ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta dopravní

Ústav aplikované informatiky v dopravě

Nový algoritmus pro symbolickou regresi pomocí genetického programování s upřesňováním číselné hodnoty koeficientů

Ing. Vladimír Hlaváč

Praha, 2017

Studijní program: Inženýrská informatika

Specializace: Inženýrská informatika v dopravě a spojích

Školitel:
doc. Dr. Ing. Tomáš Brandejský
Ústav aplikované informatiky v dopravě

Fakulta dopravní
České vysoké učení technické v Praze
Konviktská 20
110 00 Praha 1
Česká republika

Prohlašuji, že jsem tuto práci vypracoval samostatně, s použitím literatury a zdrojů, které jsou v textu práce a v seznamu zdrojů uvedeny.

V Praze dne Vladimír Hlaváč

Název práce: **Nový algoritmus pro symbolickou regresi pomocí genetického programování s upřesňováním číselné hodnoty koeficientů**

Autor: Ing. Vladimír Hlaváč
Praha, 2017

Školitel: doc. Dr. Ing. Tomáš Brandejský

Pracoviště: Ústav aplikované informatiky v dopravě

Fakulta dopravní

České vysoké učení technické v Praze

Konviktská 20

110 00 Praha 1

Česká republika

Studijní program: Inženýrská informatika
Specializace: Inženýrská informatika v dopravě a spojích

Anotace

Práce se zabývá aplikací genetického programování pro řešení symbolické regrese, kde je třeba kromě tvaru funkce najít i vhodné koeficienty v minimálním potřebném, ale dostatečném počtu. Kombinací metod genetického programování a exponencionované gradientní metody (exponencionated gradient descent) je dosaženo vysoké robustnosti při zachování přijatelné rychlosti běhu programu. Pro zrychlení konvergence je nově navržena mutace spočívající v odříznutí kořene stromu zápisu funkce.

Dále je popsán nový přístup k vyčíslovaným konstantám symbolické regrese, které jsou nově zaznamenávány značkami přímo do prefixového zápisu stromu funkce v rámci genetického programování a chápány jako násobící koeficienty, které jsou pro každého jedince zaznamenány v odděleném poli s například maximálně deseti hodnotami. Program se k upřesnění koeficientů pro každého jedince vrací po každém cyklu genetického programování. Oba cykly se tak prolínají a tím umožňují jak zrychlit běh programu, tak odstraňovat z populace funkce, které využívají koeficientů v nadbytečném počtu. V práci jsou popsána i další opatření pro řešení problémů lokálního minima.

Součástí práce je vytvoření aplikace pro symbolickou regresi funkce o dvou proměnných, umožňující jak zápis statistických hodnot o průběhu evoluce do souboru, tak 2D a 3D zobrazení každé jednotlivé navržené funkce, tak i průběžné přenastavování parametrů a opětovné spouštění evolučního cyklu genetického programování.

Klíčová slova: Genetické algoritmy, Genetické programování, Gradientní metody, Exponencionované gradientní metody, Symbolická regrese, Vyčíslování konstant, Evoluční výpočetní metody, Dopravní tok, Hustota provozu, Indukční smyčka.

Title: **New algorithm for symbolic regression using genetic programming with adjustment of values of coefficients**

Author: Ing. Vladimír Hlaváč
Prague, 2017

Tutor: doc. Dr. Ing. Tomáš Brandejský

Department of Applied Informatics in Transportation

Faculty of Transportation Sciences

Czech Technical University in Prague

Konviktská str. 20

110 00 Prague 1

Czech Republic

Abstract

The work describes the application of genetic programming for symbolic regression, where it is necessary to find a function and its coefficients. There should be a minimum, but necessary amount of coefficients. A combination of genetic programming (GP) and exponencionated gradient descent (EGD) method provides high robustness while maintaining an acceptable speed of the program. To speed up the convergence, a newly designed mutation has been introduced - the cut-off of the root of the function tree.

A new approach of the recording of the constants into the prefix record of the function tree using the 8^{th} bit of symbols (otherwise, the ASCII code is used) is presented. Constants itself are represented as multiplicative (coefficients) in the marked points. It is a good representation of a physical quantity concept as a multiplication of a unit and a value. Constants for each of generated functions (the individual) is kept in a separate array, and in each of the GP cycle, only a limited number of the EGD cycles are executed, so the exact value of coefficients will be evaluated after more GP cycles. This inter-mixed cycle approach allows not only speed up the main GP cycle, but simplifies rejection of individual with unnecessary coefficients included (more complex function is less efficient for most of the applications of the function). The work also describes other methods to solve the local minimum problem.

As a part of this work, new program for symbolic regression has been written. The program includes comprehensive user interface allowing setting up many GP and EGD parameters, including variable EGD step (the simulated annealing methodology), fitness evaluation method, function (genome) size penalization, etc. The program can write statistical values to a .csv file, so graphs of the GP progress can be generated. 3D and 2D graphs of the source data and generated function are available, so as a generation of the function for the MS Excel and as a C++ function (text to file/clipboard).

Keywords: Genetic algorithms, Genetic programming, Gradient descent, Exponentionated gradient descent, Symbolic regression, Evaluation of the constants, Evolutionary computing, Traffic flow, Traffic density, Induction loop.

Obsah

Seznam použitých zkratek a symbolů

ASCII	American Standard Code for Information Interchange, základní znaková sada PC
CSV	Comma-separated values, textový soubor s daty oddělenými čárkami
DNA	Deoxyribonukleová kyselina
EGD	Exponencionovaný Gradient Descent
ES	Evoluční strategie
ftp	file transfer protocol, způsob komunikace se serverem
GA	Genetické algoritmy
GD	Gradient Descent, gradientní metoda hledání optima
GP	Genetické programování
GPA-ES	Genetické programování v kombinaci s evolučními strategiemi
GPA-GD	Genetické programování v kombinaci s gradientní metodou
MLP	multi-layer perceptron, vícevrstvá neuronová síť
n	počet (jedinců v populaci i vzorků v souboru naměřených dat)
NP	nondeterministic polynomial, nedeterministicky polynomiální
PC	Personal computer, osobní počítač, zde zejména stolní kancelářský
PWM	Pulse Width Modulation, pulzně šířková modulace
q	evoluční tlak
RAD	rapid application development
RNA	Ribonukleová kyselina
T2	turnajová selekce - vítěz je použit jako zdroj pro nového jedince a poražený je odstraněn
T2+M	turnajová selekce - vítěz je použit jako zdroj pro nového jedince, který je vytvořen mutací a nahradí poraženého
T4	dvě turnajové selekce, poražení jsou nahrazeni potomky vítězných
T4+C	dvě turnajové selekce a křížení, poražení jsou nahrazeni potomky vítězných získaných křížením
VC dimenze	Vapnik–Chervonenkis dimension, míra komplexnosti funkce

(většina symbolů je popsána v místě výskytu a dále se v práci neopakují)

1. Úvod

Nejen v oblasti dopravní telematiky, ale i v řadě dalších technických odvětví je užitečné hledat funkční závislosti z naměřených dat. V mnoha případech nám heuristické vzorce mohou pomoci navrhnout řešení, které se pak přesnějšími metodami ověří (například metodou konečných prvků). Jinde je třeba modelovat procesy a bylo by užitečné je mít popsány jednoduchou funkcí. Pro problémy modelování systémů se dvěma proměnnými jsou k dispozici zejména neuronové sítě. Další zajímavou metodou je proložení polynomem, což pro více proměnných provádí v Matlabu nadstavbová funkce *polyfitn* (dostupná https://www.mathworks.com/matlabcentral/fileexchange/34765-polyfitn, 3.9.2016). Obě metody jsou přijatelné v řadě případů, zejména, pokud můžeme zůstat v prostředí Matlabu. Pokud se mají nalezené funkční závislosti používat v jiných programech, je model získaný neuronovou sítí obtížně přenositelný. V oblasti simulování dopravních toků pak je dalším problémem, že procesy jsou často periodické. Funkce, které používají obě výše uvedené metody, z principu periodický model vytvořit nemohou, a řešením pak je používat nějakou vnější nadstavbu pro získaný model, využívající apriorních znalostí, nebo vhodným způsobem předzpracovávat data. Zcela nové možnosti pro hledání závislostí nabízí symbolická regrese, reprezentovaná zejména metodou genetického programování. Úkolem symbolické regrese je zjistit nejen tvar hledané funkce (symbolický zápis), ale také hodnoty konstant a koeficientů, pokud se v zápise funkční závislosti použijí.

V rámci předkládané práce byl vytvořen program pro realizaci vyhledání funkce včetně jejích koeficientů (symbolické regrese) s využitím metod genetického programování. Program samotný byl vytvořen v prostředí Lazarus (http://www.lazarus-ide.org/). Jedná se o open-source prostředí pro Free Pascal. Lazarus představuje typické RAD (rapid application development) prostředí pro rychlou tvorbu aplikací. Hlavní výhodou je, že Pascal je samodokumentovatelný, zápis jazyka je velmi přehledný a programy jdou velmi snadno ladit (odstraňovat chyby). Oproti Borland Delphi je možné programy překládat i v prostředí Linux (a údajně i pro Mac OS, což nebylo možné ověřit). Přesvědčil jsem se, že přeložené programy jsou srovnatelně rychlé, jako v Delphi.

Pro Pascal není k dispozici takové množství hotových knihoven, jako pro C++. V tomto případě byly všechny potřebné komponenty napsány přímo v Pascalu (bez využití cizích knihoven), právě díky jeho jednoduchosti. Je ale možné vzít přeloženou knihovnu v objektovém tvaru a hlavičkový soubor přepsat do Pascalu. Tím je možné mít části programu v jiném programovacím jazyku, pokud se dají dohledat hotové.

Pascal umožňuje velmi pohodlnou práci s řetězci, pokud programy používají starý typ řetězce, vytvořený pro Turbo Pascal. Toho je využito v této práci pro reprezentaci hledané funkce i jejího čitelného zápisu. Z toho ovšem vyplývá omezení na délku řetězce 255 znaků, které navíc s ohledem na vnitřní algoritmy programu nelze využít všechny.

Jako součást této práce jsou zahrnuty i výukové programy, které jsem vytvořil v průběhu studia této problematiky. Jsou určeny jednak pro výuku genetického programování v rámci magisterského kurzu umělé inteligence, jednak byly použity pro hledání vhodných nastavení genetických algoritmů. Právě nastavení parametrů evoluce je v literatuře obtížně dohledatelné a většina zdrojů vychází ze zkušeností pouze s některými kombinacemi. Například často je uváděno, že turnajová selekce by měla být srovnatelná s ruletovou, díky tomu, že chybějící možnost ovládání evolučního tlaku (evolution press control) turnajová selekce kompenzuje možností udržování větší populace. Experimenty získané v této části dokumentují, že při srovnatelných podmínkách je ruletová selekce až čtyřikrát rychlejší a

přitom je odolnější oproti uváznutí v lokálním minimu. Tato nastavení by na úplném algoritmu genetického programování, kdy jeden průběh evoluce trvá minimálně hodiny, nešlo s rozumnou spolehlivostí ověřovat.

Jako návody k použití všech předkládaných programů slouží prozatím tato práce. Zdrojové soubory všech předkládaných aplikací jsou na přiloženém CD, zdrojové kódy jsou podrobně komentované (zejména tam, kde se jedná o implementaci algoritmů GA/GP/EGD). Funkce jednotlivých ovládacích prvků jsou navíc při běhu programu vysvětleny krátkou nápovědou ve žlutém poli, která se objevuje po najetí myší na prvek (může obsahovat i několik řádek textu).

Do seznamu literatury je odkazováno textově (zdroje nejsou číslovány). Tento způsob má výhodu pro čitelnost textu, případný zájemce si nemusí pamatovat, co znamená které číslo. U prací více autorů je v několika odkazech využito označení zkratkou názvu publikace. Případné číslo za čárkou označuje, na kterou stránku se reference odkazuje. Kromě uvedených zdrojů jsem v některých případech používal pro orientaci v problematice (například kódování DNA, 3D promítání apod.) Wikipedii (www.wikipedia.org), reference jsou přímo v textu práce. V seznamu použité literatury také chybí přednáškový kurz Algoritmů na MIT (MIT Course Number 6.046J / 18.410J, Prof. Charles Leiserson, Prof. Erik Demaine, Massachusetts Institute of Technology, Cambridge, MA.), který bych doporučil jako dobrou on-line orientaci v problematice algoritmizace.

Zájemcům o problematiku genetického programování bych doporučil publikaci “Riccardo Poli, William B. Langdon,Nicholas F. McPhee, John R. Koza, "A Field Guide to Genetic Programming".“ [AField], která je dobře strukturovaná a navíc dostupná on-line. Existuje také velice rozsáhlý archiv na adrese http://www.genetic-programming.com/, spravovaný přímo profesorem Kozou a jeho spolupracovníky. V současné době již bohužel není aktualizován.

1.1 Cíle práce

- Prostudovat algoritmy, používané pro symbolickou regresi metodou genetického programování
- Navrhnout možnosti, jak dané algoritmy zrychlit
- Vytvořit aplikaci, vycházející z těchto algoritmů
- Demonstrovat zpracování dopravních dat (pro testování jsou k dispozici intenzity a obsazenosti z křižovatek na ulici Zborovská v Praze 5) pomocí uvedených metod, tedy vyhledávání závislostí mezi jednotlivými veličinami

2. Genetické algoritmy

Genetické algoritmy jsou heuristická výpočetní metoda, snažící se najít řešení (v prostoru řešení) pomocí metod, analogických k principům evoluční genetiky (citace: [Mitchell]).

Mezi základní principy patří křížení, mutace a zvýhodnění nejlepších jedinců v průběhu evoluce. Důležitá je také diverzita populace (udržování šíře genomu, v chovatelství tomu odpovídá problém úzké příbuzenské plemenitby). Oproti přirozenému výběru můžeme šíři populace, míru zvýhodnění jedinců při vytváření potomků a počet mutací řídit podle okamžité situace. Proces probíhá cyklicky a jeho průběh se označuje jako evoluce.

Genotyp – fenotyp

Jedinec je obecně popsán genomem. V některých případech mohou být genomy jedinců porovnávány přímo, většinou se ale jedná jen o popis, jak jedince vytvořit. Jedinec, vytvořený na základě genotypu, se označuje jako fenotyp. Při porovnávání kvality jedinců pak porovnáváme vlastnosti fenotypů, a na základě tohoto srovnání používáme odpovídající genotyp. Podrobněji viz následující podkapitola.

Účelová funkce (fitness)

Účelová funkce popisuje, jak moc nalezený jedinec odpovídá našim požadavkům. Pokud se provádí konverze genotyp → fenotyp, pak je samozřejmě vyhodnocován vytvořený jedinec (fenotyp).

Účelová funkce zpravidla vrací reálné číslo jako jediné kritérium, ale pro evoluční výpočetní techniky to není nezbytné. Postačí existence funkce, která umožní porovnat, který ze dvou jedinců je lepší (proto se tato tzv. *účelová funkce* někdy také označuje jako kriteriární).

Pokud evoluční algoritmus zahrnuje třídění populace, pak možnost vyjádřit hodnotu účelové funkce číselně znamená, že tuto funkci musíme v průběhu třídění vyčíslit jednou pro každého jedince. Pokud funkce umožňuje pouze porovnání, obecně potřebujeme O(n.log n) těchto porovnání (minimální počet porovnání při třídění, pokud o souboru tříděných hodnot nemáme apriorní informace).

Funkce přípustnosti (feasibility)

Geny jsou většinou tvořeny abecedou symbolů (popis může zahrnovat reálná čísla, proto nemusí být úloha diskrétní). Pokud některé kombinace symbolů či hodnot jsou nepřípustné, je třeba vytvořit funkci přípustnosti, a u každého nově vytvořeného jedince kontrolovat, zda ji splňuje. Příkladem je případ, kdy prostor hledání je jednoznačně omezen danými podmínkami. Představujeme si je často tak, že hledáme řešení v neobdélníkové oblasti, ale souřadnice jsou Euklidovské; pokud je vygenerován jedinec mimo oblast, je vyřazen. Jiným důvodem může být vyřazení jedince, jehož genotyp nedává smysl, resp. nelze jej použít pro vytvoření fenotypu, například proto, že danou mutací nebo metodou křížení byl vygenerován symbol, který není ve slovníku (například binární mutace místo písmena ASCII vygeneruje řídicí znak). V současné době převažuje spíše tendence používat takové metody mutace a křížení, které nemohou vygenerovat neplatného (invalid) jedince.

U funkce přípustnosti (feasibility) se jedná o tzv. tvrdé omezení. Nově generovaný (mutací, křížením) jedinec ovšem může mít vlastnosti, o které by bylo škoda přijít. Proto je v případech, kdy to lze, někdy přímé vyřazení nahrazováno výrazným zvýšením (snížením, hledáme-li maximum) hodnoty účelové funkce (fitness), což se označuje jako pokutová funkce (penalty). Pokud se vrátíme k příkladu s neobdélníkovou oblastí, pak může

penalizace zohledňovat vzdálenost od oblasti, kde hledáme řešení. Pokud účelová funkce vrací číslo, je k němu často výsledek penalizační funkce připočítáván (odčítán, hledáme-li maximum). Může se stát, že i po penalizaci jedinec má natolik výhodné vlastnosti, že jej přesto mezi populací řešení ponecháme, a tím zachováme jeho genetický přínos. Je naděje, že v další generaci vzniknou jedinci, kteří některé výhodné vlastnosti převezmou, ale vygenerují se v oblasti, kde je řešení přípustné (feasible). Tvrdým omezením bychom o ně přišli. Při ukončení vývoje je však třeba zkontrolovat, zda nalezené řešení leží v přípustné oblasti, nebo zda je porušení této podmínky natolik výhodné, že je třeba od ní upustit. Příklad – při sestavování rozvrhu je striktní požadavek, že někteří vyučující nemohou být na dvou místech současně. Některá kombinace rozvrhu ale může být tak výhodná, že se vyplatí tuto podmínku obejít například přijetím externisty pro výuku v danou hodinu.

Ani pokud účelová funkce vrací hodnotu, zpravidla nevíme, jaká je hledaná (nejlepší dosažitelná) hodnota této funkce. V řadě technických problémů to ovšem není třeba. Například při řešení střihového plánu pro oplechování křídla letadla z ekonomického hlediska stačí, pokud se nalezená skladba bude nejlepšímu možnému řešení blížit. Obdobné je řešení problému obchodního cestujícího. Stejně nevíme, zda matematický popis reálného problému je přesný, takže se stačí hledanému optimu nákladů na cestování jen co nejvíce přiblížit. U rozvrhu by optimální řešení, kdy všichni vyučující učí jen dopoledne a co nejméně dnů v týdnu, sice mohlo být zřejmé, ale takový rozvrh zřejmě nebude splňovat podmínky přípustnosti, například na ČVUT by narazil na kapacity místností, zejména laboratoří.

Hledání řešení

Myšlence genetických algoritmů odpovídá úvaha, že hledaný jedinec bude nejvíce podobný těm jedincům, kteří nejlépe vyhovují účelové funkci. Z matematického hlediska hledáme extrém poblíže míst, kde se při předchozích pokusech ukázaly být dosud nejextrémnější hodnoty (navzdory názvu hledáme častěji minimum účelové funkce, maximum je méně obvyklé).

Genetické algoritmy pracují s populací, kdy populaci představuje řada možných řešení.

Řešení se opakovaně kombinují, pozměňují a vyhodnocují. Kombinacemi částí dvou různých jedinců (označovanými jako křížení) zpravidla přibývají další možná řešení. Stejně tak při mutacích (pozměňování genomu cílenou změnou) můžeme uchovávat původního i nového jedince. Při následném vyhodnocení pomocí účelové funkce pak ponecháme jen x nejlepších jedinců a cyklus opakujeme. Jinou možností je generační přístup, kdy ponecháme jen nově vytvořené jedince a staré zahodíme. Jednotlivým krokům se samozřejmě budeme věnovat podrobněji.

Selekce

Selekce (výběr) slouží ke zvýhodnění těch řešení, která lépe vyhovují účelové funkci (fitness). Může být pozitivní (pro křížení se vybírají lepší jedinci) i negativní (z populace jsou vyřazováni horší jedinci), nejčastěji se používá kombinace obojího. Další důležitou funkcí selekce bývá udržení přípustné velikosti populace. Podrobněji viz dále.

Křížení

Křížením vytvoříme z genomu dvou jedinců nového jedince. Tomu odpovídá představa, že hledané řešení bude obsahovat především prvky těch řešení, které mají nejlepší hodnoty účelové funkce. Při představě n-rozměrného prostoru řešení tomu odpovídá představa vyhledávání lepšího řešení poblíž míst, kde zatím byla nalezena dobrá řešení.

Pokud pro tvorbu genomu existují nějaká omezení (pevný počet znaků, nutnost přítomnosti nějakého genu), pak se snažíme genom spojovat tak, aby tato omezení byla stále

splněna. Samozřejmostí je na závěr provést kontrolu funkce přípustnosti (feasibility, viz dále). Nepřípustní noví jedinci nejsou do populace zařazeni.

V některých případech je jednodušší provést nějaké (jakékoli) křížení a prostě ověřit funkci přípustnosti.

Při křížení se výchozí jedinci (genotypy, fenotypy) často označují jako rodiče a noví jedinci jako potomci, nebo po vzoru angličtiny děti (child, children, nebo offspring, potomci). Méně obvyklé je označení syn/dcera.

Symetrické křížení

Zpravidla lze, pokud křížení vytvoří přípustného jedince, z nepoužitých částí obou genomů vytvořit dalšího jedince, který se po kontrole funkce přípustnosti nechá také zařadit do populace. Při tomto vytvoření dvou jedinců mluvíme o symetrickém křížení.

Jednobodové křížení

U jednobodového křížení postupujeme tak, že do genomu potomka přepisujeme část (začátek) genomu jednoho z rodičů, a v jistý okamžik začneme přepisovat genom druhého z rodičů, od nějakého bodu až do konce. Nevyužité geny lze samozřejmě přepsat do jiného potomka, tím vznikne jednobodové symetrické třídění. Pokud to, co přepisujeme do jednoho potomka a co do druhého změníme v průběhu přepisu vícekrát, jedná se o **vícebodové křížení**. Není třeba složité úvahy k dovození, že místo vícebodového stačí vícekrát aplikovat jednobodové křížení, takže použití jednobodového křížení neubírá metodě na obecnosti. To platí za předpokladu, že potomci z prvních křížení budou evolucí ihned odstraněni; například u problémů s charakteristikou „xor“ musí dojít ke dvěma záměnám současně, aby se algoritmus dostal do jiného lokálního minima (maxima).

Mutace

Pro udržení diverzity populace a z důvodů, aby se lépe (úplněji) prohledala oblast, kde se vyskytují nejlepší řešení, je občas nutné nějakým způsobem zkusit pozměnit genom i jinak, než jen křížením z dostupných hodnot. V reálné genetice tomu odpovídá náhodná záměna jednotlivých genů, např. při ozáření, chemickém a virovém poškození apod., často v průběhu kopírování. U genetických algoritmů bývají změny i ve větším rozsahu, často dochází i k pravidelnému dogenerování zcela nových jedinců. Jedním z problémů bývá, že mutované jedince odstraní selekční tlak dříve, než se uplatní při křížení.

Nově vytvářené jedince zpravidla umísťujeme na konec populace, kterou pravidelně zmenšujeme vypouštěním nejhorších jedinců. U mutací ale přichází v úvahu i jejich umístění místo původních jedinců (před mutací).

Generační přístup

Řada aplikací genetických a evolučních technik obecně používá generační postup. Zpravidla podle následujícího schematu:

1. vytvoříme náhodnou populaci
2. seřadíme podle účelové funkce
3. vyřadíme nejhorší jedince
4. populaci doplníme o jedince, vytvořené z existujících pomocí křížení a mutací tak, že ty nejlepší z existujících použijeme jako zdrojové s největší pravděpodobností

Body 2 až 4 představují jednu generaci a v algoritmu je opakujeme. Z principu věci tato metoda (na rozdíl od například neuronových sítí) neposkytuje žádné vodítko k tomu, kdy cyklus ukončit.

Stará/nová generace

Často se používá modifikace generačního principu, kdy se pomocí křížení a mutací vytvoří stejný počet jedinců, jako je v populaci, a původní jedinci se následně smažou.

V kombinaci, kdy část populace zůstává (x nejlepších), mluvíme o stabilizační části populace.

Elitismus

Abychom zachovali alespoň nejlepší zatím nalezené řešení, často je uchováváme (nemažeme je automaticky se starší generací a neměníme je mutacemi). V přírodě tento princip má obdobu jen u evoluce bakterií, které se množí dělením, u pohlavně orientované evoluce jej nenajdeme. Ale jedná se o velmi logické opatření.

Zachovávat několik nejlepších řešení zpravidla nemá smysl, jak si později ukážeme u praktické realizace genetického programování. Často se totiž jedná o v principu stejného jedince, jen například s permutovaným zápisem (tam, kde pořadí buněk není podstatné), nebo s rozepsanými funkcemi ($2x$ nebo $x+x$ je stejné).

Ruletová selekce

V praxi bychom mohli navázat pravděpodobnost výběru jedince na hodnotu poměru účelové funkce jeho a nejlepšího z populace (poměrem či odečtením, pokud může nabývat kladných i záporných hodnot), a podle některých publikací se toto řešení občas i používá [UI3, str. 123]. V praxi se ale ukazuje, že pro zachování diverzity populace je vhodnější, pokud jedince seřadíme a pravděpodobnost výběru jim přiřadíme podle nějaké rozumné posloupnosti (v praxi se častěji používá geometrická než aritmetická). Původní autor metody tomu přiřadil představu, která nemá s ruletou (hra) nic společného – že různým jedincům je přiřazen různě široký výsek ruletového kola a kdo má větší, tomu připadne větší pravděpodobnost výběru. Ruletová selekce tedy předpokládá, že účelová funkce vrací reálné číslo, resp. že není problém ji zavolat tolikrát, že lze provést setřídění (pro velké soubory může být i $n.\log(n)$ velké číslo).

Turnajová selekce

Z populace se vyberou náhodně dva jedinci a jen tito dva se porovnají. Lepší se použije pro další vývoj, horší se z populace odstraní.

Pokud chceme vytvořit další jedince symetrickém křížením, můžeme vybrat dvě dvojice, v každé provést turnajovou selekci, tím dva jedince z populace odstranit a místo nich umístit potomky, získané křížením těch vítězných jedinců. Zpravidla se používá v případech, kdy původní prvky nelze smazat, abychom si neodstranili potenciální nejlepší řešení (bez třídění nelze zavést elitismus).

Alternativou je možnost, že křížením dvou jedinců vytvoříme nového jedince, a toho turnajovou selekcí porovnáme s jedním z rodičů, nebo dokonce s úplně jiným jedincem. Pokud potomek uspěje, zařadíme jej na toto místo do populace, jinak akci zrušíme (cancel). Řešení z předchozího odstavce je obvyklejší, protože potomek málokdy uspěje; zabráněním vytváření potomků by mohlo dojít k uváznutí (neboli zamrznutí) populace.

Evoluční bezgenerační model

Pomocí předchozího postupu můžeme náhodně provádět křížení a mutace, aniž by došlo ke zvětšení/zmenšení populace, nebo k vytvoření čehokoli, co by šlo označit jako generace. Nechají se aplikovat i další principy, například pokud jsou si jedinci v populaci příliš podobní, lze provádět více mutací a méně křížení. Protože genomy se obtížně porovnávají, považuje se za „uváznutí“ evoluce stav, kdy jsou si příliš blízké hodnoty účelové funkce (fitness). V tomto modelu pak dochází k porovnání hodnot účelové funkce jen u jedinců, srovnávaných při turnajové selekci. Teprve na úplný závěr se provede výběr nejlepšího

jedince (hledání nejmenšího prvku v souboru vyžaduje jen (*n*-1) porovnání, jedná se tedy o úlohu náročnosti O(*n*)).

Takto koncipovaná metoda vypadá teoreticky velmi zajímavě, ale při testování na programech, které jsem vytvořil pro výukové účely, jsem zjistil, že je řádově pomalejší, než generační model s ruletovou selekcí (tři až pětkrát). V mnou dále popisovaných příkladech ovšem účelová funkce vrací vždy číslo (často dokonce celé) a není tedy problém prvky třídit.

Shrnutí.

Genetické algoritmy můžeme použít, pokud:

- hledané řešení lze popsat pomocí konečné sady symbolů, které případně mohou být použity v kombinaci s (reálnými) čísly, *parametry* (pak mluvíme o n-rozměrném prostoru hledání řešení),
- hledané řešení se při změně parametrů či výměně symbolů musí chovat alespoň většinou rozumně; tento požadavek neověřujeme, pokud neplatí, poznáme to podle selhání metody (úlohy, které nejsou řešitelné metodou GA, lze zpravidla řešit jen hrubou silou, vyčíslením všech kombinací) ,
- míru správnosti náhodně navrženého řešení lze posoudit na základě *účelové (kriteriární, evaluační) funkce*, která umožňuje porovnat, které řešení je lepší/horší (nemusí vracet spojitý=číselný výsledek, stačí rozhodnutí lepší/horší, nebo i je lepší/není lepší).

Genetické algoritmy v tomto případě nahrazují hledání řešení postupným ověřováním náhodně navržených řešení (například metoda Monte Carlo). Oba postupy představují NP--úplný problém. Ze zkušeností vyplývá, že genetické algoritmy naleznou rozumné řešení za mnohem kratší dobu, byť tuto skutečnost nelze zaručit. Na řadě úloh se známým řešením bylo zdokumentováno, že zpravidla velmi rychle naleznou i správné řešení, ale matematická záruka, že se tak vůbec kdy stane, zde není.

Genetické algoritmy představují výpočetně poměrně náročné řešení, takže pokud můžeme použít gradientní metody, dáváme jim přednost. Gradientní metody (například neuronové sítě) **nelze** použít v případě, kdy prostor pro hledání řešení není spojitý, zejména účelová funkce není spojitá, nebo je špatně definovaná, a také v případě, že prostor, kde hledáme řešení, má mnoho lokálních extrémů (minim nebo maxim, podle definice účelové funkce).

Evoluční výpočetní techniky (genetické algoritmy, genetické programování, gramatická evoluce) používáme tam, kde ostatní metody umělé inteligence nelze použít nebo z nějakých důvodů selhávají, protože se jedná o (z hlediska strojového času) zpravidla nejnáročnější dostupné řešení. Pokud úloha není řešitelná ani genetickými algoritmy a příbuznými metodami, pak ostatní méně náročné metody selžou také a úloha s aktuálně dostupnými prostředky nemá zřejmě řešení.

Klíčové problémy genetických algoritmů jsou

- nalezení vhodného popisu problému, tj. sady symbolů,
- stanovení správné účelové funkce; její opakovaný výpočet je nejnáročnější částí algoritmu, a často zabírá většinu výpočetního času během hledání řešení.

2.1 Genotyp, fenotyp

Jednotliví jedinci v populaci jsou popsáni genetickým kódem. Často můžeme jedince popsat tak, že možné řešení lze přímo dosadit do genetických algoritmů (například hledáme

kombinaci čísel). V jiných případech bude jedince v paměti počítače lépe zapsat pomocí symbolů, popřípadě s použitím parametrů. Jedinec je pak z tohoto zápisu generován nějakým jednoznačným vztahem. Po vzoru genetiky pak jedince označujeme jako *fenotyp* a jeho vzor v paměti, podle kterého je vytvořen, jako *genotyp* [GP, str. 46]. Do účelové funkce pak vstupuje podle vzoru sestavený jedinec, *fenotyp*.

Často je možné, že jeden nebo alespoň funkčně stejný fenotyp může být popsán různými genotypy (různou kombinací symbolů). Čím delší genotyp povolíme, tím více funkčně stejných jedinců můžeme získat. Podle experimentů popsaných v [Langdon 02], počet těchto možných řešení roste exponenciálně s přípustnou délkou genotypu, a genotyp se může během genetické evoluce dále měnit, zatímco fenotyp je podrobený účelové funkci a konverguje. Důsledkem je, že při selekci může být jedinec se stejnou hodnotou účelové funkce, ale kratším genotypem, vyřazen. Stojí za úvahu, zda by proces neměl být modifikován tak, aby při stejné kvalitě jedinců neměl být porovnáván i genotyp. V genetickém programování, kde je situace obdobná [GP, str. 191], jsou občas používány pokutové funkce, které mají bránit příliš rychlému nárůstu délky genotypu při křížení (hodnota účelové funkce je pro účely porovnání jedinců navýšena o délku genomu, ponásobeném vhodným koeficientem; při výpise nalezeného řešení by měla být zobrazena původní hodnota účelové funkce, která zde často znamená chybu interpolace; zahrnutí délky genomu do zobrazované chyby by uživatele jen mátlo).

Genetický kód je nejčastěji tvořen řetězcem symbolů (string), které popisují, jak je jedinec vytvořen. Známý příklad řešení problému obchodního cestujícího, stejně jako často i příklad rozvrhu (výuky, využití místností, ...), může být tvořen přímo seznamem hodnot, tj. do účelové funkce vstupuje přímo genotyp (který je tedy identický s fenotypem). Genetický kód pak představuje řetězec symbolů, které mohou nabývat jen konkrétních přípustných hodnot a kombinací (dáno *slovníkem*, v případě nepřípustnosti některých kombinací je třeba popsat i *gramatiku*). Při generování nového jedince (zejména u počáteční populace) není problém postupovat podle těchto pravidel a vytvořit jen přípustné jedince. Při křížení je třeba v případě potřeby uplatnit *restrikci* (funkce přípustnosti, *feasibility*). Pokud generující funkce vygeneruje nepřípustného jedince, musí se zahodit a generovat znovu. Při praktické realizaci je třeba dbát na to, aby generující funkce měla nějakou rozumnou pravděpodobnost, že vytvoří přípustného jedince, jinak může algoritmus výrazně zpomalit, nebo dokonce zastavit (příklad: u jednobodového křížení, kdy je požadavek, aby z jednoho rodiče se převzal alespoň první a z druhého alespoň poslední znak genomu. Touto podmínkou může být současné vytvoření dvou přípustných potomků vyloučeno).

U prvních aplikací se sice genetický kód dekódoval po symbolech, ale křížení a mutace se prováděly binárně. Řešením pak může být (kromě funkce restrikce a vyřazování vadně mutovaných jedinců) také vhodně zvolený slovník, kdy v binární formě má každý symbol význam (u biologické DNA mají některé trojice (kodony) stejný význam, takže počet možných generovaných aminokyselin je menší, než 4^3-1, u většiny organismů 20 [GP, str. 41]). Tento způsob kódování se dodnes často používá při simulování umělého života (artificial live), kdy například je přípustných právě 16 symbolů, což reprezentuje 4 bity datového prostoru. Jakákoli binární mutace i křížení tedy vedou opět jen na dekódovatelné jedince. U genetických algoritmů, kde je řešený problém předem dán, například seznam vyučovaných předmětů prvního stupně základní školy je například 9, se musí regulérnost nového jedince kontrolovat (funkce přípustnosti, *feasibility*).

2.1.1 Genom s částmi nevyužitými ke kódování

Uvedený přístup předpokládá, že genom jedince obsahuje pouze informace, které jsou využité při tvorbě jedince (přenos genotyp → fenotyp). Pak ale mohou být zápisy jedinců různě dlouhé, což komplikuje mnohé operace. Někdy se proto pracuje s genotypem konstantní délky. Není-li část pro generování jedince třeba, ignoruje se. Naopak, pokud je genotyp příliš krátký, často se čte znovu od začátku. Analogii v přírodě bychom mohli vidět v tom, že jen část genomu má dnes známý význam, ovšem je možné, že zbylým datům zatím nerozumíme a nějaký význam mají. Analogii čtení znovu od začátku lze najít u některých bakterií, kde chromozom je kruhový a čte se stále dokola.

Pokud je část genotypu, která se nevyužije při kódování jedince, ale přesto se přenáší do potomků, pak zjevně nepodléhá evolučnímu tlaku. V tom případě může jednou dojít k jejímu použití a zde se zatím mohou nashromáždit jakákoli nesmyslná, nebo dokonce destruktivní data.

Příkladem je použití zápisu stromu z genetického programování metodou, označovanou v algoritmizaci jako heap, hromada. To je způsob zápisu, kdy z polohy prvku ve stromu snadno určíme polohu generujícího řetězce (jako struktura je ovšem binární hromada setříděná, [Alg-BH], což u genomu funkce nepřichází v úvahu). Problém je, že většina funkcí má jen jeden argument (dva mají v základní sadě jen operace sčítání, odčítání, násobení a dělení), a druhá větev obsahuje jen neužitečná data (*padding, vata*), která se ale zúčastňují křížení (ačkoli nepodléhají evolučnímu tlaku). Terminální symboly dokonce nevyužijí ani jednu větev.

Recyklace se čtením genomu znovu od začátku může v případě použití tohoto algoritmu pro zápis stromu (heap, hromada) dokonce vygenerovat nekonečně složitého jedince. Tento problém je třeba nějak ošetřit, například v druhé polovině zápisu hromady mohou být přípustné jen terminální symboly.

Jiným problémem jsou nefunkční části genomu, označované v angličtině jako introny[1]. Vznikají v případě, že kódování na fenotyp umožňuje vygenerovat nadbytečnou část, například u genetického programování násobení jedničkou nebo přičtení nuly [GP, str. 186]. Někdy může jít i o škodlivou část, například opět u GP druhá mocnina, která se ihned odmocní, ale díky rozsahu hledání funkce v kladných číslech se tento nesmysl ihned neprojeví. Při dobře sestavené účelové funkci, která především ve vhodném poměru zohledňuje i délku genomu, by si s nimi v tomto případě měly genetické algoritmy poradit samy.

2.1.2 Křížení.

Výkonnost genetických algoritmů závisí právě na schopnosti poskládat nové řešení z kousků, které se osvědčily v předchozích generacích.

U biologických organismů se do potomka kopírují (do nově sestavovaného chromozomu) geny vždy celé [Dawkins], tj. pokud gen reprezentuje generování proteinu a má počáteční a koncový symbol (kodon), nikdy nedojde k jeho přerušení. Výsledný chromozom, pokud není zasažen mutacemi, pak vždy vytváří jen proteiny, dostupné v předchozí generaci. Obdobně lze postupovat i u genetických algoritmů, a dělit genetický kód jen v místech, kde to dává smysl. V řadě aplikací jednotlivé geny obsahují jakousi hlavičku a parametry. Je logické pak při křížení poskládat nového jedince tak, že od každého z rodičů převezmeme vždy celé geny. V genetickém programování tomu odpovídá tzv. kontext zachovávající křížení.

V prvních aplikacích genetických algoritmů se používalo symetrické jednobodové křížení [GP, str. 96], kdy se vzalo prvních *k* znaků genetického kódu prvního rodiče a

[1] Pojem intron je převzat z biologické terminologie, kde označuje části RNA, které nekódují bílkovinu; v případě biologického ekvivalentu ale nevíme, zda tyto části RNA nebo výchozí DNA nemají nějaký jiný význam (v [GP, str. 45] najdeme současně i termín Junk DNA).

doplnilo se genetickým kódem druhého rodiče od znaku *k*+1. Současně se generoval druhý potomek, který obsahoval nepoužitou část genetického kódu druhého rodiče a zbytek genetického kódu prvního rodiče.

První generalizací bylo zavedení vícebodového křížení. Kódy dvou rodičů jsou přepisovány do dvou potomků tak, že se (minimálně jednou) změní, z kterého rodiče se přepisuje do kterého potomka. Při zobecnění, kdy genom nemusí být vždy stejně dlouhý, se mohou střídavě přepisovat buď do jednoho, nebo do druhého potomka různě dlouhé úseky symbolů. Pokud symbol obsahuje parametry, kopíruje se vždy včetně nich. Zpravidla se dodržuje, že celý genom každého rodiče je využit, a ty geny, které se nepřepíší do jednoho z potomků, se použijí v druhém (symetrické křížení). Současně zpravidla platí, že žádný z genů se nezkopíruje do obou. Součet délek genomů rodičů a potomků je tedy stejný. Často dochází k nárůstu délek genomů, ale to bývá způsobeno tím, že v řadě případů jedinci s delším genomem lépe vyhovují účelovým funkcím (např. v genetickém programování).

Obr. 2.1 Schematická ukázka symetrického křížení. Nesymetrické by generovalo jen jednoho potomka, v případě více potomků by bylo spouštěno opakovaně znovu. Barvy slouží jen pro zvýraznění, o kterého rodiče se jedná, pro generování jedince nemají význam. Je třeba zdůraznit, že zde se jedná o v literatuře typický příklad, kdy má genom předem danou („konstantní“) délku.

Pokud se **nejedná** o permutační úlohu, kdy by se hledalo pořadí symbolů v genomu, ale o kombinační, může být pro funkčnost genetické metody užitečné občas zamíchat i pořadím genů. To lze považovat za mutaci, a provést náhodně na ***k*** potomcích. Protože tento typ mutace může být žádoucí, oproti klasické mutaci, kdy je jeden symbol náhodně (bez konkrétního důvodu) nahrazen jiným, je možné tento druh mutace provádět s větší pravděpodobností. Zatímco náhodná mutace zpravidla nemá překročit několik procent a často se po dlouhé úseky evoluce nepoužívá vůbec, cíleným mícháním pořadí symbolů můžeme měnit například až desetinu potomků pravidelně (například jako součást křížení).

Z představy křížení je zjevné, že vícebodové křížení má stejný efekt, jako jednobodové, aplikované několikrát po sobě. Vícebodové může vést rychleji k řešení z hlediska počtu generací, ale představuje složitější algoritmus. S ohledem na spolehlivost (robustnost) řešení a při posuzování celkové doby běhu programu místo počtu generací (který uživatele programu nemusí zajímat) vychází obě metody srovnatelně.

Poznámka – při praktických testech se často projevuje, že potomci mají v průměru horší hodnotu účelové funkce, než rodiče. V [GP, str. 159] je doporučeno řešení, kdy se vytvoří více různých potomků (body pro křížení se v genomu vybírají náhodně) a z nich se do populace zařadí jen dva nejlepší. Zejména u generačního modelu, pokud se odstraňuje celá původní generace, to může být užitečné opatření.

2.2 Algoritmus

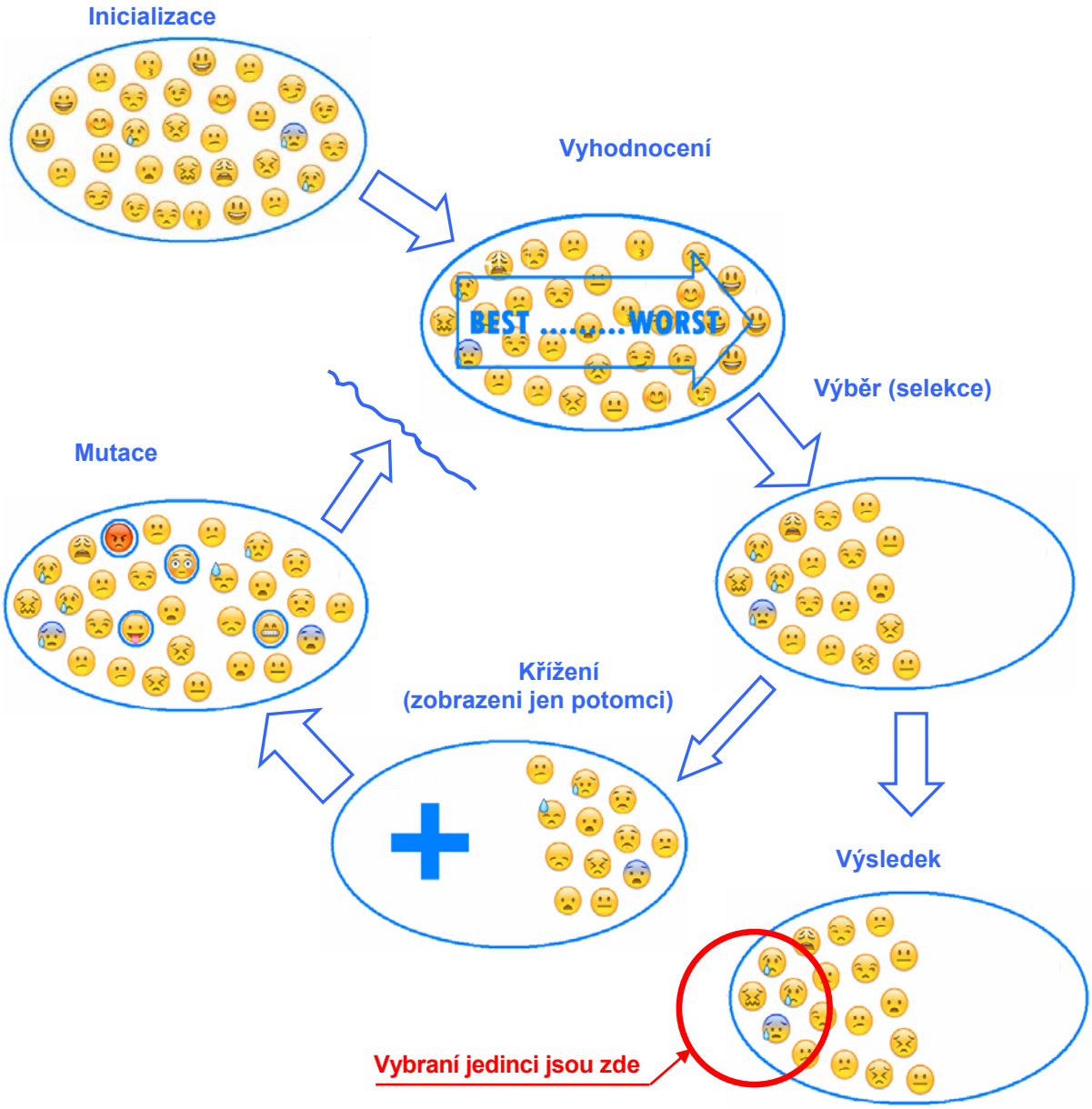
2.2.1 Inicializace populace

V prohledávaném stavovém prostoru vygenerujeme počáteční populaci zpravidla náhodně. Je-li vážný důvod se domnívat, že některé hodnoty lépe odpovídají předpokládanému řešení, můžeme při generování náhodných čísel použít jinou distribuční funkci, než v programovacích jazycích nejdostupnější rovnoměrné rozdělení.

Velikost populace je oblíbeným problémem genetických algoritmů. Větší populace značně zpomaluje řešení, minimálně při evaluaci je nutné použít nějakou metodu třídění a ta má v nejlepším případě asymptotickou náročnost O(n log n) [Vazirani, str. 59]. Malá populace zase vede k prohledání jen části stavového prostoru, zejména, pokud je počet vygenerovaných jedinců jen o málo vyšší, než počet parametrů stavového prostoru, kde hledáme řešení. Význam slov malá a velká je velmi problematický a pro jednotlivé úlohy je zřejmě nejlépe jej vyzkoušet. Z analogií s jinými metodami umělé inteligence by mohl vyplývat například pěti- nebo desetinásobný počet jedinců ve srovnání s dimenzí stavového prostoru hledaného řešení [Abu-Mostafa]. Při použití paralelních genetických algoritmů na clusterech jsou možné i mnohem větší populace, například [Koza 1997] uvádí 640 tisíc jedinců (návrh elektronických obvodů, na clusteru 8x8 80-MHz PowerPC 601 trvala každá evoluce několik dnů). [BT13] demonstruje možnost použití populací od 3 do 13 jedinců v genetickém programování, při testování se pro změnu dochází až k 14 tisícům generací a populace se vyvíjí převážně mutacemi.

2.2.2 Evaluace, selekce

Tyto dvě úlohy se obvykle oddělují. Evaluaci dělat tak či onak musíme, protože potřebujeme uvolnit místo pro nově generované jedince. S vědomím, že i dosti neúspěšný jedinec může nést vhodný gen, není podstatné, zda vyhodnotíme kvalitu jedinců zcela správně. Toho se dá využít při modifikovaných postupech, kdy řazení jedinců podle úspěšnosti nemusí proběhnout úplně. Například při požadavku na smazání poloviny populace můžeme vzít náhodného jedince a všechny ostatní s ním porovnat. Pokud bude horších například 40 až 55%, tak je z populace odstraníme a nepřesnost neřešíme. Pokud oddělíme menší část populace, pak z větší vybereme další prvek a jen v této části pokus o nalezení dělicího bodu zopakujeme. Teoreticky je maximální výpočetní náročnost stále minimálně O(n log n), jako při hledání mediánu v nesetříděné populaci, prakticky ale proběhne jen několik cyklů (díky rozptylu přípustného řešení [zdroj: vlastní výzkum]). Pokud ovšem nesetřídíme soubor, nemůžeme uplatnit základní výhodu ruletové selekce, přidělení vyšší hodnoty pravděpodobnosti při použití jako zdroj pro novou generaci těm nejlepším jedincům.



Obr. 2.2 Typický cyklus genetického algoritmu. Obrázky patří prvnímu průchodu, po aplikaci mutací (čtyři mutovaní jedinci označeni) by se pokračovalo vyhodnocením nové populace, mnohem bližší hledanému řešení.

2.2.3 Elitismus

Aby se při dalším postupu neztratilo potenciálně již možná nalezené řešení, ukazuje se jako žádoucí u těch metod, které nahrazují celou populaci zkříženými jedinci (jedna z variant generačního přístupu), nejlepšího jedince zachránit. Je otázka, zda má smysl cíleně zachovat více jedinců. Například u genetického programování mohou vygenerovaní jedinci *sin(2x)* a *2sin(x)cos(x)* vracet stejnou hodnotu účelové funkce. Zápis jedinců může být ovšem natolik komplikovaný, že stejné řešení nemusí být možné počítačem snadno rozlišit. Například řešení rozvrhu výuky, kde je přehozený čtvrtek a středa, je vlastně identické, ale přizpůsobit program tomu, aby podobné jevy kontroloval, by jej neúměrně zdrželo (předpokládáme, že každý jedinec se k tomuto stavu dostal postupným vývojem z jiných náhodně vygenerovaných podmínek).

Testování vhodného nastavení parametrů je například v práci [Baluja] na straně 3 (108x60 testovacích běhů). Z tohoto testování vyplývá především poznatek o kladném vlivu elitismu (prakticky většina běhů prezentovaných v [Baluja] vychází lépe s elitismem než bez něj). U ostatních parametrů to tak průkazné není (například nejhorších 10 nastavení obsahuje nízkou míru mutací, pod 0,1%, ale u první třetiny nejlepších nastavení na nastavení míry mutací prakticky nezáleží).

Někdy se jako kompromis zachovává například 10% populace, a tato část se nazývá stabilizační. Při experimentech, popsaných v dalších kapitolách, jsem dospěl k závěru, že odstraňování celé nebo větší části předchozí generace vede k příliš rychlému poklesu šíře genomu (diverzity). Přebírání některých jedinců z předchozí generace beze změny, zpravidla náhodně s vyšší pravděpodobností jedinců s lepší hodnotou účelové funkce (viz následující odstavec, ruletová selekce) se označuje jako *reprodukce* [Koza 1995].

2.2.4 Ruletová selekce, Roulette-wheel selection

Název pochází od představy autora této metody, že na kole rulety jsou namalované různě široké výseky (na skutečném ruletovém kole *(hazardní hra)* jsou všechna pole stejně široká).

Selekce se dělá společně s generováním nové generace. Základem je úplné setřídění jedinců podle účelové (fitness) funkce. Následně přiřadíme jednotlivým jedincům pravděpodobnosti, od nejvyšší po nejnižší. Pro přiřazení pravděpodobností se často používají číselné řady, nejčastěji geometrická, méně často aritmetická. Řadu samozřejmě použijeme obráceně, kdy nejhorší jedinec bude mít nejmenší pravděpodobnost výběru. Skutečná pravděpodobnost výběru je hodnota, přiřazená řadou danému jedinci, podělená součtem řady. S touto pravděpodobností vybereme vylosovaného jedince (v [UI3, str. 136] je tento postup uveden jako pořadová selekce a za ruletovou je označeno řešení, kdy podíl na pravděpodobnosti výběru jedince je dán přímo účelovou funkcí, a nevýhody tohoto řešení jsou zde popsány).

Pokud nový jedinec má vzniknout mutací, použijeme vylosovaného jedince jako vzor, pokud křížením, musíme zvolit stejnou metodou druhého jedince. Je otázka, zda při výběru dvakrát stejného jedince nemáme druhý výběr opakovat. Některé metody křížení totiž mohou generovat nový prvek i ze dvou stejných (zpravidla se nějaká část zopakuje, nebo naopak vynechá), ale to vede ke ztrátě šíře prohledávaného prostoru (diverzity populace).

2.2.5 Turnajová selekce

Vybereme náhodně dva nebo dva-a-dva jedince a jen ty porovnáme. Vítěze použijeme pro generování další generace [UI3, str. 137]. V této metodě tedy nemusíme třídit celou populaci.

Příklad – křížení: Vybereme dva jedince, porovnáme, lepšího si zapamatujeme jako prvního rodiče, horšího označíme jako smazaného. Totéž provedeme s jinými dvěma náhodnými jedinci, čímž získáme dva rodiče a dva zrušené prvky. Z genomu rodičů vytvoříme dva potomky, ty zapíšeme do pozic, označených jako smazané. Velikost populace se tedy touto metodou nemění. Základní požadavek je hlídat, aby byli vybráni náhodně čtyři různí jedinci (v případě vygenerování shody generování náhodného čísla zopakovat). Všimněte si také, že druhý rodič nemusí být z hlediska účelové funkce lepší, než jedinec, který neuspěl v prvním porovnání (a naopak). Popsaný postup ale zaručuje, že nejlepší jedinec v populaci zůstane zachován. Další výhodou je, že v porovnání s ruletovou selekcí se stále drží maximální velikost populace (bezgenerační algoritmus).

Wikipedia (cit. 17.1.2017) popisuje variantu, kdy turnajovou selekcí vybíráme rodiče (nebo vzor pro mutací vytvořeného jedince) pro potomky, ze kterých postupně vytváříme novou generaci. Po jejím vytvoření starou generaci smažeme. Při takto chápané turnajové selekci pak můžeme do porovnání (turnaje) vzít více jedinců, a jen nejlepšího z nich použít pro vytváření nové generace. Tím se samozřejmě zvyšuje evoluční tlak, byť si jej stále

nemůžeme nastavovat dle potřeby, jako u ruletové selekce. Nevýhodou je, že takto lze evoluční tlak jen zvýšit; zvýšení evolučního tlaku vede vždy ke ztrátě diverzity populace a tím k horším výsledkům evoluce, jak prokazují výsledky v této práci (kapitola 4).

2.2.6 Porovnání ruletové a turnajové selekce

Náročnost výpočtu genetických algoritmů spočívá hlavně na vyhodnocování účelové funkce. Protože ruletová selekce předpokládá data nejprve setřídit, je její výpočetní náročnost O(n log n). U turnajové selekce, například podle popsaného postupu, je to jen v řádu O(n). Při nižších počtech jedinců v populaci by se sice v turnajové selekci provedlo dokonce více porovnání (každý prvek je vylosován (resp. vybrán) v průměru (resp. přesně) 2x), při větších počtech je porovnání naopak méně. Tento závěr odpovídá faktům uvedeným v [UI3, str. 137].

Při praktických testech na třech řešených úlohách (kapitola 4) jsem ale změřil naprosto opačné výsledky. Turnajová selekce vychází v průměru pětkrát pomalejší při srovnatelné výkonnosti (grafy v kapitole 4) a navíc mimořádně citlivá na špatnou volbu parametrů, například úrovní mutací, která, je-li příliš nízká, zhoršuje diverzitu populace, nebo naopak při příliš vysokých úrovních ztrácí dobrá řešení. V uvedené literatuře navíc popsaná alternativa nezachovává nejlepšího jedince (na rozdíl od mých programů), takže při její realizaci by bylo možné očekávat ještě horší výsledky.

U ruletové selekce se uplatní výhoda, že si můžeme pravděpodobnostní zvýhodnění (distribuční funkci) zvolit, a tím řídit evoluční tlak na hůře hodnocená řešení (oproti literatuře dává lepší výsledky pomocí ruletové selekce evoluční tlak spíše snižovat).

Také v případě, že účelová (fitness) funkce nevrací jen minimálně požadovanou informaci, že jedinec A je lepší/horší/stejný než jedinec B, ale přímo číselnou hodnotu, kterou lze jedinci přiřadit bez ohledu na hodnocení ostatních, nemusí být seřazení všech prvků podle tohoto jediného čísla časově náročné (nejprve všechny ohodnotíme včetně penalizačních funkcí a přegenerování vadných řešení, a pak provedeme třídění běžně dostupnými metodami pro čísla).

Praktické porovnání nabízí kapitola 4.

2.2.7 Pravděpodobnost mutace

Z biologické analogie je zjevné, že mutace vedou zpravidla ke generování vadných jedinců, ale jejich podíl na vývoji je nezastupitelný. Z toho, ale i z praktických pokusů, vyplývá nutnost držet úroveň mutací nízkou.

Z počátku prohledávání nemusí být mutace nutné vůbec, protože u prvních generací je vygenerovaná genetická šíře dostatečná. Selekcí se ovšem snižuje; když si začnou jedinci být příliš podobní, je na čase zařadit i mutované jedince. Úroveň mutací je doporučováno držet nízkou, v jednotkách procent. U velmi malých populací je možné mutovat několik málo jedinců jen každou m-tou generaci. Podle [AField, str. 26, note 4] jsou ovšem u realizací, pracujících s malou populací (uvádějí podstatně menší než obvyklých 500 a více), mutace hlavním zdrojem vývoje. Otázka je, co je to malá populace, viz odstavec 2.2.1 (obecně záleží na konkrétním problému, typicky je hranice mezi 100 a 500 jedinci). Dokladem, že malé populace s dominantní rolí mutací mohou poměrně rychle nacházet řešení, je [BT13]. Autor testoval velikosti populace od 3 do 13 jedinců s dobrými výsledky, pokud je mírou úspěšnosti rychlost nalezení řešení, nebo počet provedených vyhodnocení účelové funkce. Počet generací je samozřejmě větší.

Nové geny, získané mutacemi, se do hlavní populace dostanou až za několik generací, pokud ovšem nejsou vyřazeny ihned po vytvoření („vadný“ nebo „příliš špatný“ jedinec).

Důležitým prvkem je zde *elitismus*, tj. „nejlepší jedinec nesmí být mutován“. Pokud ovšem mají být nově vytvářené prvky generovány i mutací a výsledek se píše na stejné místo do paměti (což je jinak vhodná metoda proti zamrznutí populace), pak tento prvek nebude

jako zdroj nikdy použit. Řešením je například výjimka, že výběr elitního prvku (nebo prvních 10% jedinců z populace) vede ke generování mutovaného jedince za konec populace.

Dobrým měřítkem potřebnosti mutací může být hodnota účelové funkce, pokud vrací číslo. Pokud tato hodnota přestane klesat (stoupat), pak je třeba zařadit do systému mutace [Srinivas]. Pokud je diverzita populace příliš malá (posuzujeme například podle poměru směrodatné odchylky a průměrné hodnoty, tzv. variační koeficient, popřípadě podle poměru nejlepšího jedince a průměru populace), pak lze intenzitu mutací dále zvyšovat [UI3, str. 149]. Z druhé strany je možné, že pokud ani mutace nezajistí lepší řešení, že nejlepší možný jedinec již byl nalezen (globální extrém (optimum) daného problému). Průběh evoluce s takto řízenou úrovní mutací je na obrázku 2.3, převzatém z [UI3].

Je dost velká pravděpodobnost, že mutovaný jedinec bude mít horší vlastnosti, než původní. Přesto může obsahovat přínosné informace. Je tedy zdůvodnitelné, aby se mutovaní jedinci zúčastnili křížení a nové geny se tak přenesly do hlavní populace. Toho lze dosáhnout přizpůsobením pořadí operací při cyklu genetické evoluce.

U turnajové selekce jsou prvky odmazávány v průběhu algoritmu. Mutace normálně provádíme tak, že porovnáme dva jedince, horšího smažeme a místo něj zapíšeme mutaci lepšího jedince. Alternativně můžeme provádět mutace častěji a náhodně s jakýmkoli prvkem, ale nového jedince ponechat místo původního jen v případě, že zmutovaný je podle účelové funkce lepší. V tom případě původní jedinec (zdroj mutovaného) zaniká.

2.2.8 Stárnutí (aging)

Jedním z podstatných problémů genetické evoluce je „zamrznutí“ populace, kdy v hlavní části populace se usadí jedinci, kteří jsou lepší, než většina nové populace, vznikající křížením a mutacemi, a brání tak další evoluci. Nemůže k tomu dojít u generačního modelu, pokud je vždy celá původní generace mazána a nahrazena novou. Tím ovšem často dochází k opačnému extrému, kdy z populace zbytečně vyřazujeme velmi kvalitní jedince.

