TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky a mezioborových inženýrských studií

DIPLOMOVÁ PRÁCE

Optimalizace parametrů systému komprese diagnostických dat COMPAS

Diagnostic Data Compression System COMPAS – Optimization of Parameters

Liberec 2005 Jan Jaderný

Zadání

Poděkování

Rád bych touto cestou poděkoval vedoucímu diplomové práce Prof. Ing. Ondřeji Novákovi, CSc. za podnětné rady, odborné připomínky a zodpovědné vedení. Dále bych chtěl poděkovat týmu projektu REASON na naší univerzitě za umožnění vypracování diplomové práce, ochotu a přístup při její realizaci, zejména pak Ing. Jiřímu Jeníčkovi.

Velký dík také také patří za poskytnutí výpočetních prostředků Ing. Petru Tomkovi z Katedry modelování na naší univerzitě a Davidu Spudichovi.

Další poděkování patří paní Evě Laurinové, Ing. Holadovi a Pavlu Knoblochovi.

Chtěl bych také poděkovat celé své rodině za všestranou podporu při studiu i při tvorbě této diplomové práce.

Tuto práci bych chtěl věnovat památce mého otce, jehož život pro mě vždy byl a bude velkým příkladem a inspirací.

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce.

Datum

Podpis

Anotace

Diplomová práce ze zabývá problematikou komprese testovacích vektorů. Cílem diplomové práce bylo prozkoumat možnosti změn hodnot parametrů, které určují pořadí kódování testovacích vektorů systémem COMPAS, a také navrhnout kritérium určující toto pořadí tak, aby výsledné testovací sekvence byly pro různé obvody co nejkratší. Práce je založena na software COMPAS – kompresoru testovacích vektorů.

Bylo zkoumáno současně používané kritérium a několik nově navržených, nejlepší z nich bylo doporučeno pro budoucí použití v softwaru COMPAS. Pro tyto účely byl také vytvořen program POET, který implementuje genetické algoritmy použité pro optimalizaci parametrů těchto kritérií.

Efektivita doporučeného kritéria byla ověřena pomocí benchmarkových obvodů ISCAS85 a ISCAS89.

Abstract

The thesis deals with methods and objectives of test pattern compression. The aim was to explore possibilities of changing parameters' values which determine the ordering of test patterns being coded by system COMPAS and also to propose the optimal formula which would lead to the shortest test sequences generated for different circuits. The work is based on software COMPAS – test pattern compressor.

The formula currently in use and several new were tested, the best formula was proposed to future use in software COMPAS. The program POET was created for this purpose, it implements genetic algorithm used for these optimizations of formulas' parameters.

Efficiency of the proposed formula was verified using ISCAS85 and ISCAS89 benchmark circuits.

Obsah

1 Úvod **11**

I Úvod do problematiky **13**

2 Testování číslicových obvodů **13**

2.1 Základní pojmy 13

2.2 Kombinační a sekvenční obvody 14

2.3 Metody aplikace testu 14

2.4 Problémy testů 15

2.5 Vestavěné diagnostické prostředky 16

2.6 Komprese a kompakce 17

3 Software COMPAS **19**

3.1 Popis metody komprese 19

3.2 Schéma testu 20

3.3 Srovnání s jinými metodami komprese 21

3.4 Verze softwaru COMPAS 23

3.5 Určení hodnoty bitu posloupnosti v jednom kroku 24

3.6 Kritérium ohodnocení 26

3.6.1 Původní kritérium 27

3.6.2 Příklad výpočtu kritéria 27

3.6.3 Další kritéria 29

3.7 Použití softwaru COMPAS 29

4 Genetické algoritmy **33**

4.1 Charakteristika 33

4.2 Základní podoba algoritmu 34

4.2.1 Zakódování jedince 35
4.2.2 Ohodnocení jedince 35
4.2.3 Vytvoření nového jedince 36
4.2.4 Selekce 36
4.2.5 Náhradová strategie 38
4.2.6 Zastavovací pravidlo 39
4.2.7 Parametry algoritmu 39
4.3 Další rozšíření 40

II Řešení **42**

5 Cíl diplomové práce **42**

6 Program POET **43**
6.1 Základní úvaha optimalizace 43
6.2 Popis programu 44
6.3 Účelová funkce 45
6.4 Ukončovací pravidlo 47
6.5 Použití programu 48
6.6 Konfigurační soubor 48

7 Srovnání verzí softwaru COMPAS **48**

8 Jednoparametrová kritéria **53**
8.1 Kritérium 1 53
8.2 Kritérium 1.10 55
8.3 Kritérium 1, ustálená oblast 61
8.3.1 Ověření ustálené oblasti 61
8.3.2 Ustálená oblast versus parametry obvodu 63

8.4 Kritérium 1.R 65

9 Kritérium bez parametrů **69**

9.1 Kritérium 0 69

9.2 Srovnání s jednoparametrovými kritérii 69

10 Dvouparametrová kritéria **74**

10.1 Kritérium 2.C 76

10.2 Kritérium 2.B 85

10.3 Zhodnocení kritérií 2.C a 2.B 88

11 Výběr kritéria **89**

12 Použité prostředky **90**

12.1 Programové prostředky 90

12.1.1 COMPAS, Hope, Atalanta 90

12.1.2 GAlib 90

12.1.3 g++, Matlab, Open Office 90

12.2 Hardwarové prostředky 91

13 Závěr **92**

III Přílohy **98**

A Přehled všech kritérií **98**

A.1 Původní kritérium 98

A.2 Kritérium 1.10 98

A.3 Kritérium 1.R 98

A.4 Kritérium 0 98

A.5 Kritérium 2.B 99

A.6 Kritérium 2.B 99

B Doby běhu SW COMPAS **100**

C Implementace účelové funkce **101**

D Přehled parametrů programu POET **104**

E Obsah přiloženého CD **105**

E.1 Adresář Diplomová práce 105

E.2 Adresář Zdrojové kody 105

E.3 Adresář Program POET 105

Seznam použitých zkratek

ATPG	Automatic Test Pattern Generator
	Automatický generátor testů
BIST	Built-In Self Test
	Vestavěné diagnostické prostředky
CUT	Circuit Under Test
	Testovaný obvod
FA	Future Array
	Pole budoucích bitů
GA	Genetic Algorithm
	Genetický algoritmus
LFSR	Linear Feedback Shift Register
	Lineární zpětnovazební registr
OS	Operační systém
SA	Signature Analyzer
	Příznakový analyzátor
SC	Scan Chain
	Skanovací, testovací smyčka (řetězec)
SW	Software
TAM	Test Access Mechanism
	Testovací mechanismus
TPC	Test-per-clock
TPL	Test Pattern List
	Seznam testovacích vektorů
TPS	Test-per-scan
UFL	Undetected Fault List
	Seznam nedetekovaných poruch

1 Úvod

Na katedře KES (Katedra elektroniky a zpracování signálů) Technické univerzity v Liberci je v rámci rozsáhlého projektu REASON[1] dlouhodobě zkoumán problém vestavěných generátorů testovacích vektorů ATPG.

Tato práce se zabývá možností změn hodnot parametrů určujících pořadí kódování testovacích vektorů systému COMPAS tak, aby výsledné testovací sekvence pro různé obvody byly co nejkratší. Dále zkoumá možnosti změny struktury samotného kritéria, které na základě hodnot parametrů toto pořadí určuje.

Práce staví na již funkčním sotwaru COMPAS (COmpressed test PAttern Sekvencer), stále vyvíjeném členy týmu REASON působícího na Technické univerzitě v Liberci. Tento software úzce spolupracuje se simulátorem poruch HOPE. Vypracování řešení si vyžádalo seznámení s výsledky práce týmu zabývajího se vývojem softwaru COMPAS.

Přínosem práce je jednak hlubší analýza problematiky volby parametrů a struktury kritéria určujícího pořadí kódování testovacích vektorů a dále také návrh konkrétního kritéria a jeho případných parametrů pro použití v softwaru COMPAS.

Prvních tři kapitoly se zabývají úvodem do problematiky testování číslicových obvodů, popisem softwaru COMPAS a úvodem do genetických algoritmů.

Kapitola čtvrtá popisuje program POET (Parametric Optimization using Evolutionary Techniques), který byl vytvořen pro účely optimalizace dvouparametrových kritérií pomocí genetických algoritmů.

Následující čtyři kapitoly se věnují návrhu a zkoumání jednotlivých kritérií a problematice volby jejich parametrů.

V další kapitole je doporučeno konkrétní kritérium pro použití v softwaru COMPAS.

Výsledky této práce budou využity v současných a budoucích verzích tohoto

[1] http://www.fm.vslib.cz/~reason

softwaru. Ty jsou vyvíjeny dalšími členy týmu, v současné době je Ing. Jiřím Zahrádkou prováděn výzkum metod interní reprezentace testovacích vektorů, komprimačních algoritmů a jejich vlivu na rychlost komprese a Ing. Jiří Jeníček se zabývá integrací simulátoru poruch do softwaru COMPAS.

Část I

Úvod do problematiky

2 Testování číslicových obvodů

2.1 Základní pojmy

V dnešní době zákazník, který kupuje jakýkoliv produkt, chce mít jistotu, že tento produkt fuguje tak, jak má. Výrobce tedy musí zajistit, aby jeho produkt byl bezchybný. Nejinak je tomu i v případě číslicových obvodů. Každý vyrobený obvod (testovaný obvod, Circuit Under Test, CUT), musí projít **diagnostickým testem**, který prokáže, že obvod je v pořádku, pracuje správně. Jedna z možností, jak test provést, je, že v jednom **kroku testu** jsou na **primární**[2] **vstupy** číslicového obvodu přivedeny hodnoty tvořící **vstupní vektor**, odezva obvodu na jeho **primárních výstupech** potom tvoří **výstupní vektor**, ten přitom musíme znát předem, abychom mohli říci, zda tento krok testu proběhl v pořádku (Takto lze provést test kombinačního obvodu, pro sekvenční obvody viz. kapitoly 2.2 a 2.3). Vhodně zvolená skupina vstupních testovacích vektorů (a jim odpovídajících vektorů výstupních) potom tvoří **test** obvodu.

Nejjednodušší by bylo vytvořit test, který prověří všechny funkce obvodu, tzv. **triviální test** (v případě kombinačního číslicového obvodu testovací vektory obsahující všechny možné kombinace vstupů), ten by ale v případě složitých obvodů byl příliš časově náročný a také objem dat (vstupní a výstupní testovací vektory) potřebný k protestování obvodu by byl neúnosný.

Lepší způsob je vytvořit test, který prověří všechny potenciální **poruchy** obvodu, které by mohly mít za následek **chybu** obvodu. Z toho už například plyne, že poruchu, která se neprojeví chybou, nebude možné zjistit. Je třeba tedy nalézt takovou

[2]primární jsou ty vstupy či výstupy, které jsou vyvedené z pouzdra obvodu

sadu testovacích vektorů, pomocí níž dosáhneme maximální **pokrytí poruch**; pokud pokrytí bude 100%, dokázali jsme vytvořit **úplný test**. Vzhledem k tomu, že jeden testovací vektor může otestovat více poruch, bude mít tento test rozhodně menší počet testovacích vektorů než test triviální. [5]

2.2 Kombinační a sekvenční obvody

Číslicové obvody lze podle chování klasifikovat na obvody **kombinační** a obvody **sekvenční**.

