual(i).fitness(f);
  }
}
```

5.5.10. Selekce

Objekt selekce je použit pro výběr jedinců z populace. Před provedením selekce je nezbytné volat metodu *update*. Tento selektor (turnajový selektor) je založen na principu ruletového kola. Vybere podle dva jedince (náhodně – ruletové kolo) a porovná je v turnaji podle hodnoty fitness funkce. Lepší z nich vítězí a stává se základem pro klonování nové generace.

```
class TournamentSelector : public GARouletteWheelSelector {
public:
  GADefineIdentity("TournamentSelector", 255);

  TournamentSelector(int w=GASelectionScheme::FITNESS) :
  GARouletteWheelSelector(w) {}
  TournamentSelector(const TournamentSelector& orig) { copy(orig); }
  TournamentSelector& operator=(const GASelectionScheme& orig)
    { if(&orig != this) copy(orig); return *this; }
  virtual ~TournamentSelector() {}
  virtual GASelectionScheme* clone() const
    { return new TournamentSelector; }
  virtual GAGenome& select() const;
};

GAGenome &
TournamentSelector::select() const {
  int picked=0;
  float cutoff;
  int i, upper, lower;

  cutoff = GARandomFloat();
  lower = 0; upper = pop->size()-1;
  while(upper >= lower){
    i = lower + (upper-lower)/2;
    if(psum[i] > cutoff)
      upper = i-1;
    else
      lower = i+1;
  }
  lower = Min(pop->size()-1, lower);
  lower = Max(0, lower);
  picked = lower;

  cutoff = GARandomFloat();
  lower = 0; upper = pop->size()-1;
  while(upper >= lower){
    i = lower + (upper-lower)/2;
    if(psum[i] > cutoff)
      upper = i-1;
    else
      lower = i+1;
  }
  lower = Min(pop->size()-1, lower);
  lower = Max(0, lower);

  GAPopulation::SortBasis basis =
    (which == FITNESS ? GAPopulation::SCALED : GAPopulation::RAW);
  if(pop->order() == GAPopulation::LOW_IS_BEST){
    if(pop->individual(lower,basis).score() <
       pop->individual(picked,basis).score())
      picked = lower;
  }
  else{
    if(pop->individual(lower,basis).score() >
       pop->individual(picked,basis).score())
      picked = lower;
  }
  return pop->individual(picked,basis);
}
```

5.5.11. Genetický algoritmus

Toto je odvozená třída omezeného páření. V tomto případě se první jedinec pro páření vybírá z množiny jedinců vybraných pro páření a druhý jedinec stejným způsobem ze stejné množiny jedinců, ovšem nesmí se jednat o identického jedince. Pak by páření nemělo smysl.

```
void
RestrictedMatingGA::step()
{
 int i, k;
 for(i=0; i<tmpPop->size()-; i++){
        mom = &(pop->select());
        k=0;
        do { k++; dad = &(pop->select()); }
          while(mom->compare(*dad) < THRESHOLD && k<pop->size());
        stats.numsel += 2;
        if(GAFlipCoin(pCrossover()))
           stats.numcro += (*scross)(*mom, *dad, &tmpPop->individual(i), 0);
        else
              tmpPop->individual(i).copy(*mom);
        stats.nummut += tmpPop->individual(i).mutate(pMutation());
      }
      for(i=0; i<tmpPop->size(); i++)
         pop->add(tmpPop->individual(i));

      pop->evaluate();            // vezme informaci o aktuální populaci pro další krok
      pop->scale();               // přepočítání na správné měříko (scale)

      for(i=0; i<tmpPop->size(); i++)
        pop->destroy(GAPopulation::WORST, GAPopulation::SCALED);
      stats.update(*pop);         // aktualizace stat. dat populace
}
```

5.5.12. Ukončení programu (Termination Function)

Tato funkce slouží k nadefinování, kdy má evoluční algoritmus skončit. V případě, že se má výpočet ukončit, funkce vrací hodnotu *gaTrue*, pokud vrátí hodnotu *gaFalse,* algoritmus bude pokračovat.

Jako ukončovací podmínka může být porovnání podílu průměrného ohodnocení (score) aktuální populace a nejlepšího jedince s hodnotou *desiredRatio* (očekávaný poměr).Pokud je tato hodnota lepší než očekávaný poměr, klonování se ukončí, jinak se pokračuje. Základním předpokladem je to, že ohodnocení populace musí konvergovat do hodnoty nejlepšího jedince.

```
const float desiredRatio = 0.95;    // zastav, když průměr populace je lepší než 95%

GABoolean
GATerminateUponScoreConvergence(GAGeneticAlgorithm & ga){
  if(ga.statistics().current(GAStatistics::Mean) /
     ga.statistics().current(GAStatistics::Maximum) > desiredRatio)
    return gaTrue;
  else
    return gaFalse;
}
```

Jiným způsobem je použití stupně rozmanitosti populace, jako ukončovacího kritéria. Algoritmus skončí pokud je stupeň rozmanitosti menší než stanovíme.

```
const float thresh = 0.01;     // stupeň rozmanitosti populace pro ukončení

GABoolean
StopWhenNoDiversity(GAGeneticAlgorithm & ga){
  if(ga.statistics().current(GAStatistics::Diversity) < thresh)
    return gaTrue;
  else
    return gaFalse;
}
```

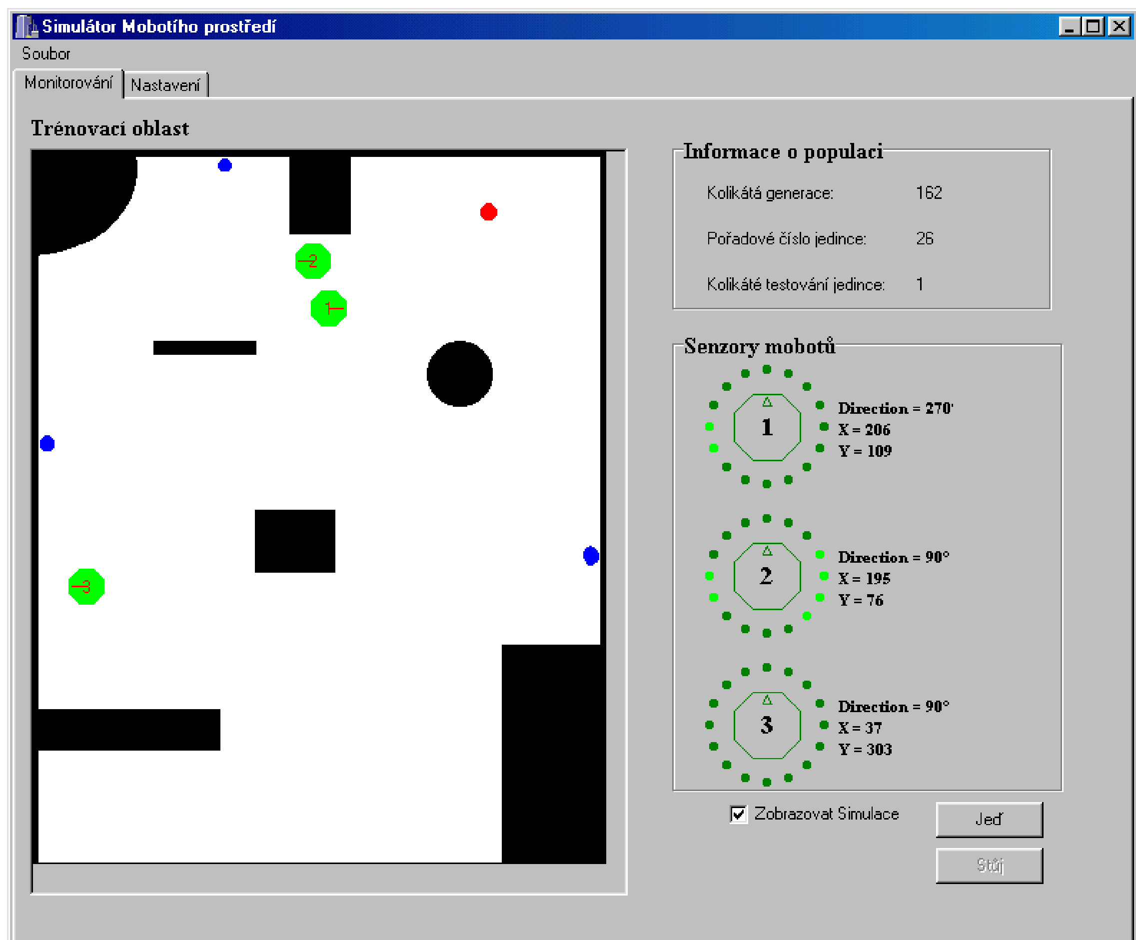
Nejrychlejší metodou (zde použitou) je odchylka populace (deviation). Tato metoda totiž nepotřebuje porovnávat každého jedince.