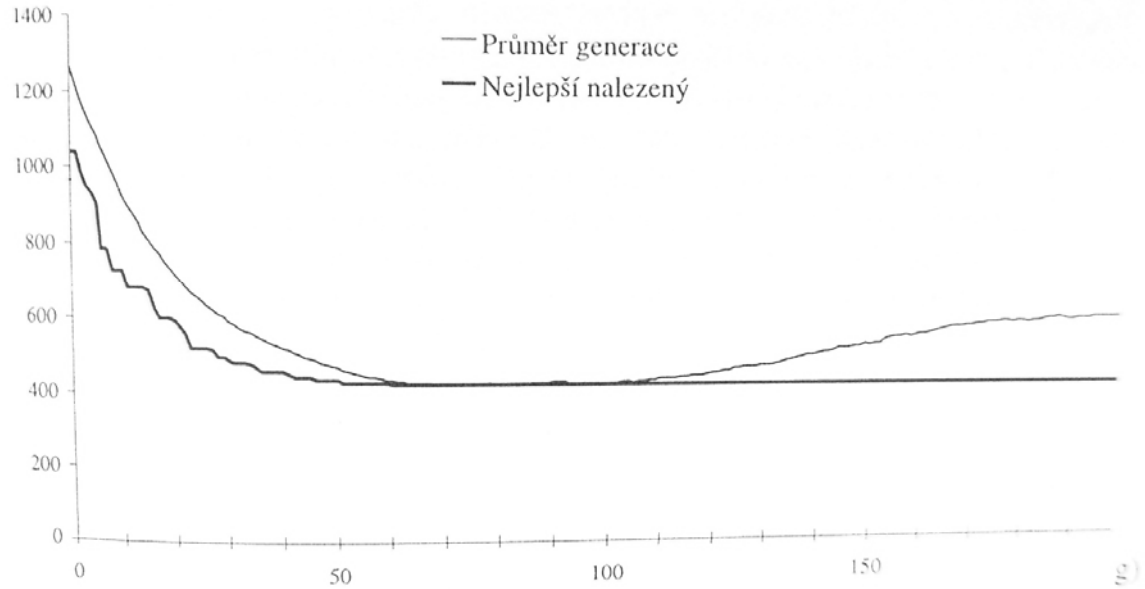
Řešením je ekvivalent biologického stárnutí populace. U jednotlivých jedinců si zaznamenáváme, jak dlouho již jsou v populaci (od vytvoření křížením, vygenerováním nového jedince nebo mutací). Odstranění z populace pak může být buďto narůstající pravděpodobností náhodného smazání z populace, nebo přičtením x-násobku věku k pokutové funkci před provedením ruletové selekce. [Ghosh] popisuje realizaci stárnutí použitím dodatečné pravděpodobnosti u ruletové selekce, ale japonský autor publikace je zatížen lokálním vztahem ke starším jedincům a z křížení vyřazuje i příliš mladé jedince.

V každém případě není žádoucí, aby podobnou penalizací byl zatížen dosud nejlepší jedinec, tj. zachováváme nebiologický koncept elitismu.

2.2.9 Ukončení běhu

S výpočtem končíme nejčastěji po předem zvoleném počtu generací, nebo po proběhnutí času, který máme na daný experiment k dispozici. V umělé inteligenci by bylo vhodné skončit spíše v okamžiku, kdy program již není schopen nalézt lepší řešení. Ačkoli hlavní výhodou genetických algoritmů je, že nemají tendenci uvíznout v lokálním extrému, žádná záruka že najdou globální extrém účelové funkce zde není. Proto má smysl skončit i v případě, kdy se již řešení nezlepšuje, resp. když ani mutace nepomohou nalézt lepší řešení problému. Při vyhodnocování statického stavu (nejlepší řešení zůstává stejné) lze vyjít z hodnot účelové funkce, pokud vrací číslo; v opačném případě můžeme za konečný stav označit ten, kdy nejlepší jedinec (zachovaný díky elitismu) je po desítky generací nehledě na mutace ostatních stejný. Pokud nevyhodnocujeme pořadí jedinců v průběhu evoluce, například při použití turnajové selekce, je třeba pro vyhodnocení těchto podmínek (nutnost mutace, dosažení konečného stavu) jedince seřadit alespoň jednou po x generacích, například každou dvacátou, nebo vyhodnotit alespoň nějaký reprezentativní výběr z populace. Na

konci evoluce ovšem vybrat nejlepšího jedince musíme (max. n-1 porovnání představuje výpočetní náročnost O(n)).



Obr. 2.3 Ukázka s nárůstem mutací. Zdroj: Mařík a kol.: Umělá inteligence ([UI3, str. 149, obr. 3.19])

2.2.10 Samčí a samičí strategie.

Někdy se v literatuře při evoluci autoři pokouší o aplikaci samčí a samičí strategie.

Populace se rozdělí na dvě části.

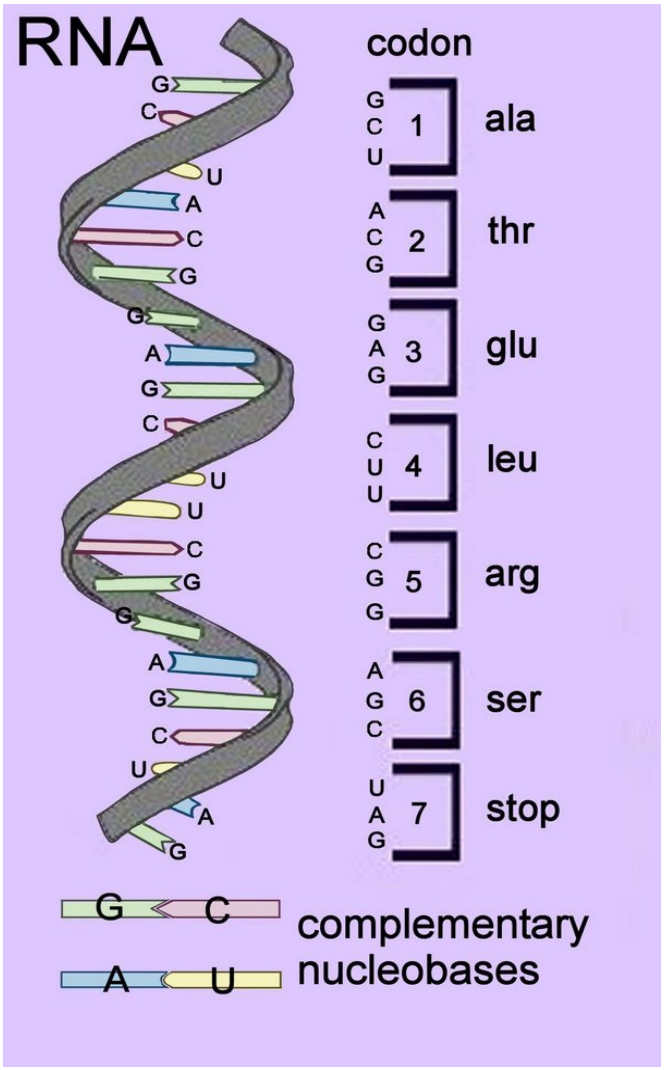
U části, označené jako samičí, se do další generace zachová velký počet jedinců, například se nahradí jen třetina nejhorších jedinců. Tato součást populace se udržuje podstatně větší než samčí a má udržovat genetickou diverzitu.

U samčí populace se zachovává jen malý počet jedinců, například desetina. Na tuto část populace pak působí větší evoluční tlak a má nás přivést k rychlejšímu řešení zadaného problému. Křížení autoři zpravidla provádí vždy mezi jedinci jedné a druhé populace. Jedinci ale žádné pohlavní znaky nemají, takže není rozdíl, zda výsledného jedince potřebujeme doplnit do jedné, nebo druhé skupiny (generuje se úplně stejně).

Oddělení samčí a samičí populace je popsáno například v [Ošmera].

2.3 Porovnání s biologickým vzorem

V biologické genetice probíhá kódování dvoustupňově, z DNA se nejprve okopíruje informace do RNA, a na základě RNA se následně sestavují výsledné bílkoviny z proteinů. Každé trojici písmen (triplet, kodon) odpovídá připojený dílčí protein odpovídajícího typu. Možných stavebních prvků (dílčích proteinů) je ovšem jen 20, některé kombinace kódují stejný protein. Speciální kombinace (UAG, UAA nebo UGA) je vyhrazena pro zakončení řetězce, resp. vede k tomu, že si kódující rybozom sáhne po elementu, který není k dispozici, a tím se generování proteinu přeruší (začátek je označen AUG, výjimečně xUG nebo AUU). U některých organismů může některý z ukončovacích kódů znamenat připojení jiné dílčí aminokyseliny, takže např. Wikipedia uvádí další dva kódované proteiny.



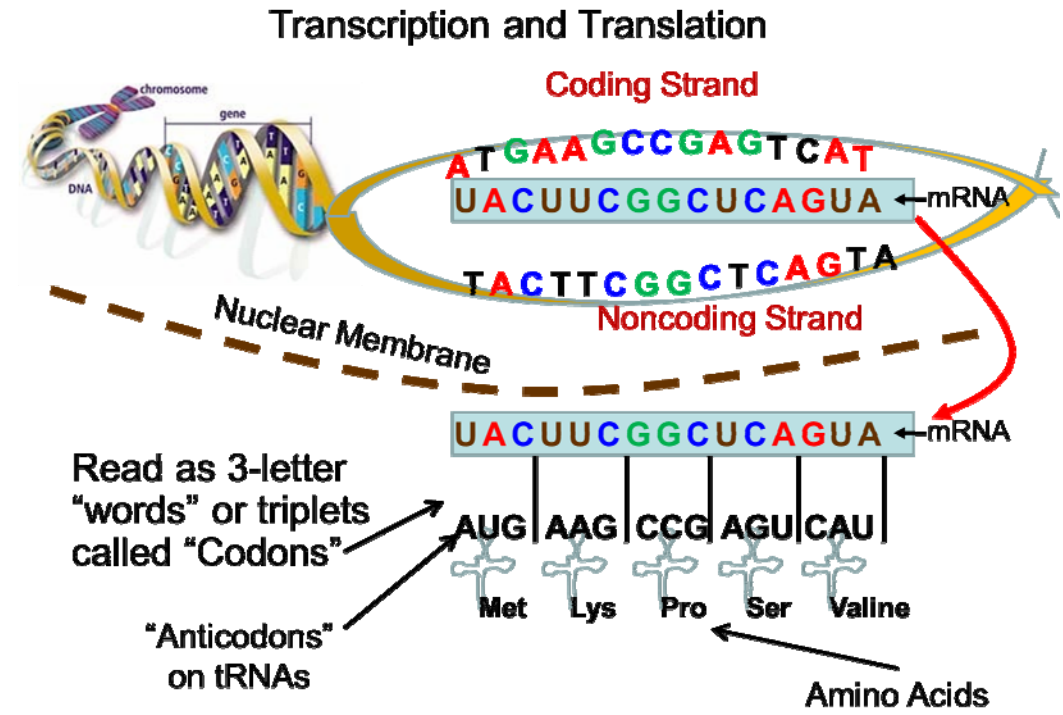
Obr. 2.4 Názorný obrázek pochází z Wikipedie.

Zajímavá je realizace křížení a mutací.

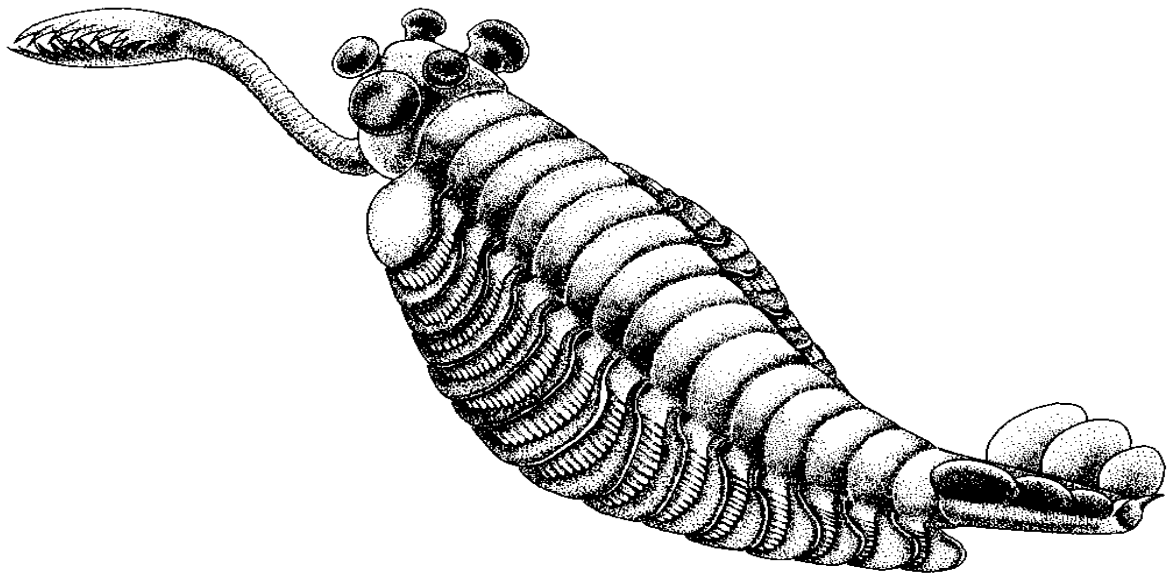
Mutace jsou obecně považovány za nežádoucí a systém se je snaží vyeliminovat. Přesto je k nim připraven sáhnout, pokud dojde ke křížení jedinců s příliš blízkým genomem (úzká příbuzenská plemenitba). Mutace v tomto případě často vedou ke generování chybných jedinců. (Většinou se tato skutečnost vnitřními procesy odhalí a dojde k potratu, jen v malém procentu případů dochází k narození defektního jedince. Biologická evoluce probíhá tak, že se porovnávají fenotypy, což jsou v tomto případě dospělí jedinci, a nejhorší jedinci by měli být z populace odstraněni přirozeným výběrem, tím, že v reálných podmínkách nepřežijí, resp. nebudou mít příležitost se dále množit.)

Křížení probíhá tak, že se při vytváření buňky, určené k množení (gameta, např. oocyt), se v procesu, zvaném meióza, spojí stejné chromozomy a opět rozdělí tak, že geny, které nejsou shodné, se dědí vždy jen po jednom rodiči (z hlediska buňky – z hlediska výsledných živočichů zde dochází k míchání genů prarodičů). Z hlediska genetických algoritmů se tedy při opětovném spojení do společné buňky s geny druhého rodiče jedná o symetrické křížení. Při meióze se do každého nově vytvořeného chromozomu předávají vždy jen celé geny (až po ukončovací sekvenci, tj. UAG, UAA nebo UGA). *Křížením se tedy geny nemění, přenášejí se vždy celé* [Dawkins]*; chromozom ovšem může obsahovat jinou kombinaci genů*. Tím je v přírodě zajištěno, že až na mutace nedojde k sestavení genu, který by nešel přepsat na protein (pokud je v části chromozomu, kde se to očekává). Teoretický pohled na problematiku viz [GP, str. 35-55].

Biologický proces s křížením, tj. množení s pohlavně rozdělenými jedinci, neobsahuje elitismus. Ekvivalent elitismu lze najít u nepohlavního množení u jednobuněčných bakterií, které probíhá dělením (evoluce zde obsahuje jen mutace).



Obr. 2.5 Zdroj: http://sphweb.bumc.bu.edu/otlt/MPH-Modules/PH/PH709_DNA-Genetics/Translation.png (15.1.2015)



Obr. 2.6 Ukázka výsledného živočicha, kódovaného geny (bez vazby na předchozí obrázek). Zdroj: www.as.wvu.edu/~kgarbutt/EvolutionPage/Studentsites/Burgesspages/images/opabinia.gif (15.1.2015)

2.4 Použití genetických algoritmů v kombinaci s neuronovými sítěmi

2.4.1 1. více různých neuronových sítí

V mnoha aplikacích se používá metoda současného trénování více neuronových sítí. Například u predikce hodnot se pak získá výsledek od každé sítě, extrémy se „zahodí“ a ze zbylých hodnot se určí průměr. Nevýhodou je násobná výpočetní náročnost (ale u neuronových sítí je výpočetně náročné hlavně trénování), výhodou je vyšší spolehlivost získaného výsledku.

V tomto případě můžeme některé prvky genetických algoritmů použít i v průběhu trénování sítě.

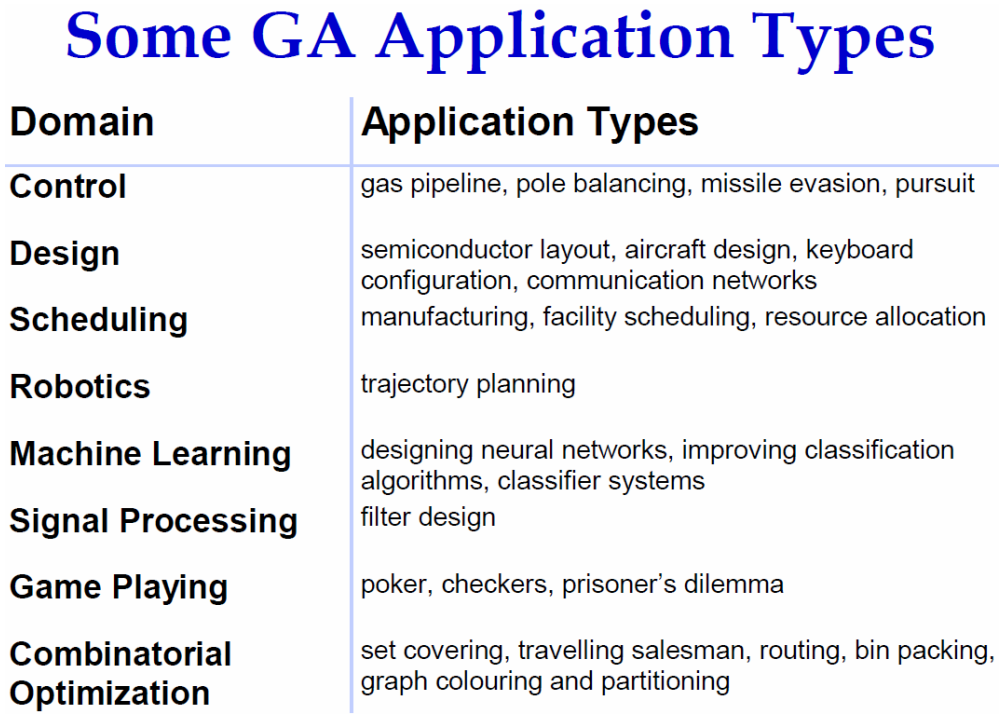
- křížení. Síť získáme jako spojení prvků ze dvou sítí. Protože význam prvků je náhodný, vzniklý jedinec je většinou horší, ale může být lepším začátkem pro další trénování.
- mutace. Náhodné změny hodnot, nebo záměny vah způsobí zhoršení výsledků, ale mohou vést k překonání lokálního minima.
- elitismus. Předchozí operace nesmí měnit prozatím nejlépe vytrénované sítě. Například nejprve odstraníme několik sítí s nejhoršími výsledky a pak je nahradíme sítěmi, vzniklými předchozími metodami. Pro jejich výběr použijeme ruletovou selekci, protože při trénování neuronové sítě (metodami odvozenými z back propagation) získáváme výslednou chybu sítě jako jediné číslo (resp. u sítě s více výstupními neurony lze jejich chyby sečíst).

Teoreticky je možné opustit při trénování neuronových sítí gradientní metody a pohlížet na popis sítě (váhy) jako na množinu reálných čísel, například zapsaných v poli. Takovou úlohu lze řešit pomocí genetických algoritmů, kdy genotyp je reprezentován polem reálných čísel. Stejný problém je pak možné řešit i hejnovými algoritmy. Oba přístupy jsou ale mnohem výpočetně náročnější a pro velký počet parametrů konvergují velmi pomalu.

2.4.2 2. uvnitř jediné neuronové sítě

Jednotlivé neurony se na výsledku sítě podílejí různě. Je tedy možné ty, které mají nejmenší přínos (na jejich výstupu, přesněji na vstupu od nich do dalšího neuronu, jsou řádově menší váhy, než od ostatních), odstranit a nahradit jiným neuronem. Ten může být náhodně generovaný, nebo vzniknout mutací nebo křížením jiných neuronů. Také je možné naopak ten neuron, který nejvíce přispívá k výsledné hodnotě (na vstupech následujících neuronů jsou největší váhy), rozdělit na dva (nelze v pravidelném poměru – vždy je třeba jeden díl zvolit náhodně a váhu pro výstup z druhého nově zavedeného neuronu určit tak, aby v součtu bylo dosaženo původní hodnoty; první generovaná váha může být i větší než od „rodičovského“ neuronu, druhá pak bude záporná. Nejedná se zde tedy o ekvivalent křížení, ale spíše dělení buněk.

U problematických aplikací neuronových sítí může být použití podobných metod úspěšnější, než obvyklá metodika simulovaného žíhání. V případě adaptivních neuronových sítí, kdy se síť současně používá (například k modelování funkce nějakého systému) a současně trénuje pak nelze metodiku odvozenou od simulovaného žíhání použít vůbec a obdobné heuristické metody (jako je zde popisovaná) mohou být jediným způsobem, jak nevytvořit síť, kterou již nepůjde natrénovat na správnou funkci.



Obr. 2.7 Stránka z přípravy na přednášku umělé inteligence, dostupná na internetu. Source: Genetic Algorithms, a Tutorial. Zdroj: utc.ac.za (University of Cape Town, South Africa) (15.1.2015). Stránka není v současné době dostupná.

2.5 Příklady použití GA v ČR

- prof. Ing. Ivan Zelinka, Ph.D., UTB Zlín, VŠB-TU Ostrava: GACR 102/09/1680, Control Algorithm Design by Means of Evolutionary Approach; návrh obvodů

- RNDr. Jaroslav Teda, Ph.D.; Zdeněk Lehocký: Optimalizace dělení materiálu – návrh pomocí GA (VÍTKOVICE ITS s.r.o.)

2.6 Praktické ukázky použití GA

2.6.1 Genetické algoritmy pro řešení permutačního problému

(Genetic algorithms – *Order-based GA*)

Obchodní cestující

Mnoho problémů lze popsat jako vyhledání pořadí prvků, z nichž žádný se nesmí opakovat, ale každý je zastoupen (právě jednou). Nejznámější je problém obchodního cestujícího [UI3, str. 271; úloha je NP-úplná], který je čistě permutační: Obchodní cestující má za úkol navštívit všechna města ze seznamu v libovolném pořadí. Ve školní podobě tohoto problému je hledaná optimální cesta dána jen tím, jak jsou města od sebe daleko, a má se tedy najít nejkratší cesta. Při řešení reálné úlohy může ostatní vlivy, například nepřípustnost cesty autem delší než 4 hodiny (vyžadována přestávka), nebo reálnou realizovatelnost cesty (jen v některých městech je možné přenocování) vyřešit vhodně definovaná funkce přípustnosti (feasibility, resp. restrikční funkce). V reálném případě by také nebylo na závadu nějaké místo navštívit zbytečně podruhé, pokud se tím zkrátí cesta (obchodní cestující vjede do města znovu, jen zde již nenavštíví zákazníky).

Úlohu **popíšeme** tak, že vytvoříme seznam míst, která má navštívit (tabulka), a každému místu přiřadíme jeho pořadové číslo, v dalším popisu 1 až M. Genotyp je tedy vektor (jednorozměrná matice) čísel, kde se každé číslo vyskytuje právě jednou. Specifické u takto definované úlohy je, že do účelové (fitness) funkce vstupuje právě tento vektor, nikoli výsledný rozpis cesty (fenotyp).

Inicializace probíhá tak, že na začátku náhodně vygenerujeme populaci N jedinců, kdy každý představuje jinou cestu. Pro generování množiny prvků s náhodným pořadím se zpravidla používá Fisher-Yatesův algoritmus [Alg-FY]. Lepší metody generování počáteční populace představují algoritmy, popsané v [Erickson, str. 441] – buď zdvojením kostry grafu (minimum spanning tree), nebo jejím doplněním na Eulerův okruh (Christofidesův algoritmus). Oba postupy je možné použít, pokud u vzdáleností platí trojúhelníková nerovnost, tj. vzdálenost mezi městy A a C není větší, než součet vzdáleností A a B a B a C. V obou případech je třeba algoritmus znáhodnit (randomizovat), protože jinak by generoval jen jediného jedince. I tak je často nutné část populace generovat zcela náhodně.

Aktuální populaci ohodnotíme. Protože účelová funkce vrací hodnotu (ve „školní“ verzi vzdálenost, v reálné náklady), přiřadíme každému jedinci tuto hodnotu. Úloha tedy vede na možnost použití ruletové (roulette-wheel) selekce.

Novou generaci N prvků při použití generačního modelu vytvoříme do nového pole řešení tak, že vybereme dva náhodné jedince (pokud vybereme dva stejné, výběr druhého opakujeme), a ty zkřížíme. Pokud chceme, aby pravděpodobnost výběru prvního prvku byla například ½, druhého ¼, atd., musíme použít vhodnou distribuční funkci (v tomto případě geometrickou posloupnost). Menší evoluční tlak, tedy pomaleji klesající distribuční funkce, bývá rozumnější. Pro výkonnost hledání správného řešení je u genetických algoritmů podstatné hlavně zachování šíře populace, nikoli rychlé potlačení nesprávných řešení (šíři populace lze částečně nahradit mutacemi, za cenu pomalejší konvergence).

Křížení u permutačních úloh probíhá tak, že vezmeme část genomu prvního rodiče, a k němu doplníme chybějící prvky v pořadí, v jakém byly u druhého rodiče. Je otázka, kam v řetězci zapsat úsek přebíraný od prvního rodiče. Při jednobodovém křížení se náhodně

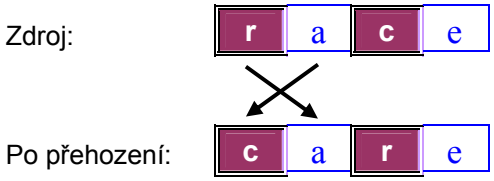
vybere jistý počet prvků, ten se převede do potomka (od začátku), a zbylé se pak přebírají z druhého rodiče. V tomto případě se pak často do dalšího potomka nejprve převezmou nepoužité prvky z druhého rodiče a doplní se nepoužitými prvky z prvního rodiče, protože obojí předchozí algoritmus již nalezl. Z jednoho křížení pak vzniknou dva potomci. U dvoubodového křížení vezmeme prvky z prvního rodiče počínaje například prvkem *j* a konče prvkem *k*, a umístíme je do potomka od pozice *l* (čísla musí být generována tak, aby byla realizovatelná). Doplníme jako v případě jednobodového křížení chybějící prvky z druhého rodiče, postupně do zatím volných pozic podle pořadí v tomto druhém rodiči. Obdobně je možné, byť méně obvyklé, i vícebodové křížení.

Po vygenerování N-1 potomků novou populaci doplníme o nejlepší řešení z předchozího kroku, abychom je neztratili (elitismus).

Obr. 2.8 Křížení v případě permutačního problému. Každý symbol může být použit pouze jednou, hledá se jejich pořadí, které nejlépe vyhovuje účelové funkci. Zdroj slov: http://www.enchantedlearning.com/english/anagram/ (dostupné: listopad 2016).

Nevýhodou křížení u permutačních úloh je, že ve skutečnosti současně dochází k zpřeházení prvků (prvky přebírané z druhého jedince vytvoří jinou cestu, protože jsou vynechány již použité prvky), které lze považovat za mutaci (viz [Koza, 1995]).

Vytvoření nové generace řešení a odstranění staré je obvyklejší, ale řešení, kdy stará populace zůstává a je doplňována novými potomky, zatímco nejhorší jsou průběžně odstraňováni, je programátorsky jednodušší. Místo dvou polí máme jen jedno, o velikosti například 2N. Do volných prvků vždy vygenerujeme N nových potomků (mutace použijeme jen v případě, že populace se nemění; mírou neměnnosti stavu může být stagnace nejlepšího řešení, měřeno účelovou funkcí – například si můžeme pamatovat hodnotu účelové funkce pro první prvek a součet dalších tří). Všechny prvky znovu ohodnotíme, seřadíme a ponecháme prvních N nejlepších. Zjevně není třeba aplikovat elitismus, v tomto případě k němu dochází automaticky.



Obr. 2.9 Ukázka typické mutace v případě permutačního problému.

Nevýhodou tohoto řešení je, že pokud může být sub-optimální řešení popsáno několika různými způsoby (mezi permutační úlohy patří například školní rozvrh), pak může zablokovat první pozice v poli a evoluci zastavit. Model s odstraňováním staré generace je na toto méně náchylný. Lze navrhnout řadu alternativních postupů, jak toto omezení obejít, například v každém kroku přegenerovat více prvků než N – například začít generováním

prvku 2N, pokračovat 2N-1, ale nezastavit se u N+1, ale vyměnit i zbývající až na první (elitismus), vždy za využití všech ostatních prvků. Tímto způsobem se nejen odstraní celá původní generace, ale dokonce se pro generování posledních prvků použijí čím dále častěji i nově vygenerovaní jedinci. Tento postup tedy ještě zvýrazní všechny vlastnosti algoritmu, urychlí hledání řešení, ale zvětší i negativní vliv na šíři populace (ekvivalent genetické diverzity). Z druhé strany takto navřené řešení, které pracuje stále s jedním polem v plném rozsahu, „in place“, je programátorsky jednodušší.

Existuje i opačný názor, podle kterého je část původní generace vždy zachována, aby se zachovala různorodost (šíře populace). Vliv na šíři (různorodost) populace je možné kompenzovat mírou mutací (pro menší populace je možné nastavit větší úroveň mutací).

Počet jedinců v populaci samozřejmě není možné plnohodnotně ničím nahradit. Z druhé strany náročnost vyhodnocení jedné generace je většinou úměrná O(n log n), což naopak vede k nasazení menších populací.

Mutace představují další složku algoritmu permutačního problému. V tomto případě je nejlepší záměna pořadí několika prvků, dvou, nebo celé sekvence. Je třeba dodat, že existuje heuristický přístup, kdy se pracuje jen s jedním řešením, a v něm se algoritmus pokouší změnit pořadí jen dvou nebo tří po sobě následujících prvků; v tomto případě se nové řešení použije jen v případě, že dojde ke zlepšení. Genetické algoritmy pracují s populací a je tedy možné doplnit takto nově generovaná řešení na konec populace a na konci cyklu ponechat v populaci jen požadovaný počet nejlepších. I u permutačního problému je rozumné, pokud se hodnota účelové funkce dlouho nezlepšuje, **úroveň mutací** zvýšit.

Ukončení algoritmu

Za nalezenou konečnou variantu (sub-optimální řešení) můžeme považovat stav, kdy ani mutace nepomohou najít lepší hodnotu účelové funkce. Zpravidla pokračujeme s využitím mutací ještě po delší dobu, než která byla potřebná pro nalezení aktuálního řešení, protože přenos nových prvků z mutací vytvořeného jedince do hlavní populace je pomalý a nespolehlivý. Často končíme také pro časové omezení, například sestavení rozvrhu může být nad výpočetní kapacitu dostupné techniky (pro sestavení rozvrhu na letní semestr je k dispozici nejvýše pět měsíců zimního semestru).

Podrobný rozbor problému obchodního cestujícího lze najít například v [TSPR], problematice se věnuje i česká [UI3, str. 143-150].

2.6.2 Genetické algoritmy pro řešení kombinačního problému

(Genetic algorithms – *Knap-sack type problems*)

Výsledné řešení je hledáno jako kombinace prvků ze sady, ze které vybíráme za nějaké omezující podmínky. Například v podnadpise zmíněný problém skládání věcí do ruksaku (batohu) má jistou množinu objektů, ze kterých si vybíráme, nějakou omezující podmínku (například maximální hmotnost či objem), a nějakou účelovou funkci (například naložit zboží v maximální ceně). Problém je NP úplný, obvykle se řeší „hrubou silou“.

Pro problém je typické, že nezáleží na pořadí prvků. Naopak, pokud dochází ke křížení, mohou se zvolit úseky náhodně a případně být i ve výsledném jedinci přehozeny.

Vzhledem k tomu, že neznáme počet prvků v hledaném jedinci, může křížení snadno generovat nepřípustné jedince, nebo naopak jedince, kteří zcela zbytečně nevyužívají část dostupného prostoru. To je třeba ošetřit funkcí přípustnosti (feasibility), resp. v druhém případě nějakým heuristickým řešením.

2.7 Paralelní genetické algoritmy

Genetické algoritmy jsou časově velmi náročné, ale na rozdíl od neuronových sítí je lze snadno dělit do více dílčích výpočetních procesů [GP, str. 279].

Rozdělení je jednoduché. Při K počítačích na každém dílčím počítači (uzlu) vygenerujeme při inicializaci jen N/K jedinců. Provedeme nějaký počet kroků (generací, například 100), a pak si vyměníme nejlepší dosažená řešení (podle [EVT, str. 185] je dalším možným řešením předávat náhodně vybrané jedince; publikace to nekomentuje, ale je možné, že jde o inspiraci z ostrovní evoluce, kde bouřky či hurikány náhodně přesouvají části populace ptáků z ostrova na ostrov).

Pokud je N/K celým násobkem K, můžeme teoreticky vytvořit novou populaci tak, že z každého počítače převezmeme N/K^2 nejlepších jedinců. Pak ji použijeme jako startovací na každém počítači znovu a budeme doufat, že vzhledem k náhodnosti křížení a mutací každý bude tvořit různé jedince a po dalších x generacích dospějí k různým výsledkům. Výpočet pak probíhá stejně, jako bychom měli vše na jednom počítači. Intuitivně lze ale usoudit, že takto zbytečně přicházíme o genetickou diverzitu.

Ostrovní provoz

Opravdu bylo prokázáno, že diverzifikace řešení na dílčích stanicích a tím spíše na celé síti je větší, pokud předáváme jen nejlepšího jedince, tedy přebíráme K jedinců, za předpokladu, že z původní populace je zachována větší část jedinců (např. $N/K > 3$, lépe 5) (podrobněji např. [AField, str. 88]). Takto diverzifikovaná populace pak tolik nevyžaduje aplikaci mutací, které jsou ze zkušeností méně efektivním nástrojem, než křížení. (diverzita, tedy rozdíly v genomech, zůstává při srovnávání mezi populacemi *různých* ostrovů).

Variantou tohoto modelu je dokonce představa sousedních počítačů – například můžeme stanice, podle toho, jak se připojují na řídící server, zapojit do jakéhosi kruhu, kdy si stanice předávají (po menším počtu generací) nejlepší prvky vždy jen s předchozím a následujícím přihlášeným počítačem (viz [Andre]; první je pak navázán s posledním do ***logického kruhu***). Ukázalo se, že „genetický tlak“ na subpopulaci (populaci ostrova) je pak nižší a síť spíše nalezne správné řešení (globální extrém), i když při výše popsaném řešení se k sub-optimálnímu řešení dojde rychleji. Zde je třeba zdůraznit, že jednak u reálných úloh nejsme schopni poznat, zda řešení je optimální, nebo suboptimální, jednak v mnoha případech nám suboptimální řešení postačuje, s vědomím, že optimální již nemůže dávat znatelně lepší výsledky (řada technických problémů, například skladba plechů na křídle letadla apod.).

Z výše uvedeného vyplývá, že logický kruh (logický kruh znamená, že každý si předává data jen s předchozím a s následujícím počítačem ze seznamu, se speciálním případem, kdy první je provázán s posledním) lze provozovat i jen se dvěma prvky a další stanice do něj mohou vstupovat a vystupovat podle potřeby, kdy si počáteční populaci buď vytvoří, a nebo načtou nějakou, která byla – dříve končícím počítačem – uložena na serveru. Můžeme tak snadno využít volnou kapacitu počítačů, například těch, které jsou přes den využívány jiným způsobem, aniž bychom předem věděli, kolik se jich nakonec zapojí. To může být výhodné u mimořádně náročných úloh, jako je například zmíněná tvorba rozvrhu. Teoretické práce ovšem často pracují s předem pevně daným počtem zúčastněných počítačů, vyhrazených jen pro tyto experimenty, a s plánovanou výměnou jedinců po pevném počtu generací (synchronní migrace, [EVT, str. 184]).

Pokud je dostupný počítačový cluster, jsou data zpravidla ukládána na sdílený disk. Každý datový balíček musí být označen jedinečným názvem, ten lze snadno generovat jako náhodné číslo. Zabránění vytvoření dvou souborů se stejným názvem zajišťuje operační systém serveru. Jinou možností je zahrnout do názvu číslo procesu. Předávání sdílených úseků dat je pak opět přes tyto soubory.

Toto řešení lze snadno nahradit spojením nevyužitých počítačů do sítě (například realizovat genetický algoritmus jako spořič obrazovky, podobně jako u projektu SETI (https://setiathome.berkeley.edu/)). V současných programovacích jazycích lze **data** nejsnáze **ukládat** na webovém nebo **ftp** serveru, které jsou často k dispozici a ukládání dat v rozumné frekvenci pro ně nepředstavuje velkou zátěž (jejich stávající funkce může být zachována). Na ftp serveru si každý počítač vyhledá korektně ukončená data (po počítači, který se korektně odhlásil), dlouho opuštěná data, nebo načte pokyny pro vygenerování nových dat (krajní varianta, pokud není možná jedna z předchozích). Server v tomto případě slouží jen jako odkladiště dat a na výpočtu řešení se nijak nepodílí. U webového serveru lze navíc vytvořit stránky pro předávání a pro zobrazení statistiky výpočtu; tento postup je ale náročnější a stojí za zvážení vyčlenit pro tento účel samostatný webový server.

Protože v relevantní literatuře jsem nenašel vhodný příklad, přikládám svůj návrh řešení logického kruhu:

Názvy souborů zvolíme začínající písmenem O (odloženo) nebo A (aktivní). Stanice, která právě uložila novou verzi svého souboru (dokončila např. 100 generací), prohlédne adresář, přejmenuje všechny, které jsou nápadně starší (například 10x starší, než byl soubor naší stanice před obnovením), a to z A*.* na O*.* (zachová zbytek jména, je-li to možné), pak si stáhne všechny názvy začínající A, seřadí je, a ke svému najde předchozí a následující (první hvězdička reprezentuje náhodně vygenerovaný název souboru, netřídí tedy podle adresy či dokonce umístění počítače). Z těch si převezme předávaná data (tedy vždy nejlepšího jedince, popřípadě několik náhodně ze souboru, pokud potřebuje zvýšit diverzitu) a vrátí se k výpočtu dalších 100 generací. Stanice tedy musí počítat s tím, že někdo její soubor přejmenuje z A*.* na O*.*, například pokud výpočty běží na pozadí a počítač řešil jiné, aktuálnější problémy (v tom případě musí vrátit název zpět, nebo, byl-li mezitím přebrán jiným počítačem – to se může poznat například podle ip adresy zapsané v souboru – zapsat svá aktuální data pod novým náhodně vygenerovaným názvem). Jiným řešením problému, aby někdo „nezabral“ cizí soubor, na kterém se byť pomalu, ale pracuje, může být místo náhodné části jména použita ip adresa, resp. její část, která se v organizaci mění (typicky maximálně šest číslic). Po převzetí opuštěného cizího souboru jej pak příslušný počítač smaže a vytvoří svůj. Pokud se původní majitel souboru vrátí, prostě jej jen zapíše zpět (jeho novou verzi). Vzhledem k charakteru zpracovávaných dat tento přístup není na závadu, i když výpočetní postup pak nemusí být optimální (na stejné části populace se pracuje dvakrát); z druhé strany genetické algoritmy jsou náhodný proces a jsou-li stejná data použita vícekrát, dojde se jistě k různým výsledkům.

Z časových důvodů jsem tento model nezkusil naprogramovat a zůstává tak jako potenciální zadání nějaké diplomové práce. Podkapitolám 2.6 a 3.1 (symbolická regrese) odpovídají výukové programy, popsané v kapitole 4.

3. Genetické programování

S první vizí genetického programování přišel John Koza v roce 1990 (podrobněji: [Koza]). Přichází s koncepcí zápisu stromem, což umožňovalo generovat programy v poněkud neobvyklém programovacím jazyku Lisp. Zpočátku se soustředil na vytváření jednoduchých funkcí, čímž položil základy řešení symbolické regrese.

Úloha vytvářet celé programy byla později znovu řešena tím, že zápis se vrátil k lineárnímu genomu, kdy genotyp jedince je překládán na fenotyp pomocí gramatiky (*gramatická evoluce*, kapitola 3.3).

3.1 Symbolická regrese

Základní řešená úloha zní: Máme průběh funkce (může být i 2D, 3D, ...), tedy její hodnoty, a „stavební prvky“, ze kterých se jí máme pokusit sestavit (sin, cos, ln, x^2, násobení, sčítání, několik různých konstant apod.). V obecné literatuře se často uvádí, že úkolem je vytvořit celý program nebo algoritmus [např. UI4, str. 129], kdy je slovník rozšířen o podmínky, ale i cyklické funkce. V tomto textu jsou problémy daného řešení popsány v podkapitole 3.3, Gramatická evoluce.

3.1.1 Zápis funkce stromem

Podle [GP, str. 101] se zápis funkce stromem objevuje již v práci [Cramer, str. 185]. Pro rozvoj genetického programování je ale výchozí prací [Koza]. Profesor J. Koza popisuje zápis funkce nebo programu pomocí stromové struktury, kdy jednotlivé parametry funkce představují podstromy (funkce sčítání má dva podstromy, z funkce sin vede jen jedna větev; jsou jistě i funkce s třemi a více větvemi, jako min, suma, ... [AField, str. 10]. Aby nedocházelo k záměně s jedincem (sestavený jedinec je také funkce), jsou tyto dílčí funkce (sčítání, sin, ln) označovány jako ***primitivní funkce***.

Ačkoli průvodní aplikací prof. Kozy bylo automatické vytváření programů v jazyce Lisp [Koza, kapitola 3], v praxi je genetické programování nejužitečnější pro hledání funkce, odpovídající zadanému průběhu [popsáno již v Koza, str. 37]. Do stromů lze snadno přepsat výrazy, zapsané v polské notaci (prefix notation), nebo v polské notaci s obráceným pořadím zadávaných dat (obrácená polská notace, postfix notation). Pro klasicky zapsané funkce (infix notation) je třeba provést transformaci, často se tak děje právě postupným sestavováním stromu funkce.

*“When **Polish notation** (Polish logician Jan Łukasiewicz invented this in 1924) is used as syntax for mathematical expressions by interpreters of programming languages, it is readily parsed into **abstract syntax trees** and can, in fact, define a one-to-one representation for the same.” [Wikipedia, https://en.wikipedia.org/wiki/Polish_notation, 15.8.2016].*

V obrácené polské notaci (postfix notation) se dříve programovaly kalkulačky firmy Hewlett-Packard (HP-35, HP-41C), naposledy asi před dvaceti lety. Nepřehledný zápis má tu výhodu, že nepotřebuje prioritu operací a závorky. Naopak, argumenty se ukládají do jakéhosi zásobníku. Kalkulačka má tlačítko „Enter“ a typické je i tlačítko pro záměnu dvou hodnot na vrcholu zásobníku, „x⇆y“. Například pro sečtení dvou čísel zadáme první číslo (napíšeme, stiskneme „Enter“), pak stejně druhé, a pak stiskneme tlačítko „+“. Výsledek se objeví na zobrazovači a současně je na vrcholu zásobníku. Obdobný systém, s maximálně osmi čísly uloženými v zásobníku, je použit u matematické části procesorů Intel, poprvé byl použit u koprocesoru i8087 (kromě toho ale existují i instrukce pro operace přímo s čísly v paměti nebo dokonce i hlouběji v zásobníku koprocesoru [8086, str. 256]).

U polské notace (Prefix notation) se nejprve uvádí, jakou chceme provést operaci, a pak doplňujeme potřebný počet parametrů. Pro vytváření stromů je tento způsob zápisu vhodnější, protože víme, kolik větví může z daného uzlu vycházet (pokud nějaká). To vedlo k použití tohoto způsobu zápisu u genetického programování (např. [AField, str. 11]).

3.1.2 Příklad zápisu/přepisu z infixové notace

Mějme funkci vyjádřenu jako

$f(x, y) = \sin x \cos y + \cos x \sin y$

Skutečné provedení výpočtu zahrnuje násobení, které se v matematických vzorcích nezapisuje. Pro přehlednost doplníme ještě závorky, kterými vyznačíme priority:

(sin x*cos y) + (cos x*sin y)

Poslední operace se zapíše první. Je jí operace součet, která má dva parametry.

Prvním (i druhým) parametrem součtu je součin. Bezprostředně po příkazu součtu tedy bude následovat příkaz k součinu, následovaný ovšem nejprve svými parametry. Součin má opět dva parametry.

Prvním parametrem součinu je výsledek funkce sinus.

Zápis tedy bude vypadat (rozlišeno závorkami):

+(*(sin(x),cos(y)),*(cos(x),sin(y)))

Počítač závorky a oddělovací čárky nepotřebuje.

Genetické programování je základní částí této práce. Podrobný popis použitých metod bude uveden až u praktických řešení, jak u výukového programu, tak řešení zadaného problému, abych se vyhnul zbytečným duplicitám.

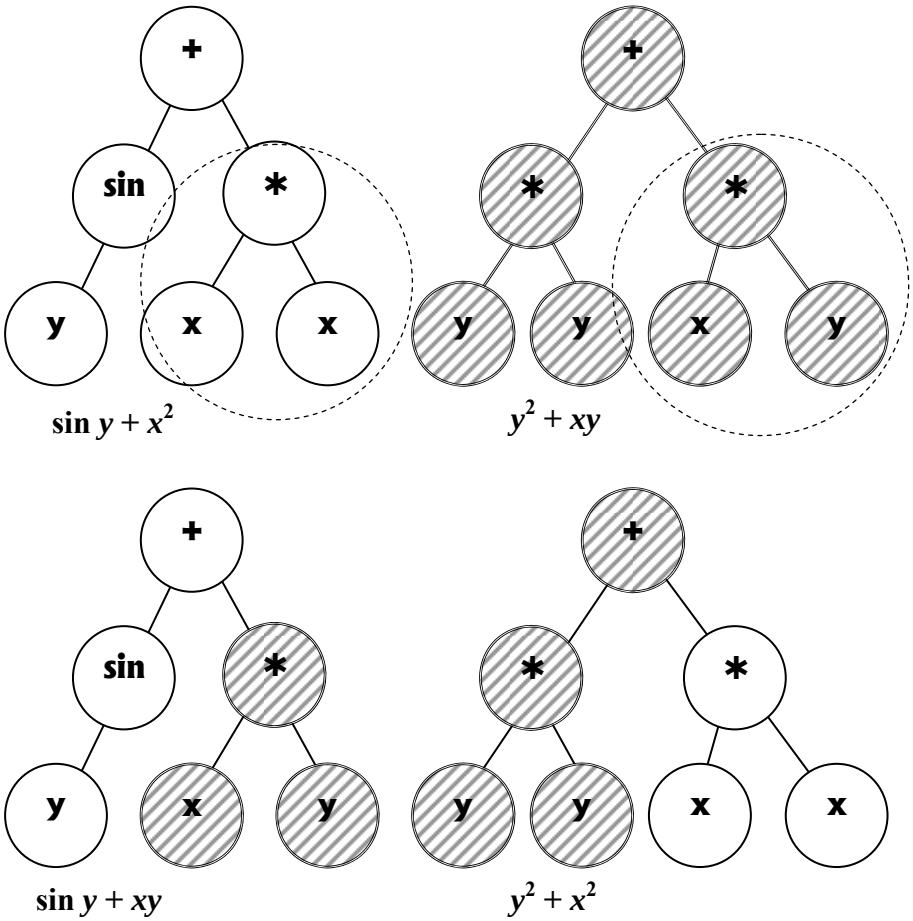
3.1.3 Křížení

Křížení podle [Koza, str. 12] představuje odtržení větve stromu a připojení do jiného stromu, resp. častěji záměnu větví mezi stromy (každý strom představuje jedno možné řešení problému): V nějakém místě stromu odřízneme jednu větev a místo ní připojíme větev z jiného stromu. To druhého stromu naopak připojíme prvně odříznutou větev [GP, str. 123]. Takovéto křížení pak u stromů označujeme jako symetrické, i když na rozdíl od genetických algoritmů se zde nezachovává velikost jednotlivých genomů. Ačkoli v součtu je délka zachována, zpravidla vznikne jeden větší a jeden menší potomek. Typickou vlastností genetického programování je, že větší strom (často současně s větším počtem nastavitelných parametrů, viz kapitola 6) snáze vyhoví účelové funkci, a průměrná délka genomu v populaci pak trvale narůstá (blowing). Pokud se jedná o nefunkční části genomu, mluvíme o intronech [GP, str. 181-199] (termín byl převzat z biologické genetiky).

Připomínám, že ač to tak na obrázcích nevypadá, při křížení není rozdíl mezi levou a pravou větví, pokud z uzlu vedou dvě (nebo dokonce více), tedy přesněji můžeme z jednoho podstromu vzít pravou větev a nahradit ji levou větví z jiného podstromu. Při vyčíslení hodnoty funkce odčítání a dělení zde později rozdíl bude.

Bývá zvykem pracovat s kopiemi jedinců, rodiče zůstávají beze změny a generují se dva potomci. Jinak by nebylo možné, aby se úspěšnější jedinec použil jako zdroj genomu víckrát. Po vygenerování celé generace se buď předchozí generace, nebo lépe po seřazení vhodný počet jedinců smaže, aby velikost populace zůstala zachována (neplatí v případě turnajové selekce, kdy je možné pracovat s konstantní velikostí populace).

„Kontext zachovávající“ křížení (např. [AField, str. 45], [GP, str. 161], [D'haeseleer]) se snaží hledat k výměně podobné podstromy. Protože se tím snižuje náhodnost výběru, nejsem příznivcem této metody.



Obr. 3.1 Ukázka symetrického křížení v genetickém programování. Použito v [Hlaváč].

3.1.4 Účelová funkce

V programu hledáme funkci, která co nejlépe proloží body, dané měřením. Ve standardní regresní analýze je tvar funkce zadaný a hledáme její parametry. To můžeme popsat vztahem

$$Y \approx f(X, \beta) \tag{3.1}$$

kde f je zadaná funkce a β je vektor hledaných parametrů. X je vektor nezávislých proměnných, Y je vektor v těchto bodech změřených dat.

V našem případě (x,y nezávislé proměnné, z závislá) by množina zadaných bodů šla popsat jako

$$\mathbf{M} = \{x_i, y_i, z_i; i = 1 \ldots n\} \tag{3.2}$$

nebo například jako výsledek hypotetické funkce G

$$z_i = \mathrm{G}(x_i, y_i) \tag{3.3}$$

Účelem programu je najít funkci f (3.1) takovou, aby její vzdálenost od zadaných bodů byla minimální. Ve výukovém případě popisovaném v kapitole 4.3 se tedy vektor parametrů β neuplatní, neznámou je naopak funkce. Hledání hodnot parametrů popisuje kapitola 6.

Účelová funkce (kriteriární funkce, *fitness*) [GP, str. 126 a 128; Koza, str. 10] je pak počítána jako odchylka generovaných a správných hodnot, například jako součet absolutních hodnot odchylek navrhované funkce ve všech bodech, kde byla reálná data získána měřením ([Koza 1995] to označuje jako *Minkowski distance*; podle Wikipedie je ovšem Minkovského vzdálenost definována s různou mocninou včetně druhé (zobecněná vzdálenost) a vhodnější je označení *Manhattan distance* (podle nemožnosti v pravoúhlé síti ulic jet kratší vzdáleností))

$$F_j = \sum_{i=1}^{n} \left| \mathrm{G}(x_i, y_i) - f_j(x_i, y_i) \right| = \sum_{i=1}^{n} \left| z_i - f_j(x_i, y_i) \right| \tag{3.4}$$

V literatuře se často uvádí součet kvadrátů, který zvýrazňuje největší odchylky a zanedbává menší nepřesnosti (odmocněním by vznikla euklidovská vzdálenost, [Koza 1995]),

$$F_j = \sum_{i=1}^{n} \left(\mathrm{G}(x_i, y_i) - f_j(x_i, y_i) \right)^2 = \sum_{i=1}^{n} \left(z_i - f_j(x_i, y_i) \right)^2 \tag{3.5}$$

[UI4, str. 130] uvádí součet kvadrátů odchylek jako jedinou možnost. Důležitý poznatek je, že hodnotu funkce musíme znát v dostatečném počtu bodů, ale další omezení není. Opravdu se může jednat o jednotlivé body a celé úseky průběhu nemusí být vůbec zadány, stejně jako u jakékoli jiné metody regresní analýzy. Lze tedy také očekávat, že rozumný minimální požadavek na počet bodů, ve kterých má být známa funkce, bude desetinásobkem VC-dimense [Vapnik] nalezené funkce, viz [Abu-Mostafa]. Problém je, že u genetického programování předem nevíme, kolik měnitelných parametrů bude mít hledaná funkce. Výběr z množiny funkcí, tedy každý prvek nalezené funkce, je třeba počítat jako další parametr.

S menší váhou je třeba účelovou funkci navýšit o délku generované funkce, aby se zabránilo nadměrnému růstu délky stromu (je vlastností genetického programování, že počet možných číselně správných řešení roste s nadbytečnou délkou funkce exponenciálně, viz [GP, str. 191]). Míru zohlednění složitosti funkce se doporučuje volit malou, aby nebylo znevýhodněno správnější řešení [UI4, str. 136, *složitost*] [AField, str. 101]. Na druhou stranu i malý rozdíl hodnoty účelová funkce způsobí zařazení na horší pozici v ruletové selekci a tím menší pravděpodobnost generování potomků. Pojem „malý“ je třeba chápat v poměru k variabilitě účelové funkce v populaci, vyjádřené například směrodatnou odchylkou, protože v konkrétním případě mohou být hodnoty účelové funkce číselně velké (podkapitola 5.14).

Pro další generace populace by bylo lépe, kdyby se účelová funkce zaměřila spíše na podobnost tvaru, než na součet odchylek. Ten lze vyjádřit například korelačním koeficientem [Koza 1995, str. 591]. Jeho výpočet je ovšem složitější – např. [Jarkovský] uvádí:

$$r_s = \frac{\sum_{i=1}^{n} x_{ri} y_{ri} - n\bar{x}_r \bar{y}_r}{(n-1) s_{x_r} s_{y_r}} \tag{3.6}$$

Ve výsledku je ovšem metoda genetického programování natolik robustní, že i při použití jednodušších metod výpočtu účelových funkcí, jako (3.4), (3.5), se dojde k srovnatelnému výsledku; větší počet generací je pak vyvážen jejich rychlejším výpočtem.

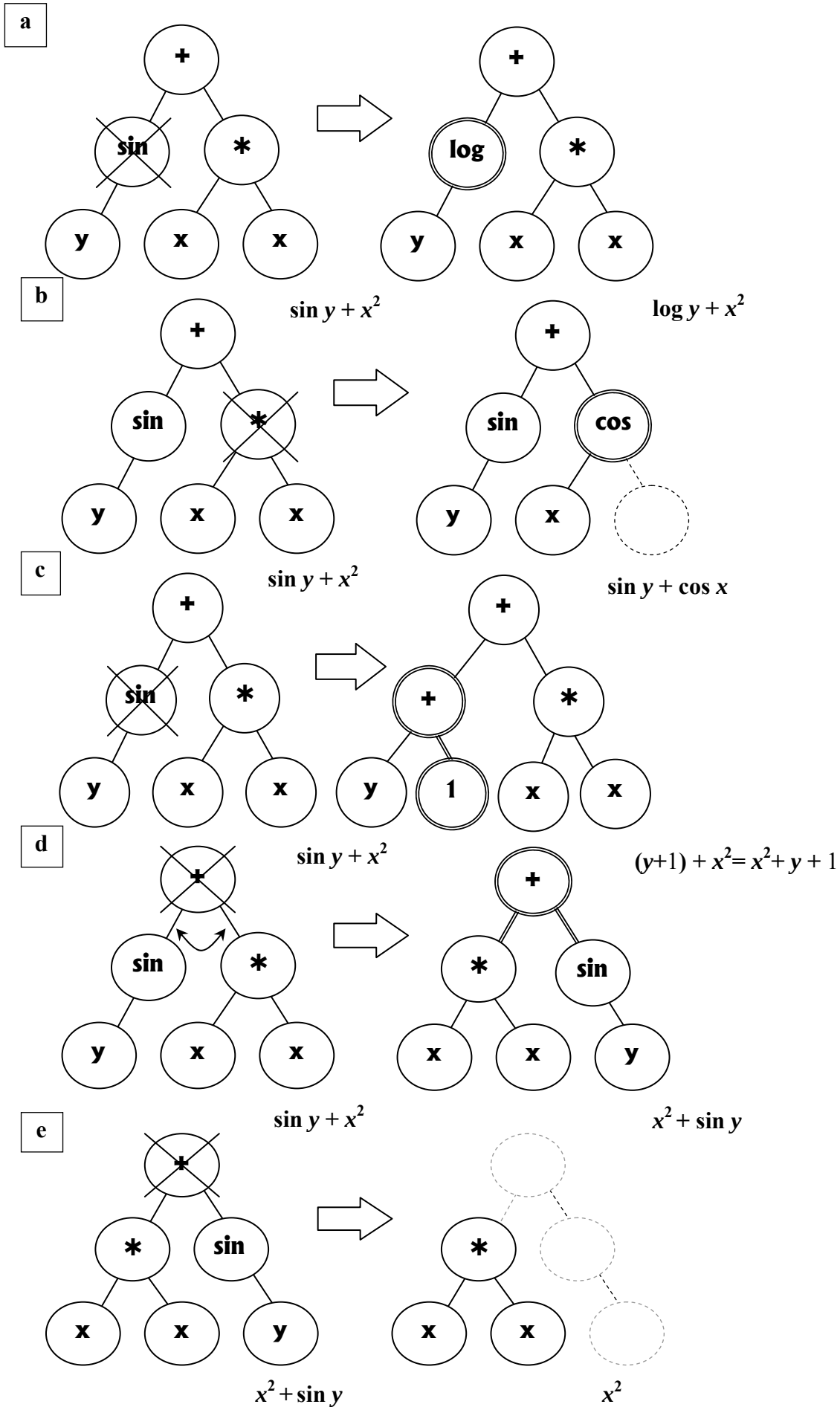
3.1.5 Mutace

Základními druhy mutace může být náhrada náhodného prvku jiným, doplnění nebo odmazání větve apod. Řada možných mutací je popsána v [AField, str. 42].
Ve výukovém programu (kap. 4.3) jsem nakonec použil následující mutace (viz obr. 3.2):

a. Jeden náhodně zvolený symbol je nahrazen symbolem ze stejné třídy (se stejnou aritou).
b. Jeden symbol je nahrazen symbolem s menší aritou, přebytečná větev je odstraněna.
c. Jeden symbol je nahrazen symbolem s vyšší aritou, chybějící větev je náhodně dogenerována.
d. Je vyhledán symbol s aritou 2 a je-li nalezen (např. sčítání), větve jsou prohozeny.

Protože byly problémy se zamrzáním části genomu, který reprezentuje kořen stromu (je na začátku, a do potomků se přenáší opět jen na začátek a beze změn), zařadil jsem dodatečně mutaci, která odřízne začátek stromu. Ačkoli pravděpodobnost dílčích mutací je malá, daný problém jsem tím částečně odstranil. Zůstává problém, že mutace nemohou změnit elitního jedince, ale zaměřují se naopak na konec populace. K rozšíření „genů“ do populace tak dochází až po několika cyklech, pokud mutovaní jedinci přežijí v populaci. Řešení tohoto dílčího problému by zřejmě vedlo na hybridní model (část populace mutovat, u části výsledek mutace přidávat na konec populace jako jednopohlavní křížení – klonování; v prvním případě přednostně volit jako zasažené horší jedince, v druhém jako zdroj naopak lepší).

V prezentované aplikaci pro vyhledání funkce o dvou proměnných (kapitoly 5, 6) je nakonec možné použít obojí přístup k mutacím, spolu s nastavením dalších parametrů. Současně jsou dostupné další mutace, například variace konstant (viz kapitola 6.6).



Obr. 3.2 Typy mutací, použité v ukázkovém (výukovém) programu (kapitola 4.3)

3.2 Genetické programování s lineárním genomem

Existují další varianty genetického programování. Například, pokud je algoritmus popsaný obdobně, jako strojový kód v počítači (častěji algoritmus zapsaný pseudokódy nebo blokovými operacemi, například žebříčkový (ladder) diagram u programovatelných automatů), může se za pomoci vhodně definovaných operací mutace a křížení zpracovávat pomocí metod, obvyklých u genetických algoritmů. Mluvíme pak o genetickém programování s lineárním genomem. Pro řešení úlohy „hledání funkce zadané průběhem“ se ovšem nehodí, takže je zde zmiňuji jen pro úplnost.

Zajímavou aplikací, kterou bych zařadil do genetických algoritmů obecně, ale autor ji řadí do genetického programování, je návrh elektronických obvodů [Koza 1997], [Koza 1998]. Zápis genomu je opět lineární.