Obvody, u nichž je odezva v určitém časovém okamžiku podmíněna pouze hodnotami, které jsou v uvažovaném okamžiku na vstupech tohoto obvodu, se nazývají obvody kombinačními.

Sekvenční obvod je takový, který kromě kombinační logiky obsahuje také vnitřní paměť (tvořenou zpravidla klopnými obvody). Znamená to tedy, že hodnoty na výstupech tohoto obvodu v určitém časovém okamžiku nezávisí pouze na hodnotách na vstupech, ale také na hodnotách paměťových prvků, neboli jinak řečeno na **vnitřním stavu** tohoto obvodu. To velmi komplikuje testování, neboť projev poruchy se může na výstupu opozdit o celou řadu taktů.

Proto se při testování sekvenčních obvodů používá diagnostický sériový přístup, který zahrnuje i **skanovací smyčku – skanovací řetězec, testovací smyčka** (Scan Chain, SC), pomocí níž je v diagnostickém režimu umožněn zápis testovacích dat do všech klopných obvodů testovaného obvodu. Tím dosáhneme toho, že obvod se v diagnostickém režimu chová jako obvod kombinační, a je možné jej tedy otestovat.

2.3 Metody aplikace testu

Při použití skanovací smyčky dále záleží na způsobu aplikace testu. Metody aplikace testu mohou být rozděleny buď na **test-per-scan** (TPS), nebo **test-per-clock** (TPC).

Při použití metod TPS jsou testovací vektory z paměti testeru (Automatic Test

Equipment, ATE) postupně načteny do skanovací smyčky testovaného obvodu, dále je proveden funkční cyklus (po každém taktu hodin je načten jeden bit), při kterém je testovací vektor přiveden na primární vstupy obvodu. Odezva obvodu se uloží do klopných obvodů a pak je za pomoci skanovací smyčky z obvodu vyčtena. Tento přístup vyžaduje provedení stejného počtu taktů hodin pro vložení každého testovacího vektoru do testovaného obvodu, jako je dlouhá skanovací smyčka, a obvykle stejný počet taktů hodin pro přečtení odezvy testovaného obvodu. To znamená, že čas potřebný k protestování velmi roste, zvlášť když je skanovací smyčka dlouhá.

Při použití metod TPC je jeden testovací vektor vložen do testovaného obvodu po jednom taktu hodin. Tyto metody vyžadují další logiku, která zaručí, že všechny odezvy testovaného obvodu jsou během testu zachyceny. [10]

2.4 Problémy testů

Díky narůstající komplexnosti integrovaných obvodů, lepší výrobní technologii a většímu počtu jader na jeden čip rostou i nároky na testování, zvětšují se objemy testovacích dat a prodlužují se časy potřebné k otestování obvodu. Proto je potřeba hledat nové techniky pro snížení objemu testovacích dat a zkrácení času potřebného k vykonání testu. Konkrétně se jedná o tyto problémy:

Omezená velikost paměti testeru: Paměť testeru musí být velmi rychlá, a proto je i drahá. Čím více máme testovacích vektorů, tím se zvyšují nároky na paměť. Při použití testeru, který nemá dostatečně velkou paměť, však dojde k tomu, že provedený test bude neúplný (neodhalí všechny poruchy) a tím nemůžeme mít jistotu, že testovaný obvod je v pořádku.

Čas přesunu dat do testeru: Čím je objem dat potřebných k protestování obvodu větší, tím déle trvá přesun těchto dat do testeru (řádově i minuty až hodiny). Je zřejmé, že pokud se podaří tento čas výrazně zkrátit, sníží se i cena celého testu.

Úzký testovací mechanismus: I když máme tester s dostatečně velkou pamětí pro uložení testovacích dat, čas přenosu těchto dat do testovaného obvodu je stále

dlouhý kvůli úzkému testovacímu mechanismu (Test Access Mechanism, TAM). Také tento čas je možné zkrátit zmenšením objemu testovacích dat kompresí. [9], [11]

2.5 Vestavěné diagnostické prostředky

Pro řešení zmíněných problémů externího testování lze použít **BIST** (Built-In self Test) neboli vestavěné diagnostické prostředky. Ty jsou používány pro testování pamětí (díky jejich pravidelné struktuře), pro testování číslicových obvodů jsou však použitelné jen s obtížemi. Jsou založeny na tom, že test generujeme v reálném čase, tedy v době testování, a to buď nedeterministicky (pseudonáhodně), nebo pomocí předpisu (algoritmu), podle něhož bude test generován (v případě pamětí se používá deterministický BIST).

Při použití nedeterministického generování testovacích vektorů vzniká problém, že některé poruchy jsou tímto způsobem obtížně detekovatelné (poruchy obtížně detekovatelné náhodným testem), pokrytí poruch je tudíž nedostatečné. Proto je potřeba navíc použít tester s deterministickými testovacími vektory. Jako nejvhodnější pseudonáhodné generátory testovací posloupnosti se používají **lineární zpětnovazebné posuvné registry** (Linear Feedback Shift Register, LFSR).

Další možností je použít techniky založené na reinicializaci zpětnovazebních posuvných registrů (Linear Feedback Shift Register Reseeding), ty předpokládají, že velká část bitů v testovacích vektorech je neurčitá (tzv. **don't care** bity, to jsou bity testovacího vektoru, na kterých nezávisí odhalení konkrétní poruchy). Vestavěný LFSR je potom inicializován tak, aby jím generovaná výsledná bitová sekvence souhlasila s deterministickými vektory na určených pozicích. Počet bitů uložených v paměti testeru je pak relativně malý, ale počet cyklů hodin potřebných k protestování může být velký. [11]

2.6 Komprese a kompakce

Vzhledem k tomu, že generování testů pomocí **automatických generátorů testů** (Automatic Test Pattern Generator, ATPG) stále dokáže držet krok se vzrůstající složitostí obvodů, testování pomocí deterministických testovacích vektorů je stále aktuální. Šetří testovací čas a také v obvodu vestavěný hardware je není příliš složitý. Problémem však je objem testu dat, který se složitostí obvodů roste (viz kapitolu 2.4). Proto pro zmenšení objemu dat, který je potřeba přenést z ATE pomocí TAM, se používají **kompakce** nebo **komprese** testovacích vektorů.

Při kompakci testovacích vektorů se využívá toho, že původní testovací vektor detekující jednu nebo více poruch obsahuje don't care bity. Sloučíme-li několik takových vektorů do jednoho, výsledný vektor potom neobsahuje don't care bity, ale detekuje všechny poruchy původních testovacích vektorů, ze kterých byl složen.

Komprese testovacích vektorů je metoda, při níž testovací vektory tvořící test jsou zakódovány do značně menšího objemu dat, která jsou uložena v paměti testeru. Původní test je potom z těchto dat generován dekódovacím zařízením v obvodu. Bylo publikováno mnoho různých metod komprese, není však jednoduché je porovnávat, protože někteří autoři ukazují efektivitu komprese na kompresi vektorů detekujících poruchy obtížně detekovatelné náhodným testem a další autoři na celých deterministických testech generovaných pomocí ATPG. Navíc použitelnost různých kompresních metod navíc není ovlivněna pouze výsledným kompresním poměrem, ale také složitostí dekompresního mechanismu, časem potřebným k testu a výpočetní náročností algoritmu pro kompresi testů. [11]

Jednou z nových metod komprese testovacích vektorů je originální metoda komprese, která je implementována v software COMPAS (viz. kapitola 3).

Pro provedení testu je potřeba mít také k dispozici kompletní odezvy na všechny vstupní vektory, tedy všechny výstupní vektory. Aby však nebylo nutné tato data uchovávat, je možné použít příznakový analyzátor (Signature Analyzer, SA), který prakticky provádí **kompresi odezvy na test**. Je založen na metodě dělení dvou binárních

polynomů, realizované v posuvném registru se zpětnými vazbami. Tento způsob dělení polynomů je známý z teorie kódování, kde se ho používá při generování systematických cyklických kódů. Komprese spočívá v tom, že analyzátor redukuje posloupnost výstupních vektorů testu na komprimovaný tvar, který člověk dokáže snadno porovnat s tvarem předepsaným, který odpovídá obvodu bezporuchovému. [5]

3 Software COMPAS

COMPAS - Compressed Test Pattern Sequencer je software pro kompresi testovacích vektorů pro obvody se sériovým diagnostickým přístupem.

Tento software je stále vyvíjen na KES (Katedra elektroniky a zpracování signálů) pod vedením Prof. Ing. Ondřeje Nováka, CSc. Ve svých pracích se jím v poslední době zabývali Ing. Miroslav Holubec (diplomová práce, viz [6]), Ing. Jiří Jeníček (diplomová práce, viz [9]) a Ing. Jiří Zahrádka (diplomová práce, viz [14]). Další práce viz [10], [11], [15]. SW COMPAS je napsán v jazyce C tak, aby jej bylo možné zkompilovat jak pro platformy Windows, tak i platformy typu UNIX.

3.1 Popis metody komprese

Metoda komprese spočívá v postupném hledání testovací posloupnosti po jednom bitu z předem připravené sady testovacích vektorů – úplného testu. Tato posloupnost při vlastním testu vstupuje do skanovací smyčky, jejíž velikost je stejná jako počet vstupů obvodu.

Na začátku algoritmu máme seznam zatím nedetekovaných poruch obvodu (Undetected Fault List, UFL), seznam všech testovacích vektorů tvořících úplný test (Test Pattern List, TPL) a ve skanovací smyčce jsou samé nuly. Krok algoritmu pak probíhá tak, že je nalezen další jeden bit posloupnosti, dále je provedena simulace a ze seznamu poruch jsou vyškrtnuty ty, které jsou simulací detekovány. Dále jsou z TPL vyškrtnuty všechny vektory, které detekují pouze poruchy už detekované. Algoritmus končí tehdy, když je TPL prázdný.

Klíčovým je rozhodovací mechanismus (viz kapitolu 3.5), který určí hodnotu dalšího bitu posloupnosti, aby ta byla co nejkratší za stejného pokrytí všech poruch.

Dekomprese testu potom probíhá tak, že posloupnost vstupuje do skanovací smyčky po jednom bitu, při posunutí posloupnosti o jeden bit je přitom vygenerován celý testovací vektor, čímž dochází ke značné úspoře paměti testeru. Při kompresi

se nevyužívá pouze kompakce testovacích vektorů, ale zároveň jejich posuv, což značně zvyšuje stupeň komprese.

Co se týče seznamu testovacích vektorů tvořících úplný test, při jeho generování některým z ATPG je na něj kladen požadavek, aby testovací vektory z vygenerovaného testu obsahovaly co nejvíce don't care bitů a aby byl pokud možno k jedné poruše vegenerován jeden testovací vektor. Některé ATPG však takové volby vůbec neumožňují.