```
const float thresh = 0.01;     // ukončí, když odchylka populace je menší než toto číslo

GABoolean
StopWhenNoDeviation(GAGeneticAlgorithm & ga){
  if(ga.statistics().current(GAStatistics::Deviation) < thresh)
    return gaTrue;
  else
    return gaFalse;
     }
```

5.6. Popis simulátoru

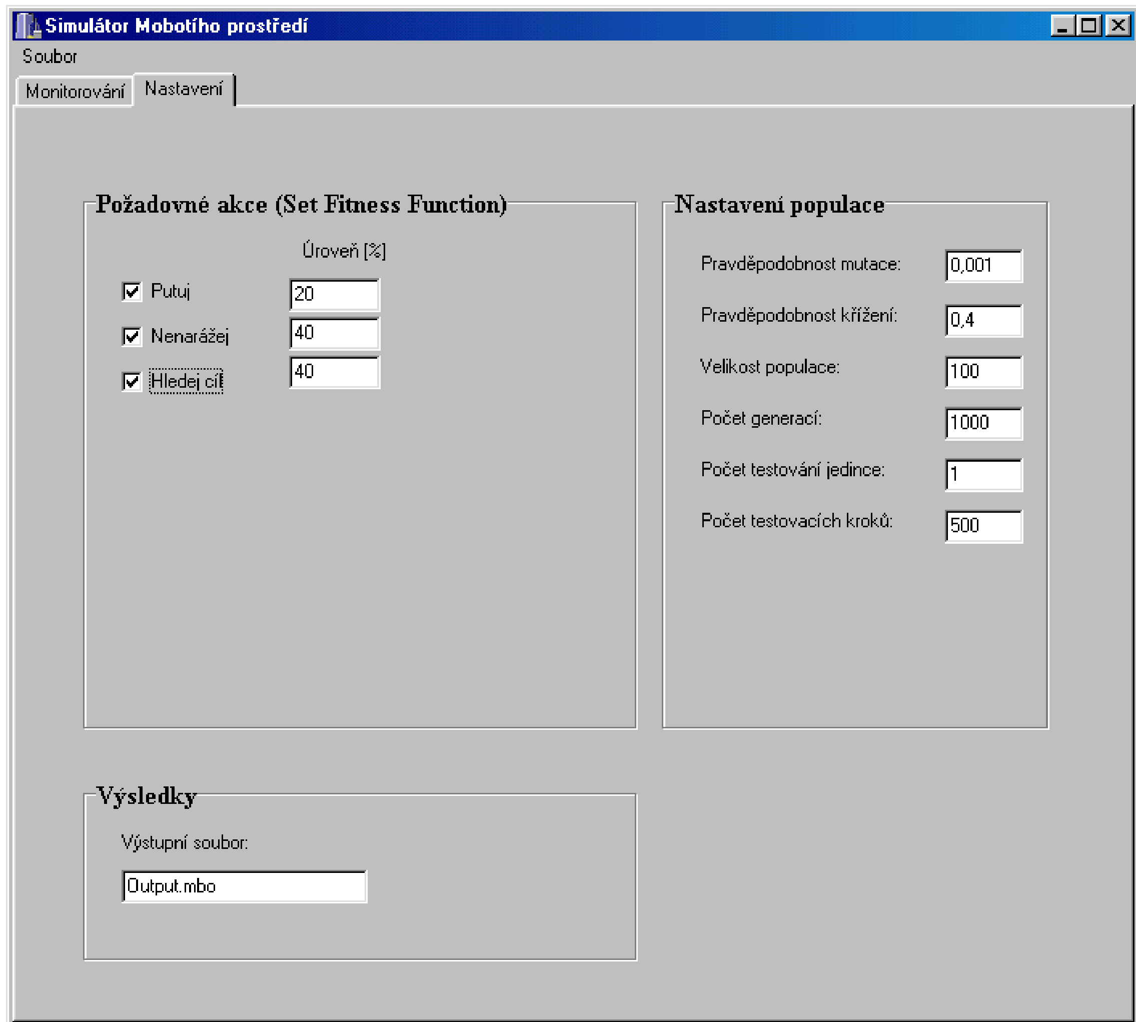
V tomto odstavci se zmíním o samotném simulátoru, krerý slouží pro ohodnocování populace jedinců. Je napsán ve vývojovém prostředí C++ Builderu 4.0 od firmy Borland z důvodu kompatibility v naší mobotické skupině.



Obr. 5.6.: Obrazovka monitorování simulátoru

Na obrázku 5.6. je zobrazeno monitorování mobotího společenství. Tato část simulátoru zobrazuje aktuální půběh testování jedinců. Testovaný jedinec je jedinec s číslem 1. Prostředí ve kterém se moboti pohybují („trénovací oblast“) lze měnit v menu *Soubor/Načti prostředí.* Jde v podstatě o bitmapu, kde volný prostor pro pohyb je označen bílou barvou, modrou barvou jsou označeny zdroje potravy (energie) a červeně cíl. Zelená barva je vyhrazena pro moboty. Všechno ostatní představuje pro mobota obyčejnou překážku. Zobrazovaní tohoto okna lze potlačit přepínačem *„Zobrazovat Simulace“*, který se nachází v pavém dolním rohu obrazovky. (Potlačení je

vhodné pro značné urychlení běhu celého programu, protože je kód optimalizovám především na zpracovávání genetického materiálu.) Dále se zde také nacházejí tlačítka pro spuštění, či pro zastavení simulace. V Pravém horním rohu obrazovky se nalézá pole, které zobrazuje aktuální informace právě testovaném jedinci populace. Další pole (*„Senzory mobotů“*) zobrazuje stav senzorů každého mobota v prostředí. Vnější okruh je tvotřen IR čidly (kolečka), kontury mobota pak dotykové nárazníky. Vedle nich je zobrazována jejich aktuální poloha úhel, určující směr jejich pohybu.



Obr. 5.7.: Obrazovka nastevení simulátoru

Obrazovka *„Nastavení“* (viz obr. 5.7.) slouží k nastavení jednotlivých parametrů simulace. V poli *„Požadované akce“* se zapínají příslušná kritéria pro ohodnocovací fitness funkci. Vedle každé z nich se nachází políčko do kterého se zadáva hodnota představující úroveň (váhu) jednotlivých kritérií (viz vícekriteriální fitness). Pole

„Nastavení populace“ slouží pro nastavení jednotlivých parametrů pro evoluční algoritmy. Jejich význam je popsám výše v textu. Hodnoty v nich na obrázku uvedené jsou experimentálně ověřeny jako nejvhodnější pro tuto úlohu. Aby genetický algoritmus správně fungoval, je potřeba mininálně 100 jedinců a počet iterací 500 (řádově). Pro správné ohodnocení každého jednotlivého mobota (respektive jeho genů) je potřeba nechat jej se v prostředí pohybovat přiměřený počet akcí. Na výpočetně slabších strojích, jako je i ten na které jsem celý program testoval, trvá výpočet desítky hodin. (Konkrétně jsem testoval na Pentium II 233 MHz, 64 MB RAM.) Tento velký nedostatek je způsoben tím, že nelze najít matematický popis hodnotící fitness funkce.

Z důvodů zrychlení algoritmu jsem také opustil od vícenásobného testování jedinců v náhodně zvolených pozicích prostředí. Místo toho se všichni jedinci testují ze stejné výchozí pozice. Kromě tohoto zrychlení se zlepší i reprodukovatelnost experimentů, protože vymizí tento stochastický prvek.

V poli *„Výsledky“* se nastavuje soubor, do kterého má být uložen genetický popis nejlepšího jedince uvedeného na přiložené disketě.

6. Závěr

V této práci byly ověřeny možnosti použití genetických algoritmů, konkrétně jejich využití v mobilní robotice. Při návrhu vlastního řídícího systému založeného na principech etologie jsem se snažil co nejvíce se přiblížit parametrům mobilních robotů existujících v současné době na katedře kybernetiky elektrotechnické fakulty ČVUT v Praze. Z důvodů časové náročnosti a návaznosti na doktoradské práci Ing. Maixnera se nepodařilo převést výsledky této práce na tyto reálné moboty. Proto jsem, na pokyn vedoucího diplomové práce, od tohoto bodu řešení ustoupil. Přesto bych chtěl v práci na této problematice dál pokračovat v rámci doktorandského studia a pokročit tak dál v tomto směru mobilní robotiky.

Dosažené výsledky pomocí genetických algoritmů jsou uspokojivé. Nicméně se během testů ukázaly některé záporné rysy evolučních algoritmů. Je to zejména množství parametrů, které je třeba správně stanovit. Patří mezi ně především problém určení ukončovacího kritéria pro případ, kdy algoritmus dostatečně rychle nekonverguje k nějakému řešení. Toto je ovlivněno volbou fitness funkce, kde je třeba správně stanovit její minimální přípustnou hodnotu. V tomto případě byla fitness funkce představována simulačním prostředím. Tato volba fitness funkce byla nezbytná pro situaci, kterou měla úloha řešit a to neznalost aprior o prostředí jako takovém a o vzájemném vztahu senzorů a prostředí. Východisko vhodnější volby fitness funkce spatřuji v nutném stupni znalostí aspoň o některých vazbách senzor + motivace -> akce. Pak by se dala fitness funkce alespoň částečně matematicky popsat a urychlit tím celý proces hledání vhodné reprezentace reakční báze mobota.