3.3 Gramatická evoluce

Gramatická evoluce [EVT, str. 257] umožňuje generovat programy, zapsané v programovacích jazycích, vycházejících z Backus-Naurovy formy [UI4, str. 148] (což je ovšem výlučně Algol; u ostatních lze zpracovávat ty části, které takto popsatelné jsou; to jsou popisy algoritmů bez deklarace a do jisté míry použití objektů a bez vstupu a výstupu dat, zejména komunikace s uživatelem). Genom je zde tvořen číselným zápisem, kdy genotyp je na fenotyp převáděn pomocí gramatiky (odtud gramatická evoluce, i když gramatika je konstantní a v průběhu výpočtu se nevyvíjí). V základní verzi se z každého čísla v genomu jedince vezme zbytek po dělení daným počtem možností, podle toho, co gramatika v daném místě (stavu kódování jedince) umožňuje. Aplikací pravidla je samozřejmě zařazení dalšího prvku, jehož vytvoření je popsáno v jiném místě gramatiky (rekurzivně; podle počtu možností určíme zbytek po dělení aktuálního genu, pokud se nejednalo o terminální symbol).

Problémem zůstává účelová (fitness) funkce – jak lze posoudit, do jaké míry je nově zmutovaný program správný. Úseky programu lze takto optimalizovat, pokud dokážeme porovnat vstupy a výstupy, například u procedur. Ale porovnáváme jen správná řešení (vrací správná data), těžko popisujeme lepší a horší řešení, když ani jedno z nich *není správné*. Problém vede často na návrh „ad-hoc heuristiky“.

BNF Examples from Java

<primitive type> ::= boolean
<primitive type> ::= char

Abbreviated

<primitive type> ::= boolean | char | byte | short | int | long | float | ...
<argument list> ::= <expression> | <argument list> , <expression>
<selection statement> ::=
  if (<expression>) <statement>
| if (<expression>) <statement> else <statement>
| switch (<expression>) <block>
<method declaration> ::=
  <modifiers> <type specifier> <method declarator>
  throws <method body>
| <modifiers> <type specifier> <method declarator> <method body>
| <type specifier> <method declarator> throws <method body>
| <type specifier> <method declarator> <method body>

Obr. 3.3 Ukázka Backus-Naurovy formy popisu syntaxe [Kent, str. 23]

3.4 Současný stav problematiky symbolické regrese s užitím genetického programování

Již publikace Genetic Programming, an Introduction [GP] z roku 1998 uvádí řadu příkladů praktického použití genetického programování, zejména v kapitole 12, s výčtem v tabulce na stránkách 342 až 345. Vychází se zde z širšího pojetí genetického programování, zejména z genetického programování s lineárním genomem, kdy za genetické programování je považováno cokoli, co hledá funkční popis řešení ve tvaru připomínajícím program. V řadě případů by šlo mluvit spíše o genetických algoritmech obecně.

[Afiled, str. 111] uvádí souhrn podmínek, za kterých je vhodné genetické programování použít, mimo jiné:

- hledané řešení obsahuje proměnné, ale jejich vzájemný vztah a ovlivňování nejsou známy,
- nalezení tvaru a velikosti řešení je součástí problému (jinak řešení spadá do GA),
- je k dispozici dostatečné množství (naměřených) dat,
- správnost a míra vhodnosti (z hlediska GA feasibility/fitness) řešení je snadno vyhodnotitelná, zatímco nalezení řešení běžnými metodami (přímo) je obtížné nebo neznámé,
- běžné matematické metody neumožňují najít vhodné řešení,
- přibližné řešení je postačující, resp. nalezení skutečného optima problému se ani neočekává,
- i malé zlepšení (účelové funkce) je významné, a také snadno vyčíslitelné.

V řadě případů, zmíněných dále v [Afiled], jde ale spíše o hledání slovního/textového popisu problému, který bych také zařadil do genetických algoritmů. Zejména, pokud popis nepoužívá vyčíslované konstanty – přírodní ekvivalent evoluce také nepracuje s konečnou délkou genomu, může narůstat i zkracovat se s průběhem evoluce.

Symbolická regrese je zmíněna přímo mezi základními problémy nejen v [Afiled, str. 114], ale i v [Koza]. Právě profesor Koza ale trval na názoru, že genetické programování žádné vyčíslování konstant nepotřebuje, ale nalezne si je samo. Myslím, že to je jeden z důvodů (spolu s velkou výpočetní náročností), proč není symbolická regrese metodou genetického programování stále příliš rozšířena. Hledáním po internetu jsem našel vesměs odkazy, kdy autor touto metodou (nebo nějakou kombinací) nalezl nějaké řešení, ale hlavním přínosem bylo, že řešení publikoval. Symbolická regrese metodou genetického programování tak stále není rutinně využívána (pro řešení běžných problémů v praxi). Žádné nové příspěvky o využití genetického programování například nejsou na stránkách časopisu Swarm and Evolutionary Computation (http://www.sciencedirect.com/science/journal/22106502/32, zde vychází publikace konference Mendel), ani na konferenci GECCO (Genetic and Evolutionary Computation Conference, příspěvky např. http://sig.sigevo.org/index.html/tiki-index.php?page=TOC+GECCO+2016)

Jako příklad open-source knihoven pro genetické programování lze uvést například řešení v jazyce Python, dostupné na stránkách projektu [DEAP]. Podle popisu umožňují snadné řešení symbolické regrese, ale bez konstant (například trigonometrické ekvivalence).

Řešení pro jazyk C++ je dostupné jako příloha [AField, str. 151], appendix B, pod označením TinyGP. Na internetové zdroje s knihovnami pro C++ odkazuje ostatně již [GP, str. 393]. Na programy pro genetické programování, jako i na knihovny pro C++, perl a samozřejmě Lisp, odkazuje stránka profesora Kozy, věnovaná genetickému programování (odkazy na adrese http://www.genetic-programming.org/gpftpsite.html, ale poslední změna

2003). Nedostatek programových řešení tedy zřejmě není hlavním problémem bránícím použití genetického programování v praxi.

3.4.1 Genetické programování s vyčíslováním hodnot konstant

Z publikací dostupných na internetu například [Lahodiuk] prezentuje takřka výukový text o genetickém programování, kde je vyčíslení koeficientů prováděno metodou genetických algoritmů. Nabízí programové řešení v jazyce Java (stažitelné jako knihovna).

V [Barmpalexisa] je popsána aplikace symbolické regrese na hledání závislostí při účincích jednotlivých léčivých látek. Hodnoty koeficientů jsou hledány v omezeném rozsahu ±127 a s krokem 1 (celá čísla), takže probíhá jen hlavní cyklus genetického programování, jak to zamýšlel profesor Koza (konstanty jsou generovány jako terminální symboly náhodně v uvedeném rozsahu). Přesto si program kombinací dvou konstant snadno může evolucí vytvořit potřebné koeficienty (např. vzájemným násobením, dělením).

[Icke] popisuje vyhledávání hodnot koeficientů pomocí metody, odvozené od regresní analýzy. Potenciálně nejlepší reference, kterou jsem našel, využívá maticový zápis a lineární regresi na vyhledání požadovaných hodnot konstant, řešení se předpokládá ve formě polynomu o více proměnných. Praktická aplikace není popsána. Odvolává se na [FFX].

[FFX] hledá funkci v pevném formátu, takže se nejedná o genetické programování, ale autor se na něj průběžně odvolává. Jak již bylo naznačeno, hledá hodnoty koeficientů v zásadě metodou lineární regrese, kterou je ovšem na nelineární problém třeba použít opakovaně (iterační princip).

V ČR se problematikou zabýval i prof. Ivan Zelinka (Fakulta elektrotechniky a informatiky VŠB-TUO, Fakulta aplikované informatiky UTB), lze dohledat použití algoritmu SOMA pro výpočet koeficientů. Pro genetické programování v této formě ale používá označení *Analytické programování* [Zelinka1]. Řadu článků publikoval prof. Pavel Ošmera, ale kromě kapitoly 11.1 v [EVT] jsou obtížně dostupné. Obdobné shrnutí obsahuje podkapitola 35.6 v [Ošmera, str. 523 knihy], která je zřejmě jen shrnutím uvedených publikací. Použitý postup s dvouúrovňovou optimalizací používá pro vyčíslení konstant diferenciální evoluci. Podle kapitoly 35.2 tamtéž je výhodou diferenciální evoluce, že nepotřebuje nastavovat parametry a pracuje bez nutnosti mutací. Celková metoda je označena jako GDE (*gramatická diferenciální evoluce*, vychází z *gramatické evoluce*).

Tato práce (následující kapitoly) vychází především z prací doc. Brandejského (Fakulta dopravní ČVUT), které hodnoty konstant vyčíslují v zásadě metodou genetických algoritmů (aplikována varianta, známá pod označením *evoluční strategie*) [BT4], [BT5], viz kapitola 6.7.

Ve všech uvedených případech zpřesňování hodnot konstant jsou konstanty reprezentovány jako terminální symboly (listy) při zápise funkce stromem. V tomto směru se jejich implementace, popsaná dále v této práci, liší.

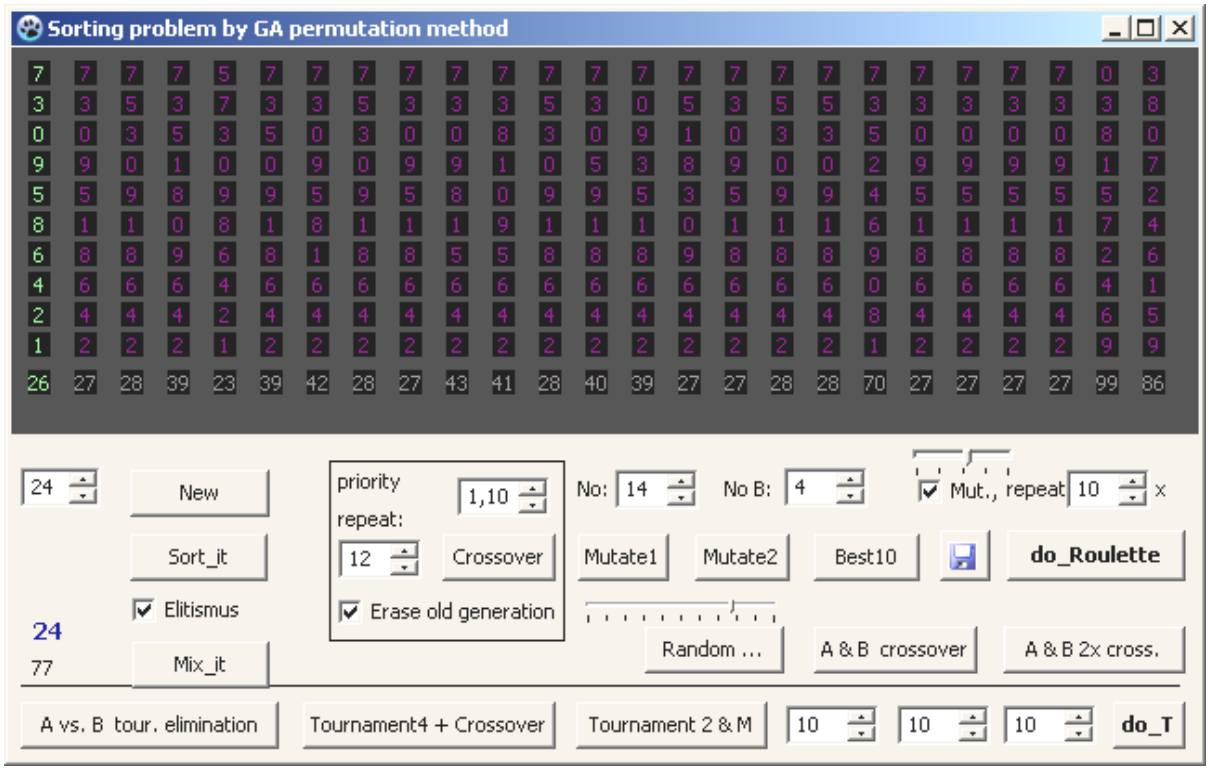
4. Genetické algoritmy – testování

Následující podkapitola demonstruje výsledky, kterých může tato metoda dosáhnout, na programech, které jsem vytvořil pro výukové účely.

4.1 Hledání řešení permutačního problému (ga.exe)

Když jsem řešil, na čem studentům chování genetických algoritmů předvést, dospěl jsem k závěru, že hlavní je, aby každý na první pohled viděl, jak daleko jsme k řešení. Jako problém jsem proto použil úlohu seřadit kombinaci čísel od 0 do 9, ale sestupně. Pro tuto úlohu by se samozřejmě tato metoda nikdy nepoužila. Samotný algoritmus navíc obsahuje v každém kroku ruletové selekce třídění (správně „řadicí algoritmus“). Z druhé strany je předem jasné správné řešení a studenti mohou sledovat, jak se mu program v jednotlivých krocích blíží.

Program umožňuje pracovat s maximálně 250 jedinci, ale zobrazuje jen 24 prvních (podle umístění v paměti, ne nutně nejlepších – pro to by bylo třeba použít tlačítko pro třídění) + případného elitního jedince. Velikost populace v tomto rozsahu lze zvolit, ale je třeba pamatovat, že při generování nové generace je nejprve tato vygenerována, a pak stará smazána, takže je třeba počítat s rezervou pro tento druh operací (lze tedy pracovat s maximálně 125 jedinci). Platí pro ruletovou selekci – turnajová umožňuje při použití navrhovaných funkcí držet velikost populace konstantní, viz dále popis funkce programu.



Obr. 4.1 Běh programu ga.exe. Cílem programu se najít jedince, který bude mít genotyp=fenotyp od devíti k nule. Ve sloupečku vždy zapsán genotyp a pod ním hodnota účelové funkce. Tmavé barvy se při výběru jedince jako zdroje pro dalšího potomka vždy o něco zjasní, při mutaci obarví (na konci jsou strakaté). Zeleně je elitní prvek, pokud není, není na jeho místě vypsáno nic.

Fitness

Účelovou funkci jsem definoval tak, že za každé porušení sekvence (za menším číslem následuje větší) se počítá penalizace 10 bodů + pozice chyby (ta je 0 až 9, přesněji 1 až 9, protože se najde až o krok pozadu, takže první prvek jedince ((v následujícím textu je jednorozměrné pole o deseti prvcích nazýváno jedinec)), s indexem 0, nemůže být penalizován).

Generování.

Na začátku je nutné vygenerovat výchozí populaci, o zvoleném počtu jedinců. Pro generování jedince je nejvhodnější Fisher-Yatesův algoritmus [Alg-FY], spočívající v tom, že ze zbytku původně seřazeného jedince jsou náhodně vybírány hodnoty a umisťovány na začátku pole (opak insert-sortu). Každý prvek je generován samostatně. Rychlost a kvalita jsou závislé na algoritmu generování náhodných čísel, na kvalitě zde příliš nezáleží, měl by tedy být volen podle rychlosti.

Křížení

Program umožňuje přidat nového potomka ze dvou jedinců, jejichž čísla si vybereme. Bod křížení si vybrat nelze, je náhodný. Nový prvek vznikne okopírováním několika (min. jednoho, maximálně osmi) prvků z prvního rodiče a doplněním nevyužitých z druhého jedince, se zachováním jejich pořadí (použita „odškrtávací“ metoda). Také mohou vzniknout dva, kdy v druhém je úloha obou prvků zaměněna (v programu jsou tlačítka pro obě možnosti; druhá reprezentuje u tohoto problému symetrické křížení).

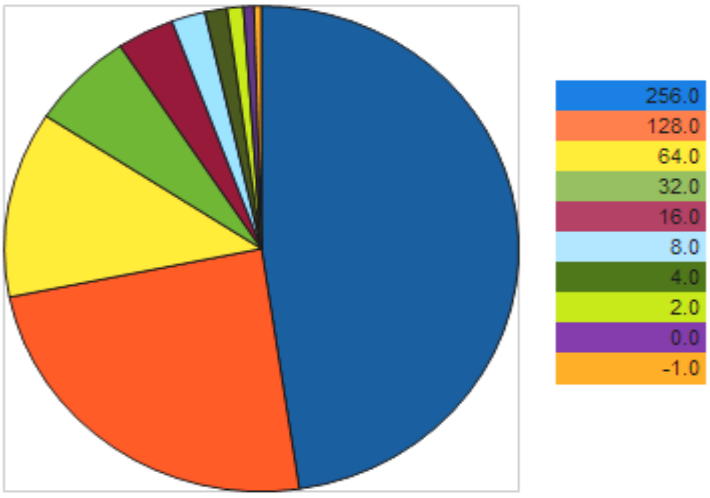
Mutace

Mutace jsou realizovány dvě; M1 představuje záměnu náhodných dvou prvků jedince. M2 přeskládá část jedince do opačného pořadí (mnohem drastičtější zásah).

Místo volení prvků k operaci zadáním čísla lze čísla jedinců vygenerovat náhodně. Aby byly přednostně generováni potomci lepších jedinců, je možné dalším tlačítkem stávající populaci seřadit.

Elitismus

Program umožňuje realizovat elitismus tím, že vybere nejlepší prvek a umístí ho mimo pole (přesněji na pozici nula). Toto řešení není formálně správně, protože daný prvek je pak v populaci dvakrát. Při tomto řešení ale je snadno možné od vyčleňování elitního jedince opět odstoupit. Elitní jedinec je ušetřen mutací a nepodléhá operacím, které staré prvky odstraňují. Jedině při operaci seřazení je nahrazen jiným, pokud se najde nějaký lepší. Nedostatkem v tomto ukázkovém programu je, že není ani zdrojem mutovaných jedinců. U takto jednoduše definovaného problému to není příliš na závadu, protože do mutací vstupují jeho potomci.



Obr. 4.2 Rozdělení pravděpodobností při ruletové selekci. Převzato z: Zepper, J.: Genetic algorithm. On-line: https://www.mql5.com/en/articles/55 (28.1.2015). Všimněte si, že velikosti výsečí neodpovídají tabulce.

Program je navržen tak, že až si studenti všechny funkce vyzkouší, mohou předvolit poměr M1:M2, pravděpodobnost mutací, počet generací a spustit algoritmus (stejně je následně možné testovat Tournament selection variantu). Obě varianty umožňují průběžné výsledky zapisovat do souboru na disku. Předpokládal jsem zpracování Excelem, takže jsem jako nyní velmi moderní zvolil formát csv (comma separated values), kdy jsou čísla oddělena středníky. Nativní formát by samozřejmě byl text s čísly oddělenými tabelátory (možné pozdější doplnění programu).

Evoluční tlak[2]

Výrazným problémem je potřeba zvýhodnit lepší jedince. Představa roulette-selection, že lepším jedincům bude přidělena větší pravděpodobnost výběru (zatímco v ruletě je pravděpodobnost padnutí všech čísel absolutně stejná), je v literatuře často ilustrována koláčovým grafem, obdobně jako na obr. 4.2. Obdobně vysoké selekční tlaky jsou popsány v [UI3, str. 134-136].

Ponechme stranou, že čísla v tabulce u obr. 4.2 neobsahují jedničku (logicky má být mezi nulou a dvojkou) a koláčový graf neodpovídá tabulce (záporné číslo). Při použití této distribuce by více jak polovina nových genů pocházela od jediného rodiče - přesněji, při volbě prvního jedince se tak stane s pravděpodobností 0,5, při druhém s podmíněnou pravděpodobností 0,5, takže nejlepší prvek by se vybral s pravděpodobností

$$p_1 = p_{1a} + p_{1b}(1-p_{1a}) = 0{,}5 + 0{,}5 \,.\, (1-0{,}5) = 0{,}75 \qquad (4.1)$$

Touto selekcí je definován evoluční tlak, u geometrické posloupnosti definovaný jako poměr mezi dvěma po sobě jdoucími jedinci:

$$p_2 = p_{2a} + p_{2b}(1-p_{2a}) = 0{,}25 + 0{,}25 \,.\, (1-0{,}25) = 0{,}4375$$
$$q = \frac{p_1}{p_2} \doteq 1{,}71 \qquad (4.2)$$

Protože je nezbytné držet velkou genetickou diverzitu populace, nelze podobnou nerovnost připustit. Řadu, která ve výsledku určuje pravděpodobnost výběru konkrétního jedince, je třeba zvolit mnohem pomaleji klesající.

[2] Někdy je za evoluční tlak označována pravděpodobnost výběru jedince pro vytvoření další generace, což oproti zde uváděné definici dále zahrnuje množství nově generovaných jedinců (mutací, křížením) dělené velikostí populace, ze které vybíráme.

Řadou může být jakákoli klesající posloupnost, například aritmetická [UI3, str. 135]. V programu jsem se ovšem držel myšlenky geometrické posloupnosti:

$$a_n = a_{n-1} q \tag{4.3}$$

Pro použití v genetické evoluci je třeba hodnoty relativní pravděpodobnosti pro jednotlivé jedince generovat opačně, od nejhoršího (s relativní hodnotou „1“) po nejlepšího, s kvocientem q. Počítač generuje náhodná čísla v rovnoměrném rozdělení, ale nikoli na intervalu <0, 1), ale jako *celé* číslo na zvoleném intervalu. Jako interval se zde nabízí součet řady [Rektorys, str. 47]:

$$s_n = a_1 \frac{q^n - 1}{q - 1} \tag{4.4}$$

Člen a_n zvolíme =1. Počet členů řady je dána aktuálním počtem jedinců v populaci. Dále je ovšem třeba ošetřit, že počítač by generoval jen celá čísla; číslo může být generováno v rozsahu například 10x větším a následně poděleno. Při použití některých knihoven počítač generuje číslo v rozsahu výsledné proměnné, například *unsigned long* (nyní integer, 2^{32}, tedy 0 až 4294967296), a výsledek je třeba podělit poměrem požadovaného rozsahu a tohoto čísla, a to tak, aby výsledek byl reálné číslo, například typ *real* nebo *double*).

Koeficient q lze v programu nastavit, studenti si mohou zvolit různé hodnoty, od 0,01 do 10. Následující tabulka ukazuje, jak při několika náhodných koeficientech stoupá hodnota q^n, která představuje relativní pravděpodobnosti výběru a určuje evoluční tlak na jedince. V našem případě potřebujeme, aby první prvek byl generován s největší pravděpodobností. Toho by bylo možné dosáhnout použitím převrácené hodnoty. Představě genetické evoluce ale lépe vyhovuje, pokud rozdíl v pravděpodobnosti vytváření potomků je největší mezi nejlepšími jedinci a nejmenší mezi těmi nejméně úspěšnými. Řešením je, že v programu jsou použita přímo tato čísla, ale jsou populaci přiřazena od konce (nejlepší prvek má poslední, tedy největší relativní pravděpodobnost být vybrán).

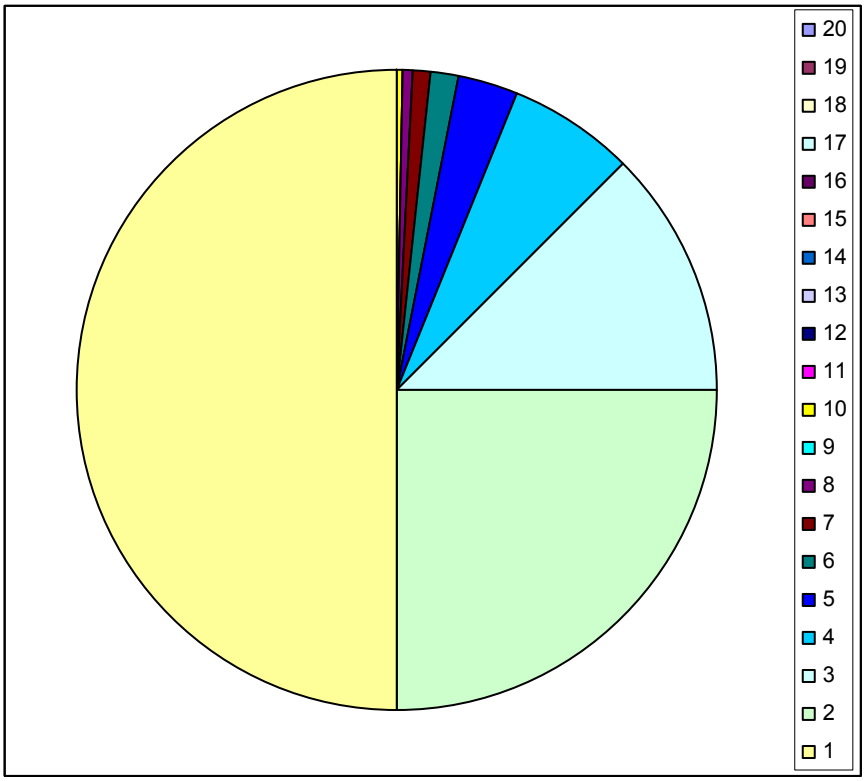
Tabulka 4.1 Výpočet členů geometrické posloupnosti

pořadí	q = 2	q = 1,3	q = 1,1	q = 1,01
1	1	1	1	1
2	2	1,3	1,1	1,01
3	4	1,69	1,21	1,0201
4	8	2,197	1,331	1,030301
5	16	2,8561	1,4641	1,040604
6	32	3,71293	1,61051	1,05101
7	64	4,826809	1,771561	1,06152
8	128	6,274852	1,948717	1,072135
9	256	8,157307	2,143589	1,082857
10	512	10,6045	2,357948	1,093685
11	1024	13,78585	2,593742	1,104622
12	2048	17,9216	2,853117	1,115668
13	4096	23,29809	3,138428	1,126825
14	8192	30,28751	3,452271	1,138093
15	16384	39,37376	3,797498	1,149474
16	32768	51,18589	4,177248	1,160969
17	65536	66,54166	4,594973	1,172579
18	131072	86,50416	5,05447	1,184304
19	262144	112,4554	5,559917	1,196147
20	524288	146,192	6,115909	1,208109

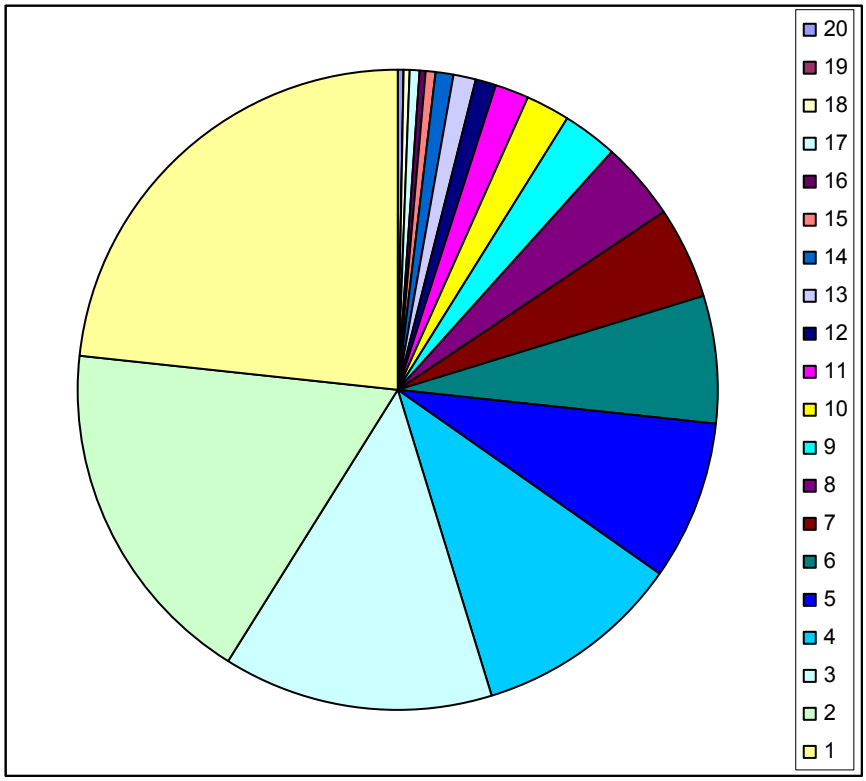
Tabulka popisuje prvních 20 prvků. Jistě bychom chtěli držet širší populaci. Z toho plyne vodítko pro výběr q. Popisovaný program ovšem neumožňuje nastavit hodnoty pod 1,01. Před tímto výpočtem jsem se domníval, že optimální hodnota (cena/výkon) by měla být okolo 1,1. Ukázky získaných dat (z několika běhů programu) na konci této podkapitoly.

Poznámka. Přepočítat tyto relativní pravděpodobnosti na skutečné by šlo tak, že by se vypočetl součet vygenerované řady a veškeré hodnoty by se jím podělily. Pro generování hodnoty s danou pravděpodobností to samozřejmě není třeba, použijí se odvozené vztahy a k výpočtu výše uvedené tabulky v programu nikdy nedojde.

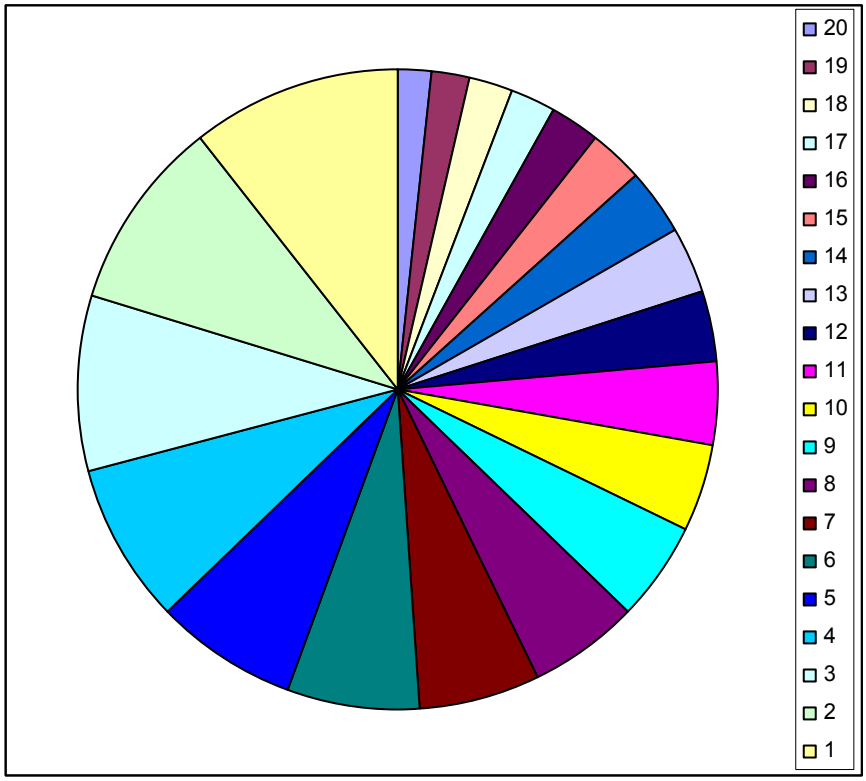
Přikládám ukázky koláčových grafů:



Obr. 4.3 Pravděpodobnosti volby (prvního rodiče) pro evoluční tlak q=2, hodnota obvyklá v literatuře



Obr. 4.4 Hodnota evolučního tlaku *q*=1,3 (další sloupeček předchozí tabulky)



Obr. 4.5 Evoluční tlak *q*=1,1, 20 prvků

Postup generování jedince pro páření s evolučním tlakem *q*:

1) spočítáme součet geometrické řady podle (4.4)

2) vygenerujeme *reálné* náhodné číslo (rovnoměrné rozdělení) v rozsahu, daném součtem s_n (ve vzorci (4.5) značeno jako y)

3) pořadí prvku x (odzadu) určíme tak, že spočítáme, kolik členů posloupnosti je třeba sečíst, abychom dostali součet, odpovídající vygenerovanému náhodnému číslu:

$$x = \frac{\ln(y(q-1)+1)}{\ln q} \tag{4.5}$$

4) pořadí prvku se ale počítá opačně:

$$i = n - x \tag{4.6}$$

5) vzoreček vrací reálné číslo, je jej třeba zaokrouhlit na celá čísla dolů. To předpokládá, že prvky jsou číslované od nuly (v programu platí, pokud se pracuje s tzv. elitismem, jinak se ještě přičítá jednička).

Turnajová selekce

Připomínám, že program obsahuje políčka pro výběr prvků A a B pro křížení a tlačítko, které tato políčka naplní náhodnou hodnotou (index v rozsahu aktuální populace).

Jako ukázku turnajové selekce program obsahuje následující tlačítka:

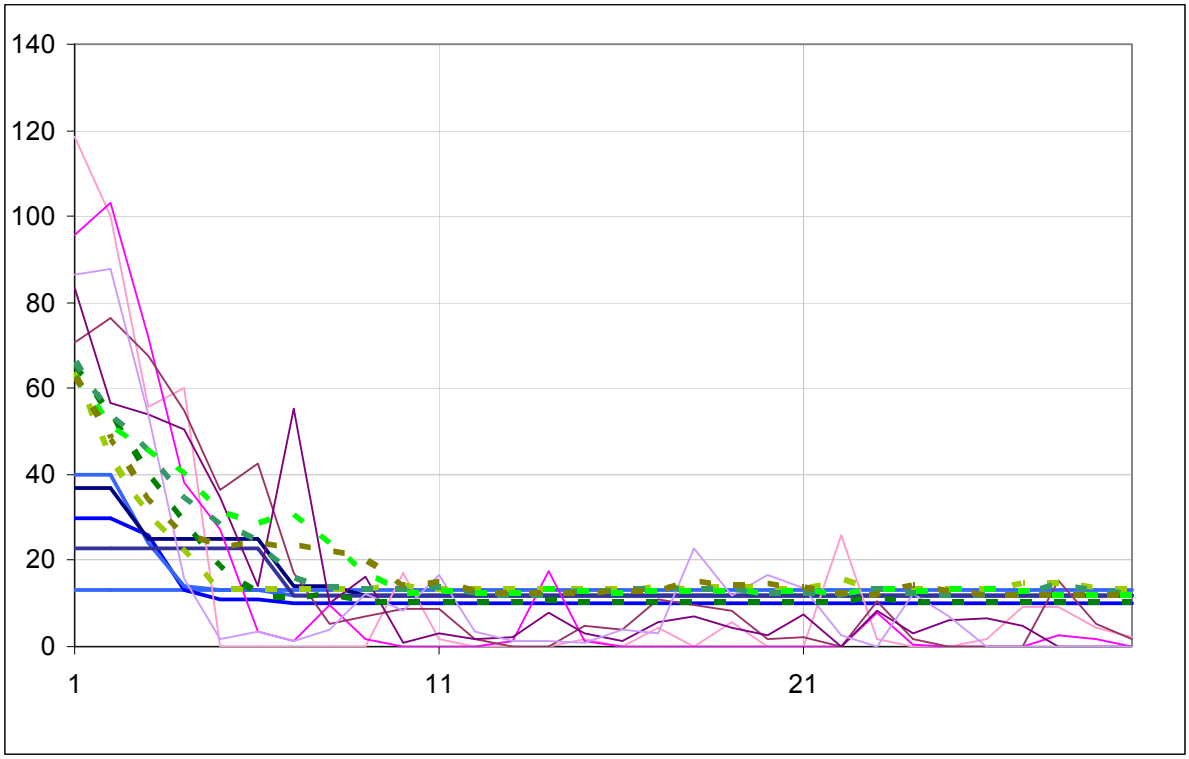
- eliminace horšího prvku z dvojice A a B (prvek je odstraněn z populace)
- výběr horšího prvku z dvojice A a B a jeho následné nahrazení mutací lepšího z prvků (metoda M2)
- vygenerování dvou dvojic náhodných prvků; z každé dvojice se horší prvek smaže a pak se křížením zbývajících prvků (které zůstávají v populaci) vygenerují dva nové.

Žádná z těchto možností nezničí nejlepší prvek. Připomínám, že turnajová selekce neví, který prvek je ten nejlepší (zachovává svého vítěze). Program to samozřejmě může po každém cyklu dopočítat a také to průběžně zobrazuje.

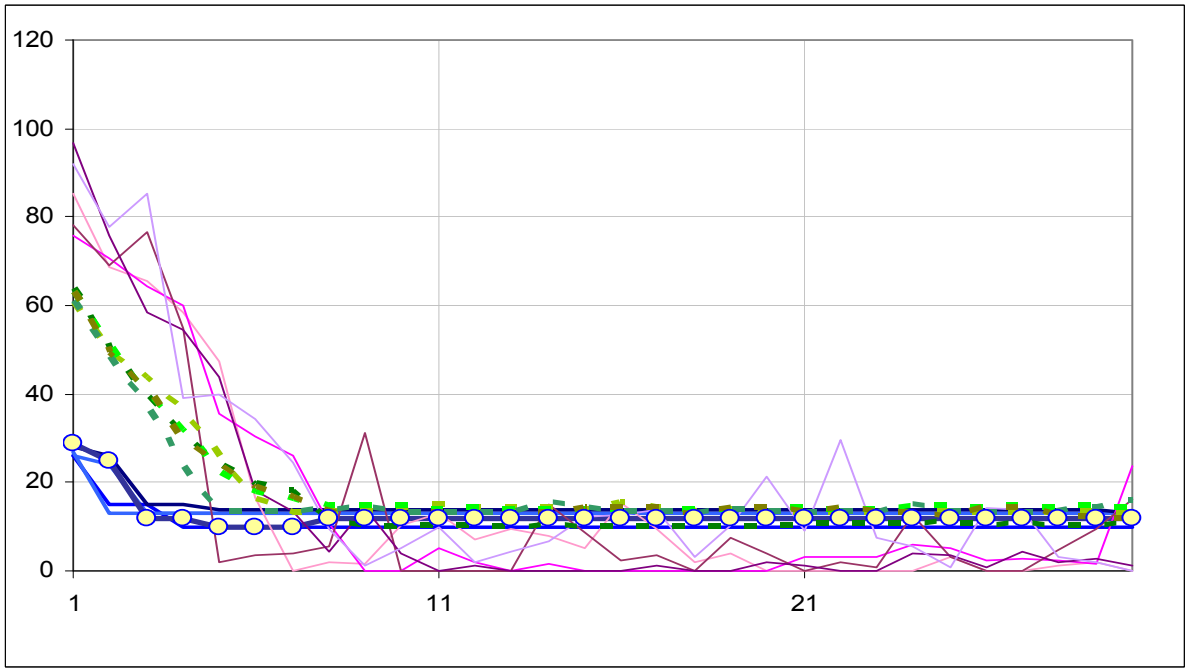
Jako ekvivalent ruletové selekce i v tournament módu (evoluce s využitím metod M2 a M4 turnajové selekce, popsaných výše) program umožňuje vícenásobné opakování, volitelně i se záznamem do souboru. Během ladění se ukázalo, že turnajová selekce ještě více než ruletová vede k rychlému zúžení genomu ([UI4, str. 167], uvádí opačný závěr, ale podotýkám, že tam popisované metody selekce jsou implementovány jinak, například postrádají elitismus, a u turnajové selekce je generována nová generace a stará mazána, zatímco v tomto programu je turnajová selekce „in-place“). Proto je nutné průběžně používat mutací. Rozdíl je v tom, že zatímco v ruletovém módu program využívá informaci o tom, nakolik monotónní je dosažená hodnota účelové funkce (a úměrně počtu stejně ohodnocených členů generuje počet mutací, ovládacím prvkem se nastavuje jen relativní intenzita), u turnajové selekce tato informace chybí a uživatel určuje přímo počet mutací „v generaci“. U turnajové selekce se pravidelně střídá několikrát provedená eliminace dvou prvků ze čtyř s následným doplněním potomky (počet provedení určí uživatel), několikrát provedená mutace (počet lze nastavit, druh mutace se nastavuje jako poměr M1:M2 v kroku pravděpodobnosti 0,1) a zápis na disk, byl-li zvolen. Počet cyklů lze nastavit, případně akci několikrát zopakovat, až se nalezne řešení.

Následující obrázky ukazují několik příkladů průběhu programu. V populaci je vždy 100 prvků, probíhá 30 generací. Základní volba byla: evoluční tlak q=1,10, pravděpodobnost mutací 50% (je třeba zdůraznit, že to je relativní pravděpodobnost, program ji zvyšuje/snižuje dle okamžité hodnoty variačního koeficientu, pokud není nulová), elitismus. Mění se jen u grafů uvedené parametry.

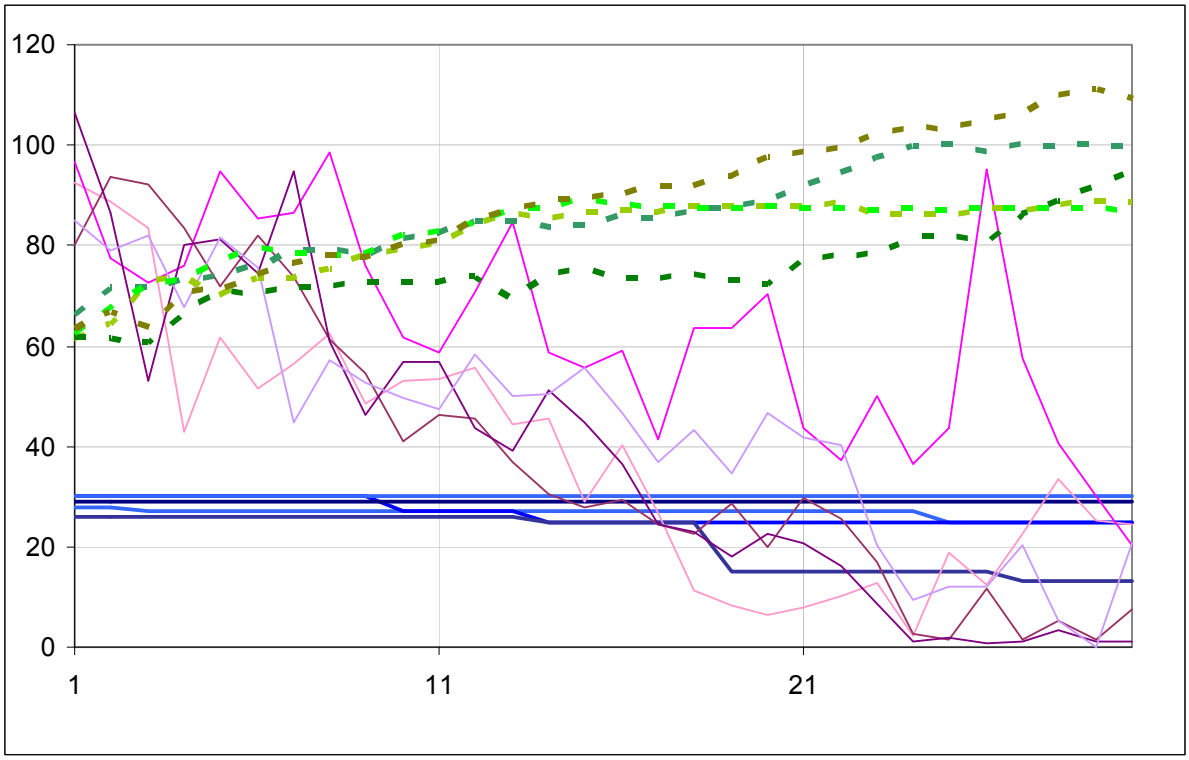
Hodnoty na svislé ose u obr. 4.6 až 4.15, 4.17, 4.20 a 4.21:
Zobrazení účelové funkce pro: (až na výjimky 5 běhů v obrázku)
fialová tenká – rozptyl, podělený dvěmi z důvodu zobrazení
modrá silná – nejlepší hodnota z populace
zelená přerušovaná – střední hodnota
Barevné značení platí pro všechny průběhy evoluce v této kapitole až do obrázku 4.21.
Vodorovná osa – průběh evoluce (zleva doprava – 30 generací), čísla uvádějí číslo generace.



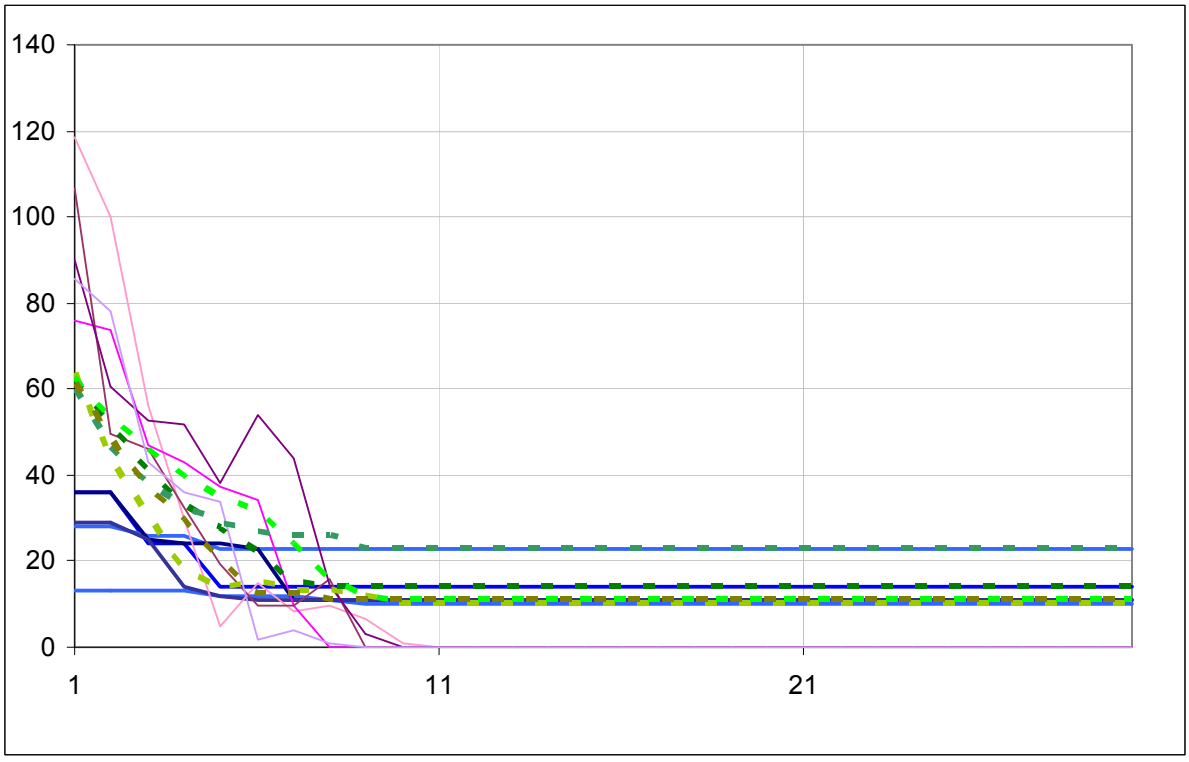
Obr. 4.6 Základní nastavení. Nejlepší výsledek 10.



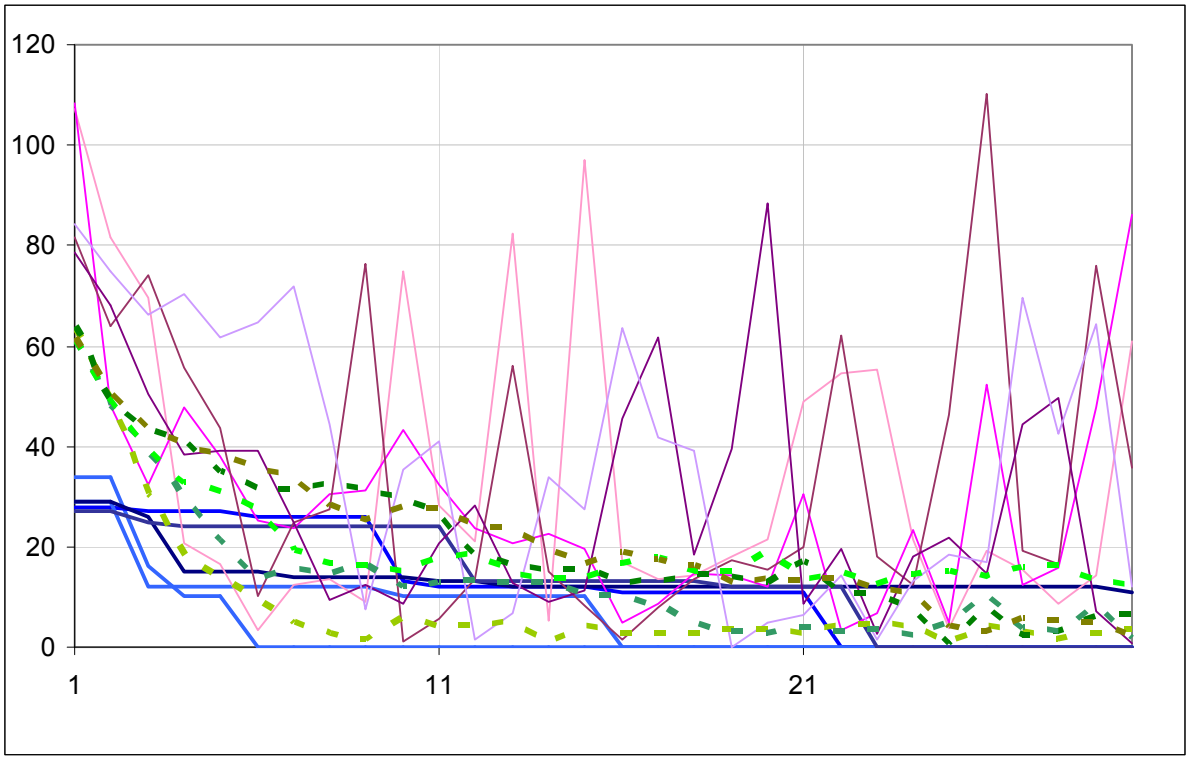
Obr. 4.7 Vypnutý elitismus. Žlutými body vyznačen průběh, při kterém došlo ke zhoršení fitness.



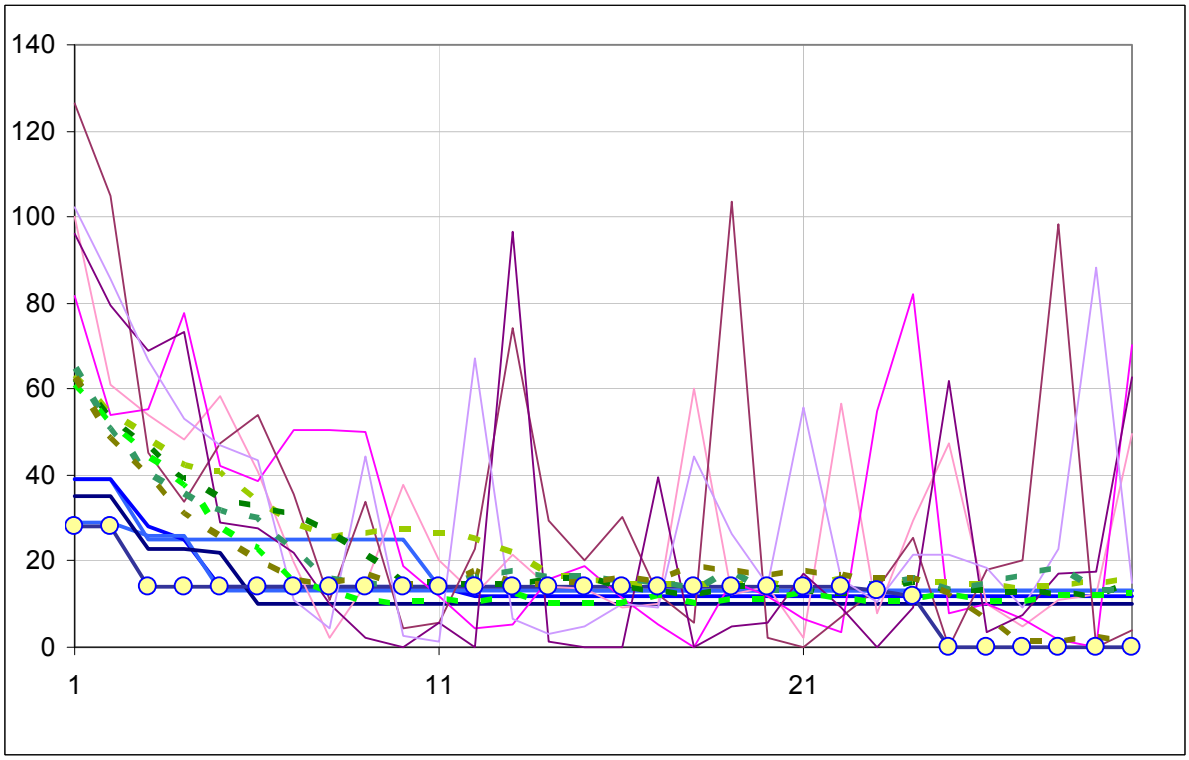
Obr. 4.8 Velmi malý evoluční tlak *q*=1,01. Nejlepší výsledek 13.



Obr. 4.9 Vypnuté mutace. Po 10 cyklech dojde k zamrznutí populace. Nejlepší výsledek 10.

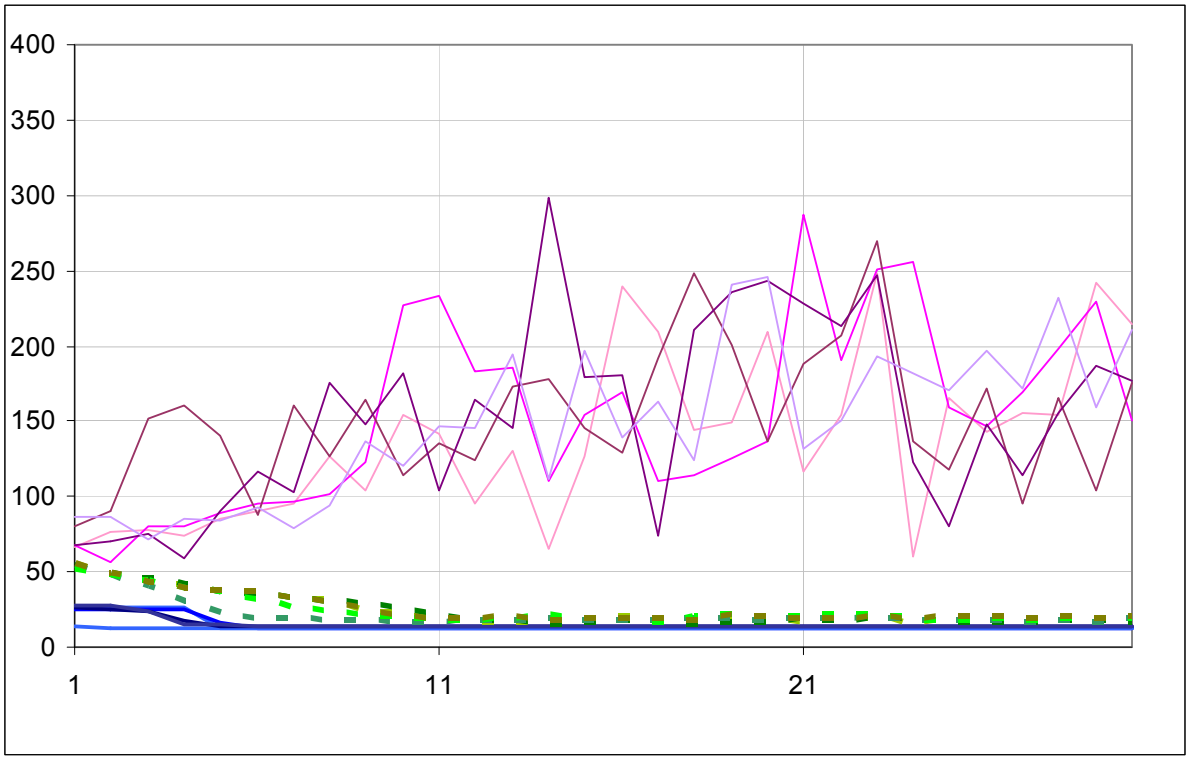


Obr. 4.10 Mutace 200% základní úrovně. Ve většině případů nalezeno řešení, nejrychleji v 6. kroku.

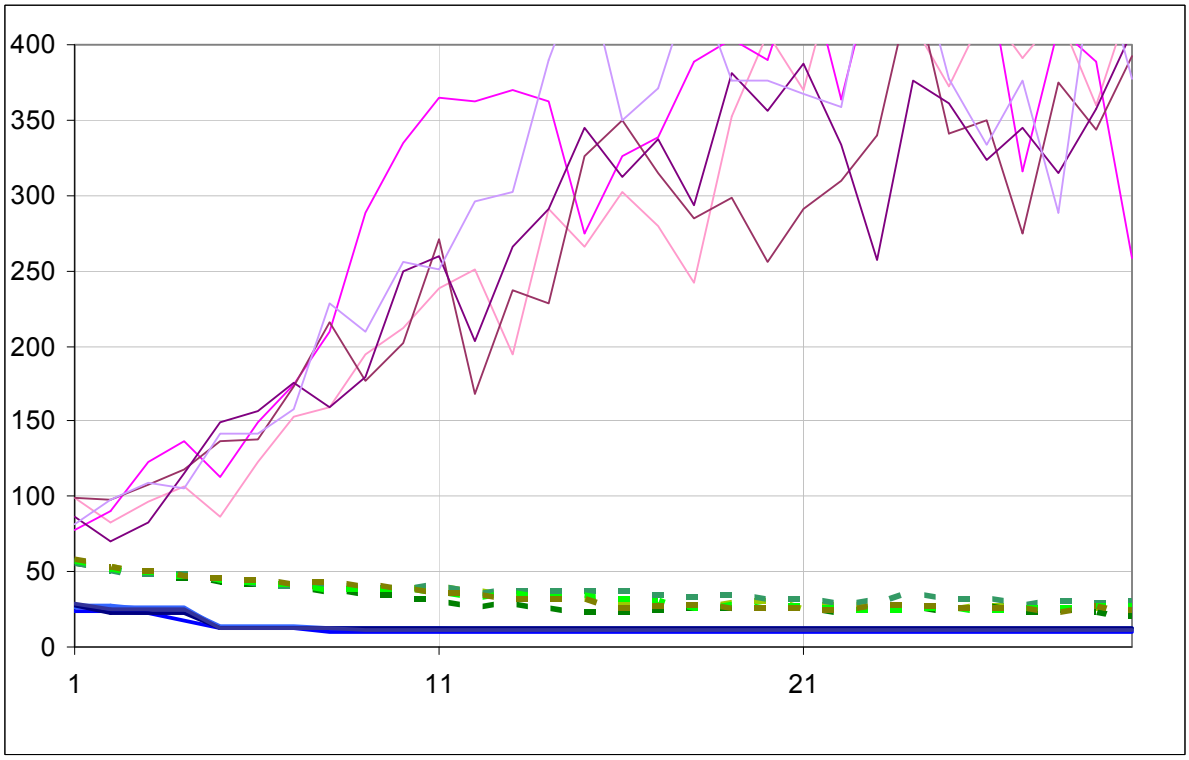


Obr. 4.11 Evoluční tlak *q*=1,3. Žluté body vyznačují případ s řešením ve 25. kroku. Za zmínku stojí, že průměrná hodnota se blíží hodnotě nejlepšího jedince, což v tomto programu způsobuje vysokou úroveň mutací (z toho pak plyne neobvykle velký rozptyl). V tomto případě ale byla populace jen 60, vzhledem k výpočtu ruletové selekce jinak v programu dochází k překročení rozsahu typu integer (viz vzorec (4.4)).

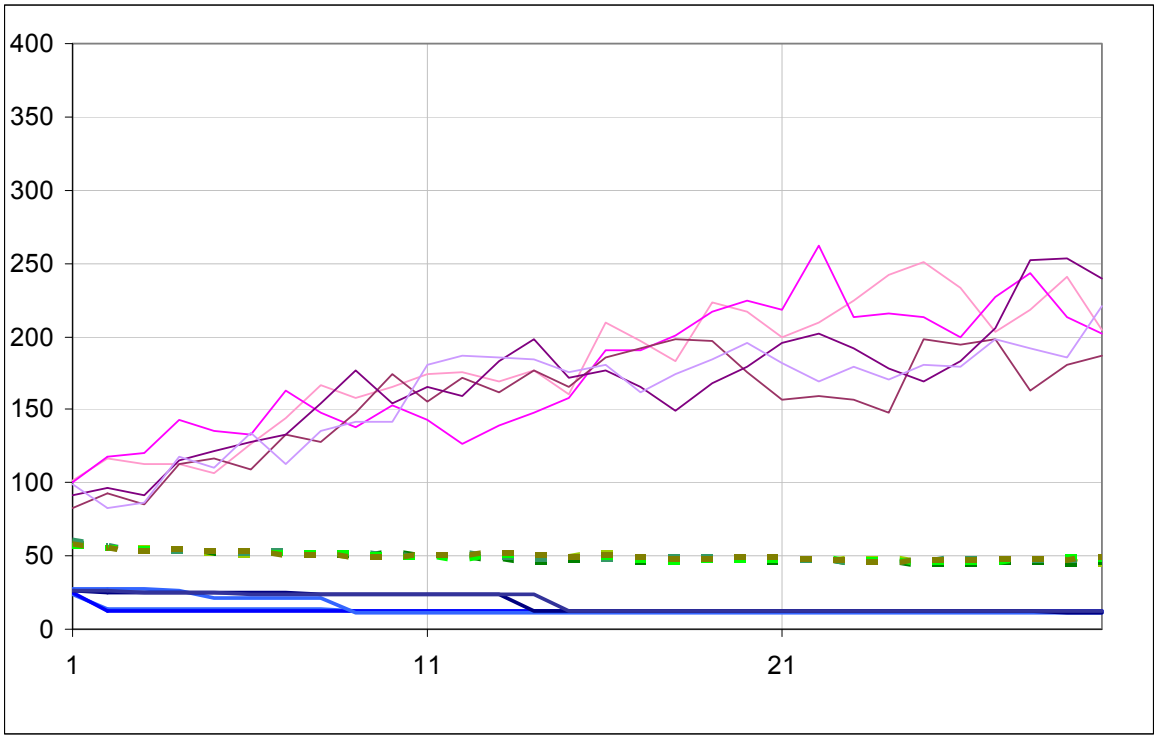
Následující grafy připadají na turnajovou selekci. Rozsah populace je 120, což je přibližně srovnatelné. Nutno podotknout, že výpočet je tři- až pětkrát pomalejší, než u ruletové selekce. Pozor na odlišný rozsah svislé osy.