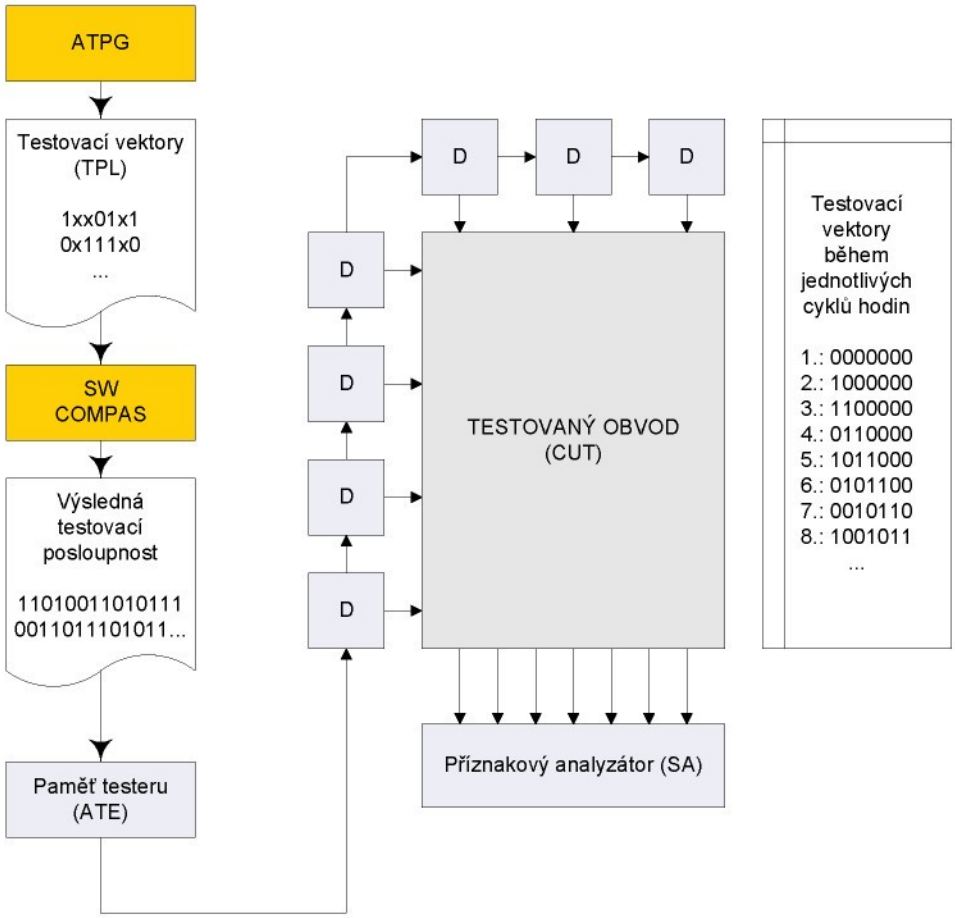
Rozdílné proti metodám běžně používaným např. při reinicializaci LSFR je i použití testovací smyčky složené pouze z D klopných obvodů oproti lineárně zpětnovazebním registrům, dekompresní hardware je tedy o dost jednodušší. Dalším velkým rozdílem je to, že po každém kroku, kdy do testovací smyčky vstoupí další bit, provádíme simulaci poruch a jejich pokrytí, což významně zjednodušuje generování testu a zkracuje délku výsledné testovací posloupnosti. Jistým omezením je ale časová náročnost simulace poruch a nutná znalost struktury obvodu.

Výhodou této metody jsou především vysoká komprese použitých testovacích vektorů, rychlé a snadné testování a velmi jednoduchý testovacím hardware, obzvláště při testování kombinačních obvodů. Při použití se sekvenčními obvody je nutné použít složitější dekompresní hardware, jedním z řešení je např. RESPIN architektura. [9], [11]

3.2 Schéma testu

Na obrázku č. 1 je schéma, jak probíhá test. První část je pouze softwarová, generování testovací posloupnosti probíhá programově. Pomocí automatického generátoru testovacích vektorů (ATPG) vygenerujeme jednotlivé testovací vektory, které jsou vstupem do kompresního programu COMPAS. Pro každou poruchu by měl být vygenerován jeden testovací vektor a je vhodné, aby v testovacím vektoru byl maximální počet neurčitých bitů (don't care bitů), které umožní překrytí s dalšími vektory. Výstupem programu je již zkomprimovaná bitová testovací posloupnost, která se uloží do paměti testeru.

Vytvořená bitová posloupnost uložená v paměti testeru vstupuje sériovým vstupem do skanovací smyčky obvodu, která je tvořena řetězcem D-klopných obvodů. S kaž-



Obrázek 1: Schéma testu

dým taktem hodinového signálu dojde k nasunutí jednoho nového bitu, a tím k vygenerování celého testovacího vektoru. Výsledné testovací vektory jsou přímo zpracovávány testovaným obvodem, odezvy tohoto obvodu pak vstupují do příznakového analyzátoru. [9], [11]

3.3 Srovnání s jinými metodami komprese

V následující tabulce jsou srovnány výsledky kompakce a komprese testovacích vektorů pomocí jiných metod. Hodnoty jsou uvedeny pro obvody z testovací sady ISCAS89, především ty s velkým počtem vnitřních hradel, a tedy s velkými nároky na generování

testu.

V tabulce č. 1 jsou uvedeny výsledné počty bitů pro některé známé metody komprimace testovacích vektorů a pro software COMPAS ve verzi Stack (popis verzí viz kapitolu 3.4).

obvod	MinTest počet bitů	Stat. Coding of Test Cubes počet bitů	LSFR Reseeding počet bitů	Illinois Scan Arch. počet bitů	FDR Codes počet bitů	Linear Decompressors počet bitů	RESPIN++ počet bitů	COMPAS počet bitů
s13207	163100	52741	11285	109772	30880	19608	26004	3968
s15850	58656	49163	12438	32758	26000	12024	32226	6832
s38417	113125	172216	34767	96269	93466	54207	89132	20329
s38584	161040	128046	29397	96056	77812	28120	63232	6150

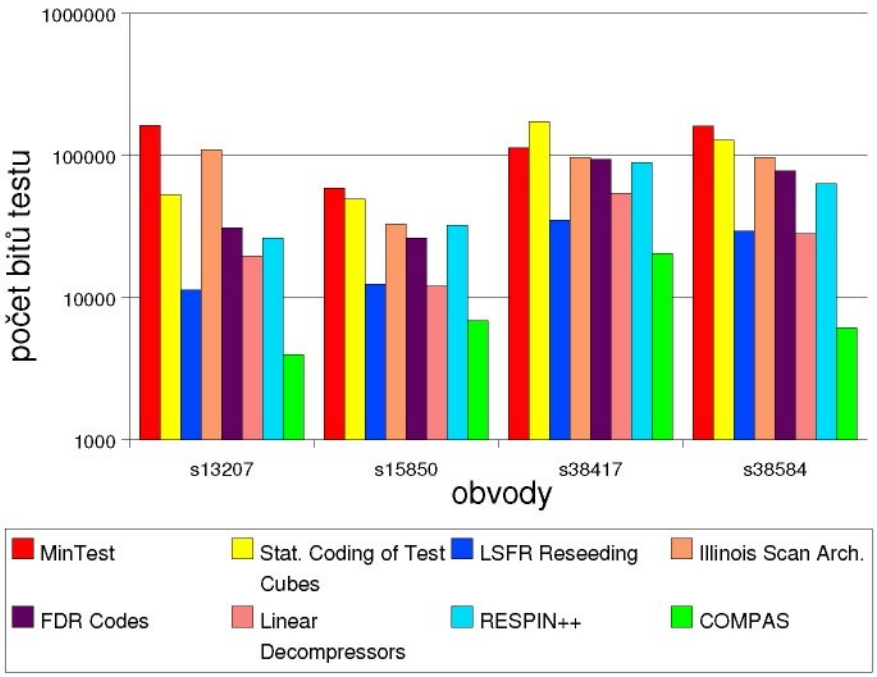
Tabulka 1: Porovnání s jinými metodami komprese testovacích vektorů

Koprese pomocí SW COMPAS dosahuje většího kompresního poměru než ostatní porovnávané metody.

Druhý sloupec této tabulky obsahuje počet bitů pro vektory generované ATGP, které tvoří úplný test a prošly pouze kompakcí. Ve třetím sloupci je velikost testu se statistickým kódováním vektorů z předchozího sloupce. Ve čtvrtém sloupci je použita metoda kombinující statistické kódování z předchozího sloupce a reinicializace LFSR. V následujících sloupcích jsou uvedeny hodnoty odpovídající metodám komprese s paralelními/sériovými testovacími smyčkami, frekvenčně orientovanými kódy a kombinačně lineární dekomprimace. Předposlední sloupec obsahuje počet potřebných bitů vygenerovaných pro RESPIN++ architekturu. Tyto výsledky jsou dosaženy s použitím smíšených testovacích vektorů, přičemž bylo použito 400 pseudonáhodných testovacích vektorů generovaných zpětnovazebními registry RESPIN++ architektury a následně byly aplikovány dekompresované deterministické vektory detekující zbývající poruchy.

Jiné smíšené (mixed-mode) metody obvykle používají větší počet pseudonáhodných vektorů, výsledky proto nejsou porovnatelné.

Z grafu na obrázku č. 2, který vychází z tabulky č. 1, vyplývá, že metody pou-



Obrázek 2: Porovnání různých metod komprese testovacích vektorů

žívající reinicializaci LFSR se vyznačují vysokým kompresním poměrem, a proto jsou vhodné k použití. Ze srovnání je vidět vyšší kompresní poměr COMPASu, dosažený současným použitím kompakce, posunu a okamžitého testování pokrytí poruch. [9], [11]

3.4 Verze softwaru COMPAS

Vzhledem k tomu, že SW COMPAS je stále ve vývoji, měl jsem k dispozici několik jeho verzí, uvádím tedy přehled těchto verzí.

- **základní verze** bez optimalizací
- verze s optimalizací používající predikci budoucích bitů posloupnosti a zásobník testovacích vektorů s nejvyšší využitelností, dále tuto verzi nazývám **verze Stack**, [6]

- **verze Stack.1**, která je malou modifikací verze Stack (šlo o drobnou úpravu, kterou provedl Ing. Jiří Jeníček)
- verze s další optimalizací, kdy se simulátor poruch nespouští v každém kroku algoritmu, dále tuto verzi nazývám **verze 3.2** (kterou v současné době vyvíjí Ing. Jiří Zahrádka, verze je tedy stále ve vývoji)
- **verze 3.2.1**, která je malou modifikací verze 3.2 (stejnou jako v případě verze Stack.1)

Cílem všech těchto optimalizací bylo kompresi testovacích dat zlepšit a současně ji zrychlit.