Přesto však spatřuji zavedení evolučních algoritmů pro řešení některých problémů v mobilní robotice za užitečné.

7. Použitá literatura

[1] Lorenz, K.: Základy etologie, Academia, Praha 1993

[2] Petrus, M.: Rozšiřující SW mobotů, vycházející z poznatků etologie a genetiky, Diplomová práce ČVUT FEL, katedra řídící techniky, Praha 1997

[3] Maixner, V.: Etologie mobotích společenství, Diplomová práce ČVUT FEL, katedra řídící techniky, Praha 1996

[4] Zach, J.: Genetické programování a jeho aplikace – Vyhledávání závislosti v datech, Diplomová práce práce ČVUT FEL, katedra řídící techniky, Praha 1999

[5] Mařík, V.,Štěpánková, O., Lažanský, J.: Umělá inteligence 1, Academia, Praha 1993

[6] Gruncl, M.: Rozšiřující SW mobotů I., Diplomová práce ČVUT FEL, katedra řídící techniky, Praha 1995

[7] Heitkötter, J., Beasley, D.: The Hitch-Hicker's Guide to Evolutionary Computation; Issue 6.3, released 30 September 1998; http://www.cs.purdue.edu/coast/archive/clife/FAG/www/

[8] Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley 1989

[9] Watson, James D.: The Double Helix, Weidenfeld and Nicholson, London 1968

Přílohy:

Slovníček základních pojmů evolučních algoritmů

Tento dodatek obsahuje slovníček nejdůležitějších pojmů, které souvisejí s problematikou evolučních algoritmů. Zejména těch, které jsou společné pro všechny typy evolučních algoritmů. Podrobnější přehled termínů týkajících se evolučních algoritmů lze nalézt v [7].

Allele (biol) Konkrétní tvar genu z nějaké oblasti chromozomu (locus). V populaci se pro daný locus může gen vyskytovat ve více tvarech – alleles. Například pro binární reprezentaci genu je každé allele 0 nebo 1.

Building block (EC) Malá, těsně seskupená, skupina společně se vyvíjejících genů. Viz v textu *Bloková hypotéza*.

Chromozome (chromozom) (biol) Jeden z řetězců DNA obsažený v buňce. Chromozom obsahuje jednotlivé geny. V EC je chromozom reprezentován datovou strukturou obsahující jednotlivé geny (např. optimalizované parametry). Chromozom může mít podobu například binárního řetězce, pole celých čísel a pod.

Classifier Systém (klasifikační systém) Systém, jehož výstupy představují klasifikaci vstupů. Např.systém, který má na vstupu senzor měřící teplotu a na výstupu klasifikaci: studená – vlažná – teplá – horká.

Crossever (křížení) (EC) Reprodukční operátor vytvářející dva potomky ze dvou rodičů, chromozomy potomků obsahují části chromozomů obou rodičů. Viz obr. 4.4.

Deception Situace, kdy kombinace dobrých *building blocks* vede na místo ke zvýšení, ke snížení hodnoty fitness funkce, což vede k selhání genetických algoritmů v některých typech úloh. Viz [8].

Diploid (biol) Buňka, která obsahuje kopie každého chromozomu. Tyto kopie obsahují stejné geny ve stejném pořadí. U mnoha sexuálně se reprodukujících druhů (např. druh sapiens) je část genů zděděna z gamety (gamete) otce (spermatu - sperm), zatímco zbylá část genů pochází z gamety matky (vajíčka – ovum).

DNA (biol) Deoxyribonukleová kyselina, makromolekula tvořená dvěmi vlákny šroubovice. Abecedu pro kódování genetických informací tvoří čtyři nukleotidové báze – Adenin, Thymin, Cytosin a Guanine. [9]

Elitism (elitářství) Mechanismus používaný v některých EA. Zaručuje, že chromozomy nejlepších jedinců se dostanou do nové populace bez modifikace genetickými operátory, což zajišťuje neklesající maximální hodnotu fitness funkce populace. Tím dochází ke zrychlení konvergence.

Enviroment (prostředí) (biol) Vše co obklopuje jedince a nutí jej k adaptaci. Prostředí může být abiotické („fyzické") nebo biotické. V obou případech organismus obývá niku (niche). Nika bezprostředně působí na výkonnost jedince (hodnota fitness funkce).

Epistasis (biol) Gen je označen jako epistatický, pokud jeho výskyt potlačuje některého genu v jiném *locusu*. (EC) V EC je *epistasis* chápána jako jakákoliv silná iterace mezi geny. Pokud pro daný problém dochází k silné *epistasis*, nemohou se formovat *building blocks* a dochází k *deception*.

Evolution (evoluce) Jedná se o „neřízený" a „necílený" proces. Nelze například říci, že cílem přírodní evoluce je lidstvo, protože výsledkem je náhodná generace biologicky odlišných organismů. Evoluce je podmíněna přirozenou selekcí a konkurencí jedinců v populaci o zdroje v daném prostředí. Ti nejlepší mají větší šanci na přežití a šíření svého genetického materiálu.

Evolutionary Strategy (evoluční strategie) Typ EA vyvinutý na počátku 60. let v Německu. Evoluční strategie kódují parametry jako reálná čísla.

Evolution Computation (EC) Metody simulace evoluce s využitím počítače.

Evolutionary Programing (evoluční programování) EA vyvinutý na počátku 60. let, podobný genetickým algoritmům, od nichž se liší reprezentací genomu a v genetických operátorech.