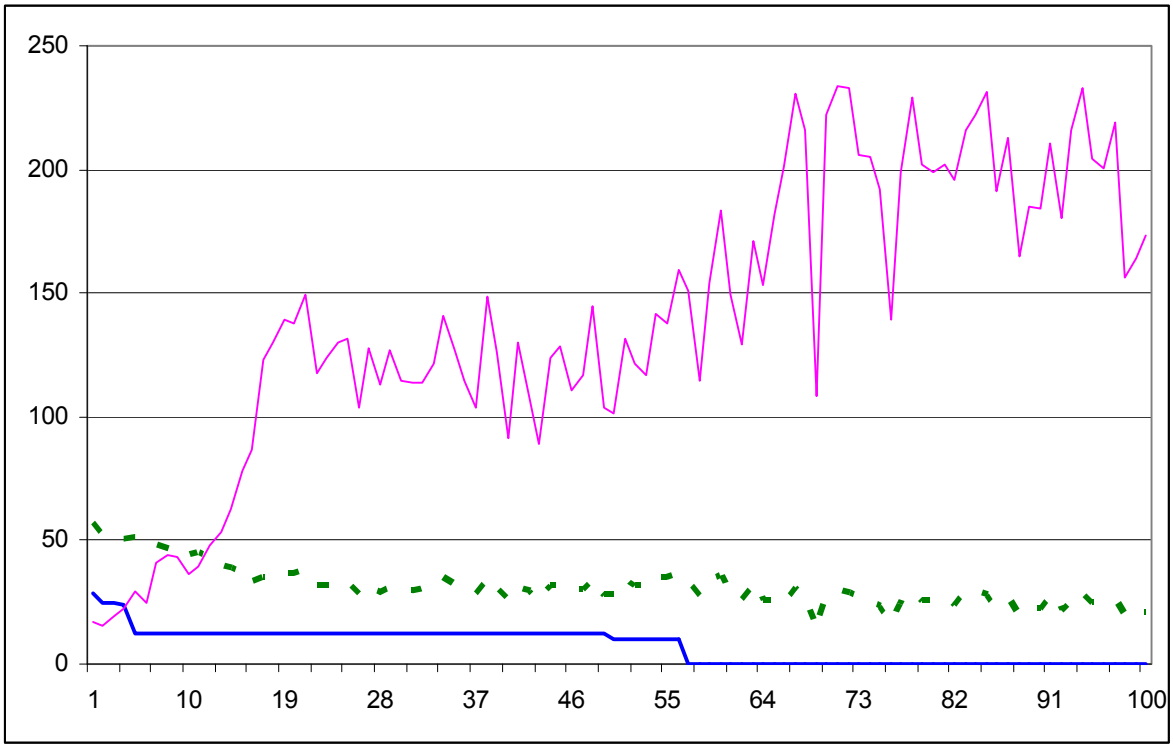
Obr. 4.12 V každém cyklu 50x T4 selekce s dvěma potomky, 10x T2 a mutace. Nejlepší výsledek 12. Pozor na jiné měřítko svislé osy než u předchozích grafů. Nízké hodnoty rozptylů z obrázků 4.6 až 4.9 jsou dány tím, že při dostatečném rozptylu program automaticky množství mutací snižuje a při příliš malém zvyšuje. U turnajové selekce toto není možné a jak zobrazují fialové tenké čáry, rozptyl dosahuje vlivem působení konstantní vysoké úrovně mutací vysokých hodnot.



Obr. 4.13 30x T4 selekce s dvěma potomky, 30x T2 a mutace. Nejlepší výsledek 10. Třikrát větší úroveň mutací, než v předchozím případě.



Obr. 4.14 10x T4 selekce s dvěma potomky, 50x T2 a mutace. Nejlepší výsledek 11. Průměr populace prakticky neklesá.



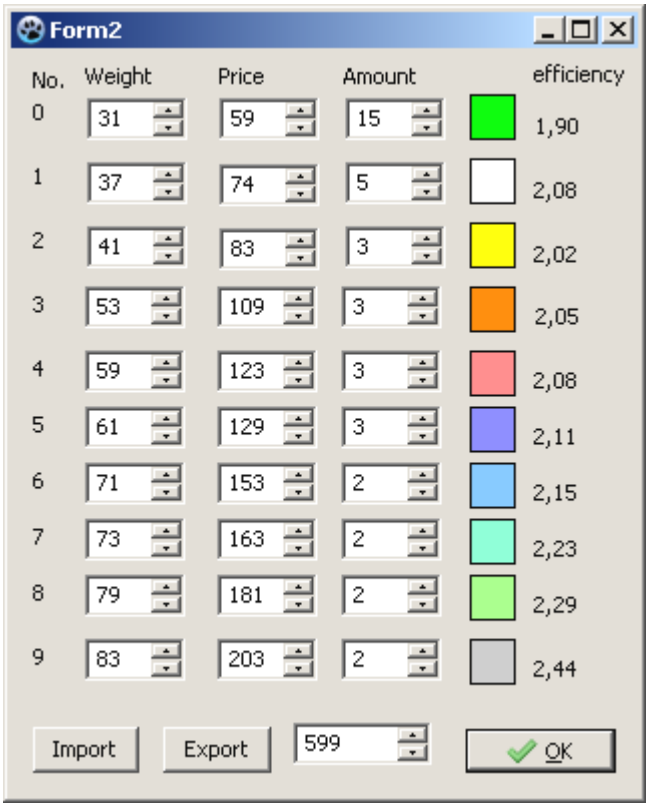
Obr. 4.15 Turnajová selekce, 30x(T4+2xC), 30x(T2+M), 100 cyklů. Tlustá modrá (nejlepší hodnota účelové funkce) označuje nalezené řešení v 57. generaci (což turnajová selekce neví, protože nezná význam účelové funkce). Zatímco průměrná hodnota (zelená přerušovaná) klesá s nalezeným řešením, rozptyl (tenká, vynesen v poměru 1:10) vlivem mutací neustále stoupá. Počet T4+2xC i T2+M je shodně 30 na jednu generaci. Populace jako v předešlých případech 120 prvků. Simulace trvala několik minut.

Předběžně lze pokus vyhodnotit jako potvrzení klíčové úlohy mutací na získání správného řešení. U následující úlohy to již takto posoudit nepůjde, protože správné řešení neznáme (proto je pro testování úloh a ladění programů užitečné používat takové zadání, kde je správné řešení známé).

4.2 Knapsack problem (program kp.exe)

Jako příklad pro nepermutační problém jsem si zvolil variantu knapsack-problému, který patří do třídy NP-úplných problémů. Je definován jako skládání předmětů do batohu. Tato představa evokuje pružnost batohu a tedy nejednoznačnost zadání, proto jsem definici problému pozměnil:

Výzkumník musí opustit základnu před zatopením na člunu a má zachránit vybavení v co největší hodnotě. Když nastoupí sám se svým notebookem a se záznamy měření, zbývá ještě *kpm* liber dostupné nosnosti (předvyplněno 599), kterou může využít na záchranu části vybavení. Každý přístroj má hmotnost, cenu (program je z výukových důvodů psán s rozhraním v angličtině, takže oboje shodně v librách), ve třetím sloupečku na ukázkovém okně programu je, kolik takových přístrojů se nachází na základně.



Obr. 4.16 Formulář pro zadání řešeného problému. Tlačítkem Export lze nové zadání uložit do souboru.

Je jasné, že pokud poměr cena/hmotnost bude u lehčích předmětů větší, program snadno najde triviální řešení. Program proto tento poměr pro informaci vypisuje v posledním sloupci (je to výsledek výpočtu, nelze jej měnit a nikam se neukládá). V průběhu testování se ukázalo, že pro dosažení více kompetitivních řešení musí tento poměr s hmotností nejen růst, ale také růst velmi pomalu.

Definici problému lze ve zobrazeném okně opravit a následně uložit na disk, respektive z disku načíst. Barva slouží pro lepší orientaci v okně výsledků, je zde zohledněno, že program je zamýšlený především pro výuku.

Uložený soubor je textový, jedná se ale o provizorní řešení, kdy na každém řádku je prostě jedno číslo. Není tak vhodný k editaci uživatelem.

Na rozdíl od minulého problému (řazení 10 čísel) lze parametry řešené úlohy (zadání) změnit a nemohu tedy obecně říci, jaké je pro jednotlivé případy správné řešení.

Fitness (účelová funkce)

Protože program účelovou funkci minimalizuje, je spočítána odečtením hodnoty naloženého zboží od maximální představitelné hodnoty, kterou program po zadání změny problému spočte jako maximální „kilogramovou" cenu přístroje, ponásobenou přípustným zatížením:

$$X_{teoretická} = \max_{i=1}^{k}(\frac{c_i}{w_i}) \cdot w_{celková} \tag{4.7}$$

Výsledek zaokrouhlí na celá čísla nahoru, protože program pracuje v cyklu s celými čísly, hlavně z důvodu zobrazování. Hodnota účelové funkce se pak spočte jako

$$f = X_{teoretická} - \sum_{j=1}^{n} c_j w_j \tag{4.8}$$

Způsob provádění křížení, kdy se vezme prvních *k* prvků z prvního rodiče a doplní se z druhého rodiče (pro změnu od pozice *k+1* do konce) a problémy s malou diverzitou genomu mne vedly k zařazení funkce, kdy po každém cyklu se u všech jedinců náhodně provede několik prohození (hodnota účelové ((*fitness,* (4.8))) funkce nezáleží na pořadí, takže změna pořadí prvků by pro řešení problému neměla mít význam). Počet prohození na jedince za generaci lze nastavit od nuly (tedy i vypnout) do čtyř. Neznamená to, že v každém jedinci se provedou například dvě prohození. Číslo se nastavuje po desetinách, nastavená hodnota se ponásobí velikostí populace a pak se provede takto vypočtený počet „zamíchání", kdy se náhodně vygeneruje číslo jedince, k němu náhodně dvě čísla a prvky na jejich pozici se vymění. V rámci náhody není zcela vyloučené, že některé další náhodné prohození prvky opět vrátí, jen se pravděpodobnost, že k tomu během cyklu dojde, blíží nule.

Protože se jedná o upravený předchozí program, opět nabízí ruletovou i turnajovou selekci. Mutace zde ovšem nemůže být reprezentována prohozením prvků u jedinců, protože na pořadí nezáleží. Jako jedinou mutaci jsem nakonec navrhl záměnu prvku náhodně za jiný. Tlačítko M1 (a odvozené funkce) vymění dva prvky, tlačítko M2 až pět prvků, ale počet záměn je náhodný (i nula).

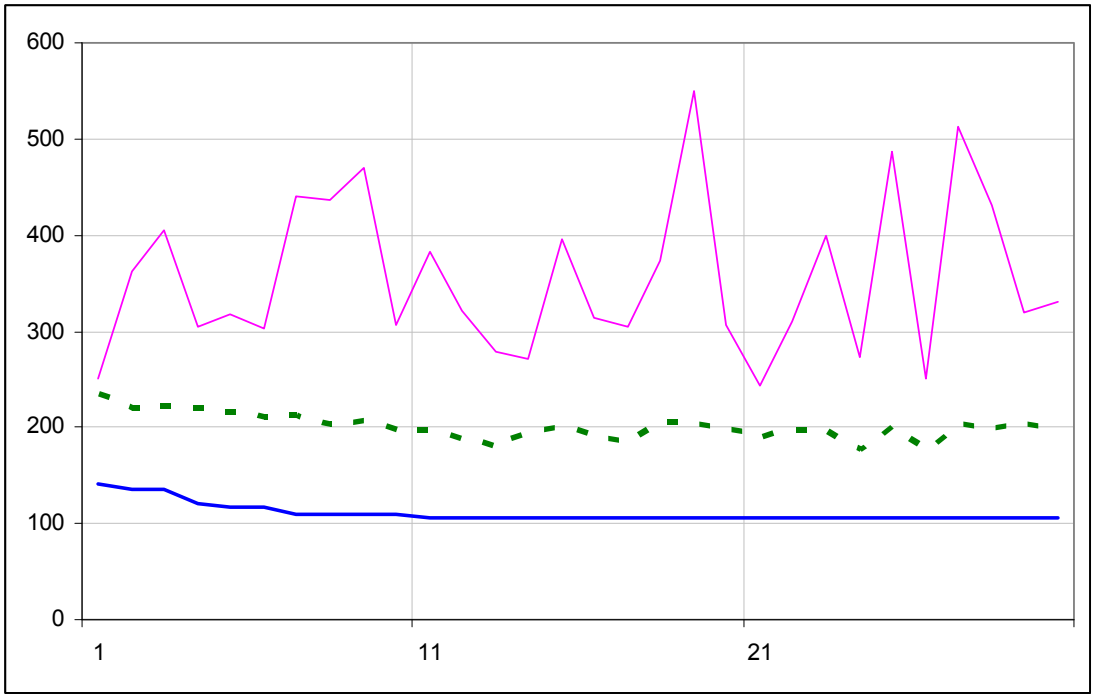
Feasibility (funkce přípustnosti)

Jak křížením, tak mutacemi může vzniknout jedinec, který není podle definice problému přípustný. Proto po každém generování proběhne kontrola přípustnosti.

1. V každém jedinci může být 11 elementů, tj. řešení může být naložení maximálně 11 přístrojů. Pokud nějaká pozice není obsazena, je nejprve náhodně doplněna jakýmkoli prvkem.
2. Zkontroluje se, zda některého přístroje není navrženo naložit více kusů, než je k dispozici. Pokud ano, nahradí se volným políčkem.
3. Spočítá se celková hmotnost nákladu a pokud přesáhne přípustnou, poslední prvek se nahradí prázdným políčkem. Člověka by logicky napadlo vyřadit ten, který má hmotnost z dostupných nejblíže vyšší, než je překročení hmotnosti. Vnášení podobných podmínek by sice nalezení řešení mohlo urychlit, ale omezovalo by to diverzitu populace, které může být užitečná v dalším křížení. Krok 3 se opakuje, je-li to třeba.
4. Prvky se „setřepou" tak, aby volnými místy popis nákladu skončil. Je to kompromis z důvodu zobrazení; každé setřídění je samozřejmě na závadu při hledání řešení, jinak by bylo samozřejmě nejlépe prvky setřídit všechny. Možné řešení by bylo třídit jen pro

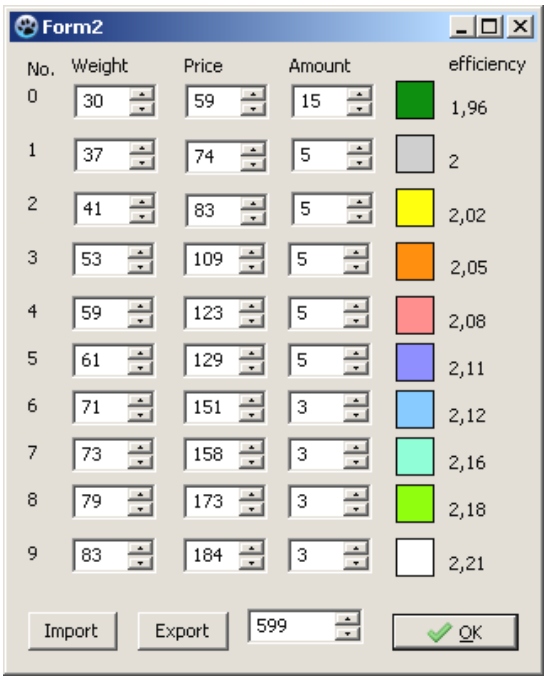
zobrazení a v paměti ponechat prvky se zamíchaným pořadím (což je možné další rozšíření funkce programu).

Při použití zadaných dat, s populací 100 jedinců a elitismem proběhla simulace s průběhem na následujícím grafu. Rozptyl (tenká fialová) je vynesen 10x zmenšený, nejlepší jedinec je modře tlustě, průměr zeleně přerušovaně.

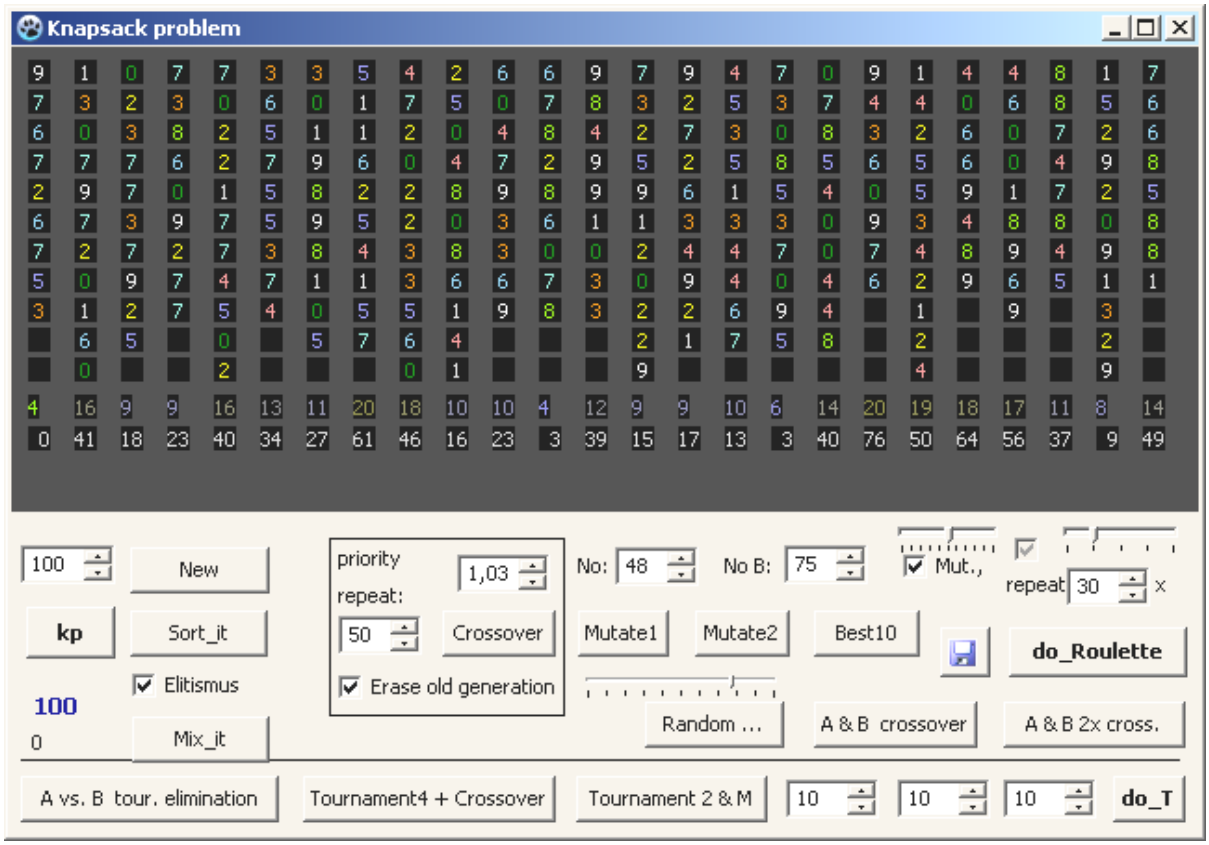


Obr. 4.17 Původní problém, 100 jedinců, roulette, *q*=1,10, řešení po 12 krocích bylo {9,9,8,8,7,6,6,4}.

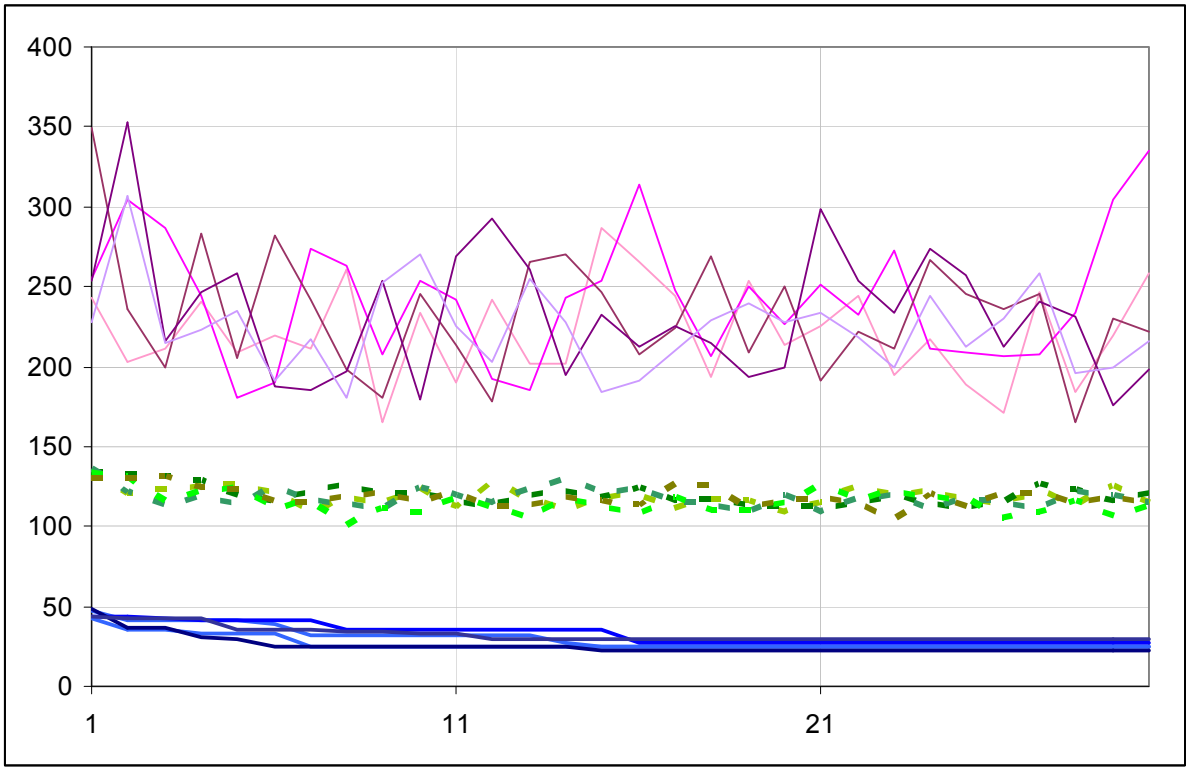
Opravené zadání snižuje rozdíl podílu ceny a váhy mezi přepravovanými předměty. Také možný počet exemplářů od každého z dražších předmětů je zvětšen.



Obr. 4.18 Okno s novým zadáním problému. Poměr cena/váha je zrovnoměrněn.

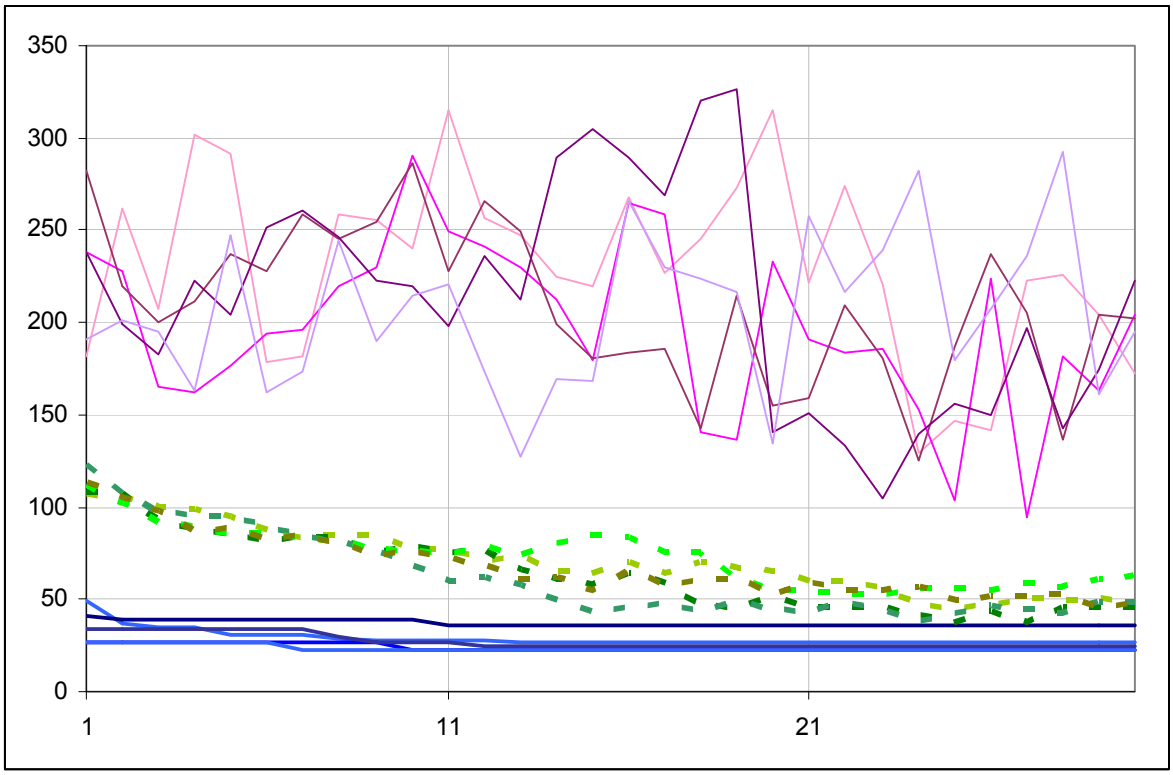


Obr. 4.19 Ukázka okna programu s náhledem populace. Max. 24 jedinců zobrazeno, 0. sloupec je elitní jedinec; je zvoleno „smazat starou generaci“, takže tento jediný přežívá stisk tlačítka Crossover nebo do_Roulette. Tlačítko kp vyvolá okno s definicí problému, viz předchozí obrázek. U jednotlivých jedinců (zvolená kombinace naložených předmětů) jsou jednotlivé předměty z definice vyznačeny jak číslem, tak barvou. Předposlední řádek reprezentuje hodnotu účelové funkce (ale podělenou deseti) a spodní řádek, kolik kapacity zůstalo nevyužito.



Obr. 4.20 Pět průběhů v jednom obrázku. q=1,03, opět 100 jedinců, 30 generací.

.



Obr. 4.21 Turnajová selekce, 30 cyklů, každý 25x T4+křížení, 25xT2+mutace. Počáteční populace by měla být srovnatelná s předchozím obrázkem (tatáž funkce), dále je vidět, že diverzita klesá rychleji (zůstává více jedinců z předchozí generace). Nutno podotknout, že doba každé simulace v turnajové selekci je výrazně delší než v ruletové, asi 100 sekund místo dvaceti.

Získaná řešení v turnajové selekci u jednotlivých běhů byla {9,9,9,8,7,6}, {9,9,9,8,8,8,6,2}, {9,9,9,8,8,8,6,2} (stejné), {9,9,9,8,7,6,5,5} a {9,9,9,8,8,8,7,1}, s hodnotou účelové funkce 27, 23, 23, 36 a 25. (zde uvedená řešení byla ručně seřazena od největších k nejmenšímu číslu, aby byla přehlednější, počítač pořadí prvků uvnitř jedince nezachovává).

4.3 Genetické programování – ukázkový (výukový) program

Pro ukázku byla vzata následující úloha:

„Vezmeme několik funkcí ze slovníku, zkombinujeme je a v Excelu připravíme tabulku hodnot. Program se pokusí uhodnout funkci“.

Za tímto účelem je třeba navrhnout slovník použitých funkcí, ze kterých se složí hledaná funkce. Návrh by mohl vypadat asi takto (druhý sloupeček – číslo nebo funkce, třetí – možné parametry):

Tabulka 4.2. Ukázka tabulky primitivních funkcí

1	1	
2	2	
3	*x*	
4	+	a,b
5	-	a,b
6	*	a,b
7	/	a,b
8	sin	a
9	cos	a
10	ln	a
11	exp	a

Je třeba rozlišovat terminály [GP, str. 109] (nemají parametry, zpravidla představují proměnnou *x* a konstanty, příkladem jsou první tři řádky v tabulce) a neterminály. Místo každého terminálu může být neterminál, bezprostředně následovaný svými parametry. Písmena „a“ a „a,b“ značí skutečnost, jestli má funkce jeden, nebo dva parametry (arita, [UI4, str. 131]). Při dvou parametrech se pak zapíše do řetězce nejprve celý výpočet prvního parametru, a pak bez další značky druhého. V tomto případě by se pak funkce zobrazovala stromovou strukturou, jak to bývá obvyklé v literatuře. Způsob zápisu je tedy takzvaná „polská notace“ (viz výklad teorie v předchozí kapitole).

Speciálním případem terminálu je hodnota parametru *x*, nezávislé proměnné hledaného funkčního vztahu. Terminální symbol *x* se v každém jedinci musí vyskytovat minimálně jednou (jedna z podmínek přípustnosti, feasibility).

Jedinec je tedy zapsán jako řetězec symbolů. Pro tuto funkci se v Pascalu hodí typ string, textový řetězec. S ohledem na snadnost ladění a řadu funkcí, dostupných z původního Turbo Pascalu, jsem se snažil vystačit s klasickým řetězcem jazyka Turbo Pascal, který má ale délku do 255 znaků ($2^8 - 1$, v nultém znaku je délka řetězce). Znaková sada reprezentuje prvních 256 znaků z aktuální znakové sady operačního systému, ale je bezpečnější předpokládat jen sadu ASCII. Protože 255 znaků není příliš mnoho, je dobré, když každý parametr/funkce je reprezentován jen jedním znakem.

Význam znaků byl zvolen tak, aby s trochou snahy bylo možné řetězec číst bez nutnosti překladu. Proto například funkci sinus reprezentuje znak „s“ a sčítání „+“. Někde to pro kolizi možné není, takže „e“ znamená funkci „exp“ („e na x-tou“), ale konstantu e znak „n“. Takové kódování je užitečné při ladění. Pro praktické použití je možné vytvořit konverzní program, který umožní funkci zadat v čitelné infixové formě. V tomto tvaru jsou jednotlivé navržené funkce i zobrazovány, ale pro jednoduchost je tento zápis doplněn o závorky.

Ve vnitřním zobrazení v programu je například sinus zaznamenán jako „s“ a kosinus jako „c“. Zápis funkce z teoretické části (odstavec 3.1.2)

$f(x) = \sin x \cos x + \cos x \sin x$

pak má v paměti počítače tvar

+*sxcx*cxsx

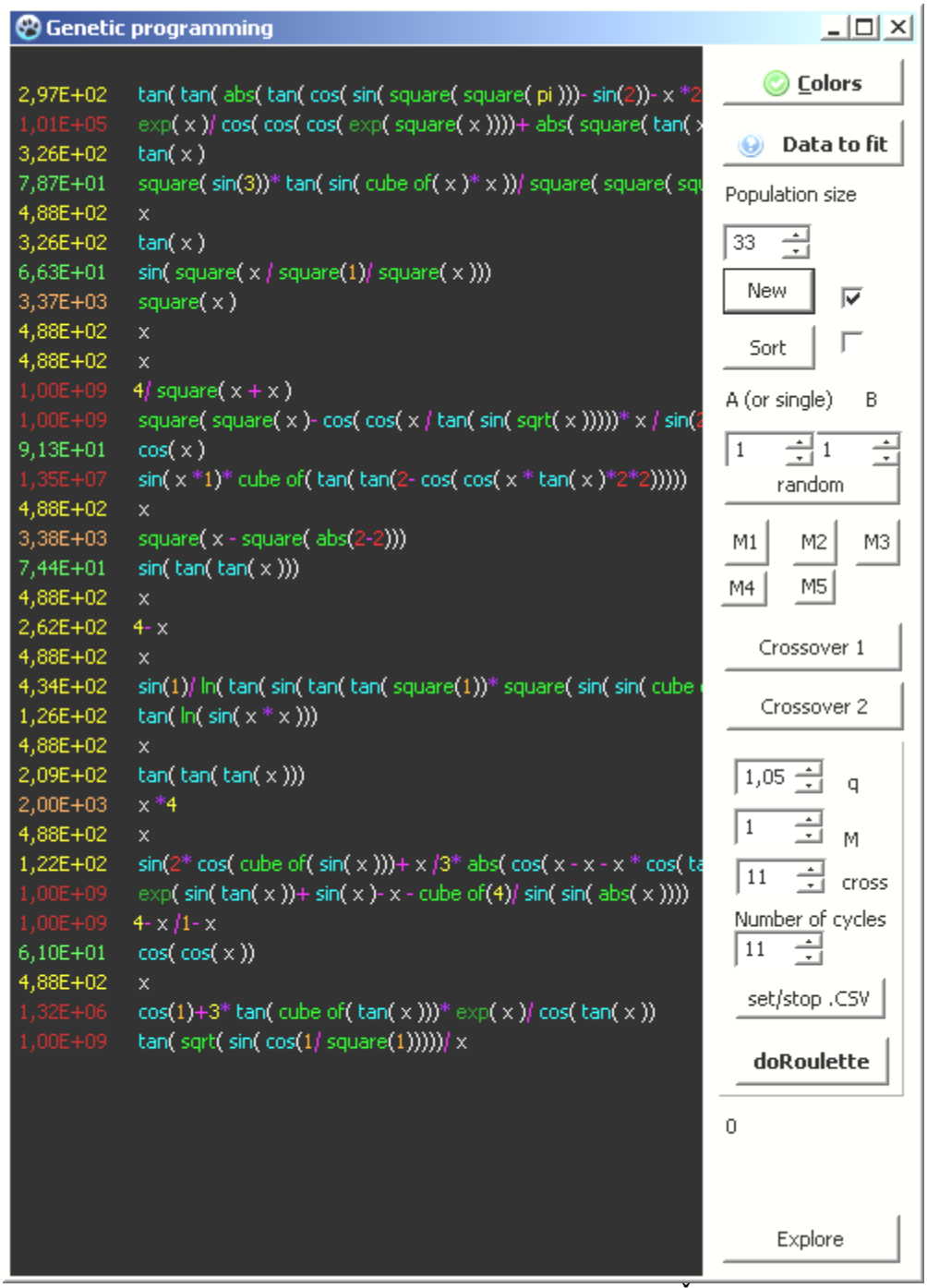
Hledanou funkci je třeba definovat zadáním tabulky hodnot pro daný rozsah. Program předpokládá stálý krok nezávisle proměnné *x* pro dělení definičního intervalu, protože takový tvar dat se v Excelu vytvoří nejsnáze (textový soubor, 1 sloupec s čísly bez záhlaví). Alternativní možností je jiné známé dělení, nebo dokonce dvojice x-y. Teoreticky lze hledat i funkci více proměnných, která ani nemusí být známa v nepřerušovaném intervalu, některé úseky/oblasti mohou chybět (obdobný problém řeší aplikace popsaná v kapitolách 5 a 6).

Účelovou funkcí (*fitness*) je součet absolutních hodnot rozdílů zadání a genetickým algoritmem sestavené funkce (3.4). Druhá mocnina by zde byla bez užitku, neboť se nejedná o gradientní metodu a absolutní hodnotu počítač spočítá rychleji (znaménko je u reálného čísla v odděleném bitu, stačí otestovat a převrátit). Vyhodnocování účelové funkce je časově nejnáročnější částí algoritmu, pomalé je samotné vyhodnocení funkce, s nutností vyhodnocení rekurzí (protože se strom větví, vedla by jiná řešení k omezení obecnosti, například omezením přípustného počtu větvení). V průběhu ladění se ukázalo, že je třeba stále kontrolovat limity dosazované funkce, protože některé funkce v hledaném rozsahu nemusí být definovány nebo mohou velmi rychle stoupat (např. „exp(exp(exp(x)))“) (viz [AField, str. 21], “*evaluation safety*”). V tom případě je hodnota nahrazována nějakou hodně velkou hodnotou, která by měla být dostatečná pro vyřazení funkce v dalším vyhodnocení. V případě popisovaného programu je zde zadáno „9e9“, devět krát deset na devátou, nelze proto hledat průběhy funkcí, které se podobným hodnotám byť řádově blíží.

Účelová funkce je vyhodnocována [GP, str. 128] na intervalech, kde je známa hodnota hledané funkce. V případě tohoto programu je to interval nezávislé proměnné, dokonce s konstantním krokem, ale v praxi může jít o několik nesouvislých intervalů (u víceparametrických funkcí dokonce oblastí) a některé oblasti mohou být posuzovány s větší váhou; typicky tabulka hodnot funkce obsahuje i nezávisle proměnnou (nebo proměnné) a účelová funkce se vyčísluje pro tytéž hodnoty, jako je tomu u programu popisovaném v kapitole 5.

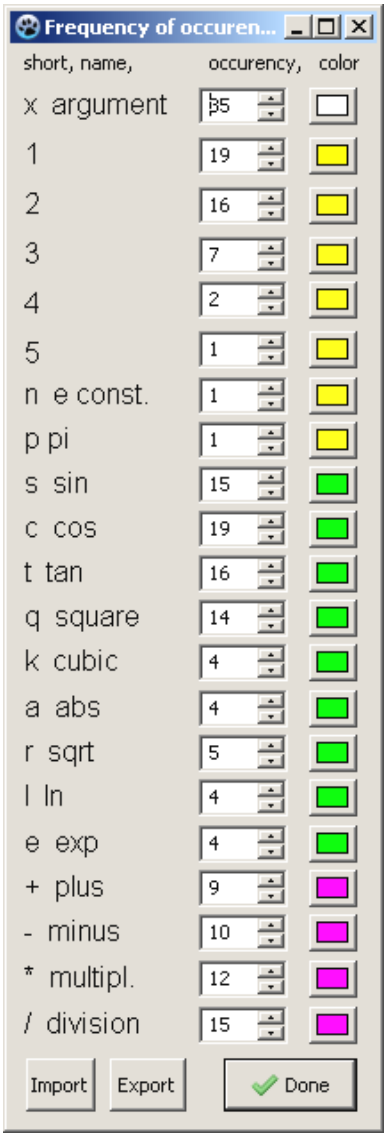
Ovládání programu

Následující obrázek ukazuje základní okno programu, sejmuté ale až po vygenerování počáteční populace (třetí krok v následujícím popisu). Program jsem se snažil koncipovat tak, aby nepotřeboval sepisovat návod. Tlačítka používáme odshora dolů, po najetí myší na většinu tlačítek se zobrazí stručná nápověda, co které dělá. Celý program komunikuje v angličtině, což má usnadnit použití při výuce – pravidelně k nám dojíždí studenti výměnných pobytů (např. program Erasmus) a ti by češtinu nezvládli.



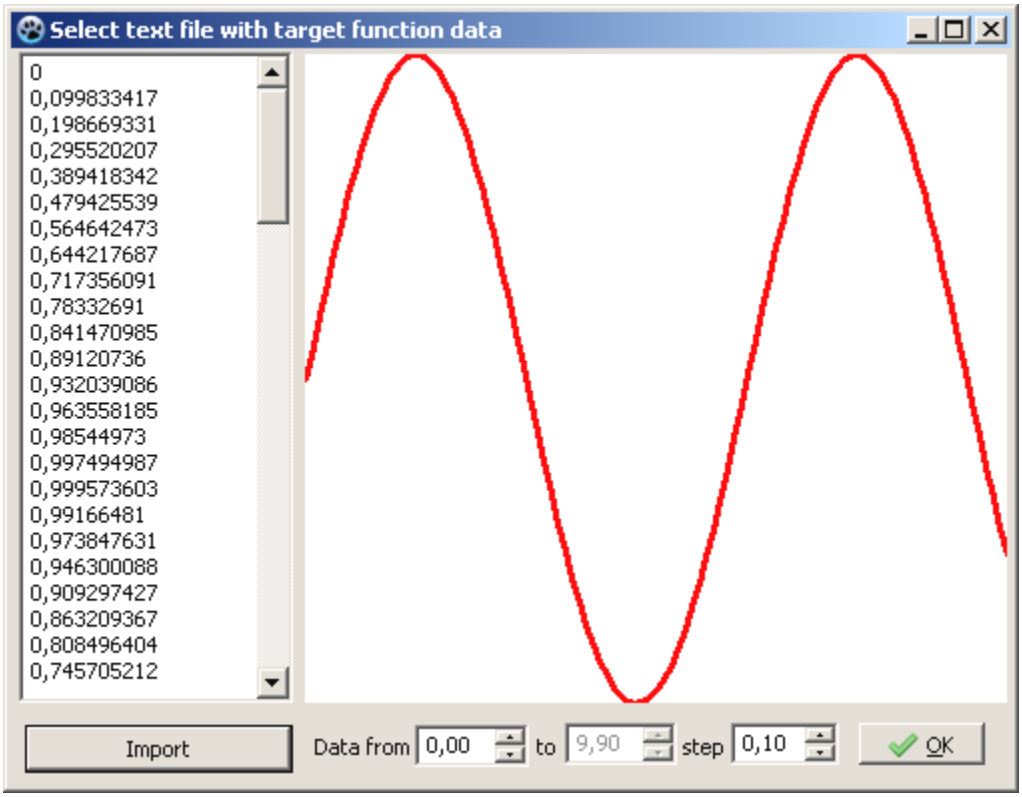
Obr. 4.22 Okno programu po vygenerování počáteční populace. Černé pozadí bylo zvoleno proto, že barvy jsou lépe rozlišitelné (výraznější), než na bílé, která by vedla na volbu tmavších odstínů.

Nejprve zkontrolujeme četnosti funkcí (tlačítko Colors). Předvyplněné počty výskytů byly získány sečtením výskytů funkce, resp. konstanty, v [Rektorys, str. 100] (přehled vztahů mezi goniometrickými funkcemi). Do seznamu byly doplněny funkce jako ln, exp, protože ty se v této části knihy nevyskytují a uživatel je může potřebovat. Uživatel si ovšem svoje funkce doplnit nemůže, lze měnit jen četnosti. Hodnoty četností lze uložit do textového souboru, respektive zde uložené zpět načíst. Soubor ovšem obsahuje jen v každé řádce jediné číslo a „ruční“ editace uloženého souboru je zřejmě neproveditelná. Výchozí stav je na následujícím obrázku. Zde je argument *x* (proměnná) bíle, ostatní „terminály“ žlutě, funkce s jedním parametrem zeleně a funkce se dvěma růžově. Barvy lze změnit kliknutím na barevný obdélníček a zvýraznit si například hledanou funkci, pokud víme, že s její pomocí jsme data vygenerovali.



Obr. 4.23 Formulář pro zadání relativních četností primitivních funkcí. Pro lepší čitelnost lze nastavit i barvy.

Další tlačítko, „Data to fit“ (ukázka vyvolaného formuláře na obrázku obr. 4.24), slouží k načtení hodnot hledané funkce. Hodnoty vygenerujeme v Excelu jako sloupeček čísel (na každém řádku jedna hodnota) a uložíme jako soubor ve formátu .txt. Po načtení do programu se nám pro kontrolu zobrazí graf hledané funkce (zadaných hodnot). Hodnoty je třeba editovat před nahráním do programu, jejich zobrazení ve formě editovatelného okna je jen pro prohlížení a případný zásah nemá na nahraná data vliv.



Obr. 4.24 Formulář pro zadání hodnot, pro které se bude hledat funkce.

Podotýkám, že program čte systémové nastavení oddělovače desetinných míst. Na některých počítačích tedy bude očekávat desetinnou tečku, jinde čárku. Alternativou by byla kontrola načítaného souboru a přizpůsobení načítaným datům.

Nesmíme zapomenout zkontrolovat meze. Zadává se „od“ a „krok“. Default je zadáno od nuly po jedné desetině, což vyhovuje pro hledání běžných kombinací goniometrických funkcí. Pokud byla data generována pro jiné meze, je to třeba zadat. Program počítá pochopitelně v radiánech (meze nemohou být součástí souboru s daty, hůře by se popisovalo, jak soubor s daty vytvořit, zpravidla by nešlo soubor generovat přímo z Excelu a ukládat jako text). Maximální počet hodnot u tohoto programu je 2020, to již by ale vyčíslení hodnoty funkce bylo dosti pomalé. Rozumné výsledky lze získat již pro 100 hodnot.

Po této inicializaci můžeme vygenerovat počáteční náhodnou generaci funkcí (ukázka například na obr. 4.24). V literatuře lze najít příklady heuristik, kdy se generují přednostně funkce ve vhodném tvaru (ze začátku se dává přednost funkcím s více parametry, pak jen s jedním a od jisté hloubky stromu se generují převážně/jen terminální symboly [UI4, str. 133]. Podobné metody zde nejsou použity, domnívám se, že nucené směrování algoritmu ke konkrétnímu tvaru omezuje obecnost. K omezení délky genomu přesto dojde, pokud přesáhne maximální zapisovatelný rozsah, 250 znaků.

Počínaje vygenerováním počáteční populace pak program po každé operaci vypisuje aktuální funkce v hlavním okně programu (je grafické, takže i když obsahuje jen text, nelze z něj ani kopírovat myší) pomocí závorek (nepíše matematické vzorce). Podobný zápis může být velmi dlouhý a nelze jej pak zobrazit celý. O něco více se zobrazí, když na funkci klikneme myší. V pomocném okně se pak zobrazí průběh hledané (zadané) funkce, průběh navržené (vygenerované) funkce a v záhlaví okna pak řádkový zápis funkce (okno lze myší o něco rozšířit, graf se nepřekreslí, ale někdy tak lze zobrazit další část zápisu funkce). Kliknutím na pomocné okno se toto zavře. Pokud jde o rozsah grafů, v pomocném okně se zobrazí celý průběh zadané funkce ((daty v textovém souboru)), a ta část vygenerované

funkce, která nevyžaduje zvětšení rozsahu grafu o více jak 50%. Je-li část vygenerované funkce mimo tento rozsah, nezobrazuje se.

Občas potřebujeme vědět pořadové číslo vygenerované funkce, například pro tlačítky vyvolané mutace nebo křížení. Řešil jsem to tak, že po kliknutí na funkci se (mimochodem ještě před jejím zobrazením) do editačního políčka, označeného jako „A“, zapíše číslo funkce.

Studenti si mohou vyzkoušet jednotlivé funkce genetických algoritmů. Zaškrtávací políčko vedle tlačítka „New“ způsobí, že při vygenerování x nových prvků (při použití tlačítek „New“ nebo „doRoulette“) se stejný počet smaže. Program ovšem pracuje nejvýše s 250 jedinci; pokud tento počet překročí, maže prvky pravidelně. Kde je to možné, přednostně ty s vyšší hodnotou účelové funkce.

Sort, tedy třídění (správně česky řazení), je další důležitou operací. Vidíme jen prvních cca 40 funkcí, takže je možná rozumné si je setřídit, abychom viděli ty nejlepší jedince. Zaškrtávací políčko vedle tlačítka Sort realizuje elitismus, opět stejně nepřesnou metodou, jako v předchozích programech – pokud se provede setřídění a pak hned nějaká další operace, tento prvek se jí zúčastní hned dvakrát (na pozici nula i jedna v seznamu), a je tím neoprávněně zvýhodněn. Programátorsky je to ale mnohem jednodušší řešení, a navíc při většině operací je prvek na pozici 1 spolu se starou generací zpravidla vzápětí smazán.

Editační políčka v další řadě umožňují zadat číslo prvku, se kterým se provede některá z následujících operací. Tlačítko „random“ je nastaví náhodně v přípustném rozsahu (ale od jedničky, tj. bez případného elitního jedince).

Tlačítka M reprezentují **mutace**, vysvětlení se objeví ve žluté „bublině“ po najetí na tlačítko.

M1 změní náhodně jeden znak v řetězci tak, že se změní symbol za symbol stejné třídy (terminál za terminál, funkce za funkci, početní operace typu sčítání (dva parametry) za jinou se dvěma parametry (obr. 3.2 a).

M2 změní náhodně některý ze symbolů (ne terminál) o třídu níže (mající o jeden parametr méně, jinými slovy o jedničku menší aritu) a přebývající větev se smaže (b).

M3 změní náhodně prvek a provede přeskenování funkce – odstraní větev nebo naopak vygeneruje novou, je-li třeba (obr. 3.2 c).

Poslední dvě byly doplněny po testování programu.

M4 odstraní hlavičku (a přeskenuje = opraví funkci). Ukázalo se totiž, že hlavička jen obtížně podléhá změnám způsobeným křížením (obr. 3.2 e).

M5 prohodí dva parametry funkce, která má dva parametry (obr. 3.2 d).

Pokud se nějakou mutaci nepodaří navrhnout, je místo mutovaného jedince použit původní nezměněný (program hledá místo pro realizaci mutace v zápise funkce na náhodně vygenerovaném místě; pokud tam není, zkusí generovat jiné náhodné místo, celkem max. 10x; v případě neúspěchu neprovede samozřejmě nic a vrátí nezměněného jedince. V [UI4, str. 138] odpovídá této (ne)mutaci *permutace*).

Všechny mutace i křížení mají jedno společné – na konci se kontroluje splnění funkce přípustnosti (feasibility). Není-li splněna, výsledek se nezapíše (poznámka – [AField, str. 21], “*type consistency*” – v literatuře je uváděna nutnost kontroly realizovatelnosti funkce, například zda pro sinus bude k dispozici argument a pro sčítání dokonce dva. V tomto programu to kontroluje zmíněná funkce přeskenování, která v případě potřeby funkci opraví – zahodí nevyužité části genomu (zápisu funkce), nebo naopak část dogeneruje, pokud

například po mutaci nějaká větev chybí. V zásadě ale tento program pracuje jen s plně definovaným genomem, je kontrolováno, aby žádná část větve nechyběla, ale také, aby součástí genomu nebyly nefunkční úseky).

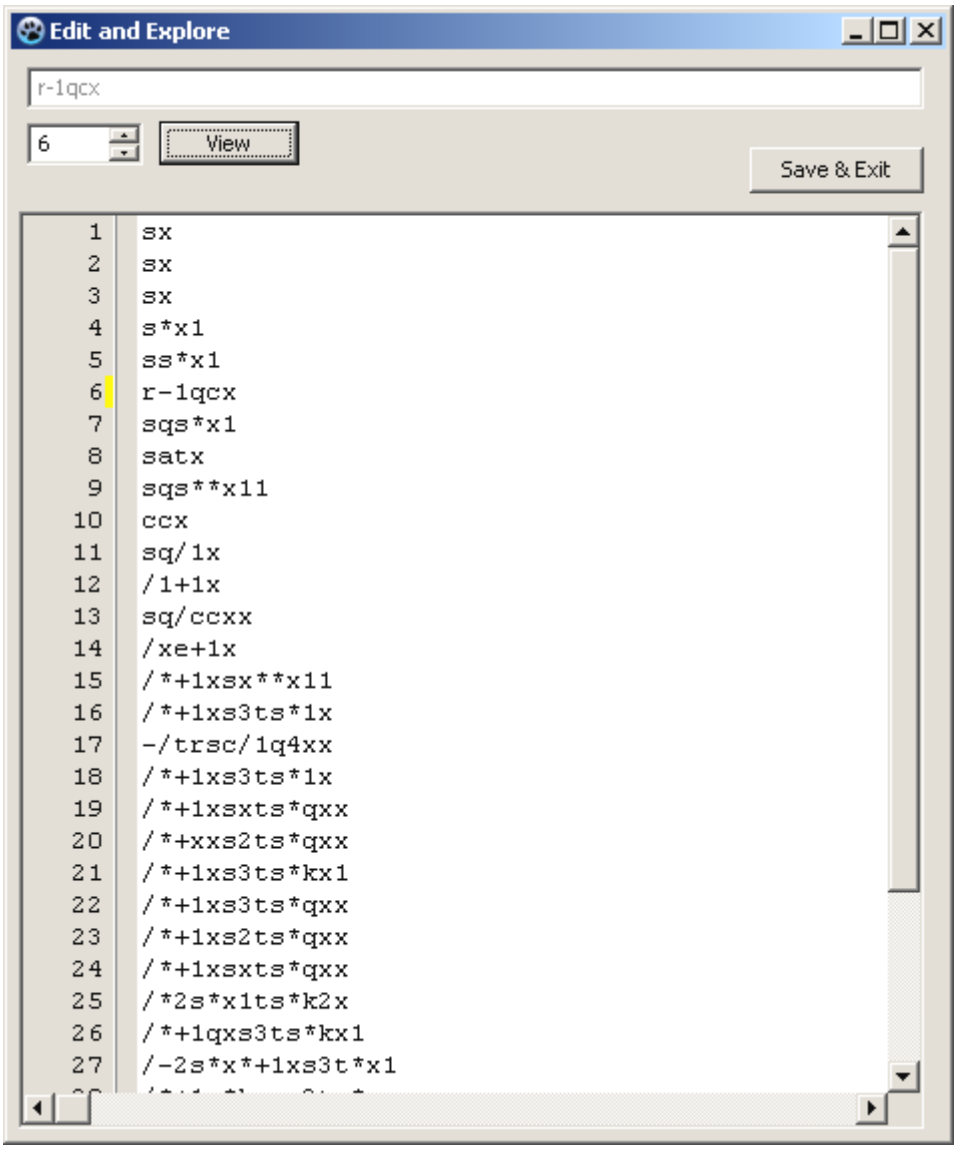
Řada možných mutací je popsána v [AField, str. 42].

Funkce přípustnosti (feasibility)

Nepřípustné funkce jsou zejména funkce, které neobsahují proměnnou *x* (ani jednou). Dále jsou nepřípustné funkce, zapsané pomocí více jak 255 symbolů. V tomto druhém případě se ale program pokusí na vygenerovanou funkci aplikovat mutaci typu M2 a po přeskenování testovat znovu. Buď bude podmínka délky splněna, nebo bude současně s korekcí smazán poslední symbol „*x*“ a funkce bude označena jako nepřípustná.

Není-li funkce přípustná, celá operace (mutace, křížení) se anuluje.

Důsledkem je, že po křížení mohou potomci být zahozeni, nebo dokonce podrobeni mutaci. Toto zjednodušení usnadňuje naprogramování algoritmů.



Obr. 4.25 Textový editor s očíslovanými řádky po jistém zácviku nabízí možnost zasahovat do zápisu funkce.

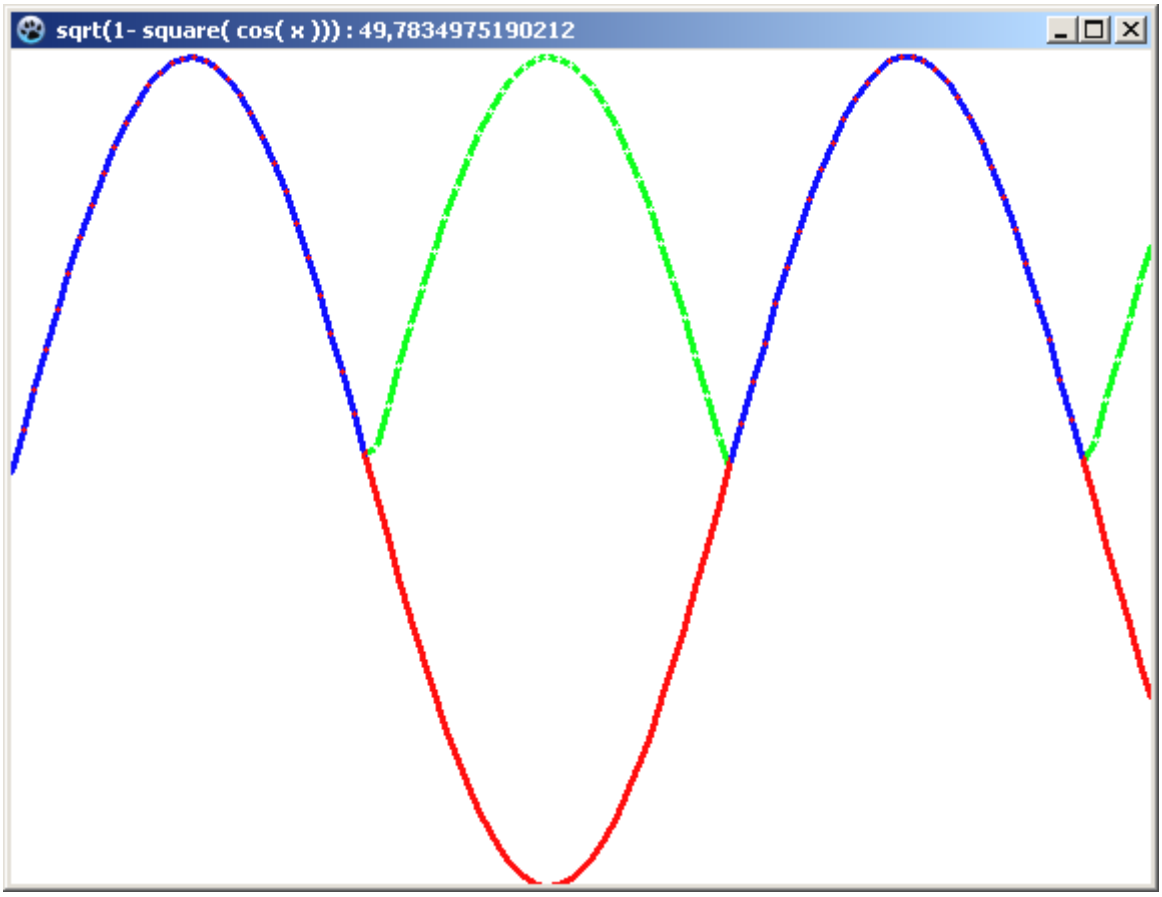
Křížení – první tlačítko vytvoří jednoho nového potomka (populace se zvětší o jeden prvek), druhé postupuje stejně, ale z nepoužitých částí genomu vytvoří současně druhého potomka.

Na panelu jsou shromážděna tlačítka pro cyklický běh programu. První je evoluční tlak (definovaný stejně, jako u předchozích programů), další je pravděpodobnost mutace (počet za cyklus – ten ale navíc záleží na diverzitě populace), poslední je počet křížení za cyklus.

Tlačítko „set/stop .CSV“ umožňuje zahájit zápis výsledků po každém cyklu do souboru (je třeba zadat jméno souboru). Opakovaným stiskem se aktuální záznam skončí, program o tom vypíše zprávu. Pak lze opakovaným stiskem záznam znovu začít do jiného souboru. Pokud soubor existuje, je přepsán (nelze pokračovat v přerušeném experimentu). Záznam neslouží k výuce, používám jej k prezentaci výsledků grafy, jako v této práci (prezentaci programů studentům předchází přednáška).

Inspekce (tlačítko Explore)

Aby byla umožněna editace záznamů (formálně nepřípustný zásah, ale pro některé školní úlohy je třeba mít možnost zápisy funkcí ručně měnit), je doplněna možnost převést záznamy na textový soubor, ten přeeditovat a uložit zpět (obr. 4.25). Okno umožňuje zadat číslo řádku (editor použitý z prostředí Lazarus umožňuje zobrazovat čísla řádků) a zobrazit si průběh funkce i její přepis do srozumitelnějších symbolů.