Všechny tyto verze používají externí simulátor poruch synchronních sekvenčních číslicových obvodů od autorů Hyung K. Lee a Dr. Dong S. Ha z Virginia Polytechnic Institute & State University, [7]

Další verze, které jsem však už neměl k dispozici (protože byly v době mé práce ještě ve stadiu vývoje, nebylo je tedy možné použít), byly tyto:

- víceprocesorová verze využívající možnosti paralelizace algoritmu, [9]
- verze s integrovaným simulátorem poruch (v současné době vyvíjená Ing. Jiřím Jeníčkem)

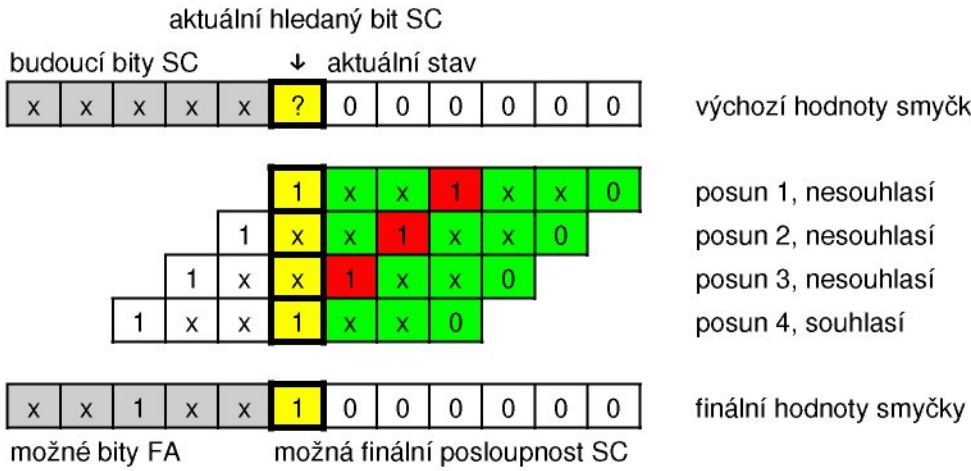
V této diplomové práci jsem pracoval se verzemi Stack, Stack.1, 3.2 a verzí 3.2.1. S verzí základní jsem již nepracoval. Pokud v dalším textu bude zmíněna verze Stack, Stack.1 a pod., vždy se jedná o verze SW COMPAS.

3.5 Určení hodnoty bitu posloupnosti v jednom kroku

Jak již bylo zmíněno v kapitole 3.1, klíčový je mechanismus, který v jednom kroku algoritmu určí následující bit posloupnosti.

Předpokladem je, že na začátku je hardwarový dekompresor testovací posloupnosti nulován, všechny bity skanovací smyčky jsou nastaveny do nuly. První testovací vektor je tedy tvořen samými nulami a není jej potřeba vyhledávat, jinak je ale zpracováván stejně jako ostatní. Je spuštěn simulátor poruch, ze seznamu UFL jsou vyškrtnuty detekované poruchy a ze seznamu TPL jsou vyškrtnuty vektory, příslušející vyškrtnutým poruchám.

Každý další bit testovací posloupnosti je vybírán následujícím postupem: Algoritmus ohodnotí všechny zbývající vektory v TPL tak, že se snaží každý vektor maximálně překrýt dosud vytvořenou testovací posloupností. Každý vektor je posouván tak dlouho, dokud jsou některé jeho bity v kolizi s aktuálním stavem skanovací smyčky (respektive části posloupnosti, která je jakoby právě v této smyčce) a dokud bit, který by měl být vybrán jako další bit posloupnosti není „0“ nebo „1“. Příklad je na obrázku č. 3.



Obrázek 3: Výběr dalšího bitu testovací posloupnosti

Po nalezení platné pozice vektoru je z posunutí a dalších případných parametrů pomocí zvoleného **kritéria** (viz. kapitola 3.6) vypočítána **vhodnost nasazení** vektoru do testovací posloupnosti.

Algoritmus pak vybere testovací vektory s nejmenším ohodnocením, tedy vek-

tory nejvhodnější. Poté algoritmus porovná počet vybraných vektorů s „1“ na právě vyhledávané pozici s počtem vektorů, které na této pozici mají hodnotu „0“. Je-li výsledný počet vektorů s hodnotou „1“ větší než počet vektorů s hodnotou „0“, testovací vektory, které mají na tomto místě jedničku, budou preferovány při hledání budoucího bitu.

Dále k takto vybraných vektorů (obvykle $k = 50$) algoritmus zjistí, zda tyto vektory souhlasí s bity, které bude potřeba vygenerovat v budoucích krocích kvůli předchozím už vybraným vektorům. Tyto bity jsou uloženy v poli budoucích bitů spolu s identifikací vektoru, ke kterému patří, a jeho hodnotou kritéria. Jestliže některá pozice v poli budoucích bitů je již nějakou logickou hodnotou obsazena, algoritmus porovná hodnotu kritéria (vhodnost) těchto dvou vektorů a bit na příslušnou pozici tohoto pole je vybrán z vektoru vhodnějšího.

Z vektorů, které souhlasí i s polem budoucích bitů, je pak vybrána opět na základě počtu jedniček a nul hodnota bitu na novou aktuální pozici posloupnosti. Pokud se stane, že není možné vybrat žádný testovací vektor, do testovací posloupnosti vstoupí bit s hodnotou „0“ (původně vstupoval náhodně zvolený bit, ale z důvodu zachování deterministicity algoritmu od toho bylo prozatím upuštěno – při dvou různých spuštěních programu by mohla být po každé vygenerována výrazně odlišná posloupnost). [11]

3.6 Kritérium ohodnocení

Kritérium určuje vhodnost nasazení konkrétního testovacího vektoru do výsledné testovací posloupnosti. Kritérium může nabývat celých hodnot větších než nula a menších než 2^{32} (dáno použitím datového typu `unsigned int`). Čím je hodnota kritéria menší, tím je daný vektor vhodnější k nasazení, a je tedy pravděpodobnější, že bude nasazen a jemu odpovídající bit se v následujícím kroku řešení zvolí.

3.6.1 Původní kritérium

Na začátku bylo zvoleno kritérium s jednou váhovou konstanou C, které mělo teoreticky vhodně popisovat parametry testovaného obvodu, mělo by prosazovat vektory s největším překrytím s aktuálním stavem testovací smyčky. Také by mělo preferovat vektory s velkým počtem určitých bitů, protože takové vektory se obecně podaří obtížněji překrýt s testovací posloupností do té doby vytvořenou. Je též pravděpodobné, že při nasazení vektorů s větším počtem určitých bitů budou pokryty i vektory s velkým počtem neurčitých bitů.

Vlastní kritérium tedy vypadá takto:

$$crit = (shift + no\_dontcare) \cdot C + global\_dontcares,$$

kde jednotlivé složky znamenají:

crit	hodnota kritéria vypočtená k příslušnému vekotru
shift	posunutí vektoru vůči stavu, kdy se úplně překrývá s SC
no_dontcare	počet neurčitých bitů v překryté části
global_dontcares	celkový počet neurčitých bitů ve vektoru

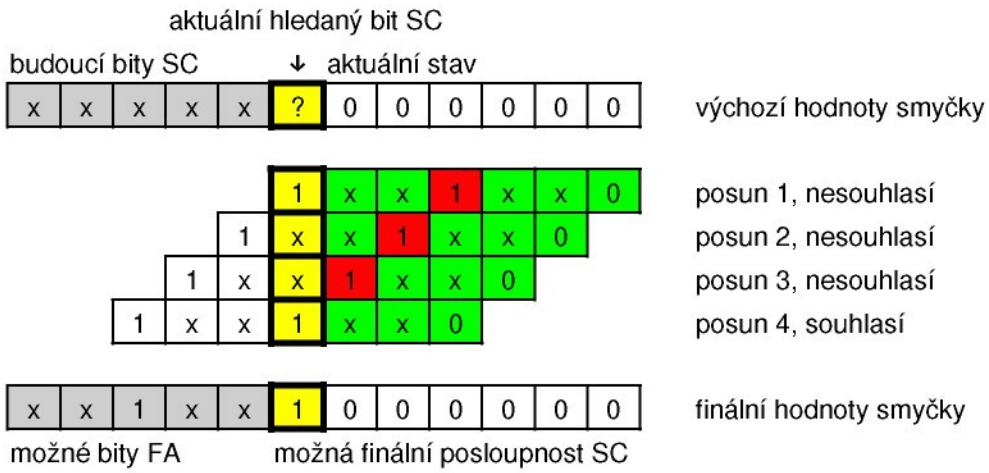
Všechny tyto složky jsou celočíselné, včetně váhové konstanty C. Už od počátku bylo kritérium navrženo jako celočíselné, a to proto, aby v každém kroku bylo vektorů s nejmenším ohodnocením více než jeden, a hodnota dalšího bitu posloupnosti tak byla vybrána na základě více testovacích vektorů (což zaručovalo, že tato hodnota zakóduje největší možný počet testovacích vektorů). [9], [11]

Volba váhové konstanty C má přitom výrazný vliv na výslednou délku testovací posloupnosti. Doposud však nebylo zřejmé, jak velikost konstanty volit nebo zda by nebylo vhodné zvolit kritérium odlišné.

3.6.2 Příklad výpočtu kritéria

Zde je uveden příklad, jak se vypočítává hodnota kritéria. Podíváme-li se znovu na obrázek, na kterém je výběr nového bitu posloupnosti (pro zopakování na obrázku č.

4), vidíme, že právě ohodnocovaný testovací vektor, který má tvar `1xx1xx0`, je možné



Obrázek 4: Výběr dalšího bitu testovací posloupnosti – výpočet hodnoty kritéria

do výsledné posloupnosti nasadit, pokud ho posuneme o 4 bity vzhledem ke konci stávající testovací posloupnosti. Proměnná *shift* má tedy hodnotu 4. Dále vidíme, že tento vektor má v části, která je překrytá s testovací smyčkou, dva neurčité bity (don't care bity), tedy proměnná *no_dontcare* má hodnotu 2. Nakonec spočítáme celkový počet těchto neurčitých bitů v celém vektoru, ty jsou čtyři, proměnná *global_dontcares* má tedy hodnotu 4. Výslednou hodnotu kritéria určující vhodnost nasazení tohoto vektoru vypočítáme dosazením do rovnice tohoto kritéria (uvažujeme přitom hodnotu konstanty například $C = 2$):

$$crit = (shift + no\_dontcare) \cdot C + global\_dontcares = (4 + 2) \cdot 2 + 4 = 16$$

Na základě této hodnoty je potom rozhodnuto, zda tento vektor bude nasazen do výsledné posloupnosti.

Nejdříve ohodnotíme zbylé vektory v TPL, mají-li ostatní hodnotu kritéria větší než 16, pak je tento vektor určen jako nejvhodnější a bude do výsledné posloupnosti nasazen.

3.6.3 Další kritéria

Problém volby kritéria byl již částečně řešen v ročníkovém projektu [8]. Bylo testováno několik dalších podob kritérií (ne vždy zcela vhodně navržených), zejména kritéria s více volitelnými konstantami (parametry). Cílem bylo s pomocí genetických algoritmů pro jeden zvolený obvod nalézt optimální hodnoty parametrů těchto kritérií. Ukázalo se však, že takto nalezené hodnoty parametrů nelze použít pro jiný obvod se stejně dobrým výsledkem jako pro obvod vybraný, na kterém probíhala optimalizace. To je způsobeno rozdílnou vnitřní strukturou jednotlivých obvodů, pro každý obvod byly optimální hodnoty parametrů jiné. Posléze bylo upuštěno od hledání optimálních hodnot parametrů pro kritéria s více jak dvěma volitelnými konstantami (kvůli nemožnosti grafického zobrazení).