Exploitation Proces prohledávání stavového prostoru, snažící se prohledat všechny dosud neprozkoumané body. Pro některé úlohy s mnoha lokálními extrémy může být tato technika, v kombinaci s některým s algoritmů náhodného prohledávání, jediným použitelným řešením.

Fitness (biol) Míra adaptivity, relativní úspěch reprodukce. (EC) Míra ohodnocující, jak jedinec dobře „řeší" daný problém. Fitness funkce je pak zobrazení chromozomu na množinu reálných čísel. Zobrazením všech chromozomů do stavového

prostoru vzniká hyperplocha, na které hledáme extrém. Nejlepší představuje řešení problému.

Gamete (gameta) (biol) Buňka nesoucí genetickou informaci jejího rodiče určenou pro sexuální reprodukci. U živočichů se samčí gameta nazývá sperma (sperm), samičí vajíčko (ova).

Gene (gen) (biol) Základní jednotka dědičnosti zahrnující segment DNA, který kóduje jednu nebo několik vztažených funkcí a zaujímá pevnou pozici (*locus*) v chromozomu. (EC) Část chromozomu obvykle kóduje jeden parametr.

Generation (generace) (EC) Jedna iterace založená na měření hodnot fitness funkce a vytvoření populace pomocí reprodukčních operátorů.

Genetic Algorithm (genetický algoritmus) Typ EA. Chromozomy jsou typicky reprezentované binárními řetězci pevné délky.

Genetic Programing (genetické programování) Typ EA podobný genetickým algoritmům, liší se datovou reprezentací, výsledkem je program (deterministický stavový automat).

Genetic Operator (genetický operátor) Operátor prohledávání stavového prostoru aplikovaný na genotyp jedince.

Genotype (genotyp) Genetická stavba jedince (organizmu) je informace obsažená v genomu.

Genome (genom) Soubor všech genů (a tedy i chromozomů), které patří příslušnému organizmu.

Hamploid (biol) Buňka obsahující jednotlivé (nezdvojené) chromozomy. Viz *diploid*.

Individual (jedinec, organizmus) Jeden člen populace obsahující chromozomy a reprezentující možné řešení problému. Jeden bod stavového prostoru.

Migration (migrace) Přemístění jedince (genů) z jedné subpopulace do druhé. Jedna z technik paralelního zpracování EA.

Mutation (mutace) Reprodukční operátor modifikující chromozom na nový změnou hodnoty genu rodičovského chromozomu.

Niche (nika) (biol) V přírodních ekosystémech existuje mnoho způsobů, umožňujících organizmu přežít. Každý takový způsob (strategie) se nazývá ekologická *nika* (ecological niche). Jednotlivé druhy, žijící v různých nikách mohou existovat vedle sebe bez vzájemného soutěžení. (EC) V EC je rozmanitost jedinců v rámci populace

vítána, protože je žádoucí prozkoumat všechny možné extrémy hodnot fitness funkce. Tyto extrémy jsou analogické nikám. Techniky jako *fitness sharing* pak umožňují prevenci před konvergencí do jednotlivých extrémů.

Offspring (potomek) Jedinec generovaný procesem reprodukce.

Parent (rodič) Jedinec účastnící se reprodukce při generování nových jedinců (offspring).

Phenotype Vlastnosti jedince.

Population (populace) Skupina jedinců schopných společné interakce (produkce potomstva apod.)

Reproduction (reprodukce) (biol, EC) Vytvoření nového jedince ze dvou rodičů (sexuální reprodukce) – potomek vznikne kombinací genetické informace obou rodičů, nebo pouze informacemi z jednoho rodiče (asexuální reprodukce), pak je potomek identický se svým rodičem. Asexuálně se v přírodě rozmnožuje velké množství druhů např. bakterií, které patří k jedněm z nejúspěšnějším druhům na Zemi.

Subpopulation (subpopulace) Populace může být rozdělena do skupin (subpopulací). Reprodukce je pak možná pouze v rámci dané subpopulace. Někteří jedinci mohou mezi subpopulacemi migrovat. Tato technika je užívána při paralelním zpracování EA.

Popis genetických tříd

V následující části jsou popsány jednotlivé třídy genetických algoritmů, aby byla orientace v textu snazší, pokusil jsem se o jejich přehledný výpis.

Globální definice typů a množin

typedef float **GAProbability, GAProb**
typedef enum **_GABoolean** {gaFalse, gaTrue} **GABoolean**, **GABool**
typedef enum **_GAStatus** {gaSuccess, gaFailure} **GAStatus**
typedef unsigned char **GABit**

Prototypové funkce

GABoolean (***GAGeneticAlgorithm::Terminator**)(GAGeneticAlgorithm & ga)
GAGenome& (***GAIncrementalGA::ReplacementFunction**)(GAGenome &, GAPopulation &)

void (***GAPopulation::Initializer**)(GAPopulation &)
void (***GAPopulation::Evaluator**)(GAPopulation &)

void (***GAGenome::Initializer**)(GAGenome &)
float (***GAGenome::Evaluator**)(GAGenome &)
int (***GAGenome::Mutator**)(GAGenome &, float)
float (***GAGenome::Comparator**)(const GAGenome &, const GAGenome&)
int (***GAGenome::SexualCrossover**)(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*)
int (***GAGenome::AsexualCrossover**)(const GAGenome&, GAGenome*)

int (***GABinaryEncoder**)(float& value, GABit* bits, unsigned int nbits, float min, float max)
int (***GABinaryDecoder**)(float& value, const GABit* bits, unsigned int nbits, float min, float max)

Globální proměnné a konstanty