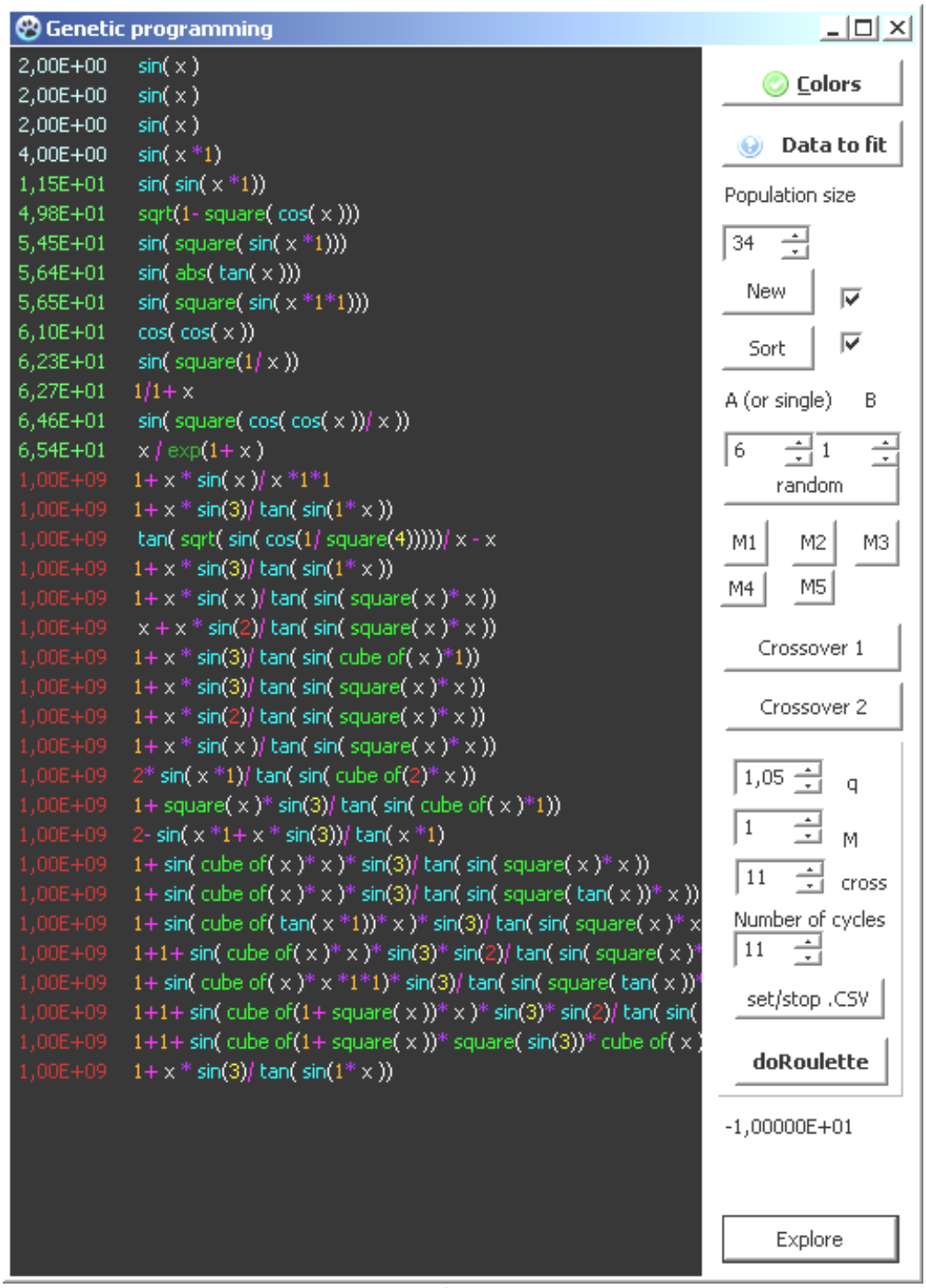


Obr. 4.26 Zobrazená funkce po výběru v předchozím okně (obr. 4.25, zápis byl „r-1qcx“) : Červeně – zadaná funkce, zeleně – generovaná funkce. Při souběhu generuje tmavě modrou přerušovanou čáru. Přerušování vzniklo náhodně – aby byl vidět souběh, použil jsem pro vykreslení funkci XOR (v Delphi/Lazarusu „xorput“), a funkce občas smaže sama sebe.

Heuristiky

Účelová funkce (*fitness*) musí nějak zahrnout délku zápisu nalezené funkce, protože výsledek chceme co nejkratší. Proto je k hodnotě rozdílu funkcí připočtena nakonec i délka řetězce [Soule]. Výsledek tedy nikdy nebude nulový.

Důvod omezení délky hledané funkce, nebo přesněji zvýhodnění krátkých zápisů funkce, je, že s délkou funkce se rychle množí způsoby, jak funkci pomocí dané množiny funkcí zapsat, podle [Hitoshi 2001] roste s délkou genomu množství vyjádření funkcí dokonce exponenciálně.



Obr. 4.27 Ukázka stavu po proběhnutí 11 cyklů

Jiným nežádoucím efektem je hromadění nefunkčních částí genomu, tzv. intronů, způsobujících, jak je všeobecně známo, také exponenciální růst délky zápisu funkce [Hitoshi 2001], [Koza 1994]. Tento efekt je označováno jako blowing). Podrobněji se touto problematikou zabývá [Langdon], kde je dokazováno, že délka intronů a tím i celková délka genomu roste v průběhu evoluce maximálně kvadraticky.

Někdy se může algoritmus takto pokoušet vygenerovat potřebné konstanty, pokud nejsou součástí výchozí sady symbolů [Koza, str. 40]. Protože parametry bývají součástí většiny reálných funkčních závislostí, je lépe o ně doplnit sadu výchozích symbolů a jejich hodnoty hledat jiným způsobem, odděleně od hlavního algoritmu genetického programování [BT5]. Obdobný postup je použit v této práci, při řešení hlavní části zadání (kapitola 6).

Po setřídění program projde všechny funkce a kde jsou za sebou dvě stejné (identické řetězce), druhou smaže. Dělá to před tím, než smaže starou generaci (pořadí vygenerovat nové – seřadit – smazat duplicitní – smazat tolik prvků z původních, kolik ještě zbývá k návratu k původnímu počtu; při mazání staré generace si tedy musí pamatovat, které prvky z ní pochází; není-li zaškrtnuto, maže nakonec ty s nejhorší hodnotou účelové funkce). Obrázky ale pochází z verze, která to ještě neumožňovala.

Zobrazení hodnoty účelové funkce je v seznamu funkcí (obr. 4.27) také barevně. Při chybě menší než 10 je světle modře (cyan), do 100 zeleně, do 1000 žlutě, vyšší řády oranžově, červeně a nad milion hnědě. Hodnota účelové funkce se při zobrazení průběhu jednotlivé funkce (zobrazí se při kliknutí na zápis, viz výše) vypisuje v záhlaví okna s grafem.

Výsledné průběhy získané ze simulací jsou obdobné, jako u předchozích. Složitější funkce hledá program obtížněji. Rychlost nalezení řešení závisí do značné míry na pravděpodobnosti, s kterou jsou generovány ty z funkcí, které hledanou funkci tvoří. Pravděpodobnosti lze samozřejmě v prvním okně měnit. Problémem je, že často hledáme funkci, kterou dopředu neznáme (jinak by nešlo o praktické použití). Dalším problémem je, že v reálném světě se funkce vyskytují s koeficienty (můžeme si je představit jako násobící (koeficienty), ale i sčítací (konstanty)). Příkladem je tentýž průběh naměřený v různých časových nebo hodnotových jednotkách. Praktické nasazení genetického programování tedy musí řešit i tento problém (zápis koeficientů do genomu, určení hodnot).

5. Program pro symbolickou regresi

5.1 Zadání

Program má vyhledávat funkční závislost pro vztah *dvou proměnných*, s možností zahrnutí koeficientů, jejichž hodnoty má také určovat.

5.2 Návrh řešení

V tomto případě již nebude výhodné, aby zadání průběhu bylo dáno jen jako sloupeček čísel, jako v ukázkovém příkladu. Aby bylo zadání omezeno co nejméně, program načítá data z textového souboru, kdy v každé řádce jsou vždy tři čísla: hodnota nezávislé proměnné *x*, hodnota nezávislé proměnné *y* a změřená hodnota funkce v tomto bodě.

Při řešení jsem se držel konceptu, že jedinec je reprezentován řetězcem znaků o délce maximálně 250 znaků, kdy každý znak reprezentuje jednu primitivní funkci (nebo terminální symbol), stejně jako v předchozí kapitole. Znaky jsou všechny ze sady ASCII, tedy sedmibitové. K tomu jsem navrhl způsob záznamu koeficientů.

Koeficienty jsou vždy výhradně násobící (z důvodu vyloučení záměny s konstantou jako terminálním symbolem používám termín koeficient). Vždy se vypočte hodnota stromu funkce v daném bodě a ta se ponásobí hodnotou koeficientu, pokud je tak dané místo stromu označeno.

Místu, kde je ve funkci použit koeficient, odpovídá nastavení nejvyššího (sedmého) bitu, který sada ASCII nepoužívá. Jednotlivé koeficienty tedy nejsou nijak rozlišeny. Proto je ke každému jedinci populace (reprezentován textovým řetězcem jako zápisem funkce, hodnotou fitness a přiřazeným atributem barvy, který se momentálně nepoužívá) přiřazeno vlastní pole o deseti reálných proměnných, které obsahují hodnoty koeficientů v pořadí, jak se uvnitř funkce použijí.

Tato koncepce záznamu koeficientů, umožňující jednoduchý záznam funkce a snadnou práci s ní, představuje hlavní teoretický přínos této práce. Ostatní principy, použité v tomto programu, již byly dříve použity jinými autory a lze je nalézt popsané v literatuře, například gradientní metody jsou základem neuronových sítí a tamtéž lze najít i princip oddělení validační sady (části) dat.

5.3 Sada primitivních funkcí a terminálních symbolů

Jedinec je popsán stromem funkce (viz kapitola 3.1.2, resp. [Koza, str. 37]). Strom se skládá z primitivních funkcí.

Primitivní funkce s aritou 2 reprezentují neterminální symboly se dvěma argumenty. Patří sem sčítání, odčítání, násobení a dělení. Program počítá v reálných číslech. Proti matematickým chybám, jako je dělení nulou, je ošetřen, ale dojde k penalizaci (samozřejmě nemůže počítat limity). Tyto funkce zastupují symboly +, –, *, /. Po nich je v řetězci (záznamu funkce) zapsán nejprve celý „levý“ podstrom a za ním celý „pravý“ podstrom.

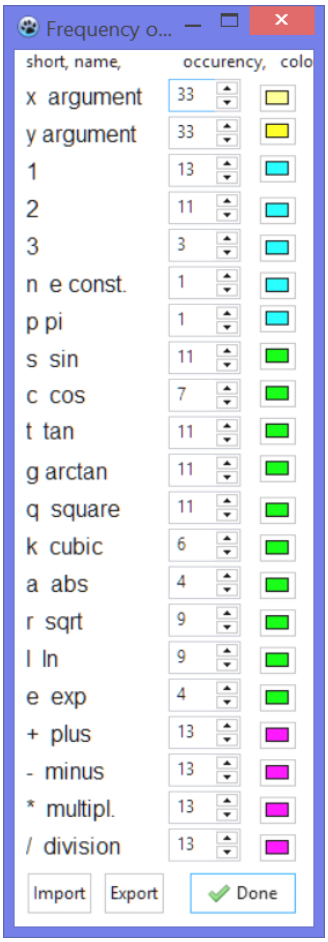
Primitivní funkce s aritou 1, tedy jedním argumentem, jsou uvedeny v následující tabulce:

Tabulka 5.1. Seznam primitivních funkcí s aritou 1 (zkratka v programu, význam, textová forma při výpisu funkce programem v čitelné infixové formě)

s	sin	`sin`
c	cos	`cos`
t	tg	`tan`
g	arctg	`atan`
q	x^2	`square`
k	x^3	`cube of`
a	$\|x\|$	`abs`
r	$\sqrt{x}$	`sqrt`
l	ln	`ln`
e	e^x	`exp`

Mezi terminální symboly (arita 0) pak patří především proměnné ***x*** a ***y***. Dalšími jsou čísla 1, 2 a 3 a konstanty e a π.

V místě neterminálního symbolu se dvěma argumenty by se tedy případné grafické zobrazení zápisu funkce větvilo. Program umí strukturu stromu do jistého rozsahu i zobrazit (viz dále).



Obr. 5.1 Zadání relativních četností primitivních funkcí a terminálů

Bez důsledku na obecnost možných řešených úloh nejsou u žádné funkce použité v genomu více jak dva parametry. Pokud je u hledané funkce například součet tří argumentů, bude hledán jako rozepsaný do dvou součtů.

Je možné některé primitivní funkce z genomu zadáním nulové relativní četnosti zcela vyřadit. Například v Excelu nemáme k dispozici třetí a v české verzi ani druhou mocninu

(program je pak může například nahradit násobením, nebo nalezne jiné řešení). Podle [BT5] je také dobré vyřadit záměnné funkce, například jednu z dvojice sinus-cosinus. Analogicky se při testování prokázalo, že program používá konstanty e a π i v místech, kde to není opodstatněné; pokud je podobná konstanta upravena násobením koeficientem, který upřesňuje její hodnotu (kapitola 6), pak by využití konstanty „1“ bylo logičtější.

Další funkce je možné přidávat jedině modifikací zdrojového textu programu. Program používám jako laboratoř, s tím, že při spouštění z vývojového prostředí lze podobné zásahy kdykoli udělat. Navrhnout program pro použití externích funkcí (například definovaných zápisem v nějakém interpretovaném jazyce) by bylo poměrně komplikované, s možnými důsledky na rychlost programu.

Zápis v paměti tvoří genom jedince. V zápise není žádný další symbol, který by nebyl třeba k popisu funkce, a také žádný nechybí (ani není použit dvakrát). Pro udržení tohoto stavu po mutacích a podobně je vytvořena funkce přeskenování, která je volána po každém zásahu do genomu nebo vytvoření nového jedince a tuto vlastnost zajistí.

Tlačítko „**Colors**“ na hlavním formuláři programu vyvolá okno, kde lze opět zadat relativní četnosti jednotlivých primitivních funkcí (obr. 5.1), obdobně, jako v kapitole 4.3, obr. 4.23.

Vedle tohoto tlačítka je zaškrtávací políčko, kde lze zvolit, že účelová funkce (*fitness*) není počítána jako součet absolutních hodnot odchylek, ale součet čtverců odchylek. V programu nebylo kam dát označení, co provádí které tlačítko. Situaci jsem řešil tím, že pro většinu prvků je vyplněn tzv. „hint“, rada, tedy text, který se po zastavení kursorem (myší) v daném místě zobrazí jako text v malém zpravidla žlutém obdélníku.

Generování nové funkce.

Zápis jedince je tvořen rekurzivně, symbol po symbolu.

Pro vygenerování nového symbolu se nejprve sečtou všechny relativní četnosti, zadané v příslušném formuláři (viz obr. 5.1) pro jednotlivé primitivní funkce a terminály.

Pak se vygeneruje náhodné číslo v takto určeném rozsahu.

Pak se postupně začínají sčítat zadané četnosti, až jejich součet překročí (nebo dosáhne) vygenerované náhodné číslo.

Index, kdy došlo k tomuto stavu, udává hledaný symbol (přeloží se tabulkou symbolů na znak).

Pokud je v rámci mutací či dogenerování třeba vybrat jen symbol z dané skupiny, například terminál, postupuje se obdobně, ale jen s částí tabulky.

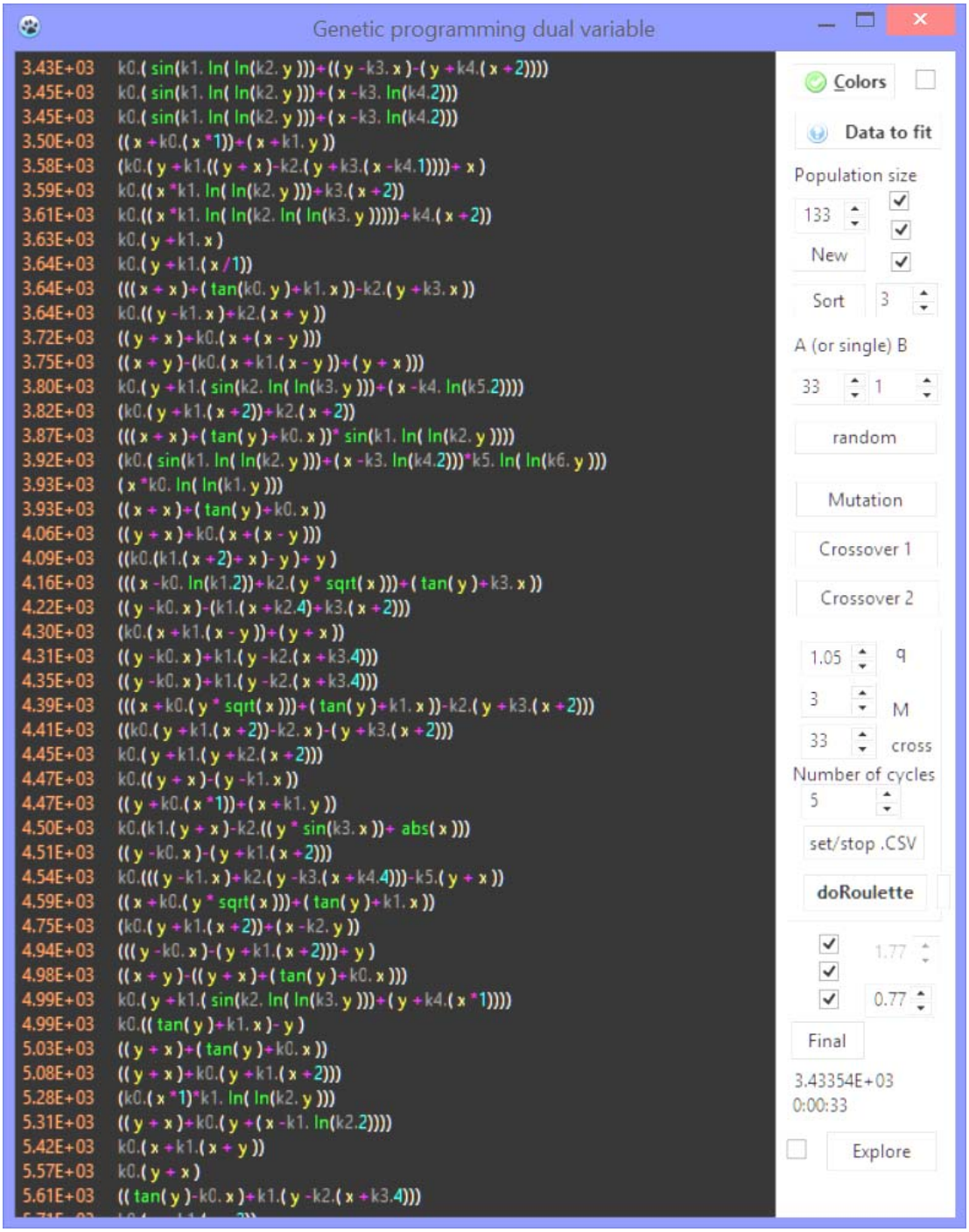
Pokud se vygeneruje symbol, reprezentující primitivní funkci s jedním parametrem, zavolá funkce rekurzivně sama sebe a výsledek volání přidá k touto funkcí vygenerovanému symbolu. Funkce tedy vrací řetězec, reprezentující celý (pod)strom.

Pokud se vygeneruje symbol, reprezentující primitivní funkci se dvěma parametry, funkce volá sama sebe hned dvakrát. Za svůj symbol doplní nejprve výsledek prvního a pak druhého volání.

Pokud by kterýmkoli z připojení symbolu byla překročena přípustná délka zápisu funkce 250 znaků, funkce vrátí chybu. Volající program na to reaguje tím, že toto generování nové funkce volá tolikrát, až vrátí platnou vygenerovanou funkci. Funkce, generující funkci, to sama nemůže udělat, protože neví, zda ji volala nějaká jiná funkce, nebo zda se jedná o rekurzivní volání.

V běhu programu se většinou generují funkce s poměrně krátkým zápisem, v řadě případů naopak pouze osamocený terminální symbol. Takovou funkci ovšem odmítne naopak kontrola funkce přípustnosti (feasibility), a i v tomto případě dojde k novému pokusu

o generování funkce. Průměrná pravděpodobnost vygenerování přípustné funkce nad 30% zajišťuje spustitelnost programu (nedojde k nekonečné smyčce).



Obr. 5.2 Hlavní okno programu po proběhnutí pěti cyklů genetického algoritmu. Nejen v tomto formuláři je minimum místa pro komentáře a vysvětlivky. Program ovšem žádný help nemá a je psán tak, aby jej ani nepotřeboval. Stručné vysvětlení funkce se objeví v klasickém malém žlutém obdélníčku po najetí myší na příslušný objekt (angl. „hint“). To platí i v ostatních formulářích programu.

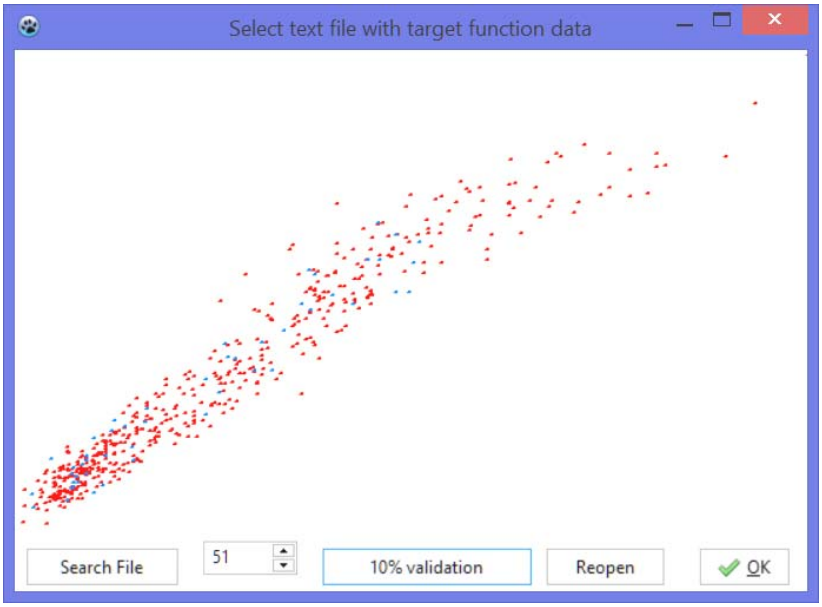
Spolu se zápisem funkce je generováno i umístění koeficientů. Každý vygenerovaný znak je s jistou pravděpodobností označen jako obsahující koeficient (přednastavena je hodnota 33%, tedy cca 1:3). U krátkých funkcí by bylo možné označit, že koeficienty jsou všude, ale v praxi by jejich přílišné množství nebylo žádoucí. V tomto stavu je program schopen generovat i funkce zcela bez koeficientů, zejména, pokud vykazují lepší hodnotu

účelové funkce (to by bylo možné, pokud by se testovací funkce vygenerovala matematicky bez šumu a tak, aby ji šlo vytvořit bez koeficientů).

Funkce přípustnosti (feasibility).

Každá funkce musí obsahovat obě nezávislé proměnné (každou alespoň jednou). Po vygenerování nové funkce je tato skutečnost testována. Pokud jedna z proměnných chybí, není pokus jako u výchozího programu opakován, ale druhá proměnná je doplněna na začátek funkce, pomocí neterminálního symbolu druhé skupiny. Protože každá funkce musí obsahovat minimálně dva terminální symboly, musí obsahovat i minimálně jeden neterminální symbol s dvěmi větvemi (arita 2).

Následně je zjištěno, zda zápis neobsahuje více jak deset koeficientů. Pokud ano, program náhodně jeden odstraní a test opakuje, až do splnění podmínky.



Obr. 5.3 Okno dat po načtení zadání ze souboru a oddělení 10% vzorků do testovací sady (modré tečky). Opakovaným stiskem tlačítka lze zvolit jiné prvky ze souboru, zvolením počtu 0 a novým výběrem se vrátit ke stavu bez rozdělení. Pořadí hodnot v poli je stále zachováváno, pamatuje se v odděleném prvku a lze je uvedeným způsobem vrátit. Hodnota účelové funkce a tedy ani generované funkce na pořadí dat nezáleží, ale použije se při animovaném vykreslování získané funkce.

Hlavní program zajišťuje, že pokud se podaří úspěšně vygenerovat funkci, je před jejím zařazením do populace vypočtena hodnota účelové funkce. Pokud se pracuje s výpočtem hodnot koeficientů (tuto funkci lze vypnout, pak mají všechny trvale hodnotu = 1), pak se program pokusí určit jejich hodnotu. Proběhne základních 10 cyklů. K upřesnění výpočtu dojde v dalším cyklu, po výpočtu jednoho hlavního cyklu genetické evoluce, kdy se vychází z hodnot, dosažených v minulém cyklu. Podrobněji viz kapitola 6.

5.4 Načtení zadání

Druhé tlačítko odshora, Data, otevře formulář pro zadání dat (obr. 5.3). Data jsou očekávána ve formě textového souboru, obsahující tři sloupečky, oddělené tabulátory (žádné další mezery). Protože je použita funkce pro načtení reálného čísla, je třeba respektovat desetinný oddělovač tak, jak je nastaven v operačním systému daného počítače. V každém řádku se očekává trojice čísel x, y a hodnoty hledané funkce. Pro kontrolu, že se nějaká data načetla, je program ihned zobrazí, ale jen v souřadnicích x a z (hodnota hledané funkce). Další zobrazení jsou dostupná dále v programu. Formulář umožňuje znovunačtení těchže dat.

5.4.1 Trénovací a testovací sada.

Z problematiky neuronových sítí i prokládání křivek polynomy metodou nejmenších čtverců je známé rozdělení dat na trénovací a testovací část. Zpravidla, pokud hledáme k daným bodům funkce model neuronovou sítí, postupujeme tak, že si množinu daných bodů náhodně rozdělíme na trénovací, vyhodnocovací a kontrolní část dat. Nejčastěji v poměru 80% pro trénování, 10 až 12% pro vyhodnocování, zda již jsme dosáhli obecného řešení, a zbytek pro závěrečné zhodnocení, s jakou přesností se nám podařilo model sestavit. Má se tím zabránit přetrénování, kdy neuronová síť spíše prokládá zadané body, než aby zobecňovala. Chyba, která je dosažena pro body, které trénujeme, se v anglické literatuře nazývá in-sample-errors, chyba kontrolní sady out-of-sample-errors. Existují metody, jak pro trénování použít celou množinu daných bodů, s tím, že pak dosaženou přesnost jen odhadujeme. Pro neuronové sítě, kde je vývoj přece jen o kousek dál, existují důkazy, že odhad takto dosažené chyby podle některé z těchto metod je pesimistický (síť je ve skutečnosti natrénována lépe).

U genetického programování nedochází k přetrénování, má tedy smysl oddělit jen jednu skupinu dat pro závěrečné zhodnocení kvality dosaženého řešení. V programu na obr. 5.3 je označena jako validační, v nových verzích je i zde termín testovací. V průběhu vývoje programu bylo ověřeno, že testování míry natrénování není možné, a význam se změnil. Je třeba dodat, že u genetického programování se tyto postupy a termíny obvykle nepoužívají.

Pokud je ve formuláři zadávání dat zvolen rozsah testovacích dat, pak při závěrečné prezentaci jednotlivé funkce grafem program zobrazuje jak hodnotu účelové funkce pro data, použitá pro její sestavení, tak i pro data z testovací sady. Pro oddělení rozumné části dat (rozsah lze v jistých mezích nastavit) se použije Fisher-Yatesův algoritmus [Alg-FY], data tedy zůstávají v tomtéž poli, jen je posunuto označení konce naplnění daty (počet vzorků pro vyhodnocování fitness funkce). Program si u jednotlivých hodnot pamatuje jejich původní index a je schopen inverzním algoritmem data vrátit na původní místa v poli (postupuje se podle zaznamenaných indexů, v principu zde probíhá insert-sort, jen není třeba porovnání, protože pozice byla před přehozením Fisher-Yatesovým algoritmem pro tento účel zaznamenána). Opakovaným stiskem lze tak vybrat jinou testovací (validační) sadu. Původní pořadí dat není podstatné pro účelovou funkci, ale použije se při vykreslování; dá se předpokládat, že data byla zadávána nějakým způsobem, takže lze při 2D vykreslení vykreslovat body s pauzou asi 0,05 s, což umožňuje při vykreslování sledovat, který vypočtený bod patří ke kterému vykreslovanému (vykreslují se současně).

Bez načtení dat nelze pokračovat, program žádná data neobsahuje a bez nich skončí chybou.

5.5 Vygenerování nové populace

Vygenerování nové populace vyžaduje naplnění daty z důvodu, že pro každou nově generovanou funkci je ihned počítána účelová funkce. Generování je pro obsluhu jednoduché, stačí zadat počet a zvolit tlačítko New. Vygenerované funkce jsou ihned zobrazeny v levé grafické části hlavního okna programu, tedy v rozsahu, pokud se vejdou. Je možné si zobrazit jen nejlepší členy populace tlačítkem Sort, případně zkopírovat nejlepšího jedince do vyhrazené nulté pozice (elitismus).

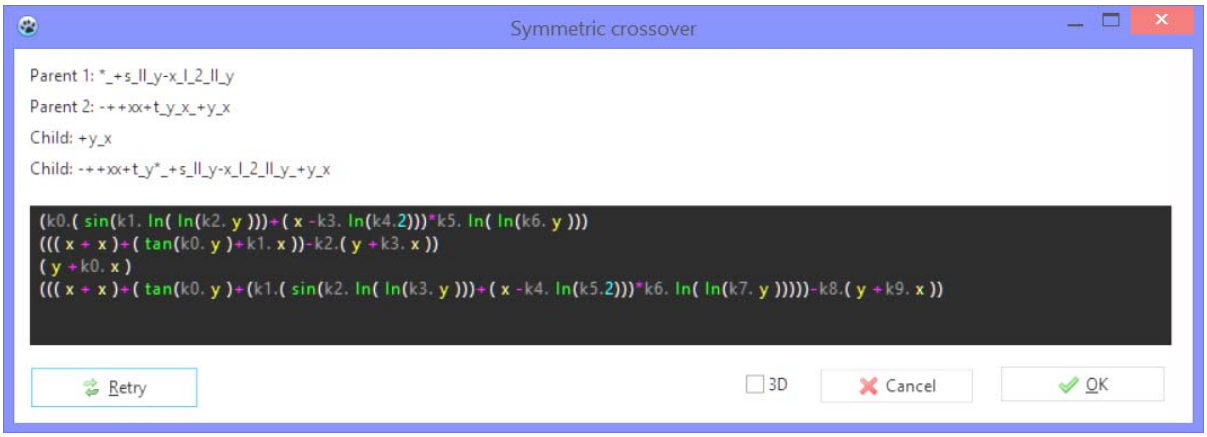
Jako třídicí (správně česky řadicí) algoritmus je zde použit obyčejný Bubblesort. Bylo by možné použít rychlejší běžně dostupné algoritmy (např. nestabilní, ale mimořádně rychlou variantu quicksoru, vytvořenou firmou Borland pro Turbo Pascal 4.0), ale vzhledem k výpočetní náročnosti ostatních funkcí pro to nebyl důvod.

5.6 Jednorázové funkce

Program vznikl úpravou školního programu, a řada funkcí tomu odpovídá. Jsou to například ukázky individuálních mutací a křížení.

Ve dvou následujících polích, označených A a B, lze nastavit číslo jedince, se kterým se pracuje. Mutace pracují s A, křížení potřebují nastavit obě pole. Pro vyplnění lze také použít tlačítko Random. Další možností je kliknout v grafické části na zápis funkce, čímž se vyplní pole A (do pole B lze číslo kopírovat). Okno, vyvolané kliknutím myší na zápis funkce, je třeba samozřejmě zavřít.

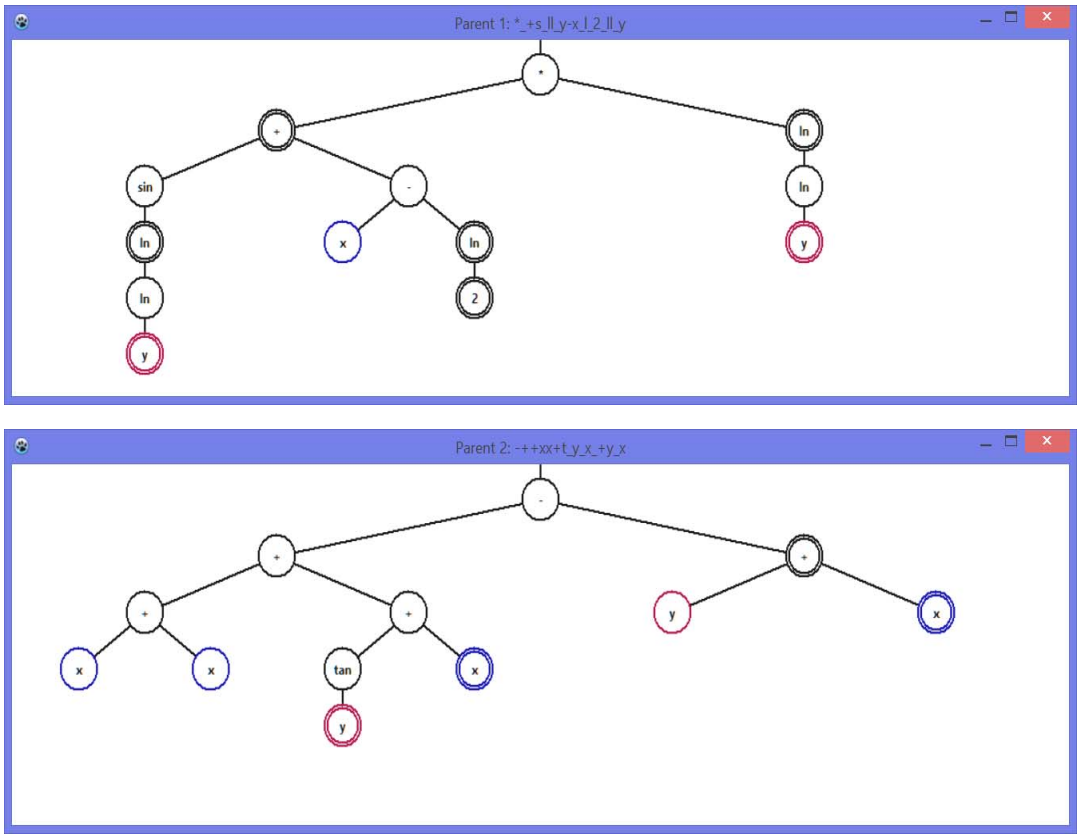
Obr. 5.4 Okno s navrženými mutacemi zvolené funkce. Nutno podotknout, že pokud kterákoli z pěti metod mutace nevrátí přípustnou funkci, program místo ní dosazuje původní funkci bez mutací. Ta je zde v první řádce. Prosté převzetí funkce bez mutace by odpovídalo reprodukci, ale to v tomto programu nemá smysl, protože program neodstraňuje starou generaci, ale vždy pokračuje s nejlepšími jedinci z populace. Mutovaným jedincem lze přepsat původního (jako v hlavním cyklu programu), nebo vygenerovaného přidat na konec. Jaký typ mutace reprezentuje které tlačítko se opět zobrazí v žlutém obdélníčku po najetí myší na příslušné tlačítko.



Obr. 5.5 Obdobné okno pro křížení. Tlačítko "Crossover1" se liší jen tím, že nezobrazuje druhého potomka. Klikem na černou plochu s texty funkcí se vyvolá graf funkce, lze zaškrtnout 3D. V černém obdélníku jsou také zápisy obou rodičů a pak zápis potomků, ale v čitelné formě. Zvolený jedinec se doplní vždy za konec populace (vytvoří se noví jedinci), pokud to nelze, výsledek se neuloží.

Jednorázové funkce Mutation a Crossover vyvolají příslušné formuláře (obr. 5.4 a obr. 5.5), kde jsou viditelné oba způsoby zápisu funkce, u tlačítek interní a v grafické části (černé pozadí) i čitelný. Při kliknutí na čitelné zobrazení (černé pozadí, zvolené barvy) se zobrazí průběh funkce, jako v hlavním okně (obr. 5.9). Při kliknutí na interní zápis v horní části formuláře se vyvolá okno se schematickým zobrazením stromu funkce (obdobné jako na obr.

5.7). U komplikovanějších funkcí nemusí být strom viditelný celý. Dvojité kolečko označuje, že u tohoto uzlu stromu (nebo terminálního symbolu) se provádí násobení koeficientem. Terminální symboly x a y jsou navíc vyznačeny barvou (fialová, modrá). Tímto způsobem je možné při výuce a s trochou štěstí i při přednášce zobrazit nejprve strom prvního a druhého rodiče, a pak potomka, jak jej navrhl počítač (lze požádat o přegenerování, ale nelze přímo určit, kde se má provést prune-and-draft operace se záměnou větví). U mutací lze sledovat funkci jednotlivých mutací na strom. Příslušnými tlačítky lze vybrat funkci pro přidání do hlavní populace a okno zavřít. První tlačítko zapíše jedince zpět bez mutace. Tomu by odpovídala funkce *reprodukce*. V tomto případě ovšem nemá smysl, program neumí mazat starou generaci (problém byl, že ji po všech opatřeních proti ztrátě diverzity, bloatu a podobně nelze rozumně rozlišit).

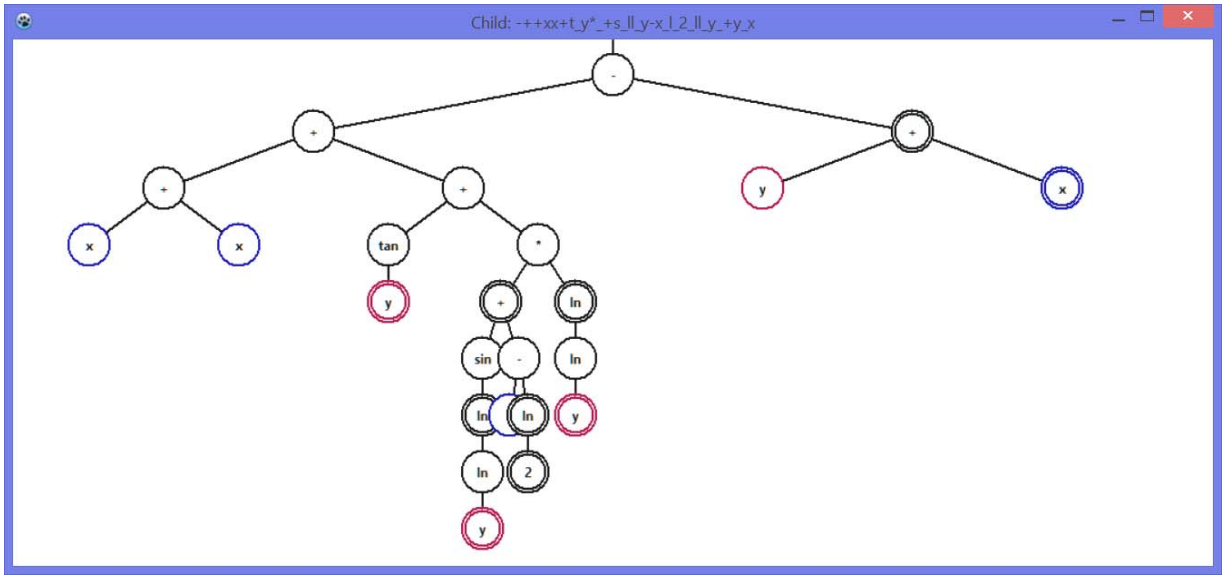


Obr. 5.6 Vygenerovaná schemata rodičů. Dvojitý kroužek značí použití koeficientu.

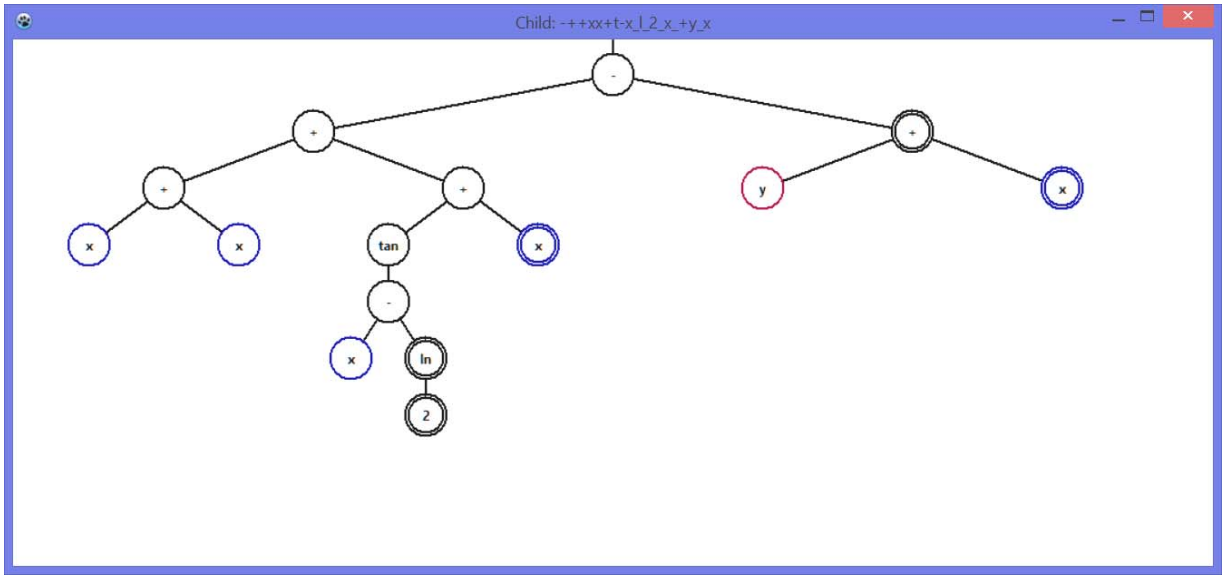
5.7 Hlavní cyklus genetické evoluce

Další tři prvky umožňují nastavit, jak bude probíhat samotná evoluce. Představují evoluční tlak q, relativní pravděpodobnost provedení některé z mutací (význam čísla je počet mutovaných prvků v jednotlivém cyklu v případě běžné diverzity populace, jinak se provádí větší nebo naopak menší počet mutací) a počet provedení operací křížení (provádí se křížení se dvěma potomky; není ale zaručeno, že dva potomci opravdu vzniknou, program operaci v případě několika neúspěšných pokusů vzdá).

Poslední je nastavení celkového počtu cyklů. Pravidelně se tedy provede několik křížení, několik mutací, výsledná populace se seřadí, zobrazí a pokračuje se dalším cyklem. Pokud někdo klikne na grafické zobrazení zápisu funkcí (vlevo v hlavním okně programu), hlavní cyklus se ukončí (po příštím zobrazení).



Obr. 5.7 Ukázka jedince, získaného z jedinců z předchozího obrázku křížením



Obr. 5.8 Jiný jedinec, vzniklý z těchže rodičů.

5.8 Záznam průběhu evoluce

Tlačítko „Start/stop CSV“ umožní zahájit průběžný zápis stavu evoluce do souboru. Ukládá se opět hodnota (účelové funkce) nejlepšího jedince, hodnota rozptylu, průměrná hodnota populace, variační koeficient, a dále délka nejlepšího a průměrného jedince a také nejlepší jedinec (text odpovídající zobrazení v okně Explore, tj. vnitřní zjednodušený zápis, ale umístění max. 10 koeficientů reprezentuje podtržítko před znakem, reprezentujícím funkci). Opakovaným stiskem se záznam ukončí. Při uložení do stejného souboru se původní smaže (nelze připisovat na konec jiného pokusu, v tomto případě by pak graf stejně nereprezentoval jeden pokus). Řádek záznamu se zapíše v každém hlavním cyklu bezprostředně před zobrazením. Data jsou určena pro zpracování v Excelu.

5.9 Účelová funkce

Jak již bylo uvedeno v odstavci 3.1.4, hledáme funkci, která co nejlépe proloží body, dané měřením. Ve standardním vztahu regresní analýzy

$$Y \approx f(X, \beta) \tag{5.1}$$

tedy hledáme nejen vektor parametrů, ale i samotnou funkci f. X je vektor nezávislých proměnných.

Výslednou funkci tedy hledáme ve tvaru

$$z_i = \mathrm{G}(x_i, y_i, \beta) \tag{5.2}$$

Algoritmus genetického programování pracuje s populací funkcí; jednotlivou funkci tedy označíme indexem například *j*, tedy f_j. V tomto programu je definice rozšířena o konstanty. Konstant je maximálně deset. Můžeme je tedy vyjádřit jako vektor konstant β

$$\beta = \{k_0, \ldots, k_9\} \tag{5.3}$$

Vektor píšeme s indexem *j*, protože každé funkci je přiřazen individuální vektor. Dosazením bodů x_i, y_i, pro které je zadána hodnota hledané funkce, pak můžeme vyjádřit hodnotu funkce v bodě jako bodový odhad hledané hodnoty závislé proměnné touto funkcí

$$\hat{z}_i = f_j(x_i, y_i, \beta_j) \tag{5.4}$$

Účelová funkce (funkce, která svou velikostí určuje, který jedinec populace je vhodnější k použití jako zdroj i jako prvek v další generaci) je pak zpravidla [GP, str. 128] definována jako součet absolutních hodnot odchylek ve všech známých bodech funkce

$$F_j = \sum_{i=1}^{n} \left| \mathrm{G}(x_i, y_i) - f_j(x_i, y_i, \beta_j) \right| = \sum_{i=1}^{n} \left| z_i - f_j(x_i, y_i, \beta_j) \right| \tag{5.5}$$

, kde z_i je známá hodnota (změřená data). Alternativně je možné použít variantu s kvadráty odchylek

$$F_j = \sum_{i=1}^{n} \left(\mathrm{G}(x_i, y_i) - f_j(x_i, y_i, \beta_j) \right)^2 = \sum_{i=1}^{n} \left(z_i - f_j(x_i, y_i, \beta_j) \right)^2 \tag{5.6}$$

Tato varianta ale zpravidla příliš zohledňuje body, kde je odchylka větší, a má smysl jen tam, kde je toto chování žádoucí. K jejímu zavedení možná přispěla podobnost výsledků s odhadem podle metody nejmenších čtverců, ale v našem případě nepotřebujeme, aby účelová funkce byla diferencovatelná. V programu si lze z těchto dvou funkcí vybrat (zaškrtávací políčko vedle tlačítka pro zadávání relativních četností primitivních funkcí, zcela vpravo nahoře).

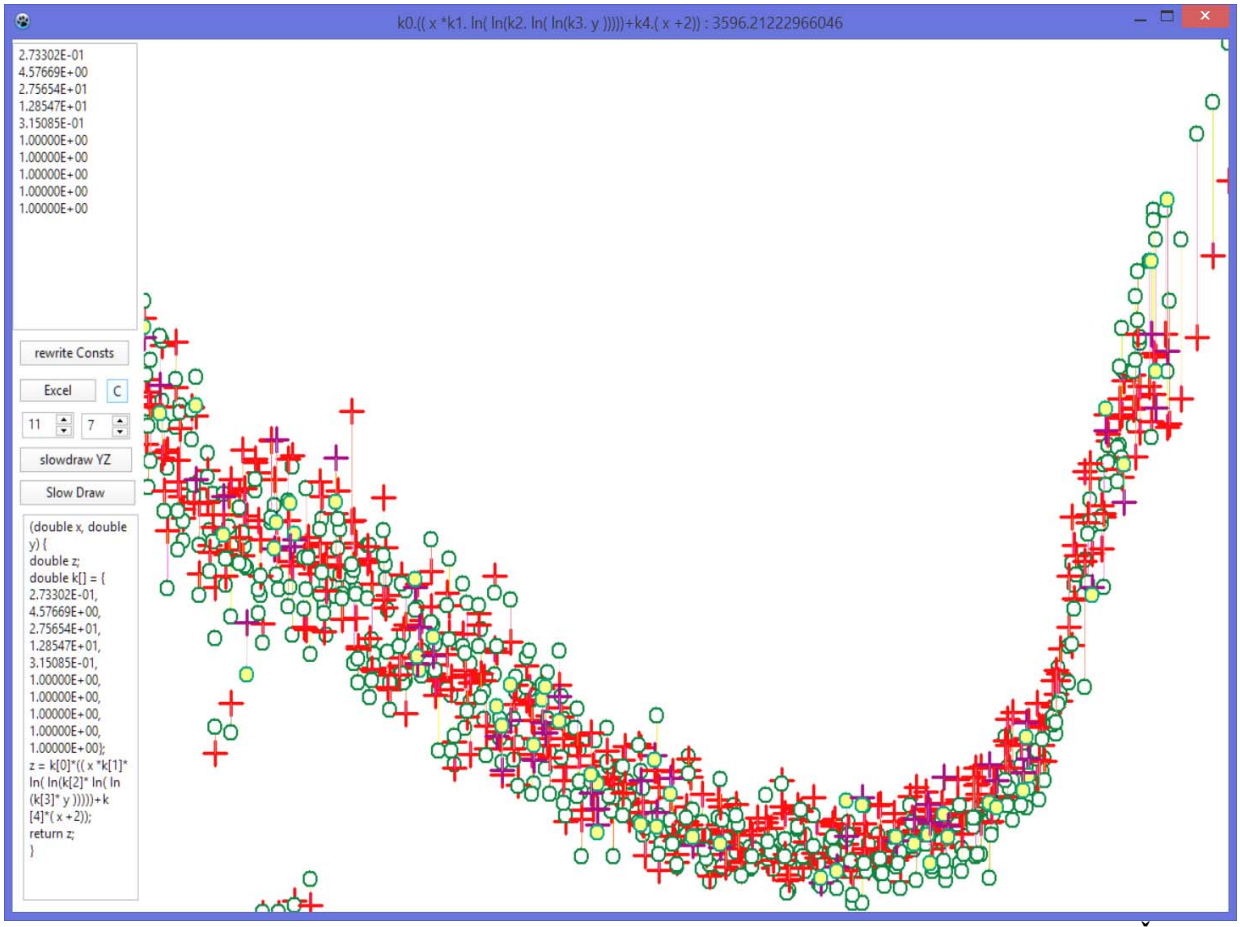
S ohledem na nutnost znevýhodnit funkce s dlouhým zápisem je zavedena pokutová funkce

$$L_j = c \,.\, \ell(f_j) \tag{5.7}$$

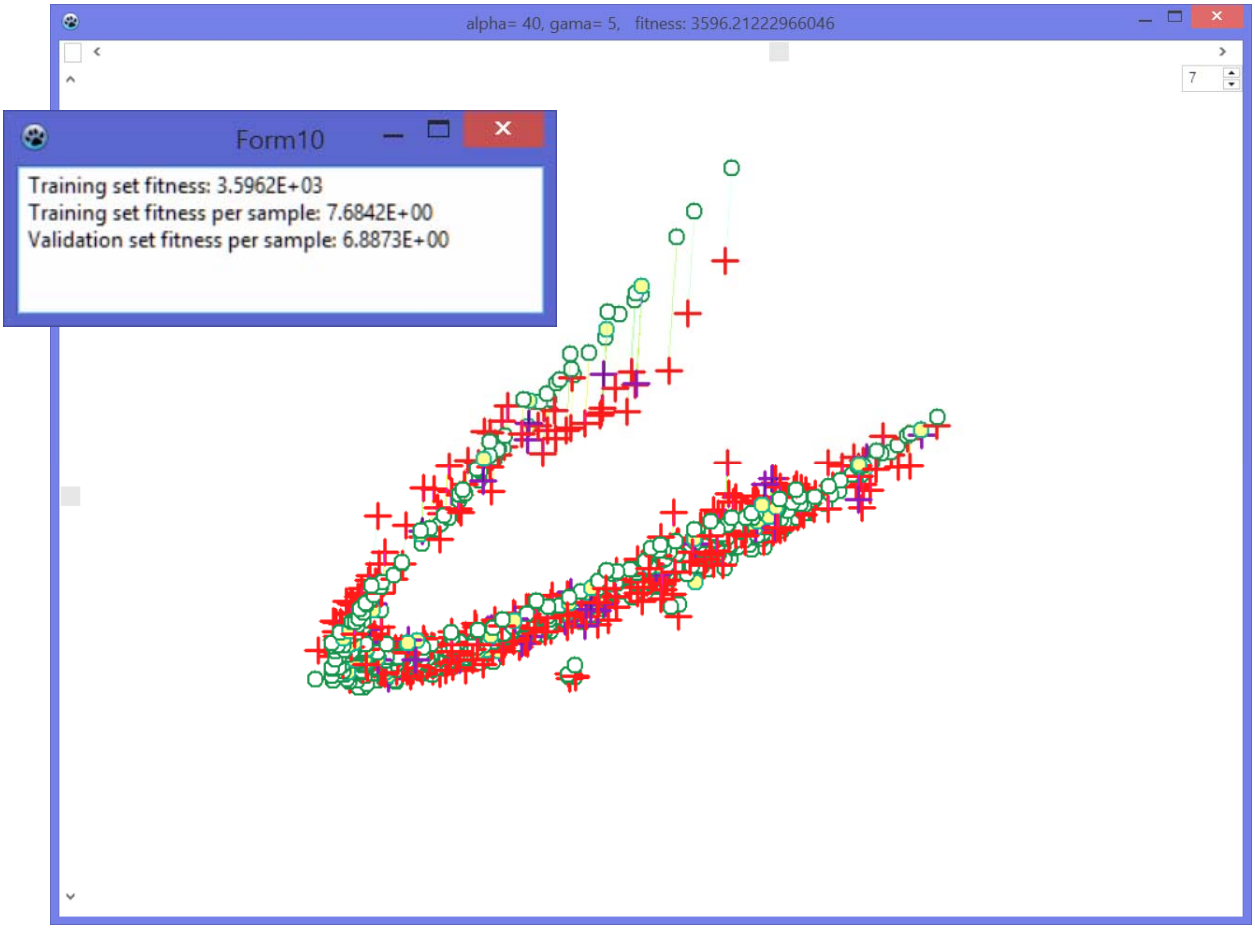
, kde ℓ vyjadřuje délku zápisu funkce a c je konstanta, kterou lze v programu nastavit. Výsledná korigovaná účelová funkce pak má tvar

$$F_{jl} = F_j + L_j \tag{5.8}$$

Program samozřejmě upřednostňuje funkce (jedince) s nejmenší hodnotou účelové funkce.



Obr. 5.9 Zobrazení výsledné funkce ve 2D nabízí přenos do Excelu nebo návrh funkce pro C++. Žlutá kolečka a fialové křížky patří do kontrolní (testovací) sady.



Obr. 5.10 3D zobrazení je vhodnější pro představu, jak výsledná funkce vypadá. Kontrolní sada je vyznačena odlišnou barvou koleček (žlutá výplň) a křížků (fialové). Při vykreslení okna, je-li zvolena testovací část dat, je zobrazeno okno s relativní chybou testovacích dat, viz obrázek vlevo nahoře.

5.10 2D a 3D zobrazení

Graf funkce je vyvolán kliknutím na její barevný zápis v grafickém okně programu (má černé nebo velmi tmavé pozadí). Které zobrazení se vybere určuje zaškrtávací políčko dole, vedle tlačítka editoru funkcí.

Pro vykreslení grafu ve 2D módu je třeba určit minimální a maximální hodnotu každé proměnné (x_{min}, x_{max}, z_{min}, z_{max}), a následně i velikost okna, kam lze vykreslovat (w je šířka, h je výška). Transformační funkce pro vykreslení bodu pak jsou:

$$x_p = \frac{x - x_{min}}{x_{max} - x_{min}}.w \tag{5.9}$$

$$y_p = \frac{z - z_{min}}{z_{max} - z_{min}}.h \tag{5.10}$$

Ve skutečnosti ovšem musíme svislou osu vynášet záporně, protože v počítači je nula na obrazovce vlevo nahoře

$$y_{p,inv} = h - \frac{z - z_{min}}{z_{max} - z_{min}}.h \tag{5.11}$$

Ve vzorcích chybí zaokrouhlení na celá čísla.

2D zobrazení dále nabízí export funkce do Excelu (funkci předpokládá nakopírovat do konkrétní buňky programu, B2, a konstanty ručně z tabulky konstant do sloupečku buněk K1 až K10. Výpočet hodnot pak záleží na uživateli; zadaná data nejsou samozřejmě exportována vůbec, předpokládá se, že byla načtena z Excelu a podobná akce tak postrádá smysl. Obdobně je řešena funkce pro použití v programech psaných C++, kde program vygeneruje funkci, kterou lze ihned použít. Tentokrát je víceřádková a je generována do příslušného okna. Pro vykreslení funkce současně s daty je pro představu lepší využít 3D zobrazení.

Pro 3D je třeba před vykreslením nejprve souřadnice transformovat, například:

$$\begin{pmatrix} x_r \\ y_r \\ z_r \end{pmatrix} = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{5.12}$$

pro rotaci okolo osy z (otočení, umožní se podívat na funkci ze strany), resp.

$$\begin{pmatrix} x_r \\ y_r \\ z_r \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{5.13}$$

pro častěji vhodnou rotaci okolo osy x (naklopení, umožní se podívat na funkci odshora). Zbývající rotace by naklápěla obrázek okolo nevykreslované osy, směřující tedy k uživateli, ve smyslu otočení jako u hodinových ručiček. Při této rotaci je stále vidět to samé, a nemá tedy praktický smysl (u grafů ji uživatel ani neočekává).

Pokud se mají vykreslovat natočené grafy, je vhodné tak činit ve zmenšeném měřítku, aby se celý natočený obalující kvádr vždy vešel na obrazovku.

U 3D zobrazení jsou všechny hodnoty před otočením z důvodu rychlosti vykreslování vypočteny předem a při zadávání natočení jsou jenom transformovány. Předem vypočtené hodnoty jsou navíc upraveny odečtením konstanty a podělením rozsahem v každém směru tak, aby rozsahu, který odpovídá vykreslovaným bodům zadání, odpovídaly hodnoty od –1 do +1. Výsledné výpočty pak vychází jednodušší. Kolem takto zvoleného bodu se souřadnicemi v počátku se pak snáze provádí rotace.

Aby byl graf přehlednější, jsou body vykreslovány po vrstvách, podle jejich hodnoty v nezobrazované ose. Pro dosažení perspektivy se relativní souřadnice po transformaci ve dvou zobrazovaných osách zmenšuje/zvětšuje podle hodnoty nezobrazované souřadnice. Korektní výpočet souřadnic je popsán například v [Venugopal, str. 605] (lze dohledat přes Googlebooks). V tomto programu je použito zjednodušení, kdy se začíná s nejvzdálenějšími body, ale transformace zmenšení ve zobrazovaných souřadnicích se provede stejná pro všechny body v určitém rozsahu, pak se zmenšující koeficient ponásobí vhodným zvětšením a vykreslení se opakuje pro další vrstvu bodů.

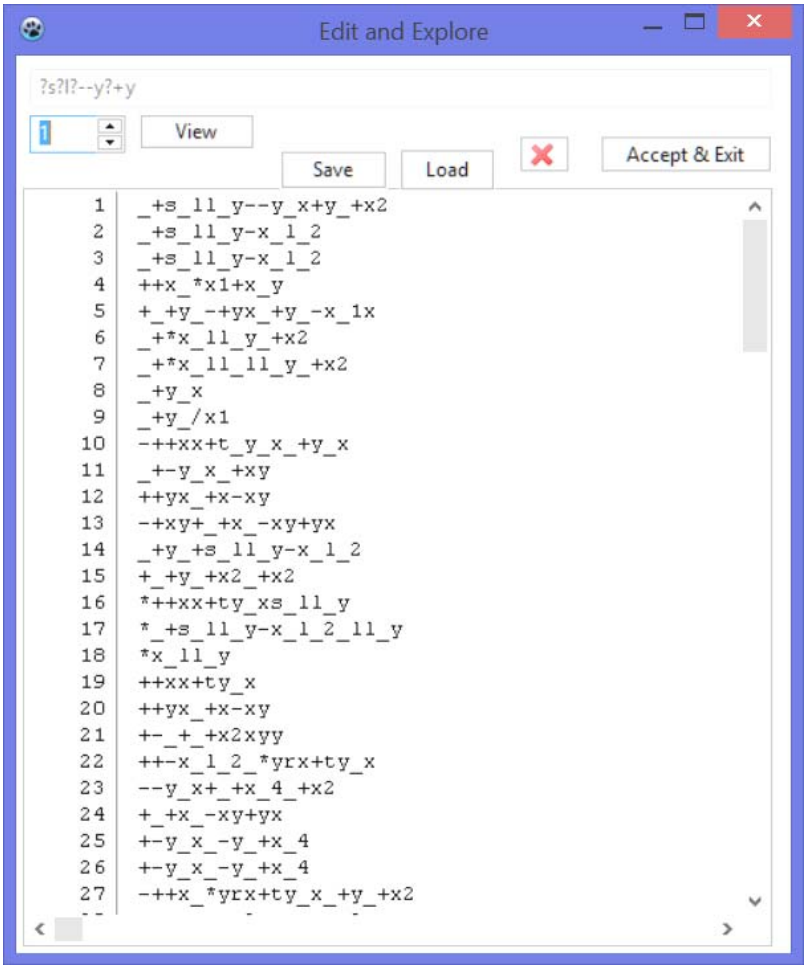
Výpočet rozsahu nezobrazované osy není třeba (body jsou spočteny pro předem daný rozsah, resp. prostorovou krychli), ale pokud by se provádělo, výpočetní náročnost by byla O(n) (ve skutečnosti 2n-2 porovnání, n je počet bodů, ale máme různé hodnoty pro zadané body a body vypočtené zobrazovanou funkcí, proto 2x). Samotné vykreslování po vrstvách, které nahrazuje třídění, které by mělo být správně před zobrazením provedeno, má opět

výpočtovou náročnost O(n), resp. O($n.m$), kde m je počet vrstev (technicky se jedná o ekvivalent bucket-sortu, [Alg-BS]). Časová náročnost vykreslení grafu je tedy dána převážně grafickými operacemi. Ve výsledku lze grafem s tisícovkou párů zobrazovaných bodů plynule pohybovat posuvníkem myší, dokonce i při zobrazení bodů plnými kroužky.