3.7 Použití softwaru COMPAS

Zde je z pohledu uživatele uvedeno, jak se software COMPAS používá.

Nejdříve je třeba musíme zajistit, aby v adresáři, kde je program COMPAS (reprezentovaný spustitelným souborem `compas.exe` v OS Windows nebo spustitelným souborem `compas` v OS typu UNIX), byl také simulátor poruch Hope, [7] (spustitelný soubor `hope.exe` (Windows) nebo `hope` (UNIX). Dále je potřeba datový soubor se strukturou zvoleného testovaného obvodu (`název_obvodu.bench`) a další datový soubor s nekomprimovanými testovacími vektory, vygenerovaný v programu Atalanta či v jiném ATPG[3].

Potom můžeme program spustit z příkazové řádky, například tímto základním způsobem (první příklad pro OS Windows, druhý pro OS typu UNIX):

```
compas.exe -f -b název_obvodu
./compas -f -b název_obvodu
```

[3]je však důležité, aby soubor měl formát, ve kterém testy generuje program Atalanta

Po parametru `-b` program očekává název obvodu, parametr `-f` znamená, že se program spustí s optimalizací pomocí predikce budoucích bitů a zásobníku vektorů s největším využitím. Pokud bychom tento parametr nepoužili, spustil by se bez těchto optimalizací, tedy prakticky by se spustila základní verze programu.

Po svém spuštění program vypíše:

```
Circuit: c432
---+ COMPAS & HOPE +---
win32 & un*x
Inputs: 36 Patterns: 518 Faults: 524
Crit = ( j + patterns[k].no_dontcare ) * C + patterns[k].global_dontcares
Constant for criterium (C):
```

Na prvním řádku je název obvodu, který uživatel zadal, na čtvrtém řádku potom počet vstupů obvodu, dále počet testovacích vektorů před kompresí a počet poruch obvodu. Na dalším řádku program vypíše použité kritérium (o kritériích a konstantách viz kapitola 3.6) a čeká, až uživatel zadá konstantu C. Tato konstanta byla doposud nejčastěji volena $C = 2,5$ či 50. Po zadání této konstanty program běží dál a provádí kompresi testovacích vektorů do bitové posloupnosti.

Konec výpisu programu vypadá takto:

```
QCircuit: c432 Inputs:   36 Done:  197 (bits) Remain:    2 (p) (   0.012 s)
Circuit: c432 Inputs:   36 Done:  198 (bits) Remain:    2 (p) (   0.012 s)
QCircuit: c432 Inputs:   36 Done:  199 (bits) Remain:    2 (p) (   0.012 s)
QCircuit: c432 Inputs:   36 Done:  200 (bits) Remain:    2 (p) (   0.012 s)
Circuit: c432 Inputs:   36 Done:  201 (bits) Remain:    1 (p) (   0.012 s)
QCircuit: c432 Inputs:   36 Done:  202 (bits) Remain:    1 (p) (   0.012 s)
Circuit: c432 Inputs:   36 Done:  203 (bits) Remain:    1 (p) (   0.012 s)
QCircuit: c432 Inputs:   36 Done:  204 (bits) Remain:    1 (p) (   0.012 s)
Circuit: c432 Inputs:   36 Done:  205 (bits) Remain:    1 (p) (   0.012 s)
Circuit: c432 Inputs:   36 Done:  206 (bits) Remain:    1 (p) (   0.012 s)
Circuit: c432 Inputs:   36 Done:  207 (bits) Remain:    1 (p) (   0.012 s)
QCircuit: c432 Inputs:   36 Done:  208 (bits) Remain:    1 (p) (   0.012 s)
Circuit: c432 Inputs:   36 Done:  209 (bits) Remain:    1 (p) (   0.012 s)
Circuit: c432 Inputs:   36 Done:  210 (bits) Remain:    1 (p) (   0.012 s)
Circuit: c432 Inputs:   36 Done:  211 (bits) Remain:    1 (p) (   0.012 s)
Circuit: c432 Inputs:   36 Done:  212 (bits) Remain:    1 (p) (   0.012 s)
Circuit: c432 Inputs:   36 Done:  213 (bits) Remain:    0 (p) (   0.012 s)
--------------------------------------------------------------------------------
```

```
Circuit: c432
Number of bits: 213
Number of random bits: 1
Relative of random bits: 0.000000
Elapsed time: 2.745076 seconds
Number of used future bits: 175
Elapsed NEAR: 0.010969 seconds
Elapsed HOPE: 2.637283 seconds
Elapsed CROS: 0.014906 seconds
```

Z tohoto výpisu vidíme, že výsledná posloupnost má 213 bitů, nárůst této posloupnosti můžeme sledovat ve sloupci `Done`. Ve sloupci `Remain` potom vidíme, kolik testovacích vektorů zbývá vyškrtnout. Výslednou bitovou posloupnost po skončení programu nalezneme v souboru `název_obvodu.result`, v tomto případě tedy `c432.result`, v podobě seznamu testovacích vektorů (komprimovaných, každý následující testovací vektor je tvořen předchozím vektorem posunutým o jeden bit a dalším bitem posloupnosti) odpovídajícím vstupnímu formátu programu Hope.

Dále už pouze zmíním další parametry, které je možné při spuštění programu použít:

`-c N` – nastavení konstanty C při spuštění programu

`-n` – program do souboru `název_obvodu.no_bits` uloží číslo vyjadřující, jak je dlouhá výstupní bitová posloupnost

`-s` – uloží do souboru `název_souboru.teststream` výslednou bitovou posloupnost ve formě po sobě jdoucích jedniček a nul

`-t` – uloží statistiku časů do souboru `název_obvodu.csv`

`-v N` – stupeň výpisů, $N = 0$ až 6, od nejméně podrobných k podrobnějším

`--ver` – vypíše, o kterou verzi programu se jedná

`--help` – vypíše popis jednotlivých parametrů

Při použití verze 3.2 je třeba místo parametru `-c` použít parametr `-c1`, navíc je zde i parametr `-c2`, který umožňuje zadat druhou konstantu (pro víceparametrová kritéria).

Navíc je potřeba použít parametr `-e`, který zaručí, že se tato verze spustí s použitím optimalizace využívající pouze občasné spouštění simulátoru poruch.

4 Genetické algoritmy

Genetické algoritmy (GA, nebo také evoluční techniky) je souhrnný název optimalizačních technik, pro které byly inspirací evoluční děje v přírodě: princip přežití nejsilnějších jedinců a princip dědičnosti jejich vlastností na další generace.

Poprvé se tato idea objevila v 60. letech 20. století, nedlouho poté v pracích zabývajících se evolučním programováním (Fogel a kolektiv). V roce 1975 svou knihou *Adaptation in Natural and Artificial Systems* dává základ genetickým algoritmům John Holland. Významnou publikaci v roce 1989 vydává David Goldberg, a to knihu *Genetic Algorithms in Search, Optimization and Machine Learning*, ve které se zabývá těmito algoritmy jakožto technikou pro řešení různorodých úloh.

Tyto algoritmy bývají náročné na výpočetní výkon, jejich použití je však dnes už možné díky stále vzrůstající rychlosti počítačů.

4.1 Charakteristika

Přirovnáme-li množinu možných řešení (kandindátů na řešení[4]) daného problému k **populaci jedinců** (jeden jedinec představuje jednoho kandidáta na řešení tohoto problému), pak právě nejsilnější (nejkvalitnější) jedinec představuje řešení nejlepší.

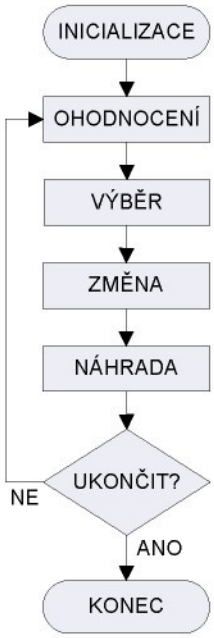
Na začátku algoritmu máme množinu (určité velikosti) náhodně zvolených možných řešení problému – počáteční populaci jedinců (její velikost zůstává v průběhu algoritmu konstantní), ve které se postupem generací vytvářejí lepší a lepší jedinci (v ideálním případě[5]), čímž konvergujeme k nejlepšímu řešení problému.

Průběh algoritmu viz obrázek č. 5.

Inicializace: Na počátku náhodně vytvoříme N jedinců (individual) tvořících populaci (population), kteří budou představovat počáteční generaci (generation).

[4]například řešíme-li v reálném oboru rovnici o jedné neznámé, pak kandidátem na řešení je jakékoli reálné číslo

[5]bohužel však tomu tak vždy není



Obrázek 5: Průběh algoritmu

Ohodnocení: Každého jedince ohodnotíme číslem, které představuje jeho kvalitu.

Výběr: Podle kvality jedinců se budou vybírat ti, kteří „přežijí“, zůstanou tedy v populaci, zbylí méně kvalitní jedinci z procesu evoluce vypadnou, tedy „zahynou“. Tento proces se nazývá **selekce**.

Změna: Některé vybrané jedince pozměníme (respektive vytvoříme jejich kopie a ty pozměníme), dále vytvoříme jedince nové (**potomci**) z jedinců existujících (**rodiče**).

Náhrada: Z jedinců, kteří prošli selekcí, a z nových jedinců se jich N vybere do další generace. Výběr jedinců řídí **náhradová strategie**.

4.2 Základní podoba algoritmu

Úplně na počátku je třeba vyřešit problémy závislé na konkrétní řešené úloze:

- zakódování kandidáta na řešení – jedince
- ohodnocení jedince
- vytvoření (změna) nového jedince z jedinců existujících

Dále musíme zvolit

- způsob selekce
- náhradovou strategii
- podmínku ukončení alogritmu – zastavovací pravidlo
- parametry algoritmu

4.2.1 Zakódování jedince

Jedinec je zakódován do **chromozomu** ve tvaru řetězce znaků

$$(s_1, \ldots, s_n)$$

pevné délky, kde znaky s_i jsou ze zvolené abecedy; nejčastěji je abeceda tvořena pouze ze dvou znaků $\{0, 1\}$.

Máme-li například úlohu, kterou řešíme na intervalu celých čísel $\langle 0, 255\rangle$, zvolíme abecedu $\{0, 1\}$ a délku chromozomu 8 znaků. V tomto případě chromozom bude představovat přímo binární podobu jedince – desítkového čísla (vždy tomu tak být nemusí, záleží na řešeném problému). Konkrétního jedince tedy dostaneme převodem chromozomu z binární podoby do desítkové (dekódování jedince).