```
char* gaErrMsg;                      // globálně definovaný ukazatel na aktual.
                                     // chybovou zprávu
int  gaDefScoreFrequency1 = 1;       // nepřekrývající se populace
int  gaDefScoreFrequency2 = 100;     // překrývající se populace

float gaDefLinearScalingMultiplier   = 1.2;
float gaDefSigmaTruncationMultiplier = 2.0;
float gaDefPowerScalingFactor        = 1.0005;
float gaDefSharingCutoff             = 1.0;
```

GAGeneticAlgorithm

Toto je popis třídy genetických algoritmů. Tvoří jádro celého programu a proto jej zde pro úplnost uvádím.

Struktura třídy

class GAGeneticAlgorithm : public GAID

Definice typů a konstant

GABoolean (***GAGeneticAlgorithm::Terminator**)(GAGeneticAlgorithm &)
enum { **MINIMIZE** = -1, **MAXIMIZE** = 1 };

Funkce

static GAParameterList& **registerDefaultParameters**(GAParameterList&);

void * **userData**()
void * **userData**(void *)

void **initialize**(unsigned int seed=0)
void **evolve**(unsigned int seed=0)
void **step**()
GABoolean **done**()

GAGeneticAlgorithm::Terminator **terminator**()
GAGeneticAlgorithm::Terminator **terminator**(GAGeneticAlgorithm::Terminator)

const GAStatistics & **statistics**() const
float **convergence**() const
int **generation**() const
void **flushScores**()

int **minimaxi**() const
int **minimaxi**(int)
int **minimize**()
int **maximize**()
int **nGenerations**() const
int **nGenerations**(unsigned int)
int **nConvergence**() const
int **nConvergence**(unsigned int)
float **pConvergence**() const
float **pConvergence**(float)
float **pMutation**() const
float **pMutation**(float)
float **pCrossover**() const
float **pCrossover**(float)

GAGenome::SexualCrossover **crossover**(GAGenome::SexualCrossover func);
GAGenome::SexualCrossover **sexual**() const;
GAGenome::AsexualCrossover **crossover**(GAGenome::AsexualCrossover func);
GAGenome::AsexualCrossover **asexual**() const;

const GAPopulation & **population**() const

```
const GAPopulation & population(const GAPopulation&)
        int populationSize() const
        int populationSize(unsigned int n)
        int nBestGenomes() const
        int nBestGenomes(unsigned int n)

  GAScalingScheme & scaling() const
  GAScalingScheme & scaling(const GAScalingScheme&)
 GASelectionScheme & selector() const
 GASelectionScheme & selector(const GASelectionScheme& s)

        void objectiveFunction(GAGenome::Evaluator)
        void objectiveData(const GAEvalData&)

         int scoreFrequency() const
         int scoreFrequency(unsigned int frequency)
         int flushFrequency() const
         int flushFrequency(unsigned int frequency)
        char* scoreFilename() const
        char* scoreFilename(const char *filename)
         int selectScores() const
         int selectScores(GAStatistics::ScoreID which)
      GABoolean recordDiversity() const
      GABoolean recordDiversity(GABoolean flag)

 const GAParameterList & parameters()
 const GAParameterList & parameters(const GAParameterList &)
 const GAParameterList & parameters(int& argc, char** argv, GABoolean flag =
gaFalse)
 const GAParameterList & parameters(const char* filename, GABoolean flag =
gaFalse);
 const GAParameterList & parameters(istream&, GABoolean flag = gaFalse);

         int set(const char* s, int v)
         int set(const char* s, unsigned int v)
         int set(const char* s, char v)
         int set(const char* s, const char* v)
         int set(const char* s, const void* v)
         int set(const char* s, double v);

         int write(const char* filename)
         int write(ostream&)
         int read(const char* filename)
         int read(ostream&)
```

Popis jednotlivých funkcí

convergence

Vrací hodnotu aktuální konvergence. Konvergence je definována jako míra nejlepšího jedince v aktuální (prozatím nejlepší) populaci.

crossover

Specifikace křížení v EA. Používám zde pouze sexuální křížení (potřeba dvou jedinců k páření).

done

Tato funkce vrací hodnotu gaTrue, jestliže bylo splněno ukončovací kritérium, jinak vrací.

evolve

Tato funkce spouští genetický algoritmus. Nejprve se spustí inicializační funkce (*initialize*) a pak se provádí postupně funkce *step* dokud není ukončovací podmínka *done* nevrátí hodnotu gaTrue.

generation

Tato fukce vrací aktuální generaci.

initialize

Inicializace genetického algoritmu. Vytvoří první generaci pomocí náhodné funkce GARandomSeed.

nBestGenomes

Specifikuje kolik nejlepších jedinců se má pamatovat v historii. Standardně nastavena na hodnotu 1.

nConvergence

Nastavuje/vrací číslo generace, ve které se má provést test na konvergenci.

nGenerations

Nastavuje/vrací číslo generace.

objectiveData

Nastavuje data všem jedinců v průběhu funkce *evolution.*

objectiveFunction

Nastavuje data všem jedinců v průběhu funkce *evolution.* Využívá funkci *objectiveData.*

parameters

Vrací parametry obsahující aktualní hodnoty parametrů genetického algoritmu.

pConvergence

Nastavuje/vrací procenta konvergence. Konvergence je definována jako poměr *n* předcházejících generací (respektive jejich nejlepšího ohodnocení) a ohodnocení nejlepšího jedince aktuální generace. Hodnota *n* je definována pomocí funkce *nConvergence.*

pCrossover

Nastavuje/vrací pravděpodobnost křížení.

pMutation

Nastavuje/vrací pravděpodobnost mutace.

population

Nastavení populace jako celku. Vrací hodnoty aktuální na populaci.

populationSize

Nastavuje/vrací velikost populace.

recordDiversity

Vhodnostní funkce specifikuje zda-li se má počítat různorodost jedinců. Slouží pro obnovu genetického materiálu.

registerDefaultParameters

Deklaruje parametry genetického algoritmu se kterými pracuje. Jsou to staticky definované funkce, užívají jméno třídy gentických algoritmů (například GASimpleGA::registerDefaultParameters(list). Jsou přednastaveny parametry: flushFrequency, minimaxi, nBestGenomes, nGenerations, nConvergence, pConvergence, pCrossover, pMutation, populationSize, recordDiversity, selectScores.

scaling

Nastavuje/vrací měřítko schematu. Musí být specifikováno měřítko schematu pro třídu GAScalingScheme. Měřítko může být měněno během výpočtu mezi jednotlivými populacemi.

selector

Nastavuje/vrací selekční schema genetického algoritmu. Selektor používá výběr jedinců před samotným křížením a mutací.

set

Nastavuje jednotlivé parametry genetického algoritmu. Argumentem je vždy jméno proměnné, která se má nastavovat. Druhým parametrem je hodnota na kterou se má nastavit.

statistics

Vrací odkaz do objektu *statistics* v genetickém algoritmu. Statistický objekt vrací hodnoty jako je: informace o nejlepším jedinci, nejhorším, průměrným, standardní odchylku a různorodost populace.

step

Jeden běh genetického algoritmu. Jinými slovy, vypočítá další populaci.

terminator

Nastavuje ukončovací funkce. Genetický algoritmus je ukončen, když ukončovací funkce vrátí hodnotu gaTrue.

userData

Nastavuje uživatelská data v genetickém algoritmu. Vlastně vytváří ukazatel na informace, které si je potřeba z nějakého důvodu zapamatovat.

GAGenome

Tento genom je virtuální základ třídy. Definuje hodnoty konstant a specifikaci prototypů funkcí této třídy. Je použit jako základ pro vícerozměrné genomy.

Genetické operátory genomu jsou funkcemi, které generují genomy jako jejich argumenty.

Struktura třídy