Bod středu perspektivy je umístěn (překrývá se s) do souřadnic (0,0,0) přepočtených hodnot zadaných dat/nalezené funkce. Cyklů pro vykreslení je v aktuální verzi 50, první vykreslení se provádí zmenšené na 0,9 násobek rozměru, tomu odpovídá změna koeficientu zmenšení v každém kroku

$$r = \sqrt[25]{\frac{1}{0{,}9}} \approx 1{,}004 \tag{5.14}$$

V programu je vypočtena přesná hodnota. Koeficient se uvedeným číslem násobí. Počet vrstev je třeba zvolit tak, aby co nejméně zdržovalo, protože se vždy provádí dvě porovnání každé transformované souřadnice, zda patří do daného cyklu, což přes rychlost operace porovnání dvou obdobně velkých čísel nějakou dobu trvá a při příliš velké hodnotě by tento čas nešel zanedbat. Příliš malé hodnoty naopak vedou k příliš viditelnému přeskakování bodů z vrstvy do vrstvy v průběhu otáčení (otestováno při devíti vrstvách).



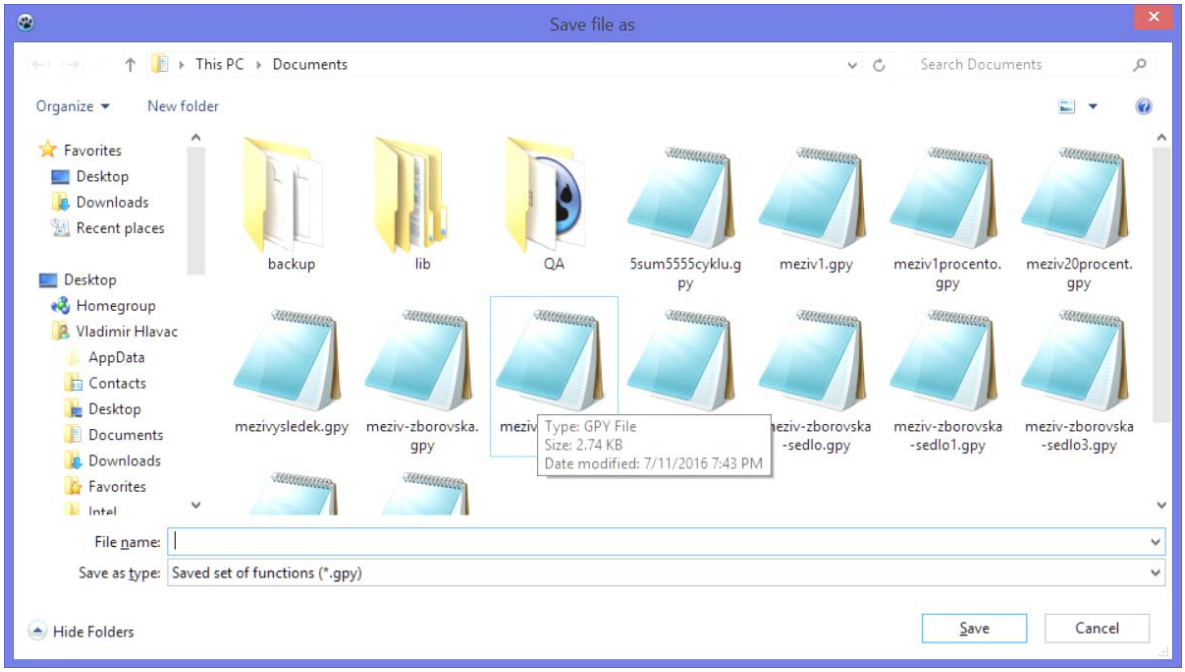
Obr. 5.11 Editor funkcí vyvoláme tlačítkem Explore. Umožňuje především uložit mezivýsledky, pokud chceme s rozpracovanou populací pracovat později znovu. Vytvořené textové soubory mají příponu .gpy a jsou přesně ve tvaru, jak jsou zobrazeny v tomto editoru (hodnoty koeficientů se po načtení/editaci dopočítají znovu).

V záhlaví 2D grafu je zapsána funkce v čitelném tvaru (infixová notace doplněná o závorky). V záhlaví 3D grafu jsou okamžité úhly natočení, jak jsou nastaveny posuvníky (v souladu s terminologií, kdy graf v základní poloze zobrazuje osy **x** a **z**, jsou úhly označeny **α** a **γ**). Označení úhlů je slovně, použití řeckých znaků by mohlo vést k nekompatabilitě, například při překladu v jiném prostředí (Linux).

5.11 Editor funkcí

Editor funkcí vyvoláme tlačítkem Explore. Pro tento formulář byl použit editor, který je dostupný jako komponenta ve standardní nabídce Lazarusu, a který umožňuje zobrazovat číslování řádek (obr. 5.11).

Aby bylo možné funkce zobrazit, je třeba oddělit zobrazení nejvyššího bitu, který označuje konstantu. Pokud je výsledná hodnota primitivní funkce (nebo hodnota terminálu) násobena konstantou, zobrazí program symbol podtržítka a za ním symbol s nejvyšším bitem nulovým. Při potvrzení editace se zápis překóduje zpět (resp. celá populace se nahradí funkcemi, zapsanými v okně textového editoru). Ručně tedy lze měnit nejen tvar funkce, ale i umístění koeficientů. Případně je zde možné si místo některé funkce zapsat vlastní řešení, pro které se ihned při uložení vypočtou hodnoty koeficientů, byly-li použity, a lze si je opakovaným vyvoláním editoru i zobrazit. Vložení vlastního řešení je nestandardní, proto není ani nijak více podporováno.



Obr. 5.12 K ukládání mezivýsledků je použita standardní funkce vestavěného editoru

Měnění hodnot koeficientů je proti smyslu tohoto programu. Technicky lze nastavit jejich počáteční hodnotu při 2D zobrazení (tlačítko „rewrite Consts“), ale jejich definitivní hodnota bude nastavena znovu v průběhu výpočtu účelové funkce, kdy dojde k jejich novému zpřesnění zvolenou metodou. Mělo by to smysl v případě, že nalezená funkce má více kombinací koeficientů, ke kterým vede upřesňující algoritmus (lokální minima). Technicky by bylo možné na hlavním panelu upřesňování koeficientů gradientní nebo iterační metodou zakázat a pak je nastavit ručně trvale, ale neumím si představit situaci, která by to zdůvodnila. Pokud to takto nastaveno není, jsou koeficienty přepočítány znovu ihned po uložení populace a uzavření okna, tedy po použití tlačítka „Accept & Exit“.

Zvolením čísla řádku vedle tlačítka „View“ lze zvolit číslo funkce a tu lze pak tlačítkem zobrazit ve 2D grafu, kde je v záhlaví funkce i vypsána v čitelném tvaru. Číslo řádku se volí podle editoru (program vnitřně odčítá jedničku, protože vnitřní rozsah pole je od nuly).

5.12 Specifický blowing

Vlastností zápisu funkce je, že může obsahovat například část ve tvaru (x–x), a pokud se podobným výrazem násobí, celý součin se tím nuluje (a výsledek je pak dán zbývající částí zápisu funkce). Pokud dojde k podobnému efektu, mluvíme o nefunkčních částech genomu (intronech) nebo o blowingu. Specifickou vlastností genetického programování je, že křížení většinou vede na vytvoření jedince, který má horší vlastnosti, než rodiče; pokud pak část genomu je kompletně nefunkční a jejím odříznutím a záměnou za jinou část se vlastnosti jedince nemění, pak je takový jedinec proti jiným, opravdu vzniklým křížením, vlastně zvýhodněn. Mít nefunkční část genomu se pak stává evoluční výhodou.

Zařazením dopočítávání koeficientů dochází k další možnosti vzniku nefunkčních částí. Pokud nějaká část stromu je pro tvar funkce nevhodná, může ji program odstranit tím, že jí postupně nastaví velmi nízký koeficient. Matematické funkce procesoru Intel pracují s přesností do 1.10^{-16} (šestnáct platných míst). Pokud po přenásobení koeficienty vyjde například hodnota jedné větve součtu (rozdílu) více jak 10^{16}-krát větší než druhá, pak se stane hodnota druhé větve pro výsledek nevýznamná (efekt se jmenuje podtečení a procesor jej při standardním nastavení ignoruje). V této větvi se pak může tvar funkce (genom) měnit jakkoli, bez vlivu na výslednou hodnotu.

Zajímavým problémem umělé inteligence by bylo podobné úseky genomu vyhledávat a odstraňovat. Tento program to samozřejmě nijak neřeší. Funkce s nadbytečnou délkou genomu jsou penalizovány a míru penalizace lze nastavovat. Pokud se zvolí dostatečně velká, program generuje jen samé krátké funkce, i když bez penalizace by delší funkcí dosáhl lepšího proložení daty.

5.13 Vyhodnocování stárnutí populace (aging)

Program umožňuje nastavit, že starší jedinci, než je nějaká nastavená hodnota, jsou z populace vyřazováni (kapitola 2.2.8). Pokud je tato volba aktivní, je možné dalším zaškrtávacím políčkem podmínky zmírnit; v tomto případě jsou pak starší jedinci než je zvolená hodnota z populace odstraňováni, ale s pravděpodobností, která je dána poměrem překročení nastavené hodnoty životnosti a touto hodnotou, tedy pro dvojnásobek životnosti je pravděpodobnost jedna.

Nastavením doporučené maximální doby života na hodnotu jedna lze (za dalších podmínek) dosáhnout stavu, kdy stará generace je (až na elitního jedince) smazána. Některé publikace doporučují toto řešení jako ochranu před zamrznutím populace.

Hlavní cyklus ruletové selekce probíhá následovně:

- je generováno m mutovaných jedinců, o počtu podrobně u kapitoly o mutacích,
- je generováno 2x k nových potomků křížením, počet k lze zadat,
- je-li to zvoleno, provede se v-krát variace konstant
- u nových jedinců jsou v okamžiku generování poprvé nastaveny koeficienty zvolenou metodou,
- pak jsou všechny prvky podle účelové funkce seřazení,
- následně jsou odzadu mazáni jedinci, kteří překročili plánovanou dobu života (je-li to zvoleno), ovšem nikdy ne tolik, aby se velikost populace snížila pod nastavenou hodnotu,
- nakonec jsou odzadu smazány nejhůře hodnocené prvky tak, aby se počet vrátil na nastavenou hodnotu (nastavená velikost populace), tentokrát již bez jakéhokoli jejich dalšího hodnocení.

Z toho je zřejmé, že pokud se nevygeneruje dostatečný počet nových jedinců, nelze dosáhnout stavu, kdy jsou v populaci jen samí noví jedinci. Přitom provedení operace křížení nebo mutace do volného prostoru za populací nezaručuje, že bude vytvořen přípustný jedinec. Pokud tedy chceme modelovat chování systému s mazáním nové generace, musíme nastavit počet nově generovaných prvků (zejména prvků, které vznikají křížením) tak, aby bylo v každém cyklu generováno více prvků, než je nastavená velikost populace.

5.14 Ukázky průběhu evoluce pro několik různých nastavení

Nastavením parametrů můžeme dosáhnout různého chování evoluce. Pro základní nastavení pro porovnání jsem zvolil parametry, ke kterým jsem dospěl při testech s genetickými algoritmy v kapitole 4, se zohledněním míry komplexnosti problému. Základním nastavením byla kombinace:

Tabulka 5.1 Základní nastavení programu pro testování vlivu parametrů

velikost populace	130
elitismus	ano
stárnutí populace (vyřazování při překročení věku)	ne
počet křížení za cyklus	70
počet mutací	5, připisovat na konec
počet mutací variabilní s variačním koeficientem	ano
vyhodnocování chyby	suma čtverců odchylek
variace konstant	30/cyklus
metodika určování koeficientů	EGD (kapitola 6.3)
snižující se krok při upřesňování („simulované žíhání“)	ano, koef. 0,33
délka zápisu jedince připočtena k účelové funkci	5x

Ostatní testovací běhy byly provedeny vždy pětkrát, při změně jen jednoho nebo vybrané kombinace parametrů.

Výsledky jsou vynášeny do grafů, kde čáry mají následující význam:

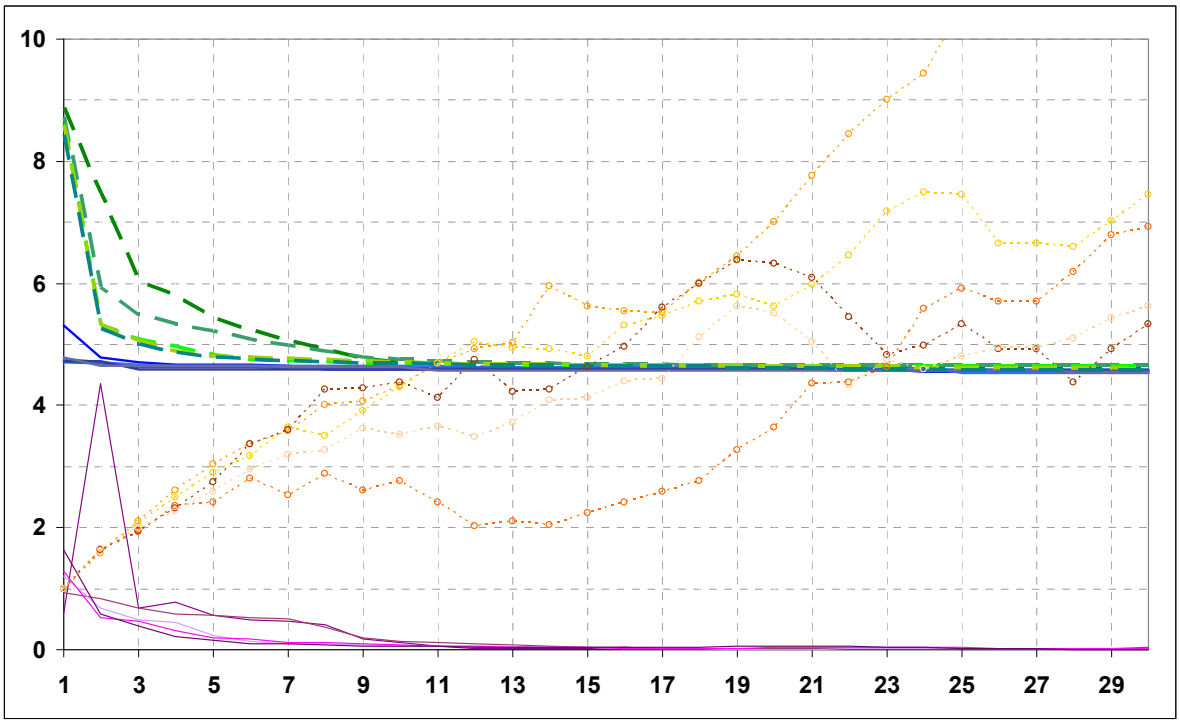
modrá silná – nejlepší jedinec. S ohledem na data vynášen dekadický **logaritmus**.

zelená přerušovaná – průměr populace. Také vynášena hodnota **logaritmu**.

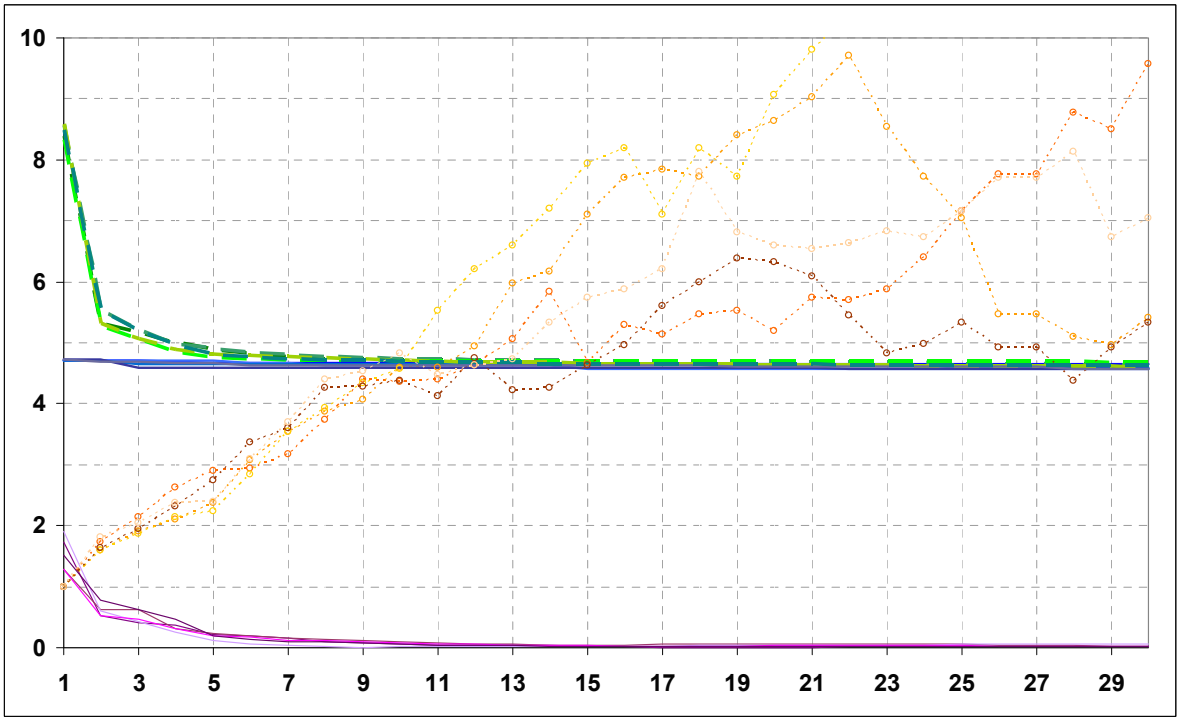
fialová tenká souvislá – variační koeficient, vynášen přímo, obvykle <10.

oranžová tečkovaná se značkami – průměrný věk jedinců v populaci.

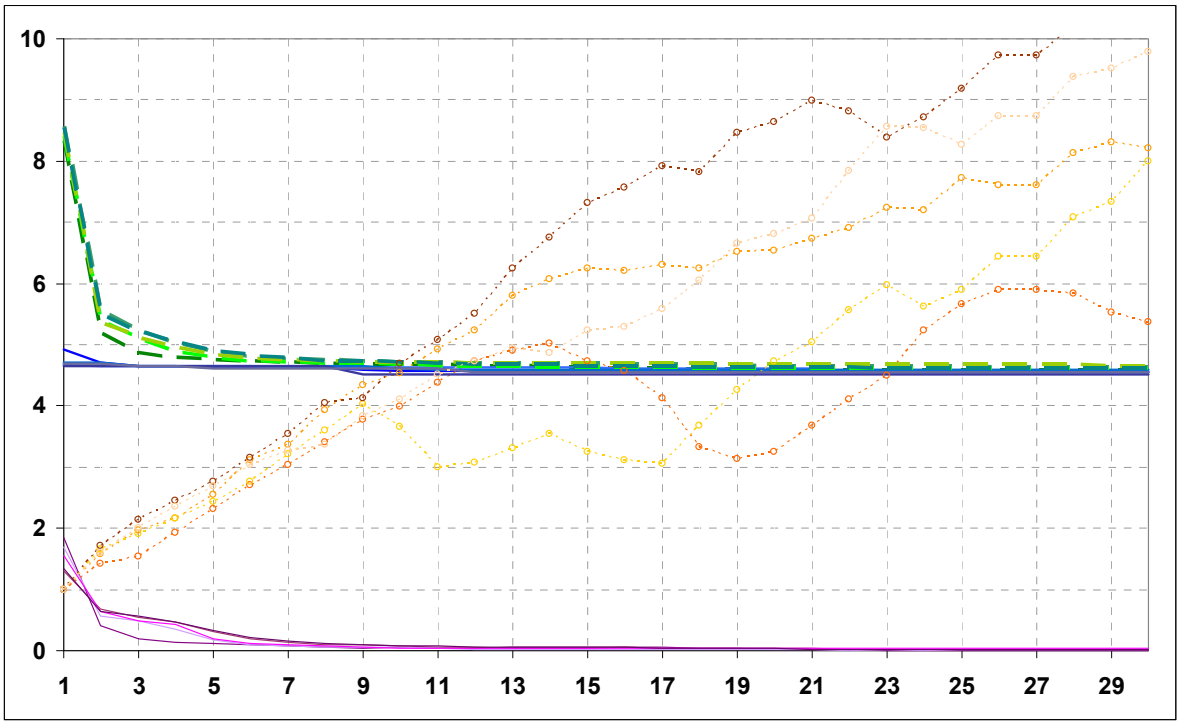
Testovací data jsou stejná, jako v [Hlaváč], v tomto případě se je jedná o dvě intenzity provozu (nezávislá a závislá proměnná) a jako druhá proměnná je použit čas bez datumu. Data pochází z úseků, kdy je nízká intenzita provozu, a vykazují tak dobrou závislost (přijatelnou úroveň šumu).



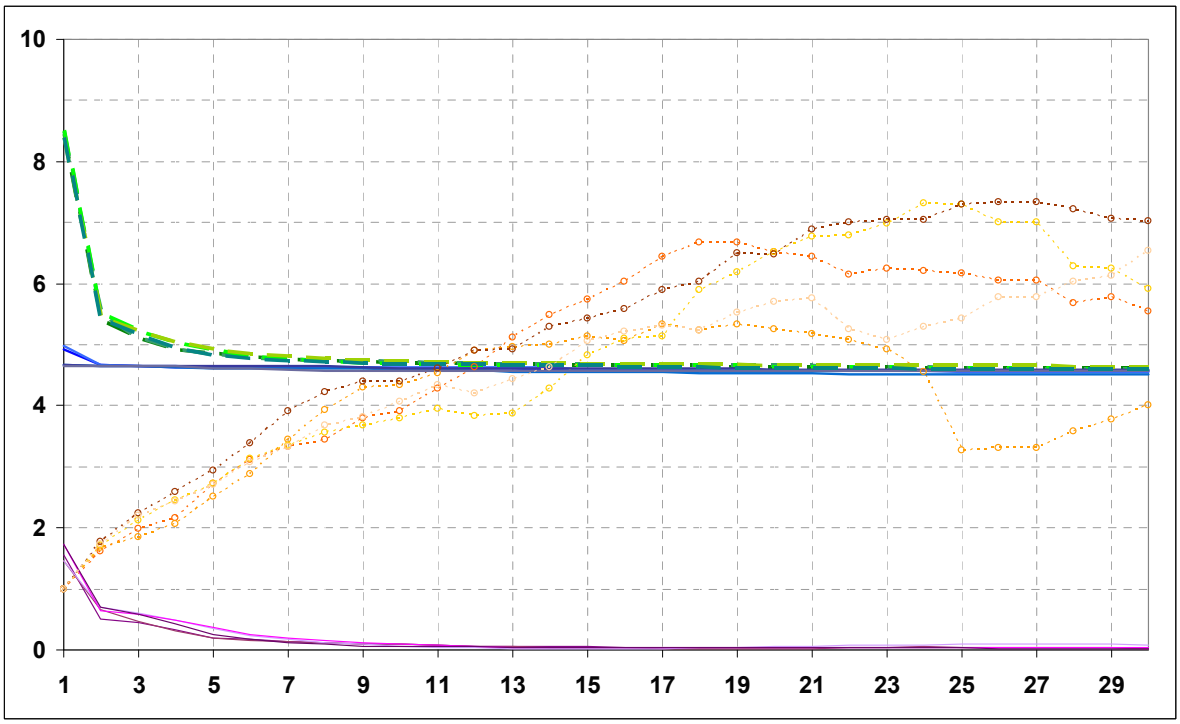
Obr. 5.13 Toto je výchozí nastavení. Populace 130 jedinců, 70 křížení v cyklu. Zelená přerušovaná (průměr populace) a modrá (nejlepší jedinec) vynášeny v logaritmickém měřítku svislé osy.



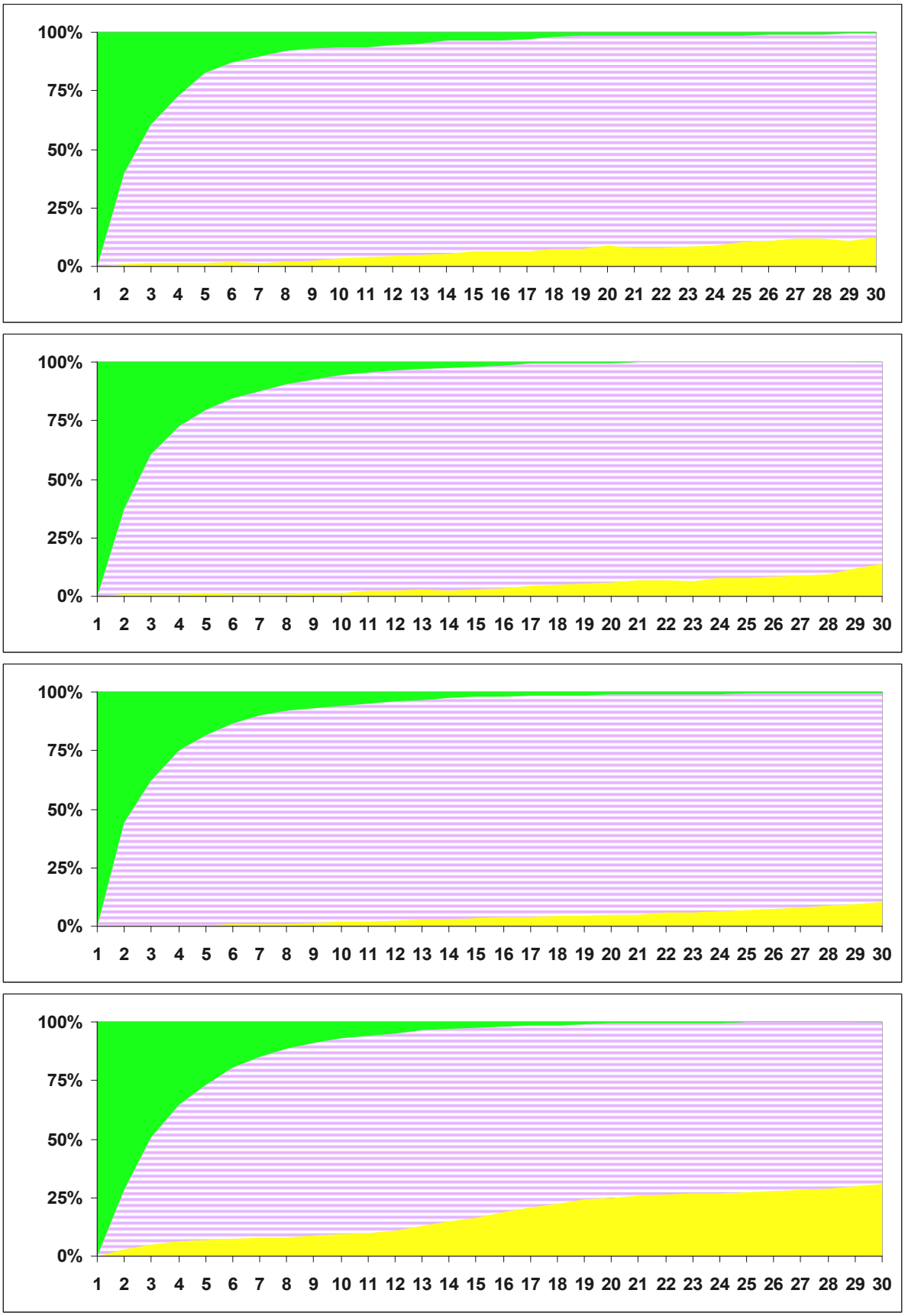
Obr. 5.14 Menší populace 60 jedinců, 33 křížení v cyklu, mutace nastaveny na 3x a proporcionálně k variabilitě. Populace není úplně zamrzlá, je vidět, že průměrný věk jedince se drží mezi pěti a osmi generacemi.



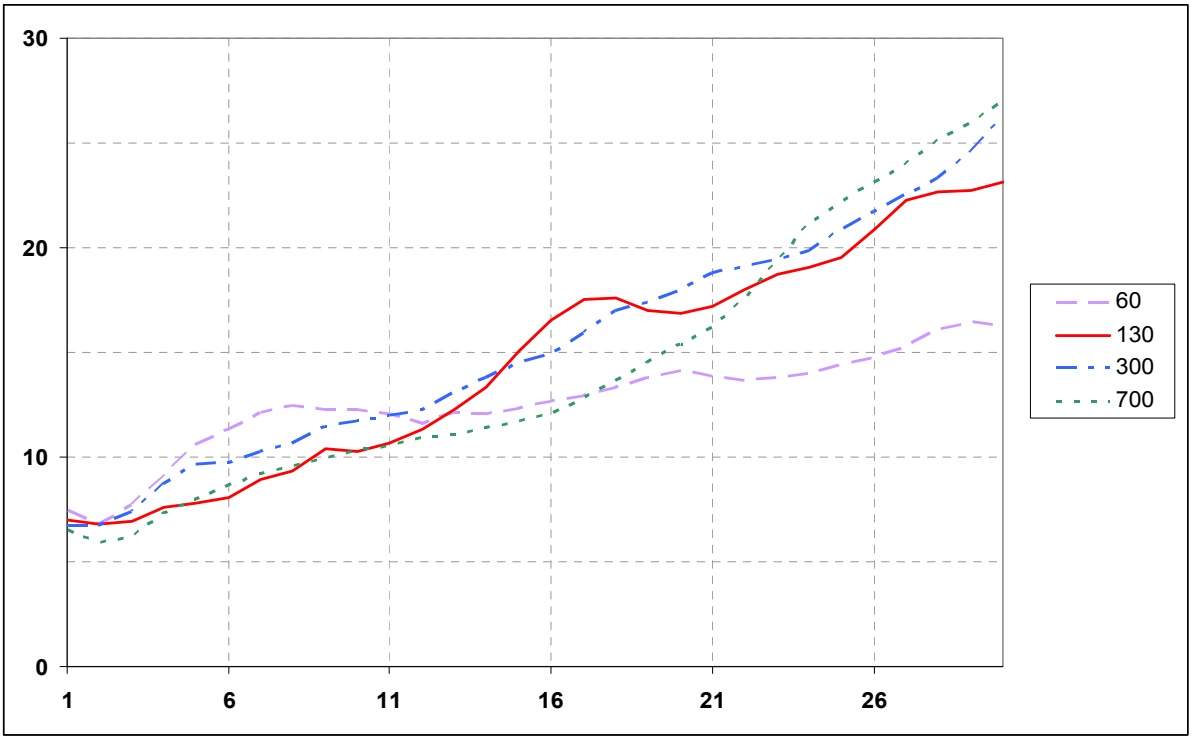
Obr. 5.15 Populace 300 jedinců, 160 křížení v cyklu, mutace nastaveny na 10x a proporcionálně variabilitě.



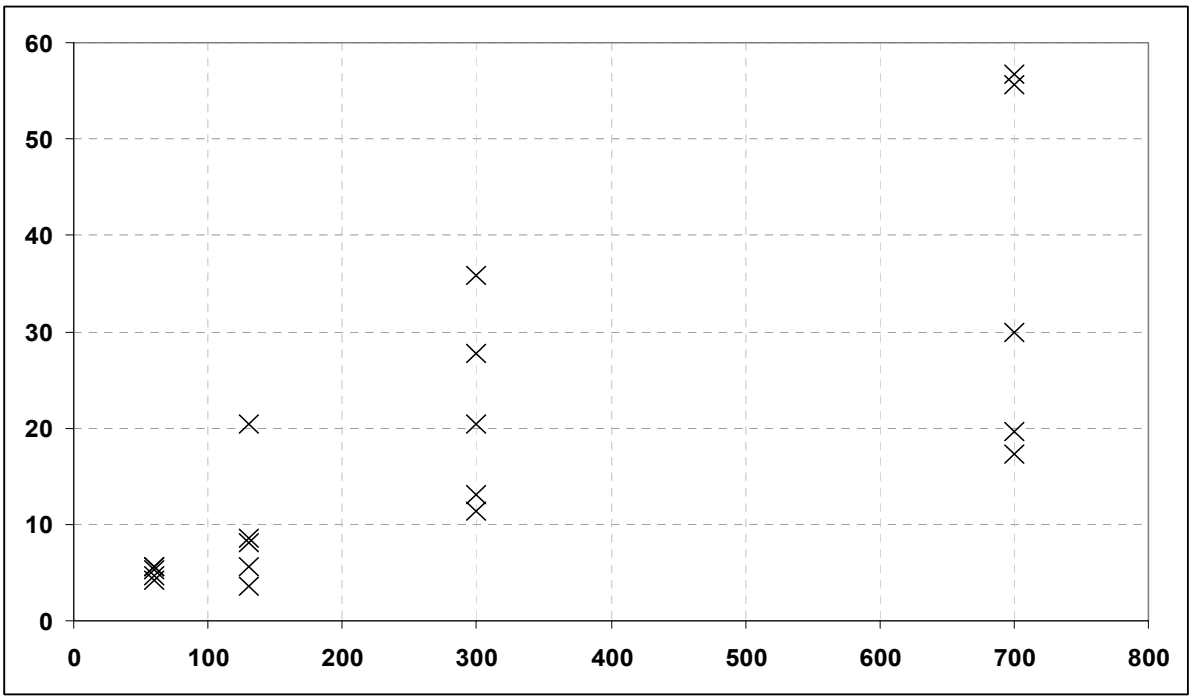
Obr. 5.16 Populace 700 jedinců, 400 křížení v cyklu, mutace nastaveny na 100x a proporcionálně variabilitě. Maximální velikost populace, kterou lze v použité verzi programu nastavit, je 1000, s velikostí bufferu pro ukládání mezivýpočtů mezi generacemi 2010. Pokud by ovšem například zvýšená intenzita mutací vyvolala generování více jedinců (může to být až 3,3násobek přednastavené hodnoty), program by je odmítl ukládat, což by mohlo mít vliv na vývoj populace. Rovněž aktivní variace konstant (viz základní nastavení) generuje další jedince za konec populace, než dojde k jejich vyhodnocení.



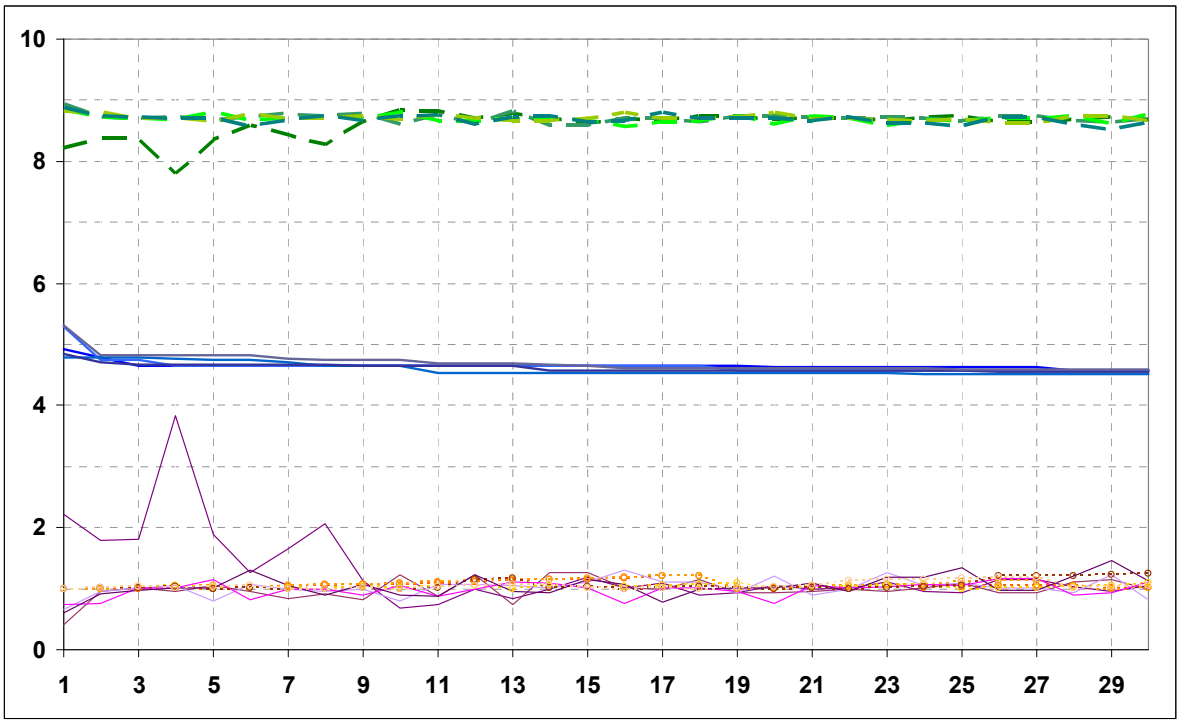
Obr. 5.17 Podíl nových (zeleně), mutovaných (žlutě) a prvků vzniklých křížením (šrafovaně fialově). Průměry přes pět běhů evoluce z předchozích grafů. Shora 60, 130, 300 a 700 jedinců. Na grafech se nejvíce projevuje koeficient mutací (relativně k velikosti populace), který je postupně 3, 5, 10 a 100. Počet provedených mutací je úměrný tomuto nastavenému číslu a převrácené hodnotě koeficientu variability populace. 100 byla asi neúměrná hodnota.



Obr. 5.18 Průměrná délka genomu (interního zhuštěného zápisu jedince) přes všech pět pokusů. Generováním nové populace jsou vytvářeni jedinci s průměrnou délkou šest až osm bytů (vliv na tuto základní délku by měl poměr četností funkcí s jedním a dvěma parametry a terminálních symbolů v okně dle obr. 5.1). Délka je penalizována (i když s ohledem na hodnoty účelové funkce je nastaven velmi malý koeficient, 5), takže může dojít i k poklesu průměrné délky genomu. Celkový růstový trend je ale zřejmý.

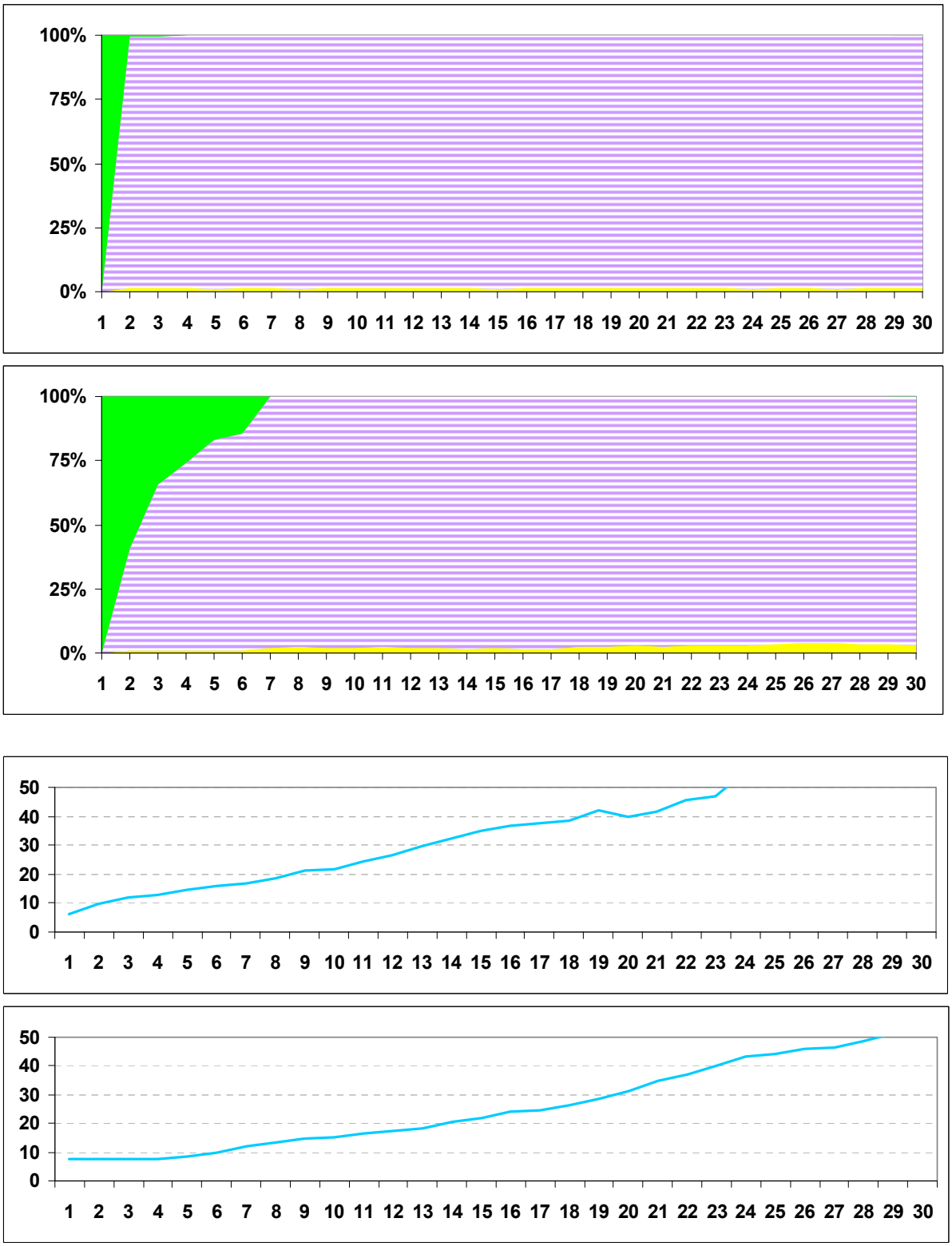


Obr. 5.19 Porovnání doby výpočtu prvních 30 generací. Vodorovně velikost populace, svisle doba výpočtu v minutách (Pascal používá interně jednotku času jeden den, hodnota je uložena jako číslo v plovoucí čárce). Je patrná očekávaná závislost doby výpočtu na velikosti populace. Konkrétní doby výpočtu pak závisí především na množství koeficientů, dopočítávaných metodou EGD v každém cyklu. S rostoucí délkou genomu zpravidla roste i počet koeficientů (pokud není dosaženo omezení uvnitř programu, zde deset). Pravděpodobnost označení položky genomu jako umístění konstanty je přednastavena na 0,3 , bylo by možné ji přenastavit.

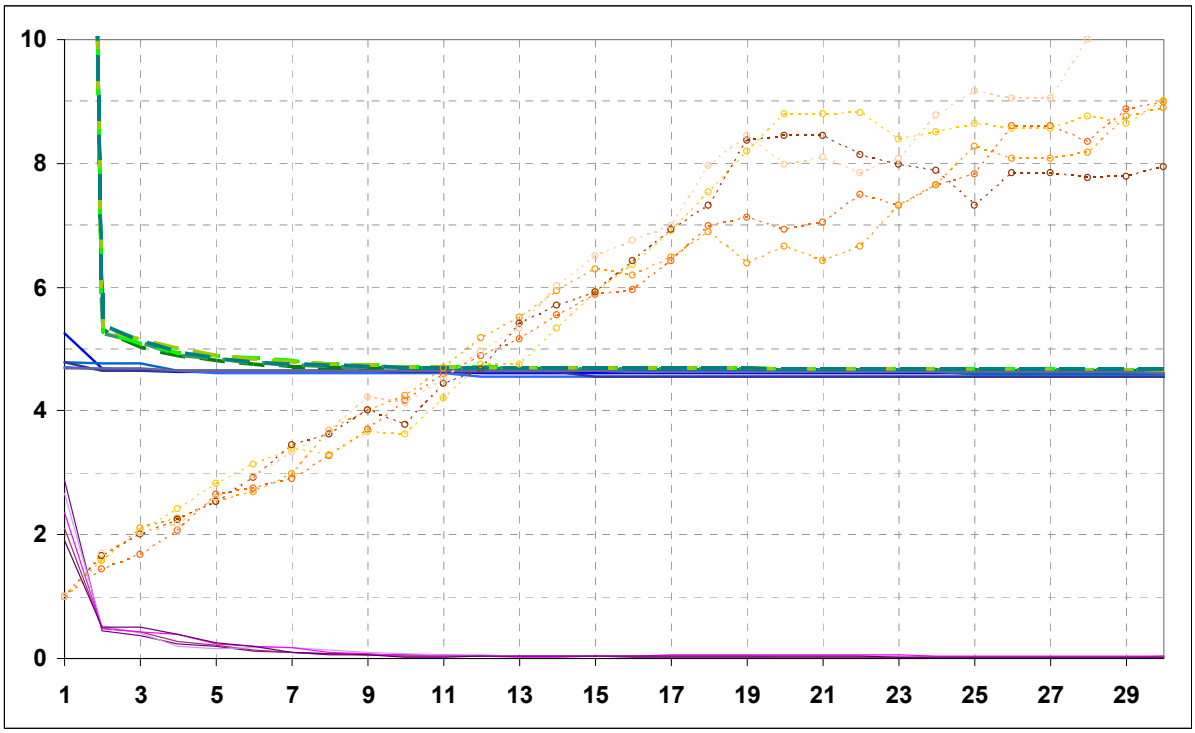


Obr. 5.20 Jednogenerační model. Podle této metodiky se celá předchozí generace (až na elitismus, tedy nejlepšího jedince) smaže a použijí se jen potomci (i vzniklí mutováním). Nutno podotknout, že v tomto programu se toto chování nahrazuje požadovaným věkem, pokud se nepodaří vygenerovat dostatečný počet nových jedinců, pak mohou starší přežít. Při populaci 130 bylo zvoleno 70 křížení +pětinásobná úroveň mutací (proporcionálně s převrácenou variabilitou), takže nových by měl být dostatek. Průměrná účelová funkce o čtyři řády větší, než v základním testu (obr. 5.13).

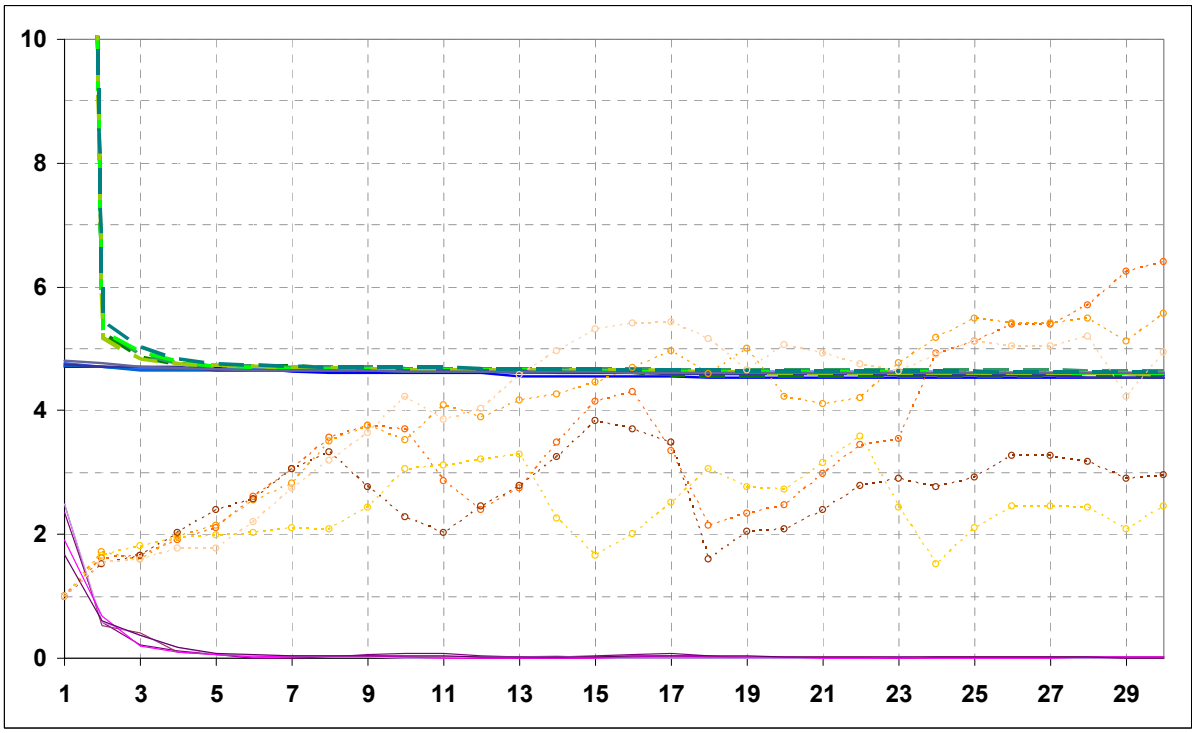
Obr. 5.21 Maximální životnost jedince v populaci je zvolena na pět, starší jsou z populace odstraňováni. Je vidět důsledek na chování populace až od šestého kroku. Jednokrokové zpoždění je dáno definicí doby života jedince v populaci, kdy se nejprve provedou statistiky, provedou se operace křížení a mutací a pak se starší jedinci mažou. Elitní jedinec je zachováván.



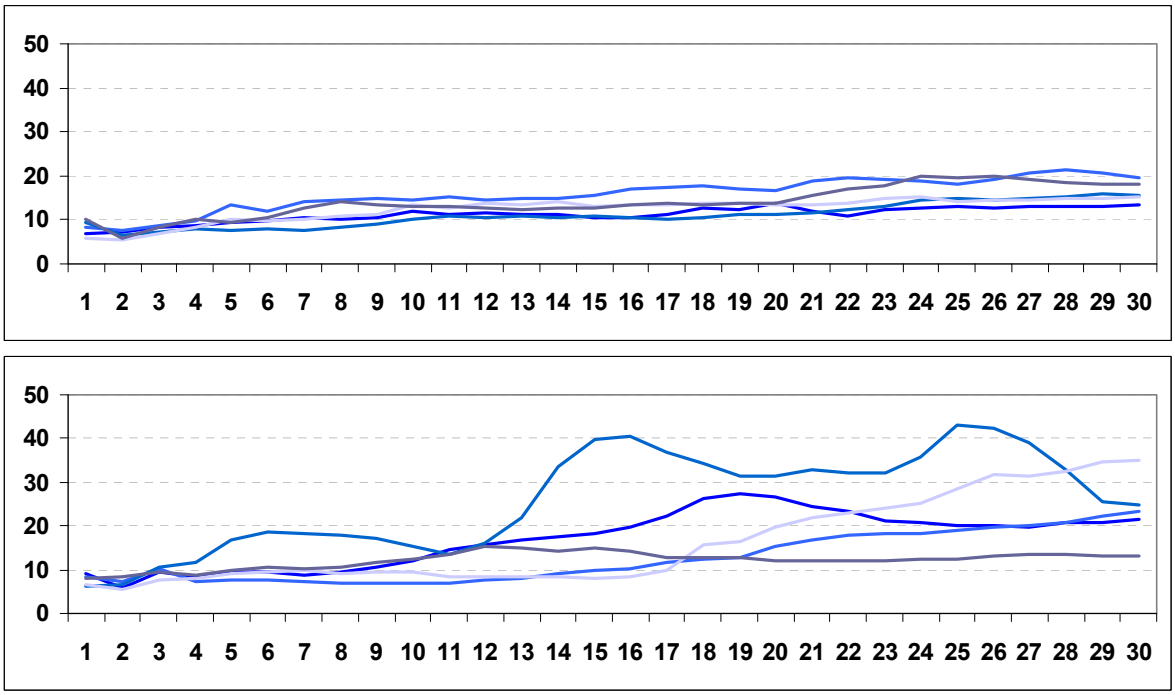
Obr. 5.22 Omezení životnosti na jednu nebo pět generací má předpokládaný vliv na složení populace. Nízká úroveň mutovaných (žlutě) je dána vysokou diverzitou populace (fialové tenké čáry na obrázcích 5.20 a 5.21). Zajímavý je vliv na průměrnou délku genomu, která má tendenci růst mnohem rychleji, než v případě, že stará osvědčená řešení v populaci ponecháváme (v porovnání s grafem na obr. 5.18, kde po 30 generacích nepřesáhne 28). Nepříliš viditelná zelená čára v horním grafu, reprezentující nově generovaný prvek počáteční sady přežívající až do 4. generace, reprezentuje elitního jedince.



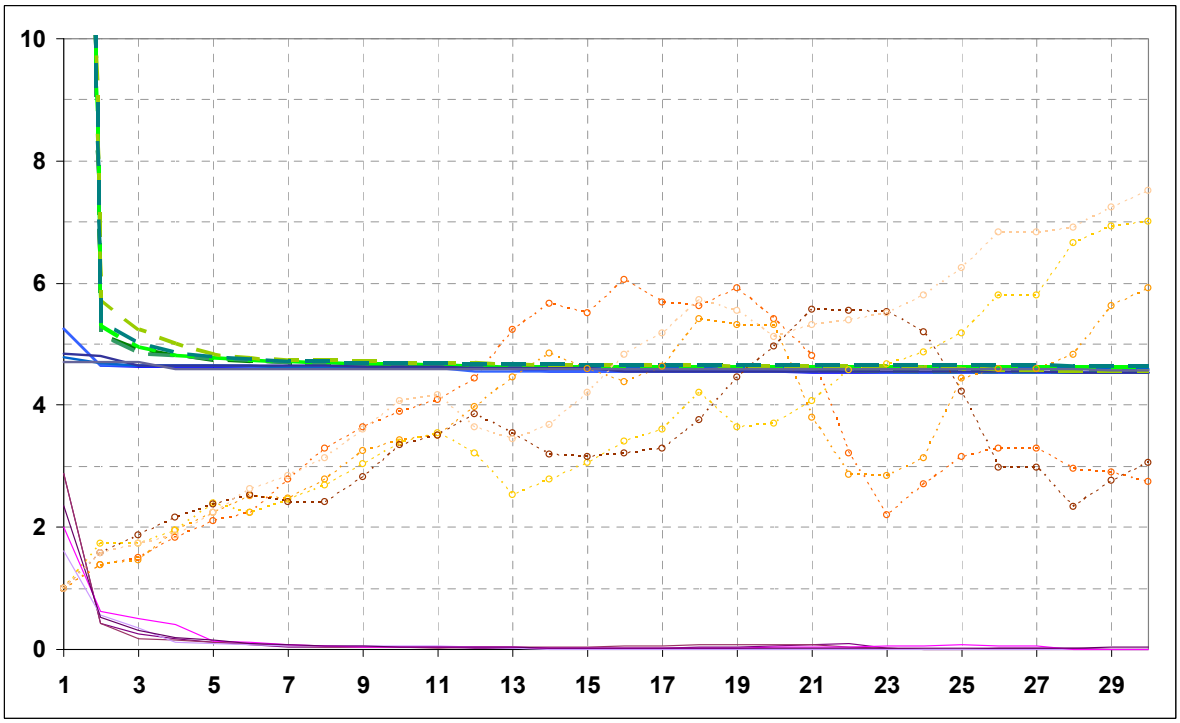
Obr. 5.23 Vliv koeficientu evolučního (v některé literatuře selekčního) tlaku (*q*) na chování populace, zde 1,01 místo základních 1,03.



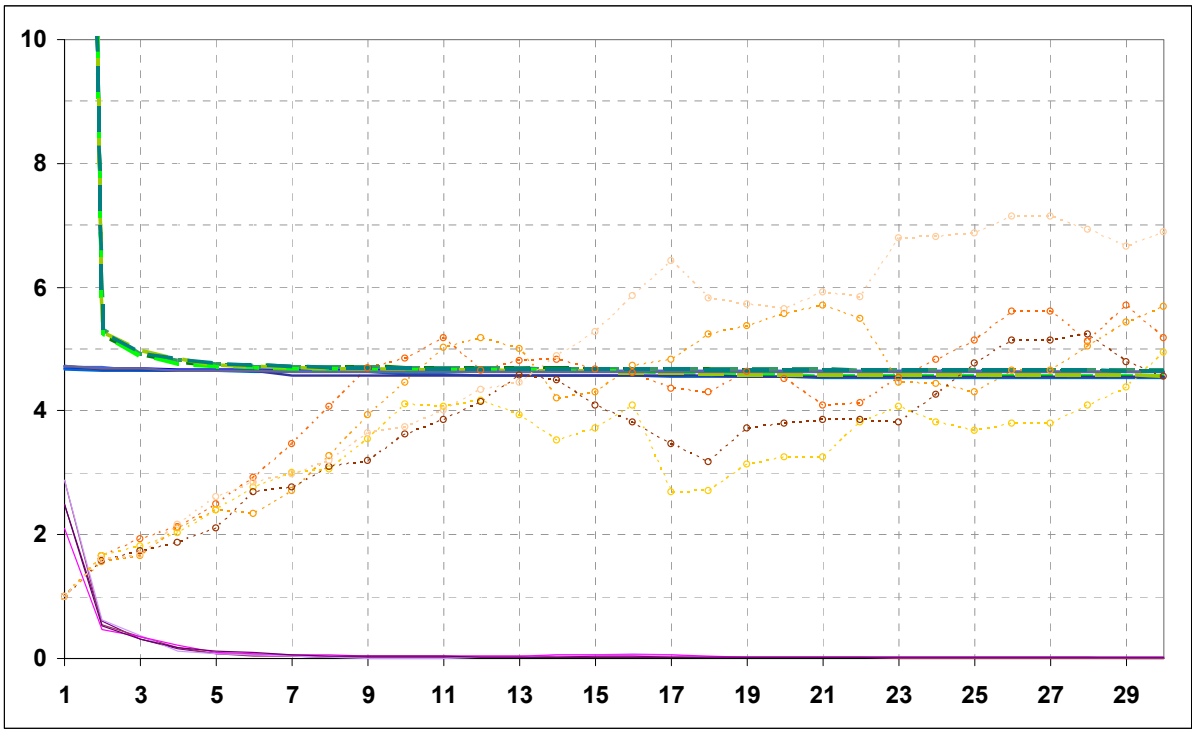
Obr. 5.24 Evoluční tlak *q* = 1,07. Evoluční tlak se projevuje překvapivě na průměrném stáří jedince v populaci (oranžová tečkovaná se značkami). Nepříznivým důsledkem je snížení diverzity populace, který není schopna zvrátit ani nastavená úroveň mutací (základní nastavení je 5 a proměnlivé s variabilitou; při nízké variabilitě populace max. 5x větší, tedy 25 mutací za cyklus (generaci)).



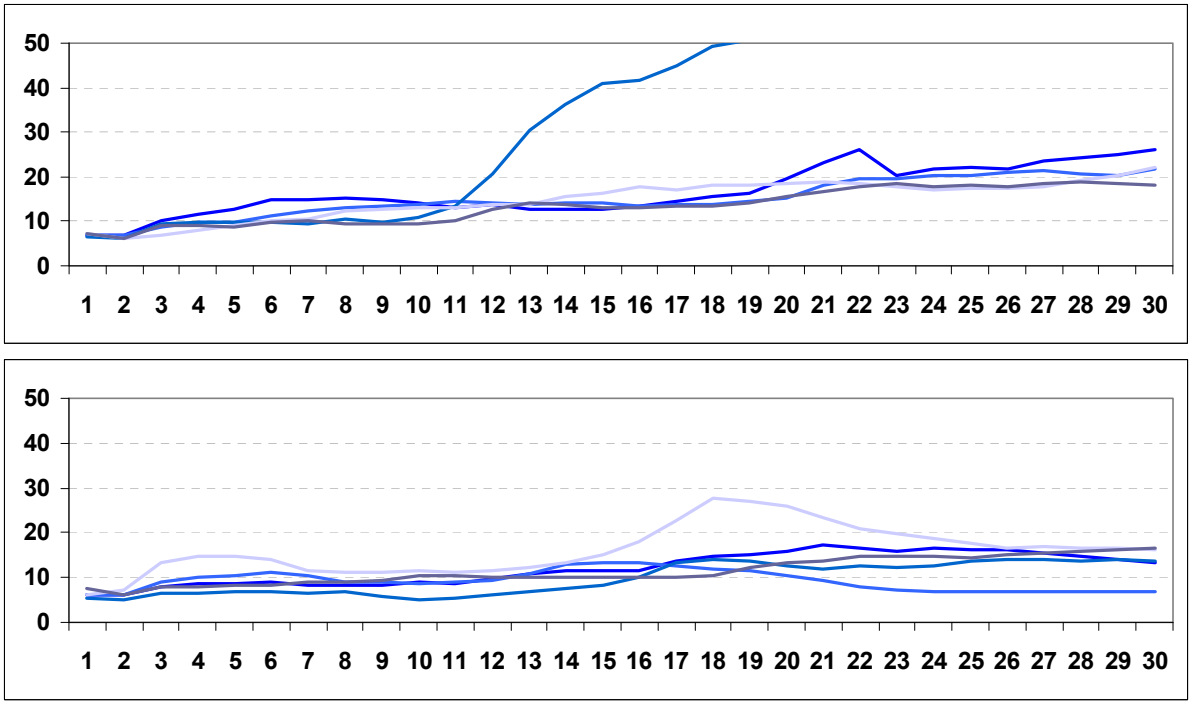
Obr. 5.25 Průměrná délka genomu při evolučním tlaku q = 1,01 (nahoře) a 1,07 (dole). Odpovídá teorii, že vyšší evoluční tlak snižuje diverzitu populace, což následně vede ke generování delších (méně efektivních) zápisů funkce. Na jednotlivých grafech je vidět, že ačkoli průměrná délka genomu v průběhu evoluce roste, může v jednotlivých případech i klesat.



Obr. 5.26 Hodnota penalizační funkce pro délku genomu nastavena na nulu. Proti obrázku 5.13 (základní nastavení) je snad poznamenáníhodný jen nižší průměrný věk jedince v populaci (programu ubylo jedno z omezení při vytváření nových jedinců)



Obr. 5.27 Graf pro vyšší hodnotu penalizační funkce (délka připočítávána s koeficientem 100) se téměř neliší.