4.2.2 Ohodnocení jedince

Dále je nutné umět jedince ohodnotit (určit jeho **kvalitu** – fitness). Musí tedy být definována ohodnocovací funkce (**účelová funkce** – objective function), která řešení dekódované z chromozomu jedince podle nějakého kritéria ohodnotí (číslem, nejčastěji

reálným). Hledáme-li například bod x_0, ve kterém funkce $f(x)$ má maximum, můžeme v nejjednodušším případě považovat za kvalitu i-tého jedince (každému jedinci v populaci odpovídá jeden kandidát na řešení x_i) přímo hodnotu funkce $f(x)$ v bodě x_i (jeho hodnota je zakódována v chromozomu i-tého jedince populace).

Právě složitost výpočtu kvality jedince je rozhodující pro časovou náročnost celého algoritmu, pokud výpočet je komplikovanější, a tedy výpočetně (potažmo časově) náročnější.

Podoba účelové funkce opět záleží na řešené úloze.

4.2.3 Vytvoření nového jedince

Podle dějů v přírodě se používají operace **křížení** (crossover) a **mutace** (mutation).

Křížení je operace, při které se z existujících jedinců (rodičů) vytvářejí jedinci noví (potomci).

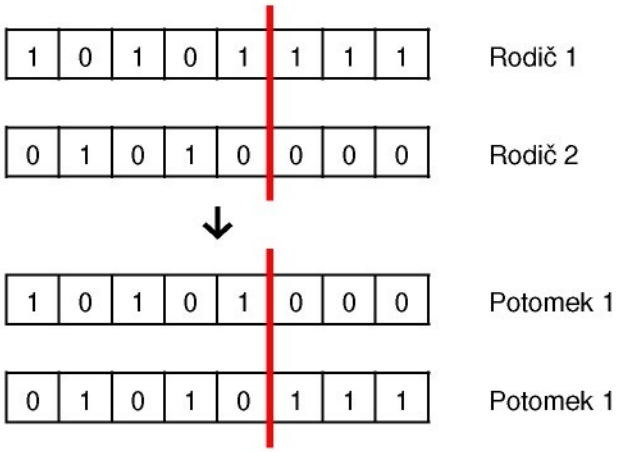
Nejčastěji se používá **binární křížení**, kdy rodiči jsou dva jedinci. Nejobvyklejším způsobem binárního křížení je **jednobodové křížení**: v chromozomech rodičů se provede řez na náhodně zvoleném místě, první potomek pak vznikne spojením první části chromozomu prvního rodiče a druhé části chromozomu druhého rodiče, druhý potomek vznikne spojením první části chromozomu druhého rodiče a druhé části chromozomu prvního rodiče (viz obrázek č. 6).

Mutace je operace, při které se provede malá změna v chromozomu jedince. Při použití binární reprezentace jedince je realizována jako negace náhodně zvoleného bitu v chromozomu (viz obrázek č. 7). Jak je patrné, může tato malá změna úplně změnit vlastnosti toho jedince.

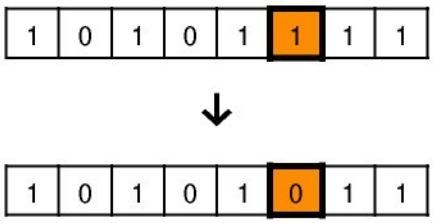
4.2.4 Selekce

Zde se uplatní kvalita jedinců, dále postoupí ti jedinci, kteří jsou kvalitnější. Obvyklé metody jsou tyto:

- selekce ruletovým kolem (**roulette wheel selection**)



Obrázek 6: Křížení



Obrázek 7: Mutace

- turnajová selekce (**tournament selection**)

Selekce ruletovým kolem: Předpokladem je, že kvalitnější jedinec má větší pravděpodobnost na přežití. Každému jedinci je tedy podle jeho kvality přiřazena pravděpodobnost, se kterou bude vybrán do další generace při jednom zatočení ruletovým kolem. Kolem točíme tolikrát, až máme v další generaci potřebný počet jedinců. Ruletové kolo je realizováno jednoduchým algoritmem využívajícím generátor náhodných čísel.

Turnajová selekce: Náhodně se vybere k jedinců (nejčastěji 2) a uspořádá se mezi nimi turnaj. Vyhrává nejkvalitnější jedinec, postoupí tedy dál do další generace. Opakuje se tolikrát, až v ní máme potřebný počet jedinců.

Jak je patrné z těchto dvou metod, někteří jedinci se budou v další populaci vyskytovat ve větším počtu než v generaci předešlé. Bez použití operátorů křížení a mutace by rychle došlo k **homogenizaci populace**, tj. v populaci by bylo čím dál tím více stejných jedinců, což by mělo za následek uvíznutí v lokálním extrému.

4.2.5 Náhradová strategie

Náhradová strategie (replacement strategy) je postup, kterým se vybírají jedinci do další generace z jedinců, kteří postoupili po selekci, a z jedinců vytvořených křížením a mutací. Nevhodně zvolená náhradová strategie může též vést k homogenizaci populace.

Možné náhradové strategie:

- náhodná (random)
- náhrada rodičů (parent)
- náhrada nejhorších jedinců (worst)
- elitismus (elitism)

Náhodná: Noví jedinci (potomci a mutanti) jsou vkládáni do populace výměnou za náhodně vybrané jedince.

Náhrada rodičů: Noví jedinci jsou vkládáni do populace výměnou za jedince, ze kterých byli vytvořeni.

Náhrada nejhorších jedinců: Noví jedinci jsou vkládáni do populace výměnou za nejhorší jedince.

Elitismus: Po vytvoření nových jedinců máme množinu jedinců, která je značně větší než zvolená velikost populace, z této „mezigenerace“ je vybrán jeden či několik nejlepších jedinců, ti jsou doplněni až do požadovaného počtu zpravidla upřednostněním potomků a mutantů.

Náhradové strategie rozlišujeme na (podrobněji popsáno v části Další rozšíření):

- generační – do další generace bude celá populace nahrazena populací následující
- postupné – změny podstoupí jen část populace

4.2.6 Zastavovací pravidlo

Jelikož algoritmus by mohl běžet do nekonečna, je třeba zvolit **zastavovací pravidlo, ukončovací pravidlo** (termination method). Ukončení algoritmu je možno volit různými způsoby:

- při dosažení postačující kvality nejlepšího jedince
- při dosažení určitého počtu generací
- pokud se po několik generací neobjevil lepší jedinec, než je dosud nejlepší

V závislosti na řešeném problému je možné zvolit i jinou podmínku, podle které algoritmus ukončíme. Také kombinace těchto podmínek mohou tvořit zastavovací pravidlo.

4.2.7 Parametry algoritmu

Chování algoritmu zásadně ovlivňují:

- velikost populace N – počet jedinců v populaci
- pravděpodobnost křížení P_x – pravděpodobnost, s jakou bude jedinec vybrán pro křížení
- pravděpodobnost mutace P_m – pravděpodobnost, s jakou bude jedinec vybrán pro mutaci

Velikost populace N je vhodné volit minimálně v řádu desítek až stovek jedinců, záleží na řešeném problému. Pokud je kódování jedince komplikovanejší, pak jich zpravidla musí být v populaci více.

Pravděpodobnost křížení P_x (každého jedince) se obvykle volí v rozsahu $\langle 0.6, 1 \rangle$, znamená to tedy, že pro křížení s $P_x = 0.9$ bude náhodně vybráno z populace zhruba 90% jedinců.

Pravděpodobnost mutace (každého jedince) je vhodné volit v rozsahu $\langle 0.05, 0.15 \rangle$. Při použití hodnoty $P_m = 0.1$ bude tedy vybrána náhodně asi desetina jedinců (zde se mohou implementace algoritmů lišit, výběr jedinců pro mutaci může probíhat z celé populace, nebo pouze z potomků).

Právě mutace působí příznivě proti homogenizaci populace, její pravděpodobnost nesmí být ale ani příliš velká, protože by potom zpomalovala konvergenci k nejlepšímu řešení.

V praxi se tyto hodnoty použijí tak, že se pro křížení náhodně vybere tolik jedinců, kolik přesně odpovídá pravděpodobnosti P_x. Stejným způsobem se vyberou i jedinci, kteří podstoupí mutaci.

Průběh algoritmu je velmi závislý na volbě hodnot těchto parametrů, stejně tak ale na volbě selekční metody a náhradové strategie.

Dále je třeba mít na paměti, že algoritmus bude mít pokaždé jiný průběh v důsledku použití prvku náhody. Je tedy potřeba, aby po změně jednoho parametru bylo provedeno statistické vyhodnocení několika průběhů algoritmu, abychom mohli určit, jaký vliv změna tohoto parametru měla. Často to však díky výpočetní náročnosti nemusí být možné.

4.3 Další rozšíření

Standardní GA využívá jedinou populaci, která po každou generaci celá podstupuje ohodnocení, selekci a změnu. Další možné varianty algoritmu jsou:

- GA s částečnou obměnou populace – **Steady-state GA**
- GA s paralelními populacemi s migrací – **Deme GA**

Steady-state GA: Tento algoritmus využívá překrývající se populace s volitelnou mírou překrytí. Každou generaci algoritmus vytvoří dočasnou populaci jedinců, tuto spojí s původní populací a pak je z výsledné populace vyloučeno tolik nejhorších jedinců, až jejich počet v populaci dosáhne původního počtu. Další parametr oproti standardnímu GA je tedy počet jedinců či procentuální část populace, která je každou generaci obměňována.

Deme GA: Tento algoritmus používá několik nezávislých populací. V každé populaci probíhá evoluce s použitím steady-state algoritmu, ale každou generaci několik jedinců migruje z jedné populace do další. Z každé populace migruje definovaný počet jedinců do populace sousední. Dalšími volitelnými parametry oproti standardnímu GA jsou tedy počet paralelních populací a počet jedinců, kteří budou každou generaci migrovat do vedlejší populace.

Dále jsou možná rozšíření operátorů mutace a křížení. Ty mohou být nadefinovány jinak, než bylo doposud uvedeno, příkladem je **vícebodové křížení**, kde řez je proveden na více než jednom místě. Další modifikací může být křížení používající více než dva rodiče.

Například je též možné použít pro metodu selekce ruletovým kolem transformaci pravděpodobnosti výběru jedince, která může mít při transformaci vhodně zvolené za následek zvýšení selekčního tlaku při výběru jedinců do další generace.

Existuje mnoho dalších možností jak tyto algoritmy modifikovat, co se týče reprezentace jedinců nebo rekombinačních operátorů, jedná se zejména o modifikace vycházející ze znalosti řešeného problému. [1], [12]

Část II

Řešení

V této části práce a částech následujících je uveden popis a výsledky práce zaměřené na kritéria. Na začátku je popis programu pro optimalizaci parametrů užitím genetických algoritmů POET, dále porovnání jednotlivých verzí software COMPAS a pak už je řešena problematika kritérií.

Jednotlivá kritéria nejsou zmiňována chronologicky, jak byla zkoumána, ale systematicky, od původního jednoparametrového přes kritérium bez parametrů až k dvouparametrovým kritériím. Je tomu tak proto, že některá kritéria byla zkoumána paralelně s jinými a k některým se naopak vracelo později. Proto by se mohlo zdát, že jsou některé poznatky uplatňovány „na přeskáčku“.