```
class GAGenome : public GAID
```

Definice a konstanty

```
enum GAGenome::Dimension { LENGTH, WIDTH, HEIGHT, DEPTH }
enum GAGenome::CloneMethod { CONTENTS, ATTRIBUTES }
enum { FIXED_SIZE = -1, ANY_SIZE = -10 }
   float (*GAGenome::Evaluator)(GAGenome &)
    void (*GAGenome::Initializer)(GAGenome &)
     int (*GAGenome::Mutator)(GAGenome &, float)
   float (*GAGenome::Comparator)(const GAGenome &, const GAGenome&)
     int (*GAGenome::SexualCrossover)(const GAGenome&, const GAGenome&,
GAGenome*, GAGenome*);
     int (*GAGenome::AsexualCrossover)(const GAGenome&, GAGenome*);
```

Funkce

```
virtual void copy(const GAGenome & c)
virtual GAGenome * clone(CloneMethod flag = CONTENTS)

        float score(GABoolean flag = gaFalse)
        float score(float s)
          int nevals()

        float evaluate(GABoolean flag = gaFalse)
GAGenome::Evaluator evaluator() const
GAGenome::Evaluator evaluator(GAGenome::Evaluator func)

         void initialize()
```

```
GAGenomeInitializer initializer() const
GAGenomeInitializer initializer(GAGenome::Initializer func)

int mutate(float pmutation)
GAGenome::Mutator mutator() const
GAGenome::Mutator mutator(GAGenome::Mutator func)

float compare(const GAGenome& g) const
GAGenome::Comparator comparator() const
GAGenome::Comparator comparator(GAGenome::Comparator c)

GAGenome::SexualCrossover crossover(GAGenome::SexualCrossover func)
GAGenome::SexualCrossover sexual()
GAGenome::AsexualCrossover crossover(GAGenome::AsexualCrossover func)
GAGenome::AsexualCrossover asexual()

GAGeneticAlgorithm * geneticAlgorithm() const
GAGeneticAlgorithm * geneticAlgorithm(GAGeneticAlgorithm &)
void * userData() const
void * userData(void * data)
GAEvalData * evalData() const
GAEvalData * evalData(void * data)

virtual int read(istream &)
virtual int write(ostream &) const

virtual int equal(const GAGenome &) const
virtual int notequal(const GAGenome &) const
```

Tyto operátory korespondují s virtuální proměnými pro práci se třídou genomu

```
friend int operator==(const GAGenome&, const GAGenome&)
friend int operator!=(const GAGenome&, const GAGenome&)
friend ostream & operator<<(ostream&, const GAGenome&)
friend istream & operator>>(istream&, GAGenome&)
```

Popis jednotlivých funkcí

clone

Tato metoda alokuje místo pro nový genom, který patří dané třídě.

compare

Porovnávání dvou genomů. Komparace vrací hodnotu větší nebo rovno 0.0, kde 0.0 znamená, že jsou dva genomy identické. Přesný význam tohoto porovnávání je uvedeno výše v textu.

comparator

Nastavuje metodu pro porovnávání.

copy

Kopíruje genom na již alokované místo genomu stejného typu. Používá operátor =

crossover

Třída genomu má definovanou metodu pro křížení.

evalData

Nastavuje data pro evoluci. Každý genom evoluce je podroben klonování, křížení a mutaci.

evaluate

Využívá genů evolučních funkcí. Vytváří novou populaci jedinců.

evaluator

Set/Get funkce používaná pro vytvoření (evaluate) nové populace.

initialize

Inicializační funkce jednoho genomu.

mutate

Funkce, která provádí mutaci genomu.

nevals

Vrací hodnotu hodnotící funkce bezprostředně po inicializaci genomu (nuluje ji).

score

Vrací hodnotu hodnotící funkce (po simulaci).

GABinaryString

Tento objekt binárního řetězce je jednoduchou implementací bitového řetězce. Každý bit je v paměti reprezentován jako jednoduchým slovem. Třída binárního řetězce definuje následující funkce (řetězec má možnost měnit svoji délku).

Konstruktory

GABinaryString(unsigned int length)
GABinaryString(const GABinaryString&)

Funkce

void **copy**(const GABinaryString&)
int **resize**(unsigned int)
int **size**() const

short **bit**(unsigned int a) const
short **bit**(unsigned int a, short val)

```
int equal(const GABinaryString& b,
        unsigned int dest, unsigned int src, unsigned int length) const
void copy(const GABinaryString& orig,
        unsigned int dest, unsigned int src, unsigned int length)
void move(unsigned int dest, unsigned int src, unsigned int length)
void set(unsigned int a, unsigned int length)
void unset(unsigned int a, unsigned int length)
void randomize(unsigned int a, unsigned int length)
```

Popis jednotlivých funkcí

copy

Kopíruje vybraná binární řetězec. Jestliže je volána s parametrem, který představuje vymezení oblasti bitů pro kopírování, zkopíruje se pouze tato oblast bitů.

bit

Nastavuje/vrací specifický bit, pozici v žetězci.

equal

Vrací hodnotu 1, pokud jsou specifikované části řetězců shodné ve všech bitech, jinak vrací hodnotu 0.

move

Přenáší počet bitů od *src* do *dest*.

set/unset

Nastavuje/ruší nastavení paramtru *a* (počáteční délka řetězce, pro funkci *randomize*).

size

resize

Navtavuje/vrací délku binárního řetězce.

randomize

Nastavuje náhodně délku řetězce. Začíná na délce udané parametrem pokud je nastavený *a*.

GA1DBinaryStringGenome

Jednorozměrný binární řetězec je odvozen od tříd GABinaryString a GAGenome. Může pracovat jak s pevnou délkou řetězce, tak i s pohyblivou délkou řetězce.

Informace jsou uložené pomocí bitů, kde každý bit představuje allelu s hodnotou 0 nebo 1.

Struktura třídy

```
class GA1DBinaryStringGenome : public GABinaryString, public GAGenome
```

Konstruktory

```
GA1DBinaryStringGenome(unsigned int x,
                       GAGenome::Evaluator objective = NULL,
                       void * userData = NULL)
GA1DBinaryStringGenome(const GA1DBinaryStringGenome&)
```

Funkce