Obr. 5.28 Průměrná délka genomu v populaci. Nahoře bez započtení do hodnoty účelové funkce, dole s penalizací stonásobkem. Průměrná hodnota účelové funkce je ovšem okolo 4.10^4, takže i tato penalizace je celkem malá. Je ovšem vidět, že v tomto případě má program tendenci vytvářet kratší jedince a průměrná délka genomu je udržována na přijatelných hodnotách (roste pomalu, nebo dokonce i klesá). Pro vyhledávání složitých funkcí, kdy proběhnou tisíce generací a je nastavena vysoká úroveň mutací, je toto chování vhodnější.

Vliv jednotlivých parametrů na výsledné hodnoty účelové funkce F_j (5.5) je shrnut v následující tabulce. Zeleně jsou vyznačeny hodnoty lepší, než celkový průměr všech zahrnutých pokusů (přibližně 4,567), červeně kurzívou horší. Již na první pohled je zřejmé, že dosažené výsledky se prakticky neliší, dokonce vždy je poměr 2:3, nebo 3:2, s výjimkou nastavení vysoké úrovně mutací. Nastavení 25x a proporcionálně až pětinásobek dává až 125 nově generovaných jedinců za cyklus, což se při velikosti populace již může projevit (výběr z většího počtu jedinců je ekvivalentem větší základní populace, a přirozeně také zpomaluje výpočet). Naopak nejvyšší hodnota u řádku „parsimony 100“ může být způsobena tím, že zde je k hodnotě účelové funkce připočítáván stonásobek délky zápisu funkce v paměti, oproti běžnému pětinásobku. Je třeba zdůraznit, že tabulka obsahuje hodnoty, vynášené do grafů, tedy dekadické logaritmy účelové funkce.

Tabulka 5.1 Dekadické logaritmy dosažených hodnot účelové funkce pro různá nastavení parametrů

nastavení	pokus č. →	1	2	3	4	5	$\overline{F}$
základní		4,554	4,532	*4,588*	*4,580*	4,561	**4,563**
lifetime omezena na 1		*4,570*	*4,571*	4,517	4,556	*4,599*	**4,563**
lifetime omezena na 5		4,548	*4,606*	*4,582*	4,564	*4,577*	***4,575***
evoluční tlak 1,01		*4,602*	4,546	*4,595*	4,547	*4,619*	***4,582***
evoluční tlak 1,07		4,524	*4,618*	*4,582*	4,534	*4,611*	***4,574***
parsimomy 0[3]		4,524	4,527	*4,583*	4,533	*4,578*	**4,549**
parsimomy 100		*4,621*	*4,633*	4,541	4,550	*4,627*	***4,594***
úroveň mutací: 0		*4,567*	4,527	*4,618*	4,518	*4,611*	***4,568***
úroveň mutací: 25, proporcionální		4,551	4,515	4,559	*4,581*	4,522	**4,546**
úroveň mutací: 5, vždy		4,548	*4,607*	4,528	*4,573*	4,529	**4,557**
úroveň mutací: 25, vždy		4,536	4,531	*4,609*	4,554	*4,604*	**4,567**

Výsledky testování chování pro různá nastavení lze shrnout tak, že algoritmus genetického programování se choval v zásadě dle předpokladů. Populace velikosti 130 se ukázala jako dostatečná, za předpokladu většího počtu mutací. Princip zachovávání jen nejnovější generace a mazání celé předchozí se ukázal jako zbytečný a spíše na závadu. Proti zamrzání populace je program možné nastavit tak, že mutace může provádět přímo na současné populaci (původní prvek není zachován) a tím, že heuristický algoritmus maže jedince, kteří mají stejný genom (bez ohledu na nastavení hodnot konstant u jedinců, resp. smaže toho, kdo má horší hodnotu účelové funkce). Po provedení testů se ukázalo, že vnitřní nastavení programu, kdy maximální počet mutací v proporcionálním módu (počet mutací nepřímo úměrný variačnímu koeficientu první poloviny populace) je shora omezen na pětinásobek přednastavené hodnoty, nemusí být pro větší počet generací dostačující a omezení by se mělo posunout například na desetinásobek. Abychom získali rozumné funkce, je třeba hlídat délku jedince (tento přístup se označuje jako „parsimony“, přeloženo do češtiny úspornost nebo skrblictví).

[3] bez penalizace délky genomu. Délka genomu se vynásobí tímto číslem a přičte k účelové funkci

6. Hodnoty konstant v genetickém programování

6.1 Význam koeficientů

V ukázkovém programu genetického programování v kapitole 4.3 nebyly parametry (koeficienty, konstanty) potřeba. Například pro hledání typické školní úlohy, goniometrické ekvivalence

$$\sin 2\alpha = 2 \sin \alpha \cos \alpha \tag{6.1}$$

(http://math2.org/math/trig/identities.htm) se obejdeme bez konstant, resp. číslice 2 bývá přímo ve skupině terminálních symbolů (resp. [Koza] uvádí, že to není nutné, protože např. číslo „1“ může program nahradit výpočtem „x/x“ a číslo nula „x–x“, ale přítomnost základních číselných hodnot zjednoduší výsledný generovaný strom funkce; jsou-li přítomny i další číselné konstanty, je tomu tak zpravidla se snižující se pravděpodobností použití během generování počáteční populace, resp. nově doplňovaných jedinců v rámci mutací).

V reálném světě se ale číselné hodnoty mění, přinejmenším s tím, jaké použijeme jednotky (času, rychlosti, ...). Proto je třeba do hledaného vztahu doplnit vhodné parametry (konstanty).

Konstanty (parametry) jsou většinou reprezentovány speciálními symboly jako listy stromu, kterým je zapsán jedince, viz např. [BT4] a [BT5]. Já jsem navrhl jiný způsob zápisu, vycházející právě z představy, že fyzikální hodnota je součin čísla a jednotky.

$$X = \{X\}[X] \tag{6.2}$$

Proto jsem parametry zvolil jako koeficient, násobící výsledek vyhodnocení hodnoty pro daný symbol. Vzhledem k potenciální náročnosti výpočtů jsem ale počet možných koeficientů omezil na deset. Pro každou hledanou funkci jsou koeficienty samostatně uloženy a při vyčíslování hodnoty účelové (kriteriární, *fitness*) funkce se vybírají v označených místech genomu postupně. Místa jsou pro jednoduchost označena nastavením jinak nevyužívaného nejvyššího bitu (celý zápis genomu je v ASCII a je tedy sedmibitový). Funkce přípustnosti kontroluje, aby značek nebylo více jak deset, a v případě překročení počtu některou (některé) odstraní (vybírá je náhodně, aby zásah do heuristiky tolik neomezoval robustnost algoritmu genetického programování).

Parametry se na začátku naplní jedničkami a lze je v tomto stavu držet vypnutím funkce hledání hodnot parametrů. V tomto stavu je program velmi rychlý, ale je využitelný spíše pro demonstrační účely, například pro proložení dat, vygenerovaných v Excelu (například při výuce jako ukázka genetického programování).

Pro dohledání koeficientů je možné vybrat metodu přímého prohledávání, nebo gradientní metodu.

6.2 Hledání koeficientů přímým prohledáváním

Tato metoda byla doplněna dodatečně, protože jsem měl problémy s laděním gradientní metody v důsledku špatně přepsané funkce do programovacího jazyka. Výhodou je, že tato metoda je velmi spolehlivá (robustní) a jednoduchá (snadno naprogramovatelná), takže také spolehlivá z hlediska běhu programu.

Přímé prohledávání (direct search) pomocí variací konstant je popsáno například v [Martin, str. 253], kde ovšem autor předpokládá postupné hledání optima, kdy je pokaždé vybrána jedna z proměnných; po nalezení extrému funkce se pokračuje další z proměnných;

po dokončení cyklu se vrací znovu k první proměnné a cyklus se opakuje, dokud u některé proměnné je její variací dosahováno zlepšení extrému lepší než ε. Například [Benke] oproti tomu předpokládá stochastické generování pomocného bodu, kterým je definován krok ve všech proměnných najednou.

V modifikaci, použité v tomto programu, se tato metoda pokouší postupně zvětšit každý z parametrů o 50% a pokud je hodnota účelové funkce lepší, použije novou hodnotu. Pokud ne, ještě zkusí stejným poměrem zmenšit. Projde všechny parametry (těch je maximálně 10) a pak začne znovu od začátku, celkem 10x. Program neřeší, zda se tím dosáhne minima. K výpočtu se vrátí v dalším kole genetické evoluce, kdy v předchozím cyklu dosažené hodnoty se použijí jako výchozí.

Hodnotu, o kterou se bude pokoušet zvětšit/zmenšit koeficient, lze nastavit (protože se jedná o násobení koeficientu, nastavujeme 1,5 a ne 50% - rozsah je od 1 (vypnuto) po 10). Pokud je v programu zaškrtnuto „simulované žíhání“, mění se koeficient v rámci geometrické řady, kdy přednastavený se použije jako třetí (termín simulované žíhání je v programu použit pro označení proměnlivého kroku ve stavovém prostoru vektoru konstant, od nepřiměřeně velkých hodnot, po velmi malé, umožňující přesné dohledání řešení).

6.3 Gradientní metoda

Gradientní metoda vychází z určení hodnot gradientu v daném bodě (použije se numerické přibližné určení diferenciálu):

$$\nabla f = \frac{\partial f}{\partial x_1} e_1 + \cdots + \frac{\partial f}{\partial x_n} e_n \tag{6.3}$$

Pro numerické vyčíslení nahradíme parciální derivace diferenciály, vypočtenými tak, že jeden každý parametr zkusíme zvětšit o hodnotu dx_i. Hodnoty pro všech deset složek vektoru koeficientů shromáždíme do pole. Protože nás nezajímá velikost, ale jen směr gradientu, je následně spočítána jeho euklidovská míra a pokud je nenulová, je jí každá složka podělena. Výsledná čísla se použijí v algoritmu gradientní metody jako bezrozměrné koeficienty.

Pokud si parametry (konstanty) představíme jako vektor parametrů, můžeme jejich novou hodnotu podle obecné gradientní metody vypočíst podle

$$x_{n+1} = x_n - \gamma_n \nabla F(x_n), \;\; n \geq 0 \tag{6.4}$$

, kde gama je koeficient učení. Jeho typická hodnota bývá 0,3. Záporné znaménko značí pohyb proti směru gradientu (pokud stejně jako v našem případě hledáme minimum funkce).

Gradientní metoda je velice robustní a jistě by našla správné hodnoty koeficientů. Naší hledané fyzikální představě ale neodpovídá přičítání hodnot polohy funkce, ale násobení. Operaci typu sčítání (ve vzorečku je odčítání) převedeme na násobení tím, že celý vztah odlogaritmujeme. To povede na vztah typu:

$$k_{n+1} = k_n / (\gamma_n^* \nabla F^*(x_n, k_n)) \tag{6.5}$$

kde ovšem „k“ (vektor koeficientů) je definované v prostoru exponenciální funkce. Výhodou je, že změna je nyní relativní k velikosti hodnoty dané konstanty.

Metoda je popsána jako Exponencionovaný Gradient Descent v [EGD, str. 20], kde se upravuje vektor vah podle pravidla:

$$k_{t+1}^{(i)} = k_t^{(i)} \cdot e^{-\nabla_\gamma (f(x,y,\gamma_t))\gamma_t^{(i)}} \tag{6.6}$$

Formálně je tedy v literatuře podroben exponenciální transformaci i samotný gradient ∇F. Robustnost metody ale zajišťuje, že i když tak neučiníme, při zachování spojitosti a znaménka gradientu v každé ose metoda stále funguje. Za první přiblížení lze považovat i to, že zatímco u obecné gradientní metody krok přičítáme, při výše popsané metodice násobíme konstantou (1+krok). Tím vlastně provedeme linearizaci náhradou exponenciály rovnicí přímky v daném bodě. V současné verzi programu ale respektuji výše uvedený vzorec. Vyčíslení exponenciály program příliš nezdržuje, hlavní časovou zátěž představuje výpočet gradientu.

Výpočet gradientu probíhá numericky, kdy zvolený krok je vždy polovinou kroku, který odpovídá předpokládanému koeficientu, kterým se následně konstanty násobí úměrně směru gradientu:

$$\beta = e^{\gamma_t} \tag{6.7}$$

Koeficient γ je přednastaven na 0,3 , lze jej změnit. V tomto případě by ale přesnost byla omezena konstantním krokem metody. Pro zpřesnění je vhodnější hodnotu koeficientu postupně zmenšovat, jak je popsáno v následujícím odstavci.

6.4 Simulované žíhání

Simulované žíhání je metoda vyhledávání parametrů, vycházející z gradientních metod [Kirkpatrick]. Podobně jako u *evolučních strategií* nebo *genetických algoritmů* (použitých pro vyhledání vektoru, tedy n-tice čísel, které nejlépe splňují kriteriární nebo účelovou funkci, zkoušíme postupně měnit hodnotu jednotlivých parametrů (složek n-tice). Simulované žíhání je inspirováno rekrystalizací kovu při zahřátí na dostatečnou teplotu a postupné ochlazování, kdy atomy při vyšších teplotách mohou měnit svou polohu v krystalové mřížce o větší vzdálenost a připojit se tak k jinému jádru krystalu. Z hlediska skutečných podmínek při žíhání je třeba dodat, že ke zvětšování krystalů dochází proto, že jednotlivé atomy kovu snáze opouštějí okrajové polohy, zejména hrany dílčích krystalů, zatímco vnitřní atomy svůj krystal opustit prakticky nemohou. Tento reálný průběh žíhání není u shodně pojmenované výpočetní metody uplatněn.

Klesající „teplota“ (ve vzorci 6.8 označeno *T*) je uplatněna dvojím způsobem. Jednak se s takto označeným klesajícím parametrem postupně snižuje hodnota, o kterou se mění jednotlivé složky vektoru *x* (zpravidla pro jednu teplotu zkusí změnit každou složku, při stochastickém přístupu je možný menší krok teploty, nebo počet změn úměrný počtu složek vektoru, zpravidla o něco větší), jednak se mění pravděpodobnost přijmutí změny, i když je nevýhodná (zhoršuje výsledek). Respektive, pokud nová kombinace položek vektoru dává lepší hodnotu kriteriární (účelové) funkce (v následujícím vzorci $f(x)$), přijme se navržená změna vždy; pokud tomu tak není, přijme se s pravděpodobností (Metropolisovo kritérium, [EVT, str. 167]):

$$P(x \to x_0) = e^{-(f(x)-f(x_0))/T} \tag{6.8}$$

Ve vzorci x je nový stav a x_0 je původní stav, popisuje tedy přechod $x_0 \rightarrow x$ a jeho pravděpodobnost, šipka by tedy měla být obráceně (vzorec je ale přesně citován).

Největší inspirací pro ostatní metody umělé inteligence bylo zavedení klesajícího kroku iterace. Obecně krok klesá nejčastěji rovnoměrně, ale může klesat i nepravidelně (stochasticky), kdy v průběhu cyklů/iterací občas může i stoupnout. Metodika simulovaného žíhání tedy předpokládá, že při gradientní metodě nejprve začneme s velkým krokem a budeme jej zmenšovat. Obdobný postup se používá i pro neuronové sítě [Mills, str. 77].

Pro realizaci tohoto postupu pro vestavěnou gradientní metodu je v programu generována tabulka alternativních kroků. Tuto funkci lze samozřejmě vypnout i zapnout. Pokud je zapnutá, použije se zadaný krok jako základní, což je v tomto programu třetí. Ostatní se spočítají tak, že každý další je 60% předchozího:

$$s_1 = \frac{step}{0.6^2}$$
$$s_{i+1} = 0.6 s_i,\ i = 1..10 \tag{6.9}$$

Skutečné násobící koeficienty se získají odlogaritmováním

$$c_i = e^{s_i} \tag{6.10}$$

Jednorozměrné pole konstant c_i se vytvoří jednorázově po startu programu, přegeneruje se po změně nastavení, výpočet exponenciály tedy nezatěžuje běh programu. V hlavním cyklu se provede vždy přesně deset cyklů zpřesnění hodnot konstant; pokud to není dostačující, v dalším hlavním cyklu programu se to provede znovu (program umožní nastavit, zda nutně opět 10x, či dokonce zda se náhodně posunout v řadě kroků a zvolit tak jemnější zpřesnění; na začátku běhu programu a při změně poměru koeficientů metody EGD dle (6.10) je jich vypočteno vždy 30). Po ukončení (nebo přerušení obsluhou) se provede celkem 25 cyklů, což v kombinaci s metodikou simulovaného žíhání umožňuje poměrně přesné donastavení koeficientů pro každou z funkcí, generovaných metodou genetického programování. To je důvodem, proč poslední cyklus trvá déle, i proč program pomalu reaguje na přerušení běhu kliknutím myší/klávesou Esc.

6.5 Lokální minimum a přesnost

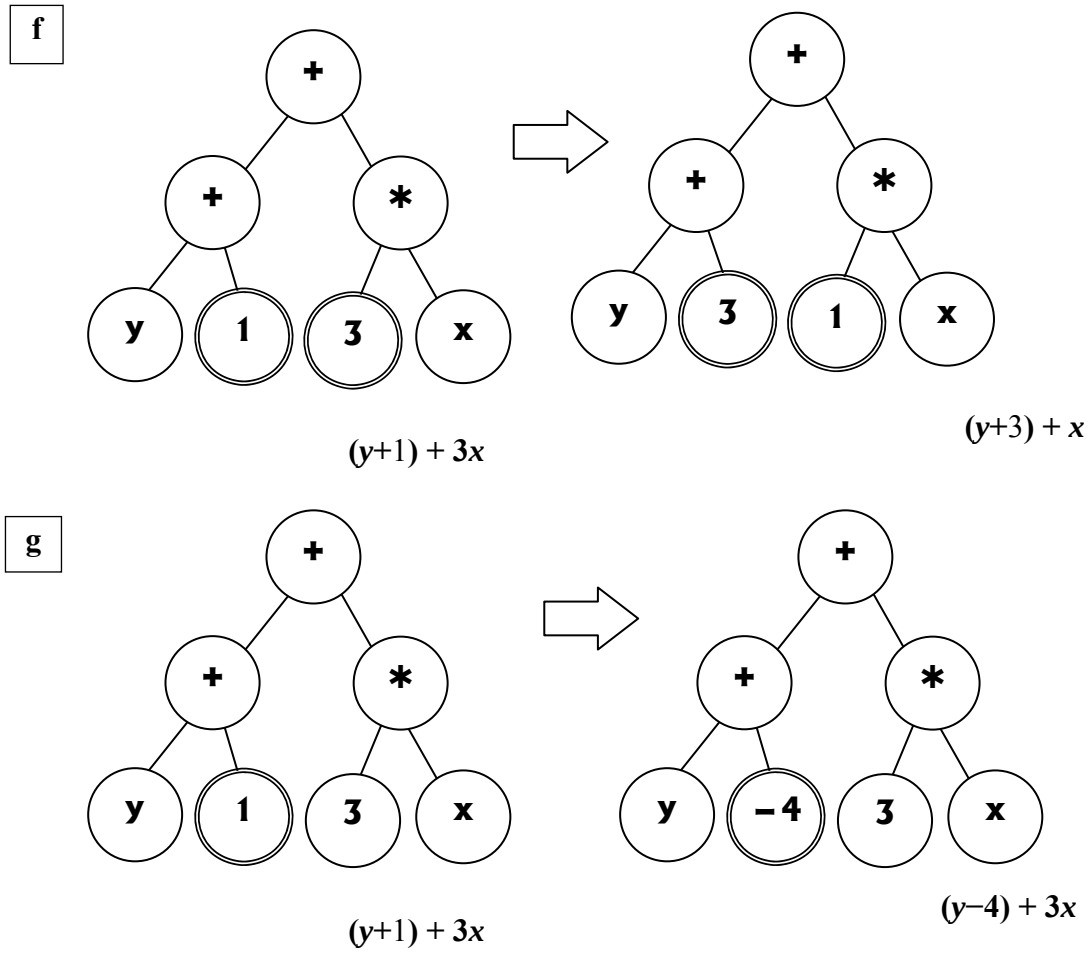
Problém úniku z lokálního minima řeší samotné genetické algoritmy tím, že funkci zkřížením zkopírují na nové místo, kde jsou počáteční koeficienty pravděpodobně stanoveny náhodně (přesněji, jsou zděděny z úplně jiné, dříve zrušené funkce). Pokud se pro nové řešení najdou lepší koeficienty, samozřejmě při provádění algoritmu ruletové selekce nahradí to staré.

Každé kolo genetického algoritmu pak probíhá tak, že nejprve se optimalizují koeficienty zvolenou metodou, pak se jedinci v populaci navrhovaných funkcí seřadí podle hodnoty v rámci určování koeficientů vypočtené účelové funkce.

Následně se provede jedna z heuristik zděděných z předchozího programu, kde, pokud jsou dvě funkce za sebou úplně stejné, pak se druhý z jedinců z populace odstraní. Tato situace může vzniknout náhodným křížením. Pokud zápis funkce je i jen drobně modifikovaný, např. (y+x) místo (x+y), tímto způsoben se neodstraní.

Následně se omezí počet jedinců (vygenerovaných funkcí) v populaci na její původně zadaný počet.

V následujícím kroku se provede zadaný počet křížení (potomci se přidají na konec populace).



Obr. 6.1 Variace konstant jako další možná mutace (f). Poslední možností je náhodná změna hodnoty.

Dále se provede zadaný počet mutací. Při mutaci v tomto programu je zmutovaný výsledek uložen na původní místo. Odpovídá to logice genetických algoritmů, ale je otázka, zda by nebylo vhodnější jej přidat na konec, mezi potomky křížení.

Pokud jsou mutace povoleny, neprovede se jejich zadaný počet, ale počet se přiměřeně zvýší/sníží podle toho, jaká je diverzita populace. Ta se určuje jako podíl směrodatné odchylky ku průměru účelové funkce, v té době již spočítané pro každého jedince (variační koeficient):

$$c_x = \frac{\sqrt{D(x)}}{E(x)} \tag{6.11}$$

Pokud tedy většina funkcí dává podobnou hodnotu účelové funkce, provede se mutací i mnohokrát více. Po testování byl výpočet variačního koeficientu omezen na první polovinu populace, která nejvíce trpí ztrátou diverzity. Ukázalo se, že variační koeficient při rozumné diverzitě je větší, než 0.1. Výsledný variační koeficient se tedy násobí deseti, a pokud je variabilní množství mutací povoleno, je generovaný počet mutací dělen touto výslednou hodnotou (pro malý variační koeficient je počet mutací zvýšen, pro velký snížen, ale v obou případech je úprava omezena na trojnásobek, resp. třetinu zadané hodnoty.

Z mutací je vyňat elitní jedinec (dosud nejlepší nalezené řešení).

Jak již bylo uvedeno, na závěr hlavního cyklu genetického programování se zpřesnění koeficientů danou metodou provede (až) 25x. Při použití metodiky snižování hodnot kroku po vzoru simulovaného žíhání tak dojde k poměrně přesnému vyčíslení konstant (výkonnost gradientní metody je až jeden bit mantisy na cyklus; dá se předpokládat, že k řádovému odhadu dojde již v průběhu prvního vyčíslení pro nově vygenerovaného nebo zmutovaného jedince). Dosahovaná přesnost je s ohledem na přesnost změřených dat většinou dostačující. Pokud není průběžné snižování hodnot koeficientů aktivní, výpočet se ukončí, pokud již nedochází k dalšímu zlepšování účelové funkce. Je třeba připomenout, že k zpřesnění koeficientů dojde u všech jedinců, kteří po skončení evoluce zůstávají v populaci. Do jisté míry je na výběru uživatele programu, kterou funkci (kterého výsledného jedince) si vybere, s možným zohledněním vhodného tvaru (matematického zápisu) výsledné funkce.

Pokud je oddělena kontrolní (testovací) část dat, pak je před zobrazením grafů (2D, 3D) zobrazeno okno s porovnáním chyby v trénovacích a testovacích bodech. Aby byly hodnoty porovnatelné, je hodnota účelové funkce pro dané body podělena jejich počtem a následně zobrazena v pomocném okně. Protože hodnota účelové funkce je dána součtem odchylek v jednotlivých bodech (rovnice 5.5, 5.6), jedná se o průměrnou chybu v dané části dat.

6.6 Variace konstant

Poslední možností pro únik z lokálního minima je variace konstant. Konstanty jsou v teorii genetického programování známy dlouho (jako koncové objekty stromu, listy), a před příchodem prací, které se je pokoušely určit přímo, jako je [BT4], [BT5], se považovala možnost změnit konstanty za jednu z mutací [GP, str. 110]. Návrat k tomuto přístupu ke konstantám v případě popisovaného programu nabízí možnost úniku z lokálního minima.

Postup je takový, že se vytvoří kopie vybraného jedince (za konec populace) a ta se podrobí náhodné změně několika konstant současně (ne všech). Důsledkem konstrukce programu ale je, že protože je vytvořen nový jedinec, okamžitě se na něj zavolá funkce vyčíslení účelové funkce, která začíná tím, že provede deset cyklů zpřesnění nově vygenerovaných konstant nastavenou metodou. Nový jedinec je tedy znevýhodněn před starší populací tím, že po prvních deseti cyklech ještě nebudou konstanty vyčísleny dostatečně přesně, ale pokud byl nový začátek blíže výhodnějšího lokálního minima účelové funkce, je velká pravděpodobnost, že při porovnání se svým vzorem uspěje lépe (a nahradí jej v populaci).

Nevýhoda tohoto postupu je v tom, že pokud obě počáteční nastavení konstant (vzor i „potomek“) vedou na stejné lokální optimum, zůstanou pravděpodobně v populaci oba, a dojde k snížení diverzity populace (bude vyřazen jiný, pravděpodobně odlišný jedinec). Variantou by bylo mutaci provést na původním jedinci, aby ke snížení diverzity nedošlo, ale to by vedlo k potenciální ztrátě lepšího původního jedince. Řešením tedy je, že po dokončení generování jedince a vyčíslení účelové funkce se k dané dvojici program vrátí a ponechá pouze lepšího jedince (buď zdroj před mutací, nebo nově vytvořeného). Obdobný postup výběru mezi zdrojem a novým jedincem při mutaci (v angličtině parent a offspring) je použit v programech doc. Brandejského [BT4].

Dalším problémem je, jak (o kolik) změnit konstanty. Nejedná se jen o velikost v logaritmickém slova smyslu, měnit se musí zkusit i znaménko. Jeden z možných postupů je udržovat si přehled o běžných hodnotách konstant v programu a nastavovat je v těchto mezích. Z opačného hlediska právě toto může být problém a má smysl hledat konstanty například o řád odlišné.

Poslední možností je dogenerovat konstanty do zápisu mutovaného jedince. Vzápětí korekční algoritmus omezí jejich počet na maximálně deset a dopočítá jejich hodnoty.

6.7 Porovnání s vyhledáváním hodnot genetickými algoritmy

Jedním z možností vyhledání vhodných hodnot konstant pro funkci vygenerovanou metodou genetického programování je i použití genetických algoritmů, nebo z nich odvozené metody označené jako *evoluční strategie* (nezaměňovat s evolučními strategiemi v biologii, jedná se o shodu názvů). Metoda je popsána například v [Hansen] (vysvětluje principy ES, doporučuje vhodné parametry populací), [Auger], dobré vysvětlení metody je v [Beyer].

Metodu zmiňuji proto, že je použita v [BT4], [BT5] a dalších, kde je její použití pro určení konstant pro funkci generovanou genetickým programováním označováno jako GPA – ES. Stejně jako v programu, popisovaném v této práci, i v [BT4] je před porovnáním účelových funkcí (fitness) jednotlivých jedinců v populaci provedeno vyčíslení konstant. Jeden z rozdílů je, že pro každého nového jedince je vyčíslení provedeno jednou – cykly se nestřídají, algoritmus ES probíhá dostatečně dlouho, aby se konstanty vyčíslily dostatečně přesně. Podle [BT12a] se provede vždy přesně 40 cyklů. Při větším množství vyčíslovaných konstant může být tento počet cyklů nedostatečný, což je forma penalizace příliš složitě zapsaných funkcí.

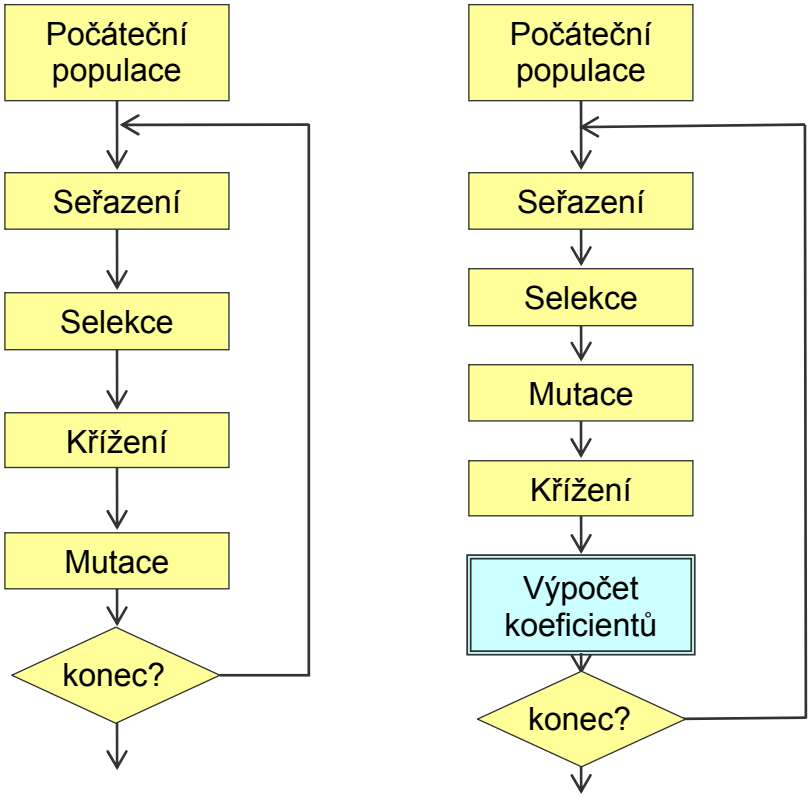
Podle [Beyer] lze metodu označit jako $(\mu/\rho + \lambda)$-ES, kdy $\mu=\rho$ a $\mu + \lambda =$(10 až 1000) [BT12a]. Postup vytváření potomků je takový, že z μ jedinců předchozí generace se vybere ρ rodičů, z těch se vypočte aritmetický průměr pro každou souřadnici a výsledný vektor konstant se použije jako zdrojový pro λ mutací. Výsledná populace v rozsahu $\mu + \lambda$ se seřadí (pro nově generované se spočítá hodnota účelové funkce) a vybere se μ nejlepších řešení jako nová populace. Provede se vždy čtyřicet cyklů. Výhodou volby $\mu=\rho$ je, že výpočet nového výchozího vektoru v cyklu proběhne pouze jednou. Genetickým algoritmům by bylo blíže použití pouze dvou rodičů, což je možná varianta algoritmu. Další možností je inteligentní křížení; u dvou vektorů si je lze představit jako hledání základu pro nové řešení nikoli jen v těžišti hodnot, ale kdekoli na přímce, spojující tyto hodnoty, a to i mimo úsečku spojující tato dvě řešení. Ekvivalentem v biologickém vzoru je křížení dvou jedinců s výraznou vlastností, například při křížení dvou neobvykle velkých psů je naděje, že kříženec bude ještě větší. Při více rodičích je možné obdobné řešení, ale je hůře geometricky představitelné (říkáme, že m jedinců svými hodnotami definuje nadrovinu v n-rozměrném prostoru, $m<n$, po které se pohybujeme nejen v části, definované těmito body, ale i v jejich blízkosti vně jimi vyhrazené oblasti).

Při teoretickém srovnání metod by výhodou GPA – ES měla být přesnost vyhodnocení konstant, kterou lze volit až do hodnot, daných výpočetními možnostmi použitého počítače [8086]. Tyto předpokládané rozdíly lze ovšem předpokládat právě proto, že program, popisovaný v této práci, nevyhodnocuje koeficienty přesně, částečně cíleně. Tím je umožněno vyřazování funkcí, které používají konstanty i v místech, kde to není třeba. Dalším důvodem je, že k zpřesnění hodnoty dojde v následujícím cyklu genetického programování, protože se vychází z hodnot, určených v předchozím cyklu. Oproti tomu v metodě GPA – ES proběhne vyhodnocení jen jednou.

Výhodou genetických algoritmů by měla být odolnost proti uváznutí v lokálním minimu (účelové funkce). Problém lokálního minima je ovšem dán zejména tvarem vygenerované funkce, kdy upřesnění koeficientů neprobíhá v prostoru zadaných dat, ale v prostoru chyb funkce, který je většinou spojitý a podobné problémy se zde nevyskytují. V krajním případě zajišťuje ochranu proti uváznutí v lokálním extrému hlavní (nadřazený) cyklus genetického programování, kdy je možné zařadit mutaci typu variace konstant a pokud to vede k lepším výsledkům, začít s hledáním hodnot koeficientů z jiného bodu, viz předchozí podkapitola. Gradientní metody se osvědčily i v mnohem obtížnějších úlohách, jako je vyhledávání vah v neuronových sítích [Gupta], kdy vzhledem k možnosti libovolného *očíslování* neuronů ve skryté vrstvě máme nutně minimálně $m!$ identických správných řešení [Abu-Mostafa] (m je počet neuronů ve skryté vrstvě, úloha je tedy z definice NP úplná), mezi kterými se hodnota

dosažené natrénované odchylky pohybuje ve větších hodnotách, často s výskytem dalších lokálních minim (*back propagation* je typický představitel gradientní metody).

K reálnému porovnání metod vyčíslování konstant ovšem nedošlo, metoda GPA – ES nebyla v době předkládání této práce v programu implementována. Její implementaci a porovnání připravuji pro prezentaci této práce pro Neural Network World [Hlaváč 2017].



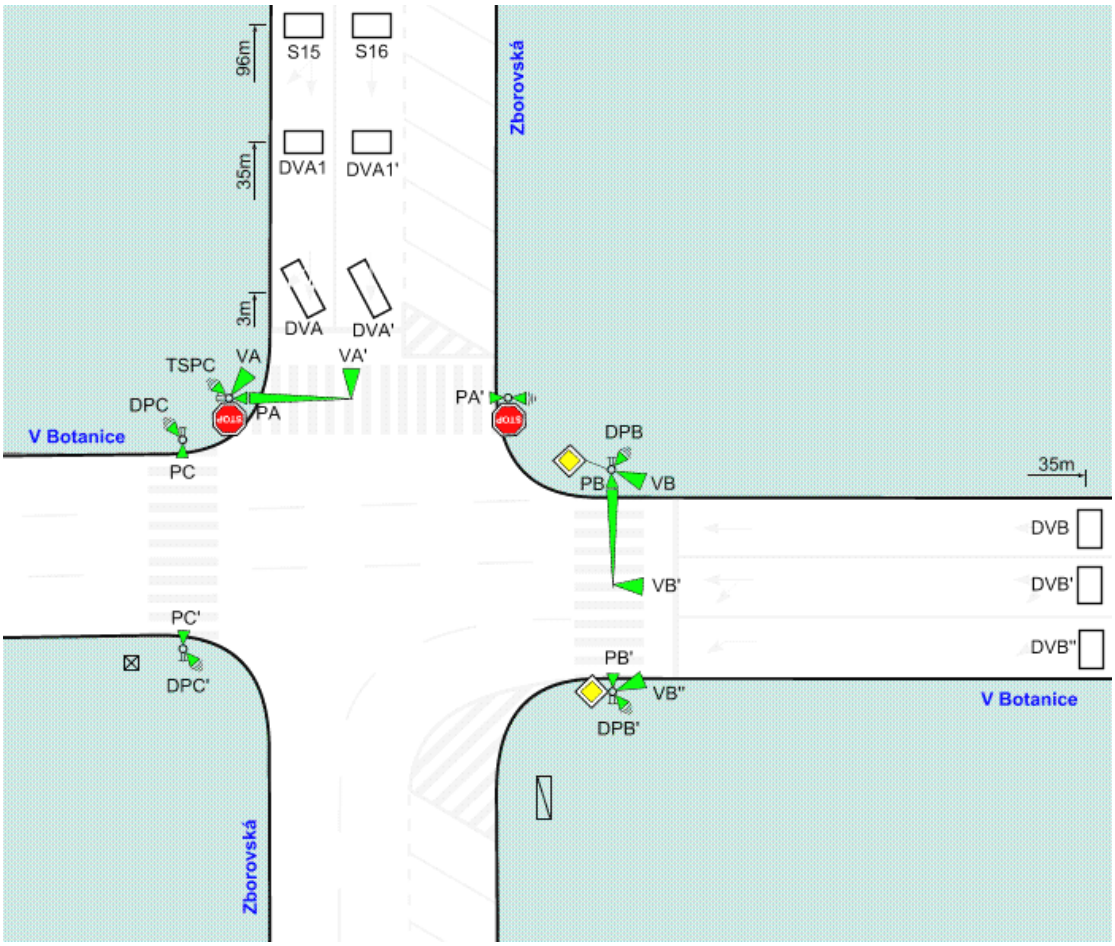
Obr. 6.2 Vývojový diagram běžného genetického algoritmu (vlevo) a rozšíření, použité v této práci. Mutace byla přesunuta před křížení po problémech s přenášením výsledků mutací do hlavní populace.

7. Ukázka použití programu

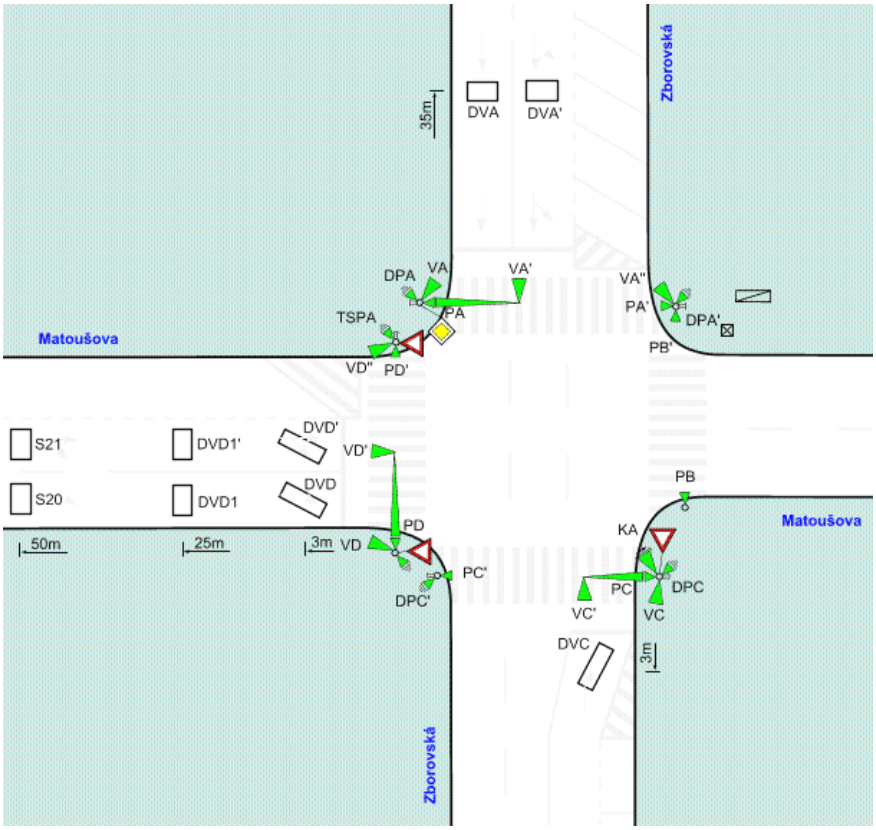
7.1 Data

Pro testování byla získána data z intenzit provozu na křižovatkách, tvořících průjezd Diezenhoferovými sady okolo budovy středočeského hejtmanství ze severu na jih, tedy z křižovatek V Botanice – Zborovská a Matoušova – Zborovská. Protože program je od začátku vytvářen pro hledání zápisu funkce o dvou proměnných, musela být vybrána vhodná kombinace dat (data a dále vložená schemata křižovatek poskytl doc. Ing. Tomáš Tichý, Ph.D., z Ústavu dopravní telematiky Fakulty dopravní).

Hodnoty intenzity a obsazenosti pro uvedené snímače jsou dostupná v excelovských souborech, agregovaná po 90 s nebo jedné hodině.

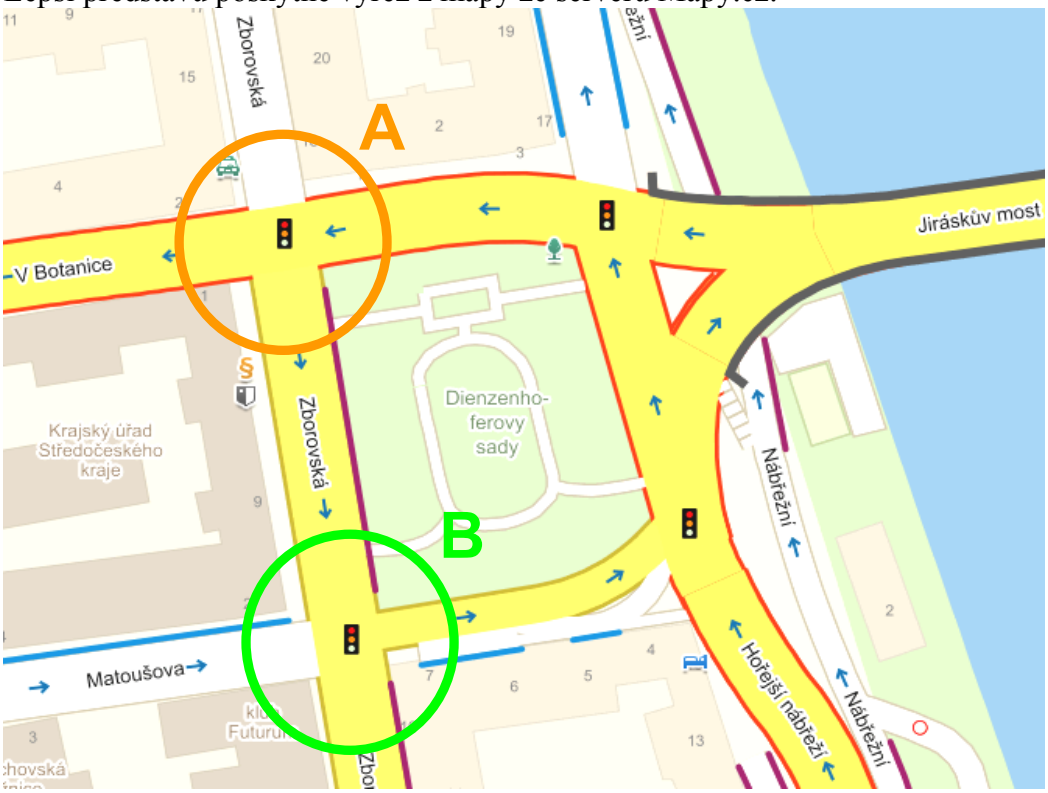


Obr. 7.1 Schéma umístění senzorů v křižovatce Zborovská - V Botanice



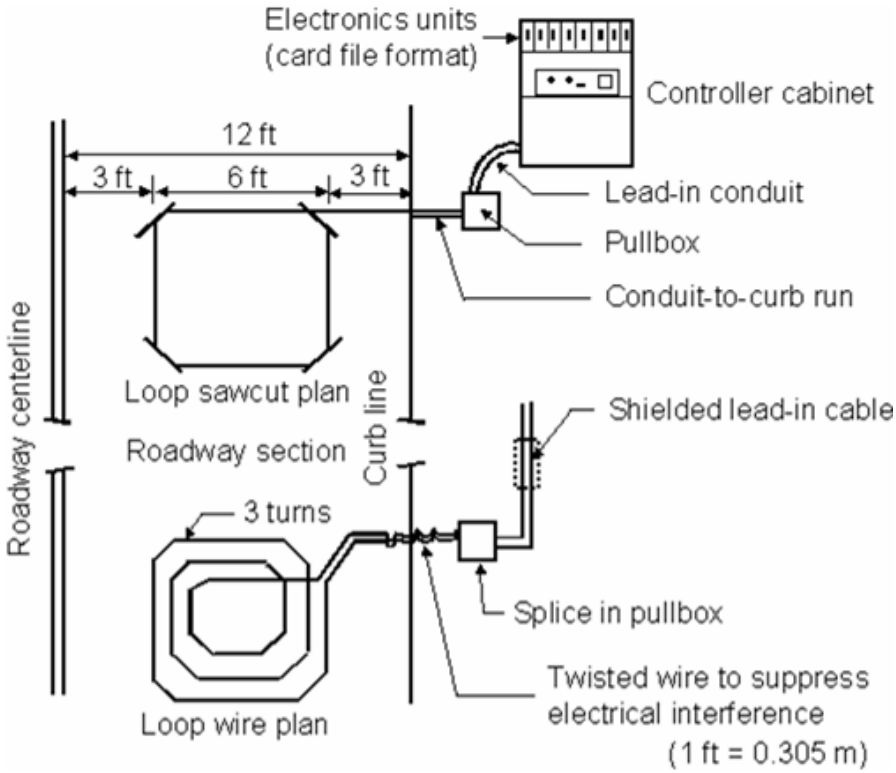
Obr. 7.2 Schéma umístění senzorů v křižovatce Zborovská - Matoušova

Lepší představu poskytne výřez z mapy ze serveru Mapy.cz:



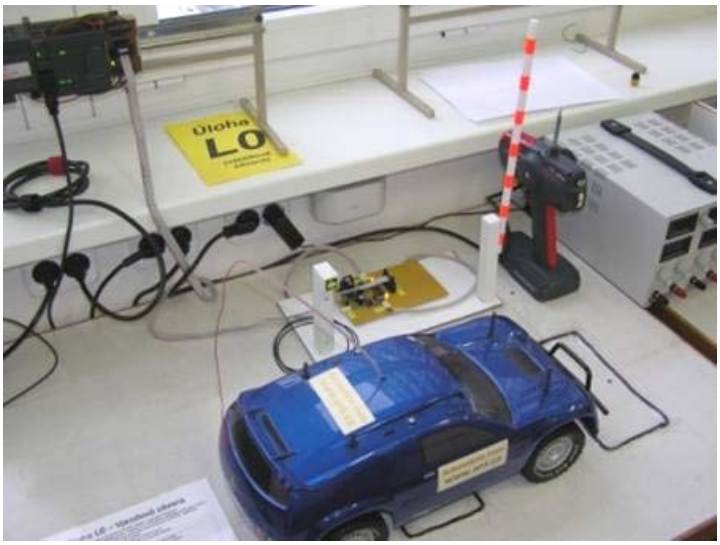
Obr. 7.3 Obrázek převzatý z mapy.cz . Server uvádí odkaz https://mapy.cz/zakladni?x=14.4086029&y=50.0749740&z=18&l=0&source=stre&id=122878 (stav k 27.11.2016). „A“ značí křižovatku na obrázku 7.1, „B“ na obrázku 7.2. Ulice Zborovská je v tomto úseku jednosměrná, resp. Diezenhoferovy sady se objíždí proti směru hodinových ručiček. Na všech křižovatkách jsou semafory a je vyznačena přednost. Hlavní silnice je ve směru velkého smíchovského okruhu, od Hořejšího nábřeží do ulice V Botanice.

Zdrojem dat jsou indukční smyčky ve vozovce. Intenzita představuje počet aut, které smyčka za daný časový úsek zaregistruje, obsazenost pak odpovídá době, po kterou vozidla nad smyčkou průměrně stojí. Konstrukci indukční smyčky naznačuje například obrázek z Wikipedie:



Obr. 7.4 Schematické zobrazení indukční smyčky. Zdroj: Wikipedie, https://en.wikipedia.org/wiki/Induction_loop (7.11.2016).

Smyčka je vložena do drážek, vyříznutých do vozovky diamantovým řezným kotoučem. Po uložení jsou vodiče ihned zality asfaltovou směsí. Podle zkušeností z praxe nedochází v těchto místech k většímu poškozování vozovky, než v místech, kde smyčka není.

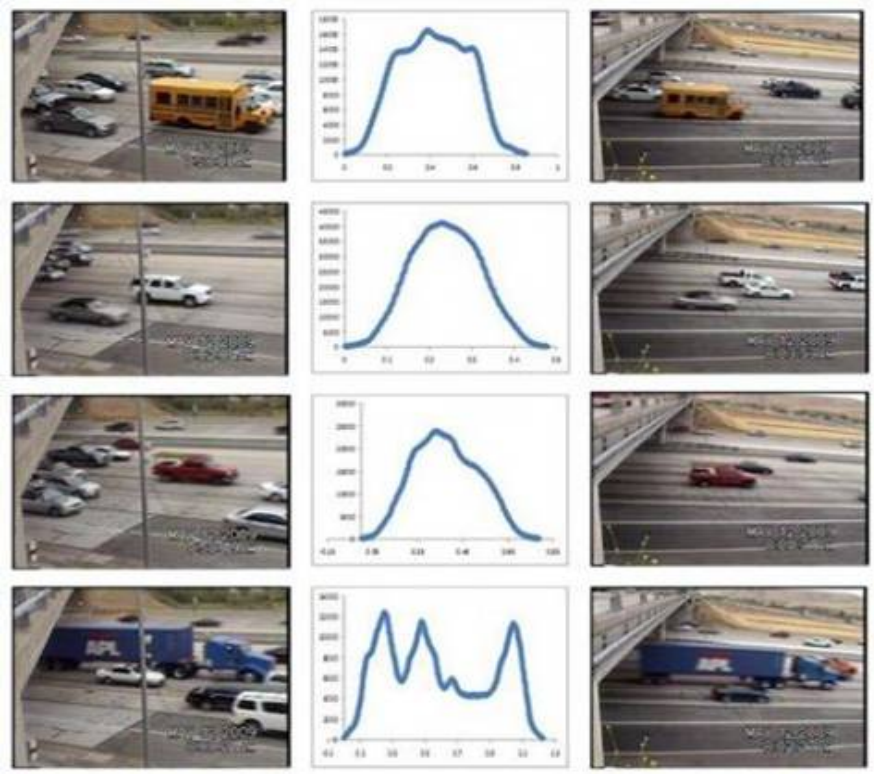


Obr. 7.5 Model vjezdové závory s dvěma indukčními smyčkami z laboratoře automatického řízení FS ČVUT (zdroj: vlab.fs.cvut.cz, stav k 27. 11. 2016).

Z druhé strany často dochází k poškození samotné smyčky nebo napojení v místě přívodu, resp. přechodu z konce zářezu ve vozovce do ochranné trubky, kterou je dále

vedena k řídicí jednotce signalizace. Proto bývá dnes často nahrazována kamerovou detekcí přítomnosti vozidla. Řešení kamerou je s ohledem na cenu instalace podstatně levnější a z provozního hlediska spolehlivější. Kamera naopak kromě požadované detekce přítomnosti vozidla mnohem častěji detekuje i jiný pohyb, jako odraz světla na mokré vozovce (nežádoucí, pokud vozidlo jede v jiném pruhu), nebo pohyb chodců ve vozovce. Vzhledem ke způsobu využití to ovšem není příliš na závadu.

Indukční smyčky měly další výhodu. Z jejich výstupu šlo poměrně snadno odhadnout, jaké vozidlo se nad smyčkou pohybovalo (obr. 7.6). Indukční smyčka musí být napájena střídavým proudem o frekvenci řádově desítek kilohertzů. Vyhodnocovaný signál je změna indukčnosti oproti klidovému stavu. Protože se jedná o vysokofrekvenční princip, detekce spočívá ve vířivých proudech, které se vytváří v jakýchkoli vodivých materiálech a které vytvářejí opačné magnetické pole, než je způsobuje. Zvyšují tím zdánlivý odpor (impedanci) cívky, kterou měříme.



Obr. 7.6 Ukázka rozlišení typu automobilu podle odezvy indukční smyčky. Zdroj: http://clranalytics.com/projects/advanced-sensor-technologies/SBIR-REID-2009 (26.11.2016)

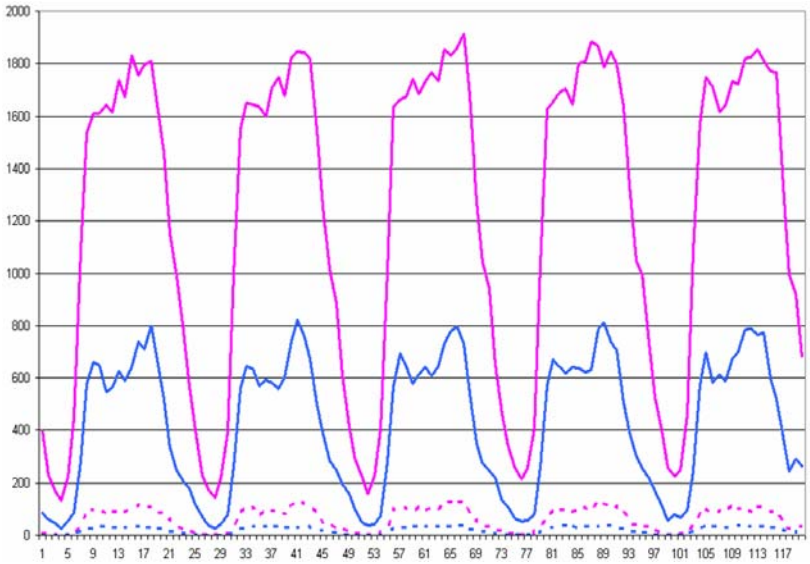
7.2 Předzpracování dat z křižovatek

Data jsou dostupná v excelovských souborech, agregovaná po 90 s nebo jedné hodině.

Při kroku 90 s jsou intenzity v řádu jednotek, v největší špičce dosahují 50. To jsou příliš malá čísla, snadno ovlivnitelná šumem.

První pokus směřoval k přímému použití hodinových intenzit. Z grafického zobrazení byla vybrána data za několik dní, které jsou nějakým způsobem reprezentativní. Zejména, jedná se o denní dobu, od cca 7:00 do cca 19:00, všední dny kromě pátku, a na grafech nejsou patrné dopravní problémy. Na obrázku 7.7 jsou ukázky průběhu z několika po sobě jdoucích dní. Pokud je více souběžných snímačů (dva pruhy vedle sebe), údaje o jejich intenzitě jsou samozřejmě sečteny. U obsazeností se jedná o průměrnou hodnotu v daném

čase, proto jsou vynášené hodnoty v tomto případě mnohem nižší (v grafu jsou ve stejném měřítku hodinové intenzity dopravy a průměrné doby stání v sekundách na uvedených snímačích).



Obr. 7.7 Průběhy intenzit (plná čára) a obsazeností (přerušovaná) ze dvou směrů příjezdu do křižovatky. Vodorovně pořadové číslo vzorku, tedy pořadí hodiny od začátku týdne.

Výsledný soubor má ovšem jen 100 hodnot. Jedno ze základních pravidel pro hledání funkční závislosti na základě zašuměných dat doporučuje mít desetkrát více hodnot, než je stupeň volnosti nalezené/hledané funkce. Tato obecná poučka (objevuje se ve všech kursech umělé inteligence napříč univerzitami, viz např. Caltech, [Abu-Mostafa]) se navíc musí brát spolu s rozumnou úvahou – čím více šumu, tím více hodnot je třeba použít. Pak již je tento počet malý.

Naproti tomu data pro 90s úseky byla příliš malá a zatížena šumem. Bylo tedy třeba přistoupit k nějakému kompromisnímu řešení, s agregací dat do delších úseků. Pro agregaci bylo použito tzv. okna, které umožňuje odstranit problémy s okamžitými ději tím, že se krajní prvky připočtou ponásobené nějakým koeficientem, a tím se okraje vyhladí. Obvyklé řešení je, že se úseky, ve kterých se získávají data, překrývají. Okna jsou reprezentována různými, převážně spojitými funkcemi. Například u Fourierovy transformace se často používá Hammingovo okno:

$$w(n) = \alpha - \beta \cos\left(\frac{2\pi n}{N-1}\right) \tag{7.1}$$

, kde $\alpha = 0{,}54$ a $\beta = 1-\alpha$.

V našem případě ovšem nepotřebujeme takové vlastnosti okna, jako je plynulost funkce, nebo její analytická integrovatelnost, proto lze použít buď přímo lichoběžníkové okno:

$$\begin{aligned} w(n) &= \frac{n}{k},\ n = 1,\ldots,k-1, \\ w(n) &= 1,\ n = k,\ldots,m-k, \\ w(n) &= \frac{m-n}{k},\ n = m-k+1,\ldots,m \end{aligned} \tag{7.2}$$

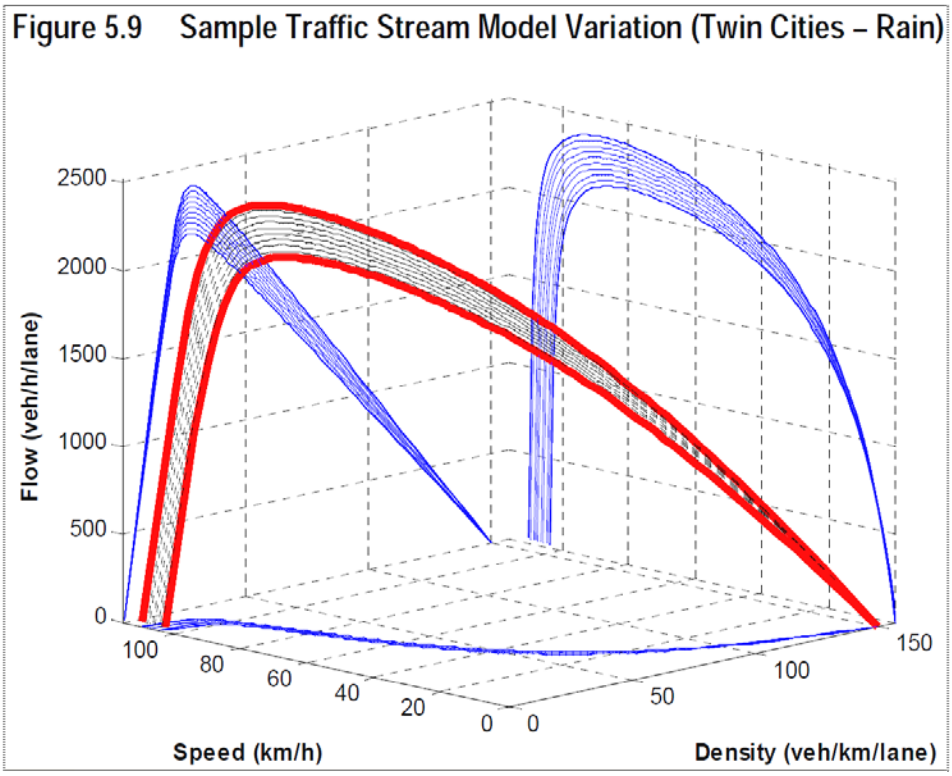
, nebo některé z oken, které mají střední část rovnu jedné, například Tukey window.

Možná reprezentace lichoběžníkového okna je okno, tvořené koeficienty (k=3, m=10):
{0,33; 0,67; 1; 1; 1; 1; 1; 0,67; 0,33}
Má celkovou délku 10 vzorků (koncové body jsou nulové), vhodné překrytí je buď krajními dvěma vzorky (žádná hodnota pak není fakticky využita vícekrát, než by měla, tomu odpovídá posunutí 7 vzorků), nebo může být překrytí až 50% předchozího a 50% následujícího vzorku (tomu odpovídá posunutí 5 vzorků).

Metodika s použitím takto definovaného lichoběžníkového okna s překrytím okrajových bodů (součet váhy = 1), tj. krok 7, byla použita na 90s intenzity provozu dvou po sobě následujících semaforů v ulici Zborovská.

7.3 Hledaná závislost

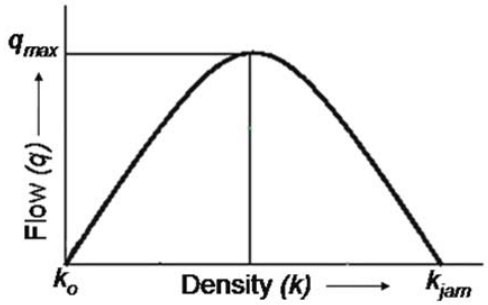
Mezi dopravními daty se pro vyhodnocení 3D závislosti (proměnná závislá na dvou nezávislých proměnných) nabízí například vazba velikosti dopravního toku, rychlosti a hustoty provozu. Hezký graf nalezl Google v práci [Rakha] (str. 74, resp. 5-12), viz následující obrázek:



Obr. 7.8 Závislost velikosti dopravního toku na rychlosti a hustotě provozu [Rakha].

V našem případě pro ukázkový výpočet nebyla k dispozici podobná data. Výše uváděná data z průjezdnosti křižovatkami nabízí jenom intenzity (jiné označení dopravního toku) a obsazenost, kterou lze považovat za úměrnou hustotě. Vzhledem ke tvaru křivky, kdy se stoupající hustotou (pravděpodobností přítomnosti vozidla v daném místě, resp. v místě detekující indukční smyčky) dopravní tok nejprve stoupá a následně po zahlcení začne klesat, není možné velikost dopravního toku použít jako nezávislou proměnnou. Vzhledem k dostupným datům jsem nakonec v tomto pokusu hledal závislost intenzity provozu na druhá z křižovatek na obsazenostech z obou větví první křižovatky. Je třeba dodat, že většina vozidel nejede tímto směrem.

K hledané závislosti, kdy podle klasické teorie je křivka dokonce obrácená parabola ([Venkatachalam], novější literatura, například [FTTF], uvádí podstatně složitější křivky), je třeba dodat, že toto platí pro volnou silnici, nikoli pro křižovatky. S ohledem alespoň na přiblížení se těmto podmínkám nebyly použity údaje obsazenosti ze smyčky bezprostředně před semafory, ale vzdálenější pomocné indukční smyčky, které se obvykle používají pro automatický noční provoz a umožňují na neobsazené křižovatce, kde je signál „stůj“ ve všech směrech, uvolnit směr, ze kterého právě přijíždí auto. Tento režim mimochodem není na daných křižovatkách použit, resp. data pochází z doby, kdy nastavení se přepínalo jen mezi dlouhými intervaly v denním režimu a krátkými v nočním (podle okamžité obsazenosti).

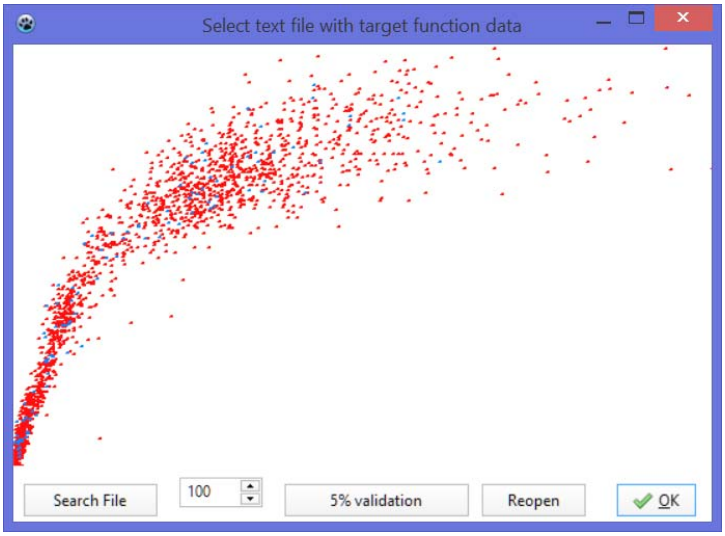


Obr. 7.9 Závislost velikosti dopravního toku na hustotě provozu [Venkatachalam, kap. 2.5.1]

Předzpracováním dat lichoběžníkovým oknem vznikl soubor s 1919 kombinacemi obsazeností z míst, označených na obr. 7.1 jako DVA1 (součet z obou pruhů tvoří nezávislou proměnnou ***x***; pomocný snímač S15 nemohl být použit, protože byl po velkou část měsíce mimo provoz) a DVB (opět součet, onačený jako ***y***). Intenzita je ze snímače, označeného na obr. 7.2 jako DVA (opět součet) a slouží jako hledaná závislá proměnná. Byly vynechány úseky, kdy některý z uvedených snímačů byl označen v datech jako „v poruše“. Všechna data pochází z května 2013.

7.4 Hledání závislosti programem gpy.exe

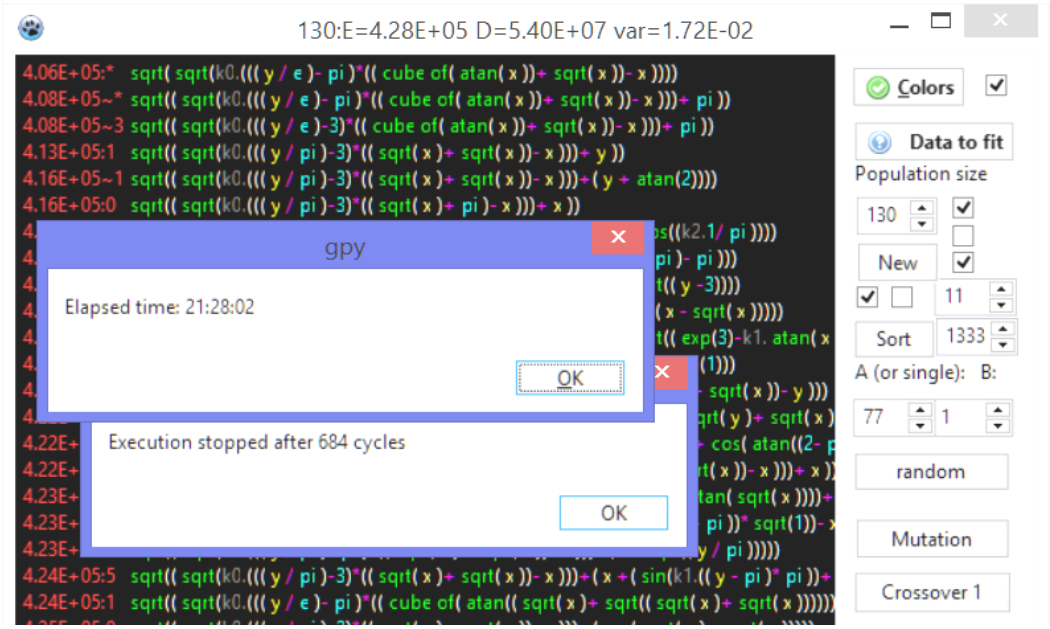
Nejprve byla načtena data. Program v okně po načtení dat zobrazí pro informaci 2D náhled dat, aby byla potvrzena čitelnost. Je-li třeba kontrolovat 3D náhled, lze tak učinit po vygenerování počáteční populace u kterékoli vygenerované funkce.



Obr. 7.10 Data po načtení a zvolení testovací sady (zobrazena modrými body)

Je-li to požadováno, je možné oddělit několik procent dat (pět, deset), která se nepoužijí na trénování a mohou později sloužit k vyhodnocení kvality nalezené funkce. U symbolické regrese metodou genetického programování to není třeba, protože nedochází k přetrénování, jak bude vidět i na výsledku tohoto pokusu.

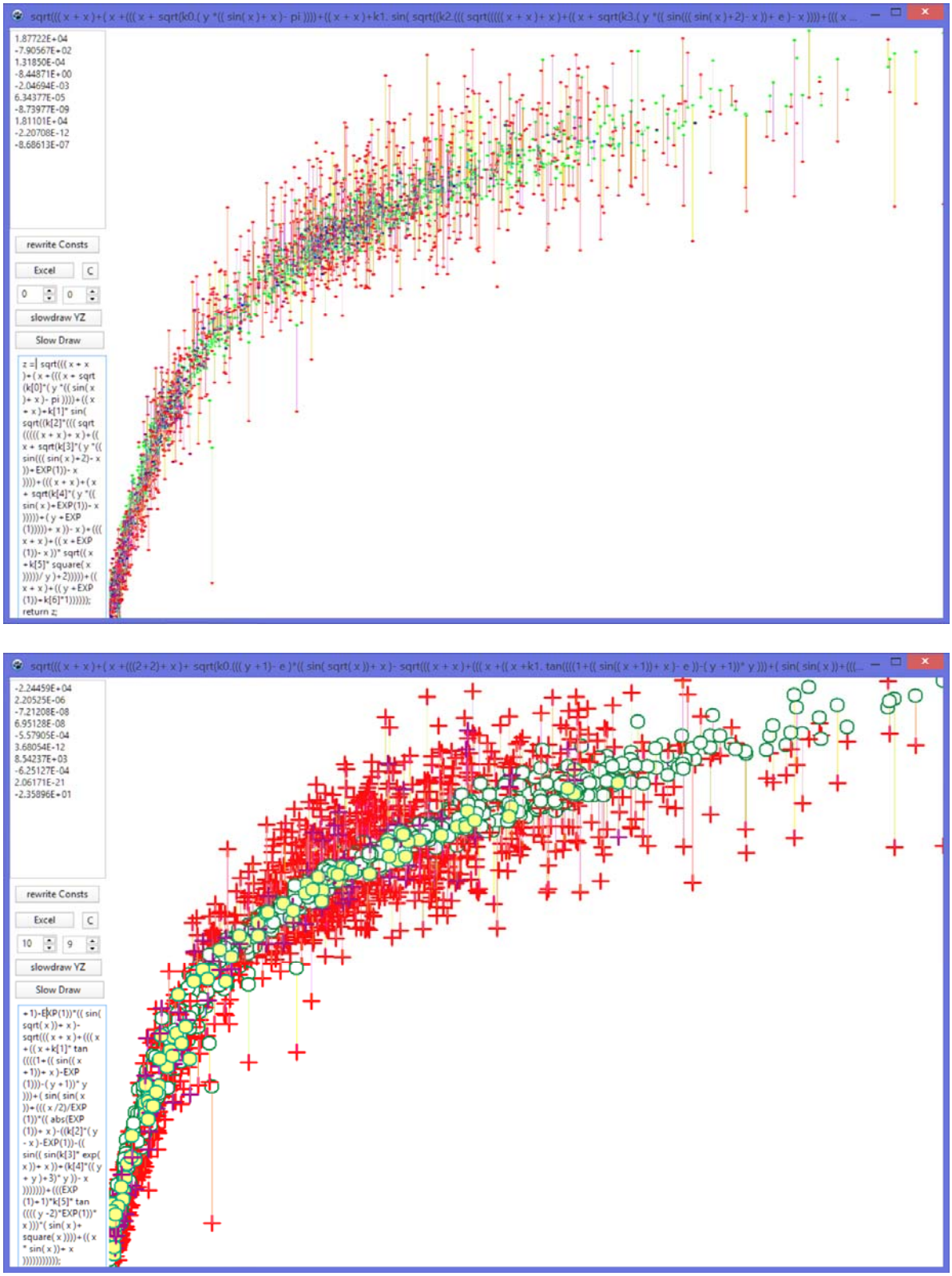
Následuje vygenerování počáteční populace, nastavení parametrů a spuštění evoluce, nejlépe přes noc. Pokud není požadován konkrétní počet generací (například testování programu, jako v kapitole 5), nastavíme zbytečně velkou hodnotu a následně evoluci zastavíme jakoukoli klávesou (aby nebyla vyvolána žádná funkce, použijeme klávesu Esc).



Obr. 7.11 Horní část hlavního okna programu po posledním přerušení. Viditelná pomocná okna se zprávou o provedeném čase a počtu generací, ale jedná se jen o údaje od posledního přerušení (a nového spuštění). Červené číslo na začátku řádku zobrazuje dosaženou chybu (přesněji řečeno hodnotu účelové funkce, zahrnuje nastavené přičtení 1333-násobku délky genomu jedince, které bylo zvoleno v poslední fázi pro získání kratšího zápisu funkce)

V seznamu funkcí v barevném okně reprezentuje číslo na začátku řádku hodnotu účelové funkce; bezprostředně následuje znak „:“ u jedince, získaného křížením, a tilda (~) u jedinců, získaných mutací (jejich velký počet na obr. 7.11 je způsoben nastavenou vysokou úrovní mutací). Další číslice je doba života jedince, * značí více jak 9 generací. Vysoké hodnoty účelové funkce jsou způsobeny zvolením varianty „suma čtverců odchylek“.

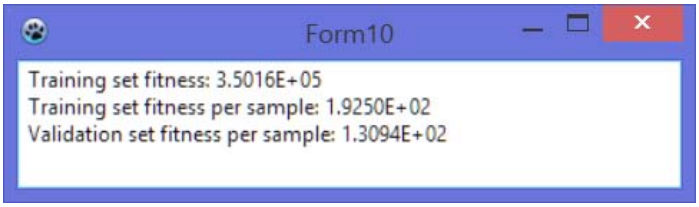
Následně je možné si vyvolat kliknutím myší na funkci (je-li zobrazena, dostupných je zde jen 31 prvních) podle nastavení 2D nebo 3D graf nalezené funkce a výchozích dat. Výchozí data jsou zobrazena červenými křížky, vypočtené hodnoty nalezené funkce zelenými kolečky (při výchozím nastavení je ale velikost těchto symbolů „0“ a program kreslí jen barevné tečky). Odpovídající si dvojice jsou spojeny tenkou čarou. U 3D grafu tato spojnice začíná v křížku, ale nekončí vždy v kolečku, v důsledku aplikovaného prostorového zkreslení (perspektivy), které tyto čáry nerespektují.



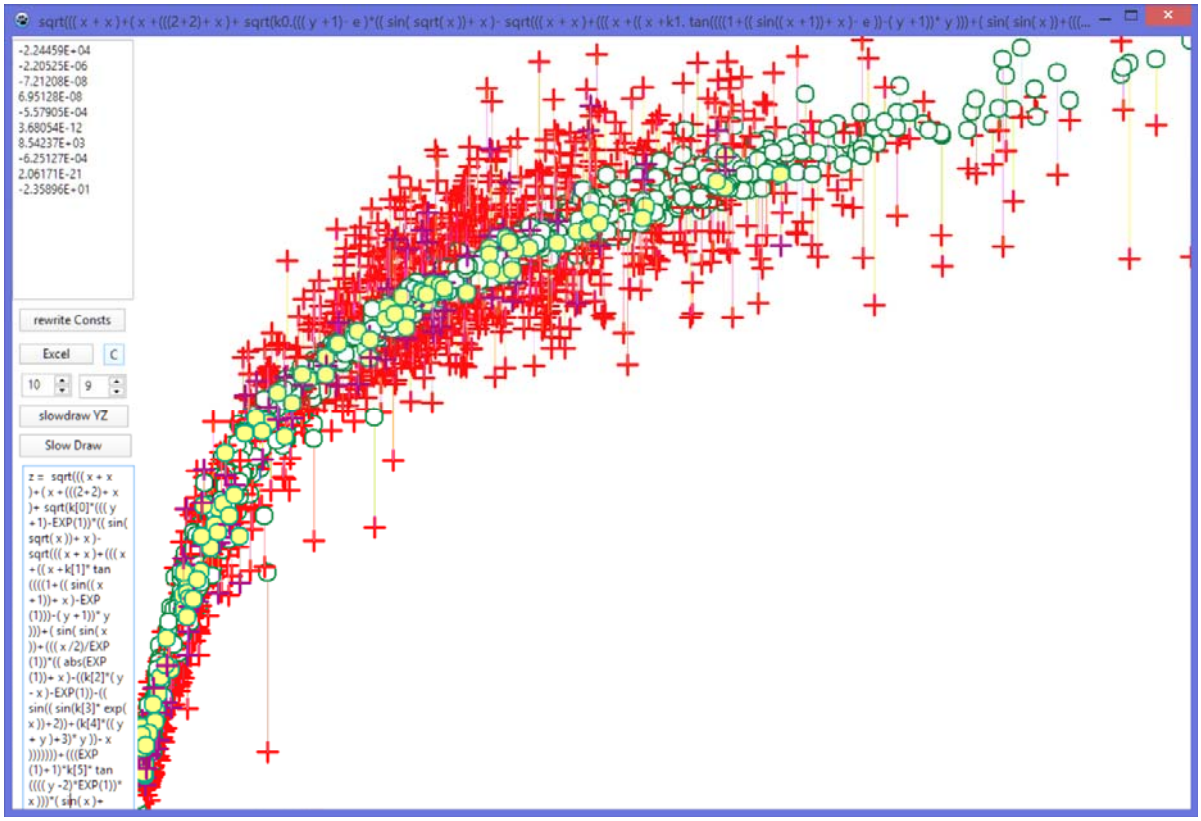
Obr. 7.12 Zobrazení nalezené funkce po prvních třech hodinách výpočtu. Nahoře zobrazení body, dole křížky a kolečky. Na dolním obrázku je lépe viditelná testovací sada, která se vykresluje jako poslední (žlutě vyplněná kolečka a fialové křížky). Je třeba dodat, že se nejedná o tutéž funkci, horní obrázek je nejlepší jedinec, dolní je druhý v pořadí. Přes zcela odlišnou funkci a nastavení konstant je způsob proložení obdobný.

Po zběžném vyhodnocení byla nastavena vyšší úroveň mutací a program byl spuštěn znovu, tentokráte přes noc. Výsledný stav je na následujících obrázcích:

Obr. 7.13 Celé okno programu po dalších skoro 21 hodinách výpočtu (viz číslo vpravo dole). Zde je vidět nastavení programu: Evoluční tlak 1,02, 130 jedinců a 77 křížení v generaci, úroveň mutací nepřímo úměrná hodnotě variačního koeficientu populace, základní (pro hodnotu 0,2) je 55, aktivní je EGD algoritmus i variace konstant.

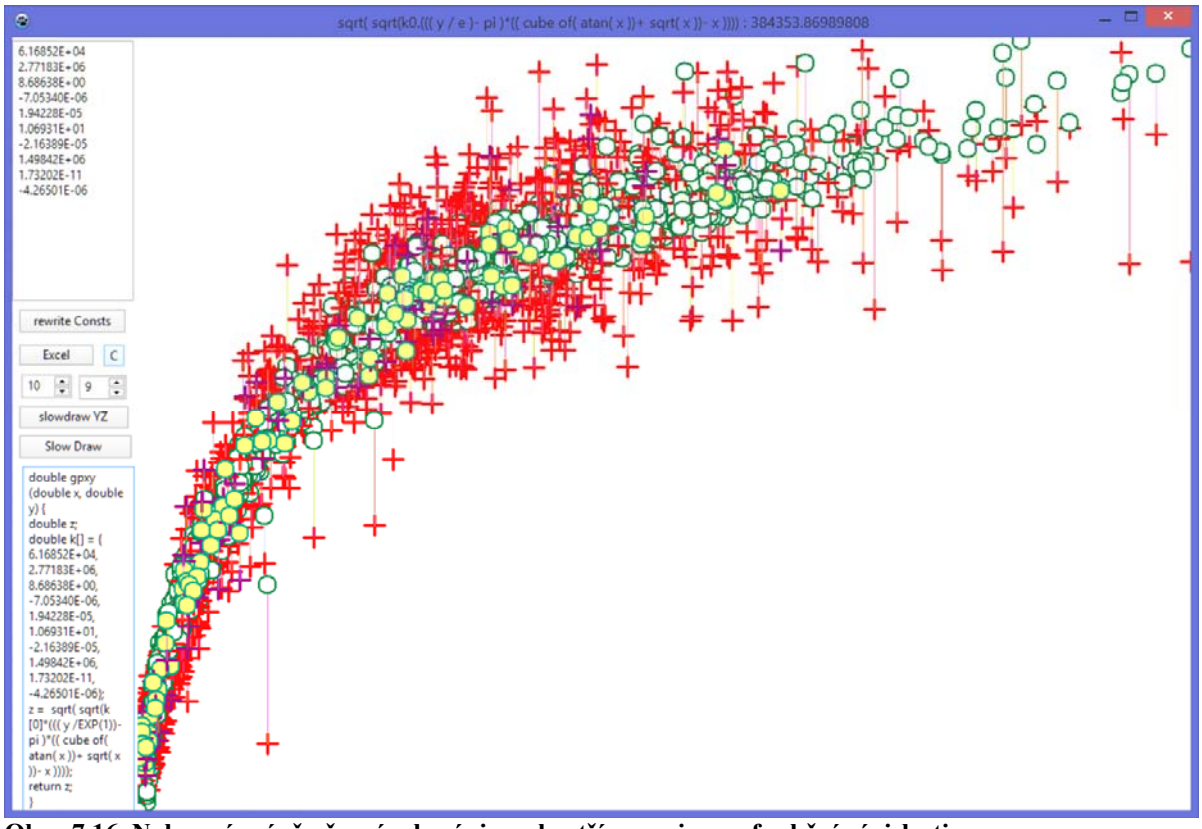


Obr. 7.14 Hodnota chyby symbolické regrese v této fázi. Nevýhodou je, že zápis funkce je velmi dlouhý a komplikovaný. Další evolucí se vlivem nastavení tato čísla zhoršila, viz obr. 7.11, ale byla nalezena méně komplikovaná funkce.

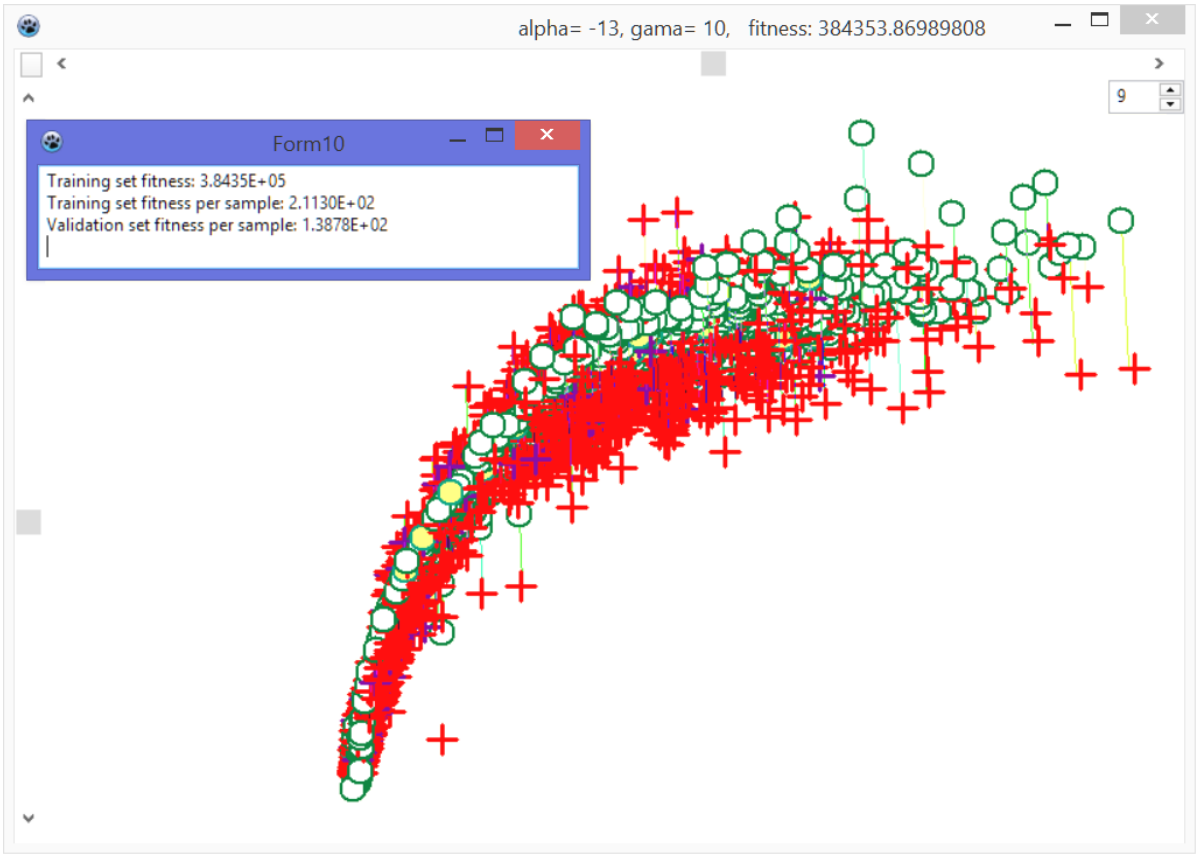


Obr. 7.15 Ukázka proložení. V záhlaví okna je vypsána jen část funkce a hodnota účelové funkce se již nevejde.

S ohledem na přílišnou délku funkce byla nastavena vyšší penalizace délky zápisu funkce a program byl opět restartován přes noc. Stav po přerušení je nesystematicky na obr. 7.11.



Obr. 7.16 Nalezená méně přesná, ale zápisem kratší aproximace funkční závislosti



Obr. 7.17 Tatáž funkce ve 3D zobrazení, posuvníky lze funkcí otáčet. V levém horním rohu pomocné okno, které se objeví současně s funkcí. Všimněte si, že chyba vztažená na vzorek (jen ta je porovnatelná) je u testovací sady menší, než u trénovací. Ve 3D zobrazení chybí zápis funkce.

Po dalších jednadvaceti hodinách evoluce byla získána funkce s poměrně krátkým zápisem pro C++:

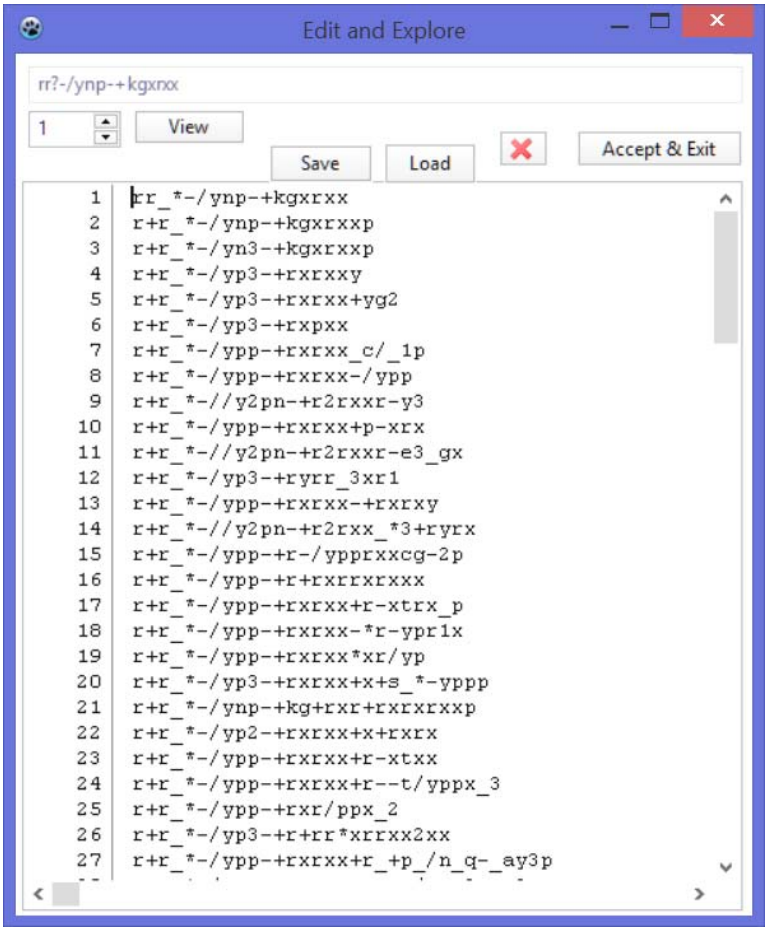
z = sqrt(sqrt(k[0]*(((y /exp(1))- pi)*((cube of(atan(x))+ sqrt(x))- x))));

tj.

$$z = \sqrt[4]{k_0 \cdot (\frac{y}{e} - \pi) \cdot (\arctan^3 x + \sqrt{x} - x)} \qquad (7.3)$$

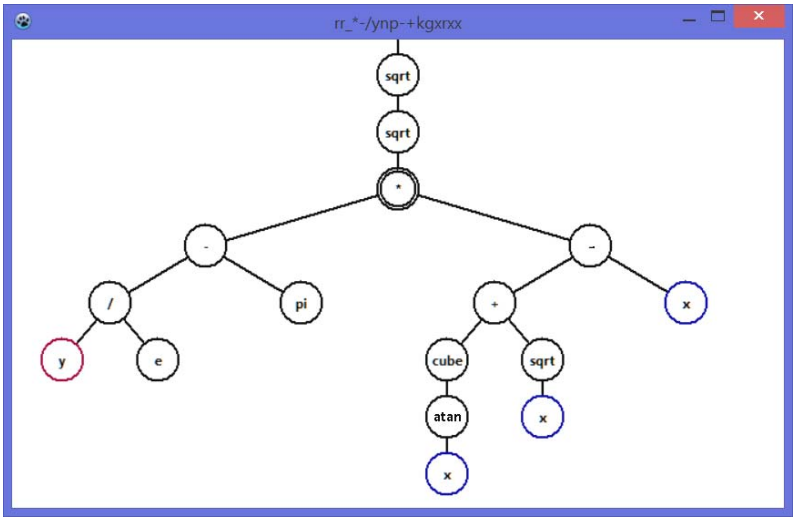
... s hodnotou jediné konstanty k_0 = 6,16852.10^4, π a e jsou známé matematické konstanty. Nedostatkem zápisu je, že ani C++, ani Excel funkci třetí mocniny nemají, takže by bylo třeba funkci ručně přepsat. Tomuto stavu by se bývalo bylo možné vyhnout tím, že by se v okně s volbou četností u této funkce uvedla nula (obr. 5.1). Řešení by pak vedlo na jinou funkci.

Pro informaci je zajímavé si prohlédnout i celou zbylou populaci, v poslední fázi zatíženou vysokou hodnotou mutací a penalizační funkce (délky). To lze tlačítkem Explore, které zobrazí editační okno:



Obr. 7.18 Okno umožňující editaci jedinců, popřípadě ukládání a načítání mezivýsledků evoluce.

Obr. 7.19 Dolní část téhož okna po posunutí na konec populace. Zběžným pohledem je vidět, že vysoká penalizační funkce na konec populace vytlačila jedince s delším zápisem a více konstantami, zatímco jednodušší jsou zobrazeny na začátku populace na předchozím obrázku. Poznámka: editor čísluje řádky, jedinci jsou číslováni od nuly, proto při 130 jedincích v populaci + elitní jedinec je poslední řádek očíslován 131.

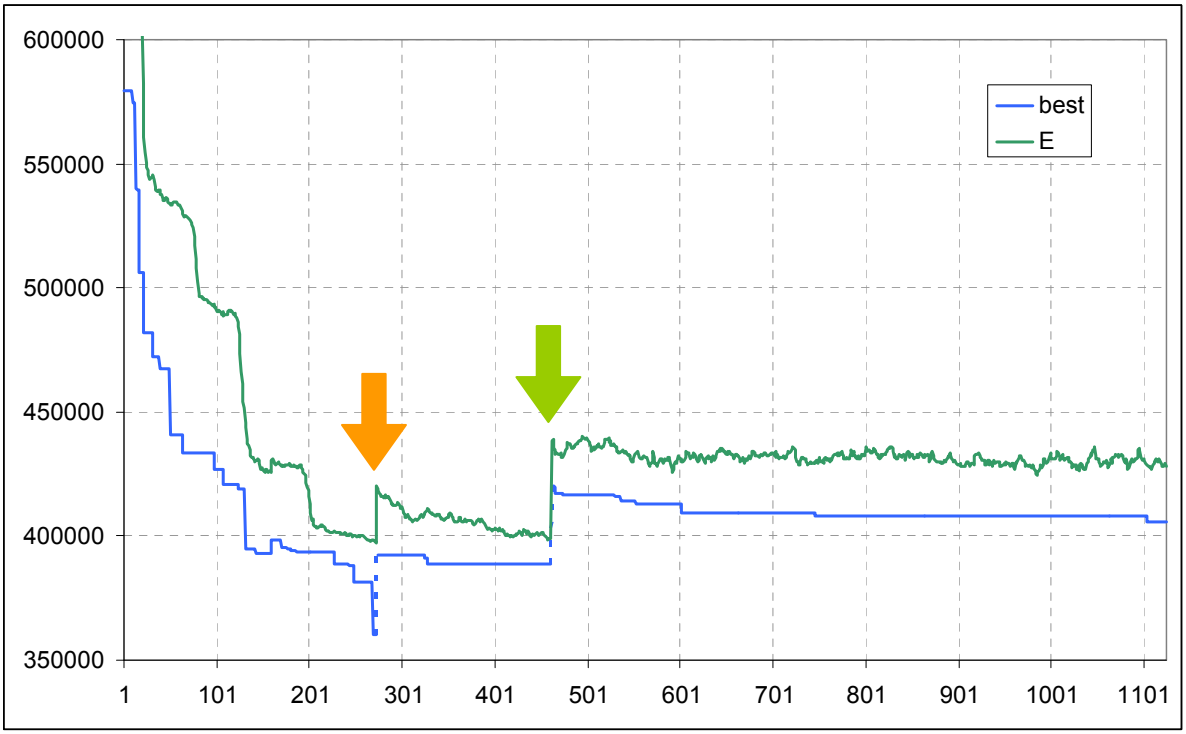


Obr. 7.20 Obrázek stromu nejlepšího jedince. Dvojité kolečko je místo násobení konstantou.

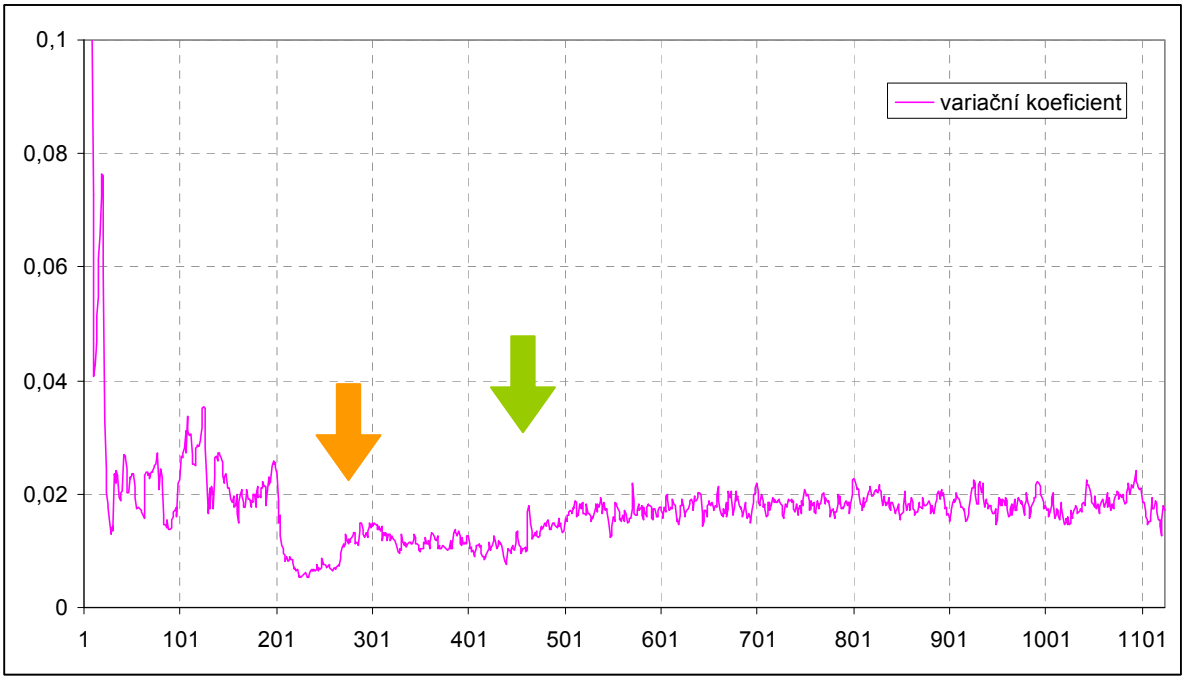
Závěrem lze říci, že by bylo ideální najít závislost, která by se stoupající obsazeností se hodnotou intenzity vracela k nule. Je ale třeba konstatovat, že symbolická regrese jen proloží data funkční závislostí, a na obr. 7.10 není patrná nějaká tendence k poklesu. Program ji tedy nalézt ani nemůže.

7.5 Průběh evoluce

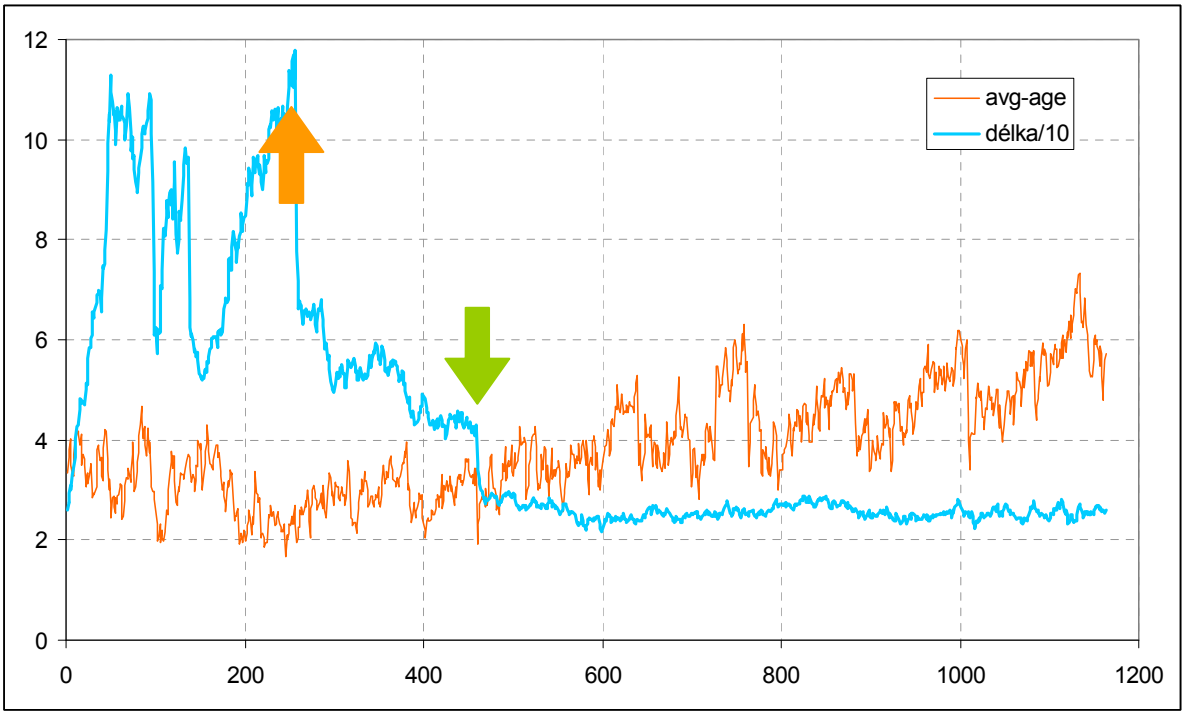
Při výše provedeném pokusu byl průběh evoluce samozřejmě zaznamenáván do souboru. V průběhu došlo ke dvěma přenastavením parametrů evoluce s cílem zvětšit diverzitu populace a snížit komplexnost nalezené funkce zvýšením penalizace její délky, po 129 generacích a následně po dalších 115 generacích. Pokud do grafu vyneseme průběh průměrné hodnoty jedinců v první polovině populace (na obr. 7.21 zeleně) a hodnotu nejlepšího (elitního) jedince (na grafu modrá křivka), jsou na obou křivkách patrné dva okamžité nárůsty. Jedná se o důsledek větší penalizace délky jedince, kdy změnou způsobu výpočtu účelové (fitness) funkce dojde ke skokovému nárůstu hodnoty. Hodnota účelové funkce elitního jedince jinak samozřejmě může jedině trvale klesat.



Obr. 7.21 Průběh hodnot účelové funkce, modře nejlepší jedinec, zeleně průměr za první polovinu populace. Oranžová a zelená šipka vyznačují okamžik ručního přenastavení parametrů.



Obr. 7.22 Variační koeficient je hodnota směrodatné odchylky dělená průměrnou hodnotou (6.11), program se jej aktivním řízením úrovně mutací snaží držet na hodnotě 0,02. V úseku před druhým přenastavením parametrů byla ale základní úroveň mutací zřejmě tak malá, že ani při nastavení jejího pětinásobku nebylo možné této požadované úrovně dosáhnout.



Obr. 7.23 Průměrný věk jedince v populaci (oranžová tenká) a desetina průměrné délky interního zápisu funkce (míra komplexnosti). Na grafech jsou označena dvě přenastavení parametrů, v obou případech se jedná o zvýšení míry penalizace délky zápisu jedince a současně mírné zvýšení úrovně mutací. Stárnutí populace na konci evoluce odpovídá stavu, kdy ani mutace nepřinášejí zlepšení dosahovaných hodnot účelové funkce. Průměrná délka života ale nemůže příliš růst, protože při tomto experimentu byla omezena na maximálně 11 generací. Po velkém počtu generací se situace ustálí tak, že je pro překročení délky života v každé generaci odstraňován přibližně stejný počet jedinců. Pokles po cca 750. generaci je v souladu se zlepšením hodnoty účelové funkce (obr. 7.21), kdy je pravděpodobně nová část struktury postupně přenášena na novou generaci, takže má lepší hodnoty, než starší jedinci.

8. Závěr

Výsledkem práce je přínos pro řešení problematiky symbolické regrese metodou genetického programování. Kombinací metod genetického programování a exponenciono-vané gradientní metody (exponencionated gradient descent) je dosaženo vysoké robustnosti při zachování přijatelné rychlosti běhu programu. Tato kombinace není zatím dohledatelná v citačních rejstřících. Nově v této práci je i mutace spočívající v odříznutí kořene stromu (začátku zápisu).

Dalším přínosem práce je nový přístup k vyčíslovaným konstantám symbolické regrese, které jsou nově zaznamenávány značkami přímo do prefixového zápisu stromu funkce v rámci genetického programování. To umožňuje velmi kompaktní zápis (jako značka je použit nejvyšší, jinak nevyužívaný bit u příslušného symbolu, který je jinak volen ze sady ASCII). Koeficienty jsou pak reprezentovány jako násobení vypočtenou hodnotou v označeném místě. Celkově je zápis ve srovnání s běžně používanými postupy i ve srovnání s gramatickou evolucí mnohem úspornější. Podařilo se také ošetřit, aby genom neobsahoval žádné nevyužívané části, ani žádný gen není použit dvakrát. Tím je ošetřeno, že celý zápis jedince podléhá evolučnímu tlaku.

Koeficienty jsou ukládány do dvourozměrného pole, jejich počet pro konkrétního jedince populace je tedy omezen (v přiložené aplikaci na deset) a reprezentován jedním řádkem v tomto poli. V každém cyklu genetického programování je pro každého jedince proveden jen omezený počet cyklů gradientního algoritmu. Program se k upřesnění koeficientů pro každého jedince vrací po každém cyklu genetického programování. Oba cykly se tak prolínají a tím umožňují jak zrychlit běh programu, tak odstraňovat z populace funkce, které využívají koeficientů v nadbytečném počtu.

Jako samozřejmost je exponencionovaná gradientní metoda implementována včetně proměnného kroku, známého z metody simulovaného žíhání. Pro řešení problému případného lokálního minima účelové funkce je navíc použito metody variace konstant (aplikované na koeficienty). Tato metoda je z podstaty stochastická, ale při dostatečném počtu cyklů v kombinaci s gradientní metodou výrazně zvyšuje pravděpodobnost nalezení globálního minima účelové funkce pro daného jedince, sestaveného metodou genetického programování.

Současně vytvořená aplikace umožňuje vyhledávat symbolickou regresí funkce dvou proměnných, jejichž hodnoty jsou známy v dostatečném počtu bodů. Výhodou genetického programování je čitelný zápis nalezené výsledné funkce. Program umožňuje vykreslovat 3D i 2D grafy a generovat jednotlivé zvolené funkce (výběrem z populace při zobrazení zápisu funkce a hodnoty její účelové funkce) ve formátu vhodném pro MS Excel (připraveno pro 3D graf) nebo jako definici funkce pro C++. Program umožňuje oddělit ověřovací (testovací) sadu. Při testování bylo prokázáno, že v nadpolovičním množství případů má nalezená funkce relativní chybu na vzorek menší pro testovací, než pro trénovací sadu (v ostatních případech je tato hodnota srovnatelná), a že tedy metoda genetického programování v souladu s literaturou ve srovnání s prokládáním polynomem nebo náhradou neuronovou sítí netrpí přetrénováním.

Program disponuje záznamem statistických dat o populaci do textového souboru (formát .CSV), čímž umožňuje testovat chování použitých metod při změně parametrů. Současně byly vytvořeny další tři programy, sloužící k ukázce evolučních metod v rámci předmětu Umělá inteligence. Z nich získané grafy o chování při různých nastaveních ukazují význam mutací a zejména výhodnost ruletové metody ve srovnání s turnajovou metodou, která je při srovnatelném výkonu přibližně pětkrát pomalejší a také více postižena všemi problémy, které tyto metody mají (ve skutečnosti může být rozdíl v tom, že ruletová metoda, využívající seřazení jedinců, může být proti negativním jevům, jako je uvíznutí populace

v důsledku ztráty diverzity, vhodným způsobem ošetřena). Na základě těchto testů by bylo možné opravit některá vžitá schémata.

Podstatnou částí této práce jsou i data, získaná pomocnými programy, popisovanými v kapitole 4, která umožňují na vhodných modelech vyzkoušet vhodná nastavení genetické evoluce. Získané informace o chování v závislosti na jednotlivých parametrech umožnily časovou úsporu při ladění hlavního programu pro symbolickou regresi. V dostupné literatuře lze sice doporučované parametry evoluce najít, ale konkrétní doporučované hodnoty se velmi (i řádově) rozcházejí. Současně byl na těchto programech otestován jako dominantní problém význam udržení diverzity populace. Tyto programy jsou včetně zdrojových kódů k dispozici na přiloženém CD. Případní zájemci si na nich tak mohou vyzkoušet další možnosti genetických algoritmů, než je aplikují na problematiku genetického programování.

8.1 Další možné rozšíření práce

Program může být rozšířen o další metody výpočtu koeficientů, a tím umožnit porovnání těchto metod. Dále je připraven na oddělené nakládání se samčí a samičí populací (chybí jen možnost rozhraní tyto parametry nastavovat). V současné době není možné spolu s aktuální populací uložit do souboru i konstanty, a tím celou rozpracovanou práci. Na zvážení je způsob realizace penalizace délky jedince a otázka vyřazování příliš příbuzných jedinců, zejména jejich detekce v populaci (vlivem přítomnosti konstant nemusí být po setřídění v populaci za sebou).

Jinou otázkou je počet nezávislých proměnných. Po dokončení vývoje by měla být zpětně odvozena varianta s jedinou proměnnou. Možnost více proměnných (například tří a čtyř) a její vliv na průběh evoluce by asi byla námětem další, podobně rozsáhlé práce.

7. Literatura

[8086] Mathur Sunil: Microprocessor 8086 : Architecture, Programming and Interfacing. PHI Learning Pvt. Ltd., New Delhi, 2011. ISBN 8120340876

[Abu-Mostafa] Yaser Abu-Mostafa: Learning from data. Machine Learning course, recorded at a live broadcast from Caltech. California Institute of Technology, 2012.

[AField] Riccardo Poli, William B. Langdon,Nicholas F. McPhee, John R. Koza, "A Field Guide to Genetic Programming". Lulu.com, 2008. pdf dostupné online http://cswww.essex.ac.uk/staff/poli/gp-field-guide/ ISBN 978-1409200734 (knihovny ČVUT: deponováno na FIT).

[Alg-BS] Pavel Stančík: Bucket sort. Zveřejněno na Algoritmy.net, online: https://www.algoritmy.net/article/152/Bucket-sort , 31.8.2016.

[Alg-BH] Pavel Stančík: Binární halda. Zveřejněno na Algoritmy.net, online: https://www.algoritmy.net/article/15/Binarni-halda, 5.9.2016.

[Alg-FY] Pavel Stančík: Fisher-Yatesův algoritmus. Zveřejněno na Algoritmy.net, online: https://www.algoritmy.net/article/43610/Fisher-Yatesuv-algoritmus, 11.8.2016.

[Andre] Andre, D., Koza, J.R.: Parallel Genetic Programming on a Network of Transputers. Výzkumná zpráva, Computer Science Department, Stanford University, Stanford, CA, 1995.

[Auger] A. Auger, N. Hansen: Theory of Evolution Strategies: a New Perspective. In: *A. Auger and B. Doerr, eds. (2010). Theory of Randomized Search Heuristics: Foundations and Recent Developments*. World Scientific Publishing, pp. 289-325. (https://www.lri.fr/~hansen/TRSH-ChapterAugerHansen.pdf)

[Baluja] *Baluja, Shumeet; Caruana, Rich (1995). Removing the genetics from the standard genetic algorithm (PDF). ICML.*

[Barmpalexisa] Barmpalexisa, P., K. Kachrimanisa, A. Tsakonasb, E. Georgarakis: Symbolic regression via genetic programming in the optimization of a controlled release pharmaceutical formulation. in: *Chemometrics and Intelligent Laboratory Systems*. Volume 107, Issue 1, May 2011, Pages 75–82 (Elsevier)

[Benke] Benke, K.K, Skinner, D.R.: A Direct Search Algorithm for Global Optimisation of Multivariate Functions. in: *Australian Computer Journal*, January 1991, pp. 73-85.

[Beyer] Beyer, H.G., Schwefel, H.P.: Evolution strategies, a comprehensive introduction. In: *Natural Computing* 1: 3–52, 2002. Kluwer Academic Publishers, Netherlands. Beyer, H.G., Schwefel, H.P.: Evolution strategies – A comprehensive introduction. in: Natural Computing (2002) 1: 3. doi:10.1023/A:1015059928466

[BT4] Brandejsky T.: Genetic Programming Algorithm with constants pre-optimization of modified candidates of new population. In: *Mendel 2004*, Brno, 2004, pp. 34-38.

[BT5] Brandejsky, T.: Genetic Programming Algorithm with Parameters Pre-optimization - Problem of Structure Quality Measuring. In: *Mendel 2005*-January *, Brno*, pp. 138-144, 2005. ISBN 80-214-2961-5.

[BT8] Brandejsky, T.: The application of behavioral features model to road traffic modeling and analysis. *Neural Network World*, 18 (1), pp. 45-53, 2008.

[BT12] Brandejsky, T.: Evolutionary system to model structure and parameters regression. *Neural Network World*, 22 (2), pp. 181-194, 2012.

[BT12a] Brandejsky, T.: Hybrid evolutionary systems for identification of nonlinear systems. *10th International Conference - PROCESS CONTROL 2012. June 12 - 15, 2012, Kouty nad Desnou, Czech Republic*

[BT13] Brandejsky, T.: Small populations in GPA-ES algorithm. in: Mendel 2013, pp. 31-36. ISBN: 978-802144755-4.

[Bukovsky] I. Bukovsky and N. Homma, "An Approach to Stable Gradient Descent Adaptation of Higher-Order Neural Units," IEEE Trans. Neural Netw. Learn. Syst., vol. 2016, no. DOI:10.1109/TNNLS.2016.2572310.

[Cramer] Cramer , N.L.: A Representation for the Adaptive Generation of Simple Sequential Programs. in: Proceedings of the 1st International Conference on Genetic Algorithms, pp. 183-187. Carnegie-Mellon University, Pittsburgh, PA, 1985.

[Dawkins] Dawkins, Richard: The Extended Phenotype. Oxford: Oxford University Press. 1982. ISBN 0-19-288051-9.

[D'haeseleer] D'haeseleer, P.: Context preserving crossover in genetic programming. In: *Evolutionary Computation*, IEEE World Congress on Computational Intelligence, 1994. Pages: 256 - 261 vol.1, DOI: 10.1109/ICEC.1994.350006

[DEAP] F'elix-Antoine Fortin, Francois-Michel De Rainville, Marc-Andr'e Gardner, Marc Parizeau, Christian Gagn'e: DEAP: Evolutionary Algorithms Made Easy. in: *Journal of Machine Learning Research*, pp. 2171-2175, 2012. http://deap.gel.ulaval.ca/doc/0.7/examples/symbreg.html (17.1.2017).

[EGD] Kivinen, Jyrki and Warmuth, Manfred K. Exponentiated gradient versus gradient descent for linear predictors. in: *Information and Computation*, 132(1):1–63, 1997. Dostupné on-line: Elsevier: http://www.sciencedirect.com/science/article/pii/S0890540196926127. doi:10.1006/inco.1996.2612

[Erickson] Erickson, J.: Algorithms. Elektronická kniha. Dostupné on-line: http://www.cs.illinois.edu/~jeffe/teaching/algorithms/ (30.11.2016)

[EVT] Zelinka, Ivan et al. Evoluční výpočetní techniky: principy a aplikace. Praha: BEN - technická literatura, 2009. 533 s. ISBN 978-80-7300-218-3.

[FFX] McConaghy, Trent: FFX: Fast, Scalable, Deterministic Symbolic Regression Technology. Chapter in: Genetic Programming Theory and Practice IX, editor: Rick Riolo and Ekaterina Vladislavleva and Jason H. Moore, Springer, 2011.

[FTTF] Levinson, David, at.al.: Fundamentals of Transportation/Traffic Flow. Publikováno na wikibooks, 2008-2014, https://en.wikibooks.org/wiki/Fundamentals_of_Transportation/Traffic_Flow (7.2.2017)

[Ghosh] Ghosh, A., Tsutsui, S., Tanaka, H.: Individual Aging in Genetic Algorithms. in: *1996 Australian New Zealand Conf. on Intelligent Information Systems*, Adelaide, Australia.

[Goldberg] Goldberg, David (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Reading, MA: Addison-Wesley Professional. ISBN 978-0201157673.

[GP] Banzhaf, Wolfgang at al.: Genetic Programming, an Introduction. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998. ISBN 978-1-55860-510-7.

[Gupta] M.M. Gupta, I. Bukovsky, N. Homma, M. G. A. Solo, Z.-G. Hou: Fundamentals of Higher Order Neural Networks for Modeling and Simulation. Artificial Higher Order Neural Networks for Modeling and Simulation, pp. 103-133, ed. M. Zhang, *IGI Global*, 2012.

[Hansen] Hansen, N., D.V. Arnold and A. Auger (2015). Evolution Strategies. In Janusz Kacprzyk and Witold Pedrycz (Eds.): *Handbook of Computational Intelligence*, Springer, Chapter 44, pp.871-898 (pdf: https://www.lri.fr/~hansen/es-overview-2015.pdf).

[Hitoshi 2001] Hitoshi Iba, Makoto Terao : Controlling Eective Introns for Multi-Agent Learning by Genetic Programming. in: *Soft Computing Agents: New Trends for Designing Autonomous Systems*, pp. 73-87, Physica-Verlag HD, Heidelberg, 2001. doi: 10.1007/978-3-7908-1815-4_3.

[Hlaváč] Hlaváč, Vl.: A program searching for a functional dependence using genetic programming with coefficient adjustment. in: *SCSP Prague*, 2016.

[Hlaváč 17] Hlaváč, Vl.: Genetic programming with either stochastic or deterministic constant evaluation. Připraveno pro časopis *Neural Network World,* zatím nepublikováno.

[Icke] Icke, Ilknur, and Joshua C. Bongard: Improving Genetic Programming Based Symbolic Regression Using Deterministic Machine Learning. in: *Congress on Evolutionary Computation* (CEC), 2013 IEEE. Pp 1763-1770.

[Jarkovský] Jarkovský, J.: Matematická biologie. online učebnice, http://portal.matematickabiologie.cz/index.php?pg=aplikovana-analyza-klinickych-a-biologickych-dat--analyza-a-management-dat-pro-zdravotnicke-obory--zaklady-korelacni-analyzy--spearmanuv-korelacni-koeficient, cit. 23.8.2016

[Kent] Kent D. Lee: Programming Languages: An Active Learning Approach. Springer Science & Business Media, Dec 15, 2008

[Kirkpatrick] Kirkpatrick, S., Gelatt Jr., C.D., Vecchi, M.P.: Optimization by simulated annealing. *Science*, 220 (4598), pp. 671-680. (1983)

[Kivinen] J. Kivinen, M.K. Warmuth: Exponentiated gradient versus gradient descent for linear predictors. *Information and Computation*, 132 (1), pp. 1-63, 1997.

[Koza] Koza, John: Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Computer Science Department, Stanford University, Margaret Jacks Hall, Stanford, CA, 1990. Dostupné on/line: < www.genetic-programming.com/jkpdf/tr1314.pdf >.

[Koza 1992] Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA: MIT Press, 1992. ISBN 978-0262111706.

[Koza 1995] Koza, J.R.: Survey of genetic algorithms and genetic programming. in: *WESTCON*, *1995 (Microelectronics Communications Technology Producing Quality*

Products Mobile and Portable Power Emerging Technologies). DOI: 10.1109/WESCON.1995.485447

[Koza 1997] Koza, J.R., at al: Automated synthesis of analogue electrical circuits by means of genetic programming. IEEE Transactions on Evolutionary Computation, 1(2), pp. 109-128.

[Koza 1998] Koza, J.R., at al: Evolutionary design of analog electrical circuit using genetic programming. Adaptive Computing in Design and Manufacture conference (ACDM-98), Plymouth, England 1998.

[Lahodiuk] Yurii Lahodiuk: Genetic-programming. publikováno: 2014 na: GitHub, Inc., https://github.com/lagodiuk/genetic-programming (17.1.2017)

[Langdon] W. B. Langdon: Quadratic bloat in genetic programming. in: *GECCO'00 Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*, San Francisco, CA, 2000.

[Langdon 02] W. B. Langdon and Riccardo Poli. Foundations of Genetic Programming. Springer-Verlag, 2002. ISBN 3-540-42451-2. Experimenty a výsledky dostupné on-line, http://www0.cs.ucl.ac.uk/staff/W.Langdon/FOGP/ , 27.7.2015

[Martin] Martin, B.: Statistics for Physical Sciences: An Introduction. Academic Press, 2012, ISBN 9780123877604

[Mills] Mills, Peter M., Albert Y. Zomaya a Moses O. Tadé: Neuro-Adaptive Process Control: A Practical Approach. Chichester, Wiley, 1995. ISBN 0471959979.

[Mitchell] Mitchell, Melanie (1996). An Introduction to Genetic Algorithms. Cambridge, MA: MIT Press. ISBN 9780585030944

[Ošmera] Ošmera, P.: Two Level Parallel Grammatical Evolution. in: *Advances in Computational Algorithms and Data Analysis.* Springer Science+Business Media B.V., 2009, ISBN: 978-1-4020-8918-3.

[Rakha] Rakha, H., et al.: Empirical Studies on Traffic Flow in Inclement Weather. Project final report, Virginia Tech Transportation Institute, 2007.

[Rektorys] Rektorys, Karel, a kol.: Přehled užité matematiky. 4. nezměň. vyd. Praha: SNTL, 1981.

[Soule] Soule, T., Foster, J. A., and Dickinson, J. (1996). Using genetic programming to approximate maximum clique. in: *Genetic Programming*, 1996.

[Srinivas] Srinivas, M. and Patnaik, L.: "Adaptive probabilities of crossover and mutation in genetic algorithms," *IEEE Transactions on System, Man and Cybernetics*, vol.24, no.4, pp.656–667, 1994.

[TSPR] Larrañaga, P., Kuijpers, C., Murga, R. et al.: Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. ***in:*** *Artificial Intelligence Review* (1999) 13: 129. doi:10.1023/A:1006529012972 http://www.dca.fee.unicamp.br/~gomide/courses/EA072/artigos/Genetic_Algorithm_TSPR_eview_Larranaga_1999.pdf (15.8.2016)

[UI3] Mařík, Vladimír a kol. Umělá inteligence*. (3).* Vyd. 1. Praha: Academia, 2001. 328 s. ISBN 80-200-0472-6.

[UI4] Mařík, Vladimír a kol. Umělá inteligence*. (4)*. Vyd. 1. Praha: Academia, 2003. 475 s. ISBN 80-200-1044-0.

[Vapnik] V. Vapnik, E. Levin, Y. L. Cun: Measuring the VC-dimension of a learning machine. *Neural computation*, 6 (5), pp. 851-876, 1994.

[Vazirani] Vazirani, U.,Dasgupta, S.,Papadimitriou, C.: Algorithms. McGraw-Hill Education, 2006, ISBN 9780073523408.

[Venkatachalam] Venkatachalam, Thamizh Arasan, and Gnanavelu, Dhivya : Concentration of Heterogeneous Road Traffic. Department of Civil Engineering Indian Institute of Technology Madras,, India.
Dostupné on-line: http://www.intechopen.com/books/advanced-technologies/concentration-of-heterogeneous-road-traffic# (3.1.2017).

[Venugopal] Venugopal: Mastering C. Tata McGraw-Hill Education, Jul 1, 2006.

[Weisser] Weisser R., Ošmera P., Matoušek R.: Transplant Evolution with Modified Schema of Differential Evolution : Optimization Structure of Controllers. In *International Conference on Soft Computing MENDEL*. Brno : MENDEL, 2010.

[Zelinka1] Zelinka I.: Analytic Programming by Means of Soma Algorithm. In: *Proc. 8th International Conference on Soft Computing Mendel'02*, Brno, Czech Republic, 2002, 93-101., ISBN 80-214-2135-5

[Zelinka2] Zelinka I.: Analytic Programming by Means of Soma Algorithm. In: *ICICIS'02, First International Conference on Intelligent Computing and Information Systems*, Egypt, Cairo, 2002, ISBN 977-237-172-3

[Zongker] Zongker, D., Punch, B., Rand, B.: Lil-gp Genetic Programming System. Michigan State University, 1996.

Přílohy

Obsah přiloženého CD

/pdf	text práce v pdf, publikace ze Smartcities 2016, publikace připravená pro časopis Neural Network World, publikace připravená pro pravidelný seminář Ústavu přístrojové a řídicí techniky FS ČVUT, teze k dizertaci, prezentace
/gpy	aplikace, popisovaná v práci, včetně zdrojových textů
/data	ukázky zpracovávaných dat. Více v adresáři /grafy .
/gp	ukázkové programy pro výuku včetně zdrojových textů
/grafy	zdrojová data a grafy, použité v této práci (Excel)