5 Cíl diplomové práce

Cílem této diplomové práce bylo nalézt takové **kritérium a jeho parametry** určující vhodnost nasazení testovacího vektoru do výsledné bitové posloupnosti, aby při použití tohoto kritéria v SW COMPAS byly výsledné testovací sekvekce pro **různé** obvody co nejkratší, a přitom aby výsledné kritérium s jeho parametry bylo použitelné i pro další neznámé obvody.

Hlavním požadavkem tedy je, aby zvolené kritérium bylo **univerzální**, a tudíž jsme mohli předpokládat, že výsledky s jeho použitím budou **stabilní** i pro jiné obvody než pro obvody trénované.

6 Program POET

Program POET (*Parametric Optimization using Evolutionary Techniques*) jsem vytvořil právě pro účely optimalizace různých dvouparametrových kritérií použitých v SW COMPAS. POET je napsán v jazyce C++ (v první verzi programu byly použity pouze standardní knihovny jazyka C, další verze už využívá knihovnu GAlib, je možné jej zkompilovat pro operační systém Linux (zde používaný kompilátor je `g++`).

6.1 Základní úvaha optimalizace

Vzhledem ke zkušenostem s víceparametrovými kritérii z předchozí práce [8] bylo zřejmé, že je nutné se zaměřit na **skupinu obvodů**, ne se pouze snažit o optimalizaci parametrů kritéria na jednom obvodu. Ukázalo se také, že je obtížné pracovat s kritérii, které mají více jak dva parametry, hlavně z důvodu nemožnosti znázornění výsledků graficky a také proto, že zkonstruovat kritérium s více jak dvěma parametry tak, aby bylo prokazatelně účinné, se ukázalo jako velice obtížné.

Dalším důležitým poznatkem z předchozí práce je fakt, že nestačí nalézt jednu optimální kombinaci dvou konstant C_1, C_2, a to z toho důvodu, že tato kombinace konstant při použití na dalších naprosto odlišných obvodech se zpravidla jako optimální nejeví.

Daleko lepší by mělo být pokusit se najít větší množství více méně optimálních kombinací konstant, které pro zvolenou skupinu obvodů vykazují dobré výsledky. Proto byla pozornost zaměřena na hledání **kandidátů** na optimální kombinace těchto konstant, které je následně potřeba ověřit na co největším počtu dalších obvodů. Přitom je kladen důraz na to, aby těchto kandidátů bylo nalezeno **co nejvíce**.

Genetické algoritmy se v předchozí práci osvědčily, proto bylo jejich použití nasnadě.

6.2 Popis programu

Po spuštění program pracuje jako klasický genetický algoritmus, tzn. po inicializaci (náhodném vygenerování populace) probíhá evoluce až do doby, kdy je splněno ukončovací pravidlo.

Hlavní rozdíl oproti [8] je v tom, že nyní program musí pracovat se **skupinou několika obvodů**, a tedy musí umět ohodnotit kombinaci dvou konstant C_1 a C_2 pro tuto skupinu obvodů. Byl tedy vytvořen konfigurační soubor, ze kterého si program načte názvy obvodů, které jsme pro optimalizaci zvolili, a další potřebné informace.

Jedinec je představován vektorem konstant (C_1, C_2) (jejich počet je dán použitým kritériem, v tomto případě vždy kritériem dvouparametrovým). Jeho ohodnocení probíhá tak, že pro všechny obvody, které byly pro optimalizaci vybrány, je automaticky externě spuštěn SW COMPAS s těmito konstantami předanými příkazovou řádkou. Jeho výstup je následně analyzován a jedinci je přiřazena **kvalita** podle dosažených délek výsledných testovacích posloupností. Tato kvalita jedince je počítána pomocí účelové funkce (viz kapitolu 6.3). Je tedy zřejmé, že pro ohodnocení jednoho jedince se SW COMPAS spustí tolikrát, kolik obvodů je ve vybrané skupině. Aby algoritmus nespouštěl SW COMPAS častěji, než je nutné, byla do programu POET začleněna datová struktura (soustava dynamických polí), která uchovává už jednou vypočtené výsledné délky posloupností.

Výstupem programu je několik souborů: statistika průběhu algoritmu, výpis poslední generace, soubory se všemi kombinacemi konstant (C_1, C_2), které byly během programu prozkoumány a seznam kandidátů na optimální kombinace konstant. Z těchto souborů jsou skripty pro interpretr `bash` a skripty pro program `Matlab` generovány tabulky a grafy.

Pro každé dvouparametrové kritérium bylo potřeba upravit SW COMPAS tak, aby při výpočtu vhodnosti nasazení testovacího vektoru používal toto zvolené kritérium.

Pro implementaci programu byly vybrány tři typy genetických algoritmů:

- standardní jednoduchý algoritmus s jednou populací (Simple GA)
- algoritmus s paralelními populacemi (Deme GA)
- algoritmus s částečnou obměnou populace (Steady-state GA)

Jako metodu selekce byla vybrána selekce turnajová, a to z toho důvodu, aby případná nelinearita v návrhu účelové funkce (hodnota účelové funkce by nebyla závislá na kvalitě jedince lineárně) neměla vliv na prováděnou selekci jedinců.

6.3 Účelová funkce

Zásadní částí genetického algoritmu je účelová funkce. Pomocí této funkce je počítána kvalita jedince, od které se odvíjí celý evoluční proces. Tato funkce obvykle přiřazuje jedinci tím větší hodnotu, čím je jedinec kvalitnější; jak bude však patrné dále, zvolil jsem v tomto případě ohodnocení opačné (čím lepší jedinec, tím menší hodnota účelové funkce). Toto musí být dále zohledněno při selekci jedinců.

Pokud bychom optimalizovali úlohu pouze pro jeden obvod, navrhli bychom účelovou funkci tak, že by hodnota kvality jedince byla přímo rovna délce výsledné testovací posloupnosti pro konkrétního jedince (tedy konkrétní hodnoty parametrů C_1, C_2) a tento obvod, tedy čím kratší délka, tím lepší jedinec. Avšak vzhledem k tomu, že bylo potřeba pracovat s více obvody najednou, bylo nutno navrhnout tuto funkci značně složitěji.

Hlavními problémy při návrhu této funkce byly tyto:

- při použití různých obvodů jsou výstupem programu COMPAS výsledné bitové délky, které mají **řádově** jinou délku (od desítek, přes stovky až k desetitisícům a více) – je tedy třeba tyto délky nějak **normalizovat** napříč obvody
- jako účelovou funkci nelze použít prostý průměr nějakým způsobem normalizovaných hodnot, stírají se přitom maximální a minimální hodnoty (přitom ty maximální jsou pro nás naprosto nežádoucí)

Problém normalizace se povedlo vyřešit po návrhu kritéria 0 (viz kapitolu 9.1). S použitím tohoto kritéria, které nemá žádný parametr, jsme dostali délky výsledných posloupností pro jednotlivé obvody, pomocí kterých je bylo už možné dále normalizovat. Výsledné délky posloupností pro dvouparametrová kritéria se tedy dále poměřují vždy k výsledkům s použitím kritéria bezparametrového. Snažíme se tedy najít takové kritérium s parametry, aby bylo lepší než kritérium 0.

K problému, jakým způsobem zahrnout do jedné hodnoty výsledky pro více obvodů, bylo přistupováno takto: Nejvíce nás zajímá případ, kdy dojde ke zlepšení výsledků pro všechny obvody, v tom případě můžeme použít prostý průměr; pokud však pro některý obvod dostaneme výsledky horší, chceme, aby tyto odchylky směrem k horším výsledkům byly co nejmenší.

Tyto dvě části přitom od sebe oddělíme tak, že hodnotu průměru normalizovaných délek tranformujeme do záporných čísel (pomocí převrácené hodnoty a znaménka minus). Hodnoty z vypočítaných odchylek směrem k horšímu přitom budou růst od nuly směrem do kladných čísel.

Algoritmus výpočtu účelové funkce tedy vypadá takto:

1. Pro všechny obvody vypočteme normalizovanou délku výsledné posloupnosti jako podíl délky výsledné posloupnosti ku délce posloupnosti s použitím kritéria č. 0

2. Pokud jsou všechny normalizované délky menší než jedna nebo rovny jedné, dosáhli jsme zlepšení (přinejhorším však stejných výsledků) proti kritériu č. 0. Hodnotu účelové funkce vypočítáme jako převrácenou hodnotě průměru normalizovaných délek a přidáme znaménko minus. Tím dosáhneme toho, že čím menší bude hodnota účelové funkce, tím lepší výsledky této hodnotě odpovídají.

3. Pokud jedna či více normalizovaných délek bude větší než jedna, znamená to, že pro jeden či více obvodů jsme dostali výslednou délku horší než při použití kritéria č. 0. Hodnotu účelové funkce spočítáme jako sumu kvadrátů odchylek normalizované délky od hodnoty jedna (přitom pro normalizované délky menší

než jedna počítáme odchylku rovnou nule) dělenou počtem obvodů. Tím dosáhneme toho, že čím větší budou odchylky normalizovaných délek od hodnoty jedna směrem nahoru, tím větší bude hodnota účelové funkce.

Označíme-li délky výsledných posloupností s použitím zvoleného kritéria L_1 až L_n, (L_i je délka výsledné posloupnosti pro i-tý obvod z vybrané skupiny n obvodů), délky výsledných posloupností s použitím kritéria č. 0 D_1 až D_n, normalizované délky výsledných posloupností N_1 až N_n a hodnotu účelové funkce F, pak algoritmus vyjádřený pomocí matematických vztahů vypadá takto:

1. $N_i = \frac{L_i}{D_i}$
2. Jestliže platí $N_i \leq 1, \forall i \in \{1, \ldots, n\}$, pak $F = -\frac{\sum N_i}{n}$.
3. Jinak $F = \frac{\sum (1-N_i)^2}{n}$, přičemž platí, že pokud $N_i < 1$, pak položíme $N_i = 1$.

Pro použití v programu už byla pouze provedena malá úpravu této funkce tak, aby hodnota účelové funkce nebyla záporná (kvalita jedince by záporná neměla být), a to tak, že byla k takto vypočítané hodnotě připočteno dostatečně velké číslo; ukázalo se, že stačí použít číslo 10.

Implementaci této funkce v jazyce C++ lze najít v příloze C.

6.4 Ukončovací pravidlo

Další důležitou částí genetického algoritmu je ukončovací pravidlo, při jehož splnění algoritmus skončí (jinak by evoluce mohla probíhat donekonečna).

Jako první se nabízí podmínka, že pokud v posledních x generacích se nepodaří najít nového kandidáta, pak algoritmus ukončíme (je pouze třeba vhodně zvolit parametr x). Jako další se ukázalo, že vzhledem k časové náročnosti SW COMPAS bude nutno omezit počet jeho spuštění. Pokud je zvolený počet spuštění překročen, algoritmus se také ukončí.

6.5 Použití programu

Program POET se spouští z příkazové řádky i s potřebnými parametry. Přehled parametrů uvádím v příloze D. Po skončení programu ten vypíše dosažené výsledky včetně počtu kandidátů, které našel (tentýž výpis je uložen do souboru `poet.statistics`). Nalezení kandidáti jsou pak v souboru `poet.candidates`. Další důležité soubory jsou `poet.no_candidates`, ve kterém jsou počty všech kandidátů po jednotlivých generacích. V souboru `poet.dat` je potom výpis, jak probíhala evoluce, se všemi důležitými ukazateli.

6.6 Konfigurační soubor

Zde je uveden příklad konfiguračního souboru `poet.cfg`, v něm je definována skupina obvodů, na které program POET provádí optimalizaci parametrů.

```
* circuit_name normalizator datafname
c432 199 c432.probe
c880 394 c880.probe
s1196_comb 726 s1196_comb.probe
s1238_comb 780 s1238_comb.probe
```

První řádek je komentář, který vysvětluje formát souboru. V prvním sloupci je název obvodu, ve druhém je hodnota, podle které je provedena normalizace, a třetí sloupec je název souboru, ve kterém se nacházejí kombinace konstant a jim odpovídající délky výsledných posloupností, které algoritmus za svůj běh prošel. Tato data program při opětovném spuštění načte, a nemusí se tedy pro tyto kombinace konstant spouštět SW COMPAS.

7 Srovnání verzí softwaru COMPAS

Protože bylo k dispozici několik verzí software COMPAS (software je stále ve vývoji), bylo třeba je nějakým způsobem porovnat. Pro porovnání byly vybrány tyto verze (více o těchto verzích v kapitole 3.4):

- **verze Stack** – verze s optimalizací používající predikci budoucích bitů posloupnosti a zásobník testovacích vektorů s nejvyšší využitelností
- **verze 3.2** – verze s další optimalizací, kdy se simulátor poruch nespouští v každém kroku algoritmu

Obě tyto verze byly později mírně modifikovány[6] (jedná se o tutéž malou modifikaci pro obě verze), v tomto porovnání ale zahrnuty nejsou. Pokud budu tedy dále zmiňovat modifikace těchto verzí, budu je nazývat verze **Stack.1** a verze **3.2.1**.

Cílem bylo porovnat rychlosti a délky výsledných bitových posloupností verze 3.2 proti verzi Stack na různých operačních systémech.

Testování probíhalo na počítači s procesorem AMD Athlon XP 2500+ (s jádrem Burton, 1.8 GHz) jak v operačním systému Linux (distribuce Mandrake 9.2), tak v operačním systému Windows XP na stejném počítači. Použity byly obvody c432, c880, c2670, c7552, s1196 a s5378 a základní kritérium (podrobnější informace o kritériích a jejich přehled viz kapitolu 3.6):

$$crit = (shift + no\_dontcares) \cdot C + global\_dontcares$$

s použitím konstant $C = 1, 2, 5, 50, 500, 1500$.

Z jednotlivých spuštění pro různé konstanty byl vypočten průměrný čas a průměrná bitová délka, které byly použity pro porovnání jednotlivých verzí. Časy byly měřeny s přesností na jednu sekundu. Výsledné časy jsou uvedeny v tabulce č. 2. Pro menší obvody nemělo smysl při této přesnosti měření časy měřit.

Porovnání výsledných délek bitových posloupností jednotlivých verzí pro některé z konstant najdeme v tabulce č. 3.

Pro větší názornost jsou průměrné časy porovnány procentuálně a vyneseny do grafu. Na obrázku č. 8 vidíme porovnání jednotlivých verzí na stejném operačním systému. Z grafu je patrné, že u verze 3.2 došlo v OS Windows ke značnému zrychlení proti verzi Stack, v OS Linux je však tato verze podstatně pomalejší než verze Stack.

[6] šlo o drobnou úpravu, kterou provedl Jiří Jeníček

obvod	verze 3.2 [min:sek]		verze Stack [min:sek]	
	Linux	Windows	Linux	Windows
c2670	0:37	0:14	0:26	0:41
c7552	17:44	0:58	1:50	2:16
s5378	0:48	0:11	0:28	0:35

Tabulka 2: Porovnání časů, verze 3.2 a Stack

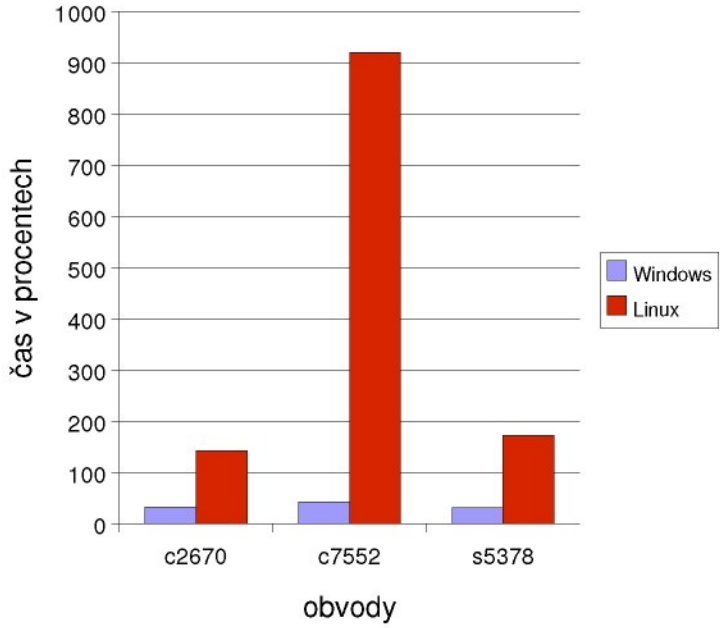
obvod	$C = 2$		$C = 5$	
	verze 3.2	verze Stack	verze 3.2	verze Stack
c432	302	223	687	190
c880	523	401	480	477
s1196	784	736	1107	725
c2670	4291	3743	4265	3759
c7552	5697	5518	5754	7312
s5378	2058	1757	1957	1933

Tabulka 3: Výsledné délky posloupností, verze 3.2 a Stack

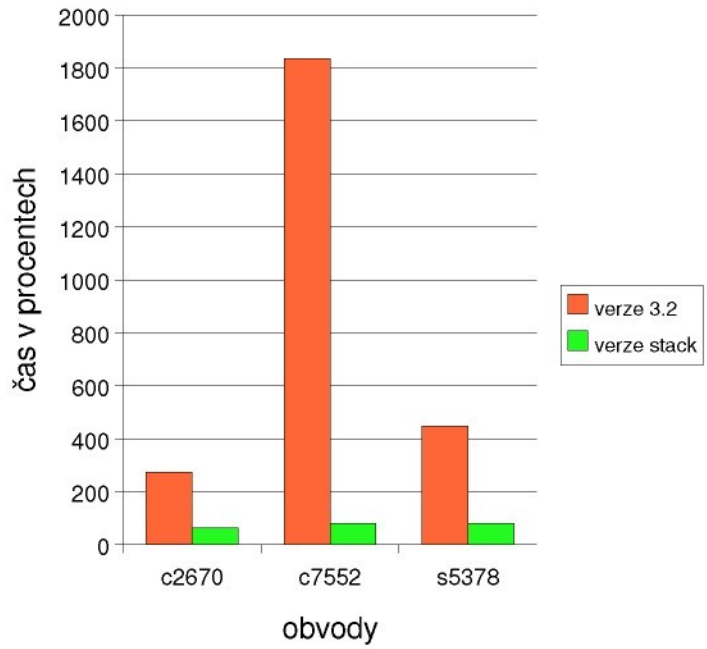
Na obrázku č. 9 vidíme porovnání časů jednotlivých verzí na operačních systémech Linux a Windows. Verze 3.2 je tedy pomalejší v OS Linux než v OS Windows, naopak verze Stack je v OS Linux rychlejší než v OS Windows.

Obrázek č. 10 ukazuje průměrné délky výsledných posloupností, a to opět procentuálně pro verzi 3.2 vzhledem k verzi Stack. Můžeme usoudit, že výsledná posloupnost je při použití verze 3.2 delší než při použití verze Stack.

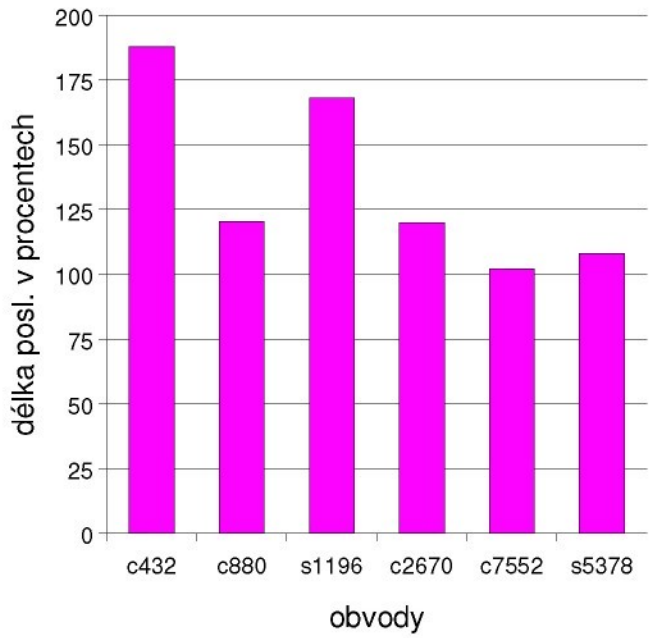
Závěrem tedy můžeme říci, že verze 3.2 je při použití v OS Windows rychlejší než verze Stack, za cenu mírného prodloužení výsledné bitové posloupnosti. Při použití v OS Linux však verze 3.2 rychlejší není, tam je tato verze pomalejší než verze Stack, dokonce sama verze 3.2 běží pomaleji v OS Linux než v OS Windows (u předchozích verzí tomu bylo vždy naopak).



Obrázek 8: Porovnání časů procentuelně, verze 3.2 vzhledem k verzi Stack



Obrázek 9: Porovnání časů procentuelně, Linux vzhledem k Windows



Obrázek 10: Porovnání délek procentuelně, verze 3.2 vzhledem k verzi Stack

Pro použití v OS Linux tedy verzi 3.2 nedoporučuji (nicméně vzhledem k tomu, že tato verze je stále ve vývoji, jistě se podaří tyto nedostatky odstranit). S výsledky a závěrem tohoto srovnání verzí jsem autora verze 3.2 Ing. Jiřího Zahrádku seznámil. Nejčastěji jsem pracoval s verzemi Stack a Stack.1, výsledky s použitím těchto verzí jsem tedy považoval za nejvíce směrodatné (hlavně z důvodu, že verze 3.2 ještě byla ve vývoji).

8 Jednoparametrová kritéria

8.1 Kritérium 1

První zkoumané kritérium, standardně používáné v software COMPAS, má tvar:

$$crit = (shift + no\_dontcares) \cdot C + no\_globaldontcares$$

V dalším textu je toto kritérium nazýváno **kritérium 1**. Konstanta C přitom byla volena víceméně náhodně, nějčastěji například