```
short gene(unsigned int x = 0) const
short gene(unsigned int, short value)
  int length() const
  int length(int l)
  int resize(int x)
  int resizeBehaviour() const
  int resizeBehaviour(unsigned int minx, unsigned int maxx)
 void copy(const GA1DBinaryStringGenome &,
          unsigned int xdest, unsigned int xsrc, unsigned int length)
 void set(unsigned int x, unsigned int length)
 void unset(unsigned int x, unsigned int length)
```

Popis jednotlivých funkcí

copy

Kopíruje vybrané bity z určeného genomu.

gene

Nastavuje specifický bit.

length

Nastavuje délku binárního řetězce.

resize

Změna délky binárního řetězce.

Genetické operátory

GA1DBinaryStringGenome::UniformInitializer
GA1DBinaryStringGenome::SetInitializer
GA1DBinaryStringGenome::UnsetInitializer
GA1DBinaryStringGenome::FlipMutator
GA1DBinaryStringGenome::BitComparator
GA1DBinaryStringGenome::UniformCrossover
GA1DBinaryStringGenome::EvenOddCrossover
GA1DBinaryStringGenome::OnePointCrossover
GA1DBinaryStringGenome::TwoPointCrossover

Implicitně nastavené genetické operátory

initialization: GA1DBinaryStringGenome::UniformInitializer
comparison: GA1DBinaryStringGenome::BitComparator
mutation: GA1DBinaryStringGenome::FlipMutator
crossover: GA1DBinaryStringGenome::OnePointCrossover

GA2DBinaryStringGenome

Jednorozměrný binární řetězec je odvozen od tříd GABinaryString a GAGenome. Může pracovat jak s pevnou délkou řetězce, tak i s pohyblivou délkou řetězce. Obdobně jako je tomu u GA1DbinaryStringGenome.

Informace jsou uložené pomocí bitů, kde každý bit představuje allelu s hodnotou 0 nebo 1.

Struktura třídy

class GA2DBinaryStringGenome : public GABinaryString, public GAGenome

Konstruktory

GA2DBinaryStringGenome(unsigned int x, unsigned int y,
GAGenome::Evaluator objective = *NULL*,
void * userData = *NULL*)
GA2DBinaryStringGenome(const GA2DBinaryStringGenome &)

Funkce

short **gene**(unsigned int x, unsigned int y) const
short **gene**(unsigned int x, unsigned int y, const short value)
int **width**() const
int **width**(int w)
int **height**() const
int **height**(int h)
int **resize**(int x, int y)
int **resizeBehaviour**(GADimension which) const
int **resizeBehaviour**(GADimension which,

```
                unsigned int min, unsigned int max)
  int resizeBehaviour(unsigned int minx, unsigned int maxx,
                unsigned int miny, unsigned int maxy)
 void copy(const GA2DBinaryStringGenome &,
        unsigned int xdest, unsigned int ydest,
        unsigned int xsrc, unsigned int ysrc,
        unsigned int width, unsigned int height)
 void set(unsigned int, unsigned int, unsigned int, unsigned int)
 void unset(unsigned int, unsigned int, unsigned int, unsigned int)
```

Popis jednotlivých funkcí

Většina funkcí je stejná jako u jednorozměrného binárního řetězce. Uvedu jen důležité funkce, které jsou doplněny navíc.

height

Nastavuje výšku binárního řetězce.

width

Nastavuje délku binárního řetězce.

Gentické operátory této třídy

```
GA2DBinaryStringGenome::UniformInitializer
GA2DBinaryStringGenome::SetInitializer
GA2DBinaryStringGenome::UnsetInitializer
GA2DBinaryStringGenome::FlipMutator
GA2DBinaryStringGenome::BitComparator
GA2DBinaryStringGenome::UniformCrossover
GA2DBinaryStringGenome::EvenOddCrossover
GA2DBinaryStringGenome::OnePointCrossover
```

Implicitně nastavené genetické operátory

```
initialization:  GA2DBinaryStringGenome::UniformInitializer
    comparison:  GA2DBinaryStringGenome::BitComparator
      mutation:  GA2DBinaryStringGenome::FlipMutator
     crossover:  GA2DBinaryStringGenome::OnePointCrossover
```

GAStatistics

Tato třída je použita pro statistické funkce, které jsou potřeba pro práci s populací, potažmo s populacemi.

Definice typů a konstant

enum { **NoScores**,
Mean, **Maximum**, **Minimum**, **Deviation**,
Diversity,
AllScores }

Konstruktory

GAStatistics()
GAStatistics(const GAStatistics&)

Funkce

void **copy**(const GAStatistics &);

float **online**() const
float **offlineMax**() const
float **offlineMin**() const
float **initial**(ScoreID w=Maximum) const
float **current**(ScoreID w=Maximum) const
float **maxEver**() const
float **minEver**() const

int **generation**() const
float **convergence**() const
int **selections**() const
int **crossovers**() const
int **mutations**() const
int **replacements**() const

int **nConvergence**(unsigned int)
int **nConvergence**() const
int **nBestGenomes**(const GAGenome&, unsigned int)
int **nBestGenomes**() const
int **scoreFrequency**(unsigned int x)
int **scoreFrequency**() const

int **selectScores**(int whichScores)
int **selectScores**() const

void **update**(const GAPopulation& pop)
void **reset**(const GAPopulation& pop)

const GAPopulation& **bestPopulation**() const
const GAGenome& **bestIndividual**(unsigned int n=0) const

int **scores**(const char* filename, ScoreID which=NoScores)
int **scores**(ostream& os, ScoreID which=NoScores)
int **write**(const char* filename) const

```
int write(ostream& os) const;
friend ostream& operator<<(ostream&, const GAStatistics&)
```

Popis jednotlivých funkcí

bestIndividual

Vrací hodnoty nejlepšího jedince populace.

bestPopulation

Vrací parametry nejlepší populace.

convergence

Vrací aktuální hodnotu, ke které populace konverguje. Počítá se jako poměr *n* posledních nejlepších jedinců (z každé populace jeden nejlepší) k hodnotě nejlepšího jedince aktuální populace.

crossovers

Vrací počet křížení od inicializace.

current

Vrací nejlepší hodnotu aktuální populace.

generation

Vrací pořadové číslo aktuální populace.

maxEver

Vrací historicky nejlepší ohodnocení.

minEver

Vrací historicky nejhorší ohodnocení.

mutations

Vrací počet mutací od inicializace.

nBestGenomes

Nastavuje počet nejlepších jedinců, kteří mají být zapamatováni.

nConvergence

Nastavuje počet populací na které má být aplikován konvergenční test.

offlineMax

Vrací průměrnou hodnotu z maximálních ohodnocení nastaveného počtu populací.

offlineMin

Vrací průměrnou hodnotu z minimálních ohodnocení nastaveného počtu populací.

online

Vrací průměrnou hodnotu ohodnocení všech populací.

recordDiversity

Tato logická (0 nebo 1) proměnná určuje, zda-li se má čí nemá počítat různorodost populace každé generace. Implicitně nastavena na False.

replacements

vrácí počet změn od počátku (inicializace).

reset

Resetování (vynulování) všech statistických proměnných a zahrne jako první pouze informace o aktuální populaci.

selections

Vrací počet selekcí od počátku (inicializace).

update

Obnoví obsah všech statistických objektů s ohledem na aktuální populaci.

Jednobodové křížení

```
GA2DBinaryStringGenome::
OnePointCrossover(const GAGenome& p1, const GAGenome& p2,
          GAGenome* c1, GAGenome* c2){
 const GA2DBinaryStringGenome &mom=
  DYN_CAST(const GA2DBinaryStringGenome &, p1);
 const GA2DBinaryStringGenome &dad=
  DYN_CAST(const GA2DBinaryStringGenome &, p2);

 int nc=0;
 unsigned int momsitex, momlenx, momsitey, momleny;
 unsigned int dadsitex, dadlenx, dadsitey, dadleny;
 unsigned int sitex, lenx, sitey, leny;

 if(c1 && c2){
  GA2DBinaryStringGenome &sis=DYN_CAST(GA2DBinaryStringGenome &, *c1);
  GA2DBinaryStringGenome &bro=DYN_CAST(GA2DBinaryStringGenome &, *c2);

  if(sis.resizeBehaviour(GAGenome::WIDTH) == GAGenome::FIXED_SIZE &&
    bro.resizeBehaviour(GAGenome::WIDTH) == GAGenome::FIXED_SIZE){
   if(mom.width() != dad.width() ||
  sis.width() != bro.width() ||
  sis.width() != mom.width()){
 GAErr(GA_LOC, mom.className(), "one-point cross", gaErrSameLengthReqd);
 return nc;
   }
   sitex = momsitex = dadsitex = GARandomInt(0, mom.width());
   lenx = momlenx = dadlenx = mom.width() - momsitex;
  }
  else if(sis.resizeBehaviour(GAGenome::WIDTH) == GAGenome::FIXED_SIZE ||
    bro.resizeBehaviour(GAGenome::WIDTH) == GAGenome::FIXED_SIZE){
   GAErr(GA_LOC, mom.className(), "one-point cross", gaErrSameBehavReqd);
   return nc;
  }
  else{
   momsitex = GARandomInt(0, mom.width());
```

```
    dadsitex = GARandomInt(0, dad.width());
    momlenx = mom.width() - momsitex;
    dadlenx = dad.width() - dadsitex;
    sitex = GAMin(momsitex, dadsitex);
    lenx = GAMin(momlenx, dadlenx);
  }

  if(sis.resizeBehaviour(GAGenome::HEIGHT) == GAGenome::FIXED_SIZE &&
     bro.resizeBehaviour(GAGenome::HEIGHT) == GAGenome::FIXED_SIZE){
    if(mom.height() != dad.height() ||
  sis.height() != bro.height() ||
  sis.height() != mom.height()){
 GAErr(GA_LOC, mom.className(), "one-point cross", gaErrSameLengthReqd);
 return nc;
    }
    sitey = momsitey = dadsitey = GARandomInt(0, mom.height());
    leny = momleny = dadleny = mom.height() - momsitey;
  }
  else if(sis.resizeBehaviour(GAGenome::HEIGHT) == GAGenome::FIXED_SIZE ||
     bro.resizeBehaviour(GAGenome::HEIGHT) == GAGenome::FIXED_SIZE){
    GAErr(GA_LOC, mom.className(), "one-point cross", gaErrSameBehavReqd);
    return nc;
  }
  else{
    momsitey = GARandomInt(0, mom.height());
    dadsitey = GARandomInt(0, dad.height());
    momleny = mom.height() - momsitey;
    dadleny = dad.height() - dadsitey;
    sitey = GAMin(momsitey, dadsitey);
    leny = GAMin(momleny, dadleny);
  }

  sis.resize(sitex+lenx, sitey+leny);
  bro.resize(sitex+lenx, sitey+leny);

  sis.copy(mom, 0, 0, momsitex-sitex, momsitey-sitey, sitex, sitey);
  sis.copy(dad, sitex, 0, dadsitex, dadsitey-sitey, lenx, sitey);
  sis.copy(dad, 0, sitey, dadsitex-sitex, dadsitey, sitex, leny);
  sis.copy(mom, sitex, sitey, momsitex, momsitey, lenx, leny);
```

```
  bro.copy(dad, 0, 0, dadsitex-sitex, dadsitey-sitey, sitex, sitey);
  bro.copy(mom, sitex, 0, momsitex, momsitey-sitey, lenx, sitey);
  bro.copy(mom, 0, sitey, momsitex-sitex, momsitey, sitex, leny);
  bro.copy(dad, sitex, sitey, dadsitex, dadsitey, lenx, leny);

  nc = 2;
}
else if(c1 || c2){
  GA2DBinaryStringGenome &sis = (c1 ?
                                  DYN_CAST(GA2DBinaryStringGenome&, *c1) :
                                  DYN_CAST(GA2DBinaryStringGenome&, *c2));

  if(sis.resizeBehaviour(GAGenome::WIDTH) == GAGenome::FIXED_SIZE){
    if(mom.width() != dad.width() || sis.width() != mom.width()){
GAErr(GA_LOC, mom.className(), "one-point cross", gaErrSameLengthReqd);
return nc;
    }
    sitex = momsitex = dadsitex = GARandomInt(0, mom.width());
    lenx = momlenx = dadlenx = mom.width() - momsitex;
  }
  else{
    momsitex = GARandomInt(0, mom.width());
    dadsitex = GARandomInt(0, dad.width());
    momlenx = mom.width() - momsitex;
    dadlenx = dad.width() - dadsitex;
    sitex = GAMin(momsitex, dadsitex);
    lenx = GAMin(momlenx, dadlenx);
  }

  if(sis.resizeBehaviour(GAGenome::HEIGHT) == GAGenome::FIXED_SIZE){
    if(mom.height() != dad.height() || sis.height() != mom.height()){
GAErr(GA_LOC, mom.className(), "one-point cross", gaErrSameLengthReqd);
return nc;
    }
    sitey = momsitey = dadsitey = GARandomInt(0, mom.height());
    leny = momleny = dadleny = mom.height() - momsitey;
  }
  else{
    momsitey = GARandomInt(0, mom.height());
    dadsitey = GARandomInt(0, dad.height());
```

```
      momleny = mom.height() - momsitey;
      dadleny = dad.height() - dadsitey;
      sitey = GAMin(momsitey, dadsitey);
      leny = GAMin(momleny, dadleny);
    }

    sis.resize(sitex+lenx, sitey+leny);

    if(GARandomBit()){
      sis.copy(mom, 0, 0, momsitex-sitex, momsitey-sitey, sitex, sitey);
      sis.copy(dad, sitex, 0, dadsitex, dadsitey-sitey, lenx, sitey);
      sis.copy(dad, 0, sitey, dadsitex-sitex, dadsitey, sitex, leny);
      sis.copy(mom, sitex, sitey, momsitex, momsitey, lenx, leny);
    }
    else{
      sis.copy(dad, 0, 0, dadsitex-sitex, dadsitey-sitey, sitex, sitey);
      sis.copy(mom, sitex, 0, momsitex, momsitey-sitey, lenx, sitey);
      sis.copy(mom, 0, sitey, momsitex-sitex, momsitey, sitex, leny);
      sis.copy(dad, sitex, sitey, dadsitex, dadsitey, lenx, leny);
    }

    nc = 1;
  }

  return nc;
}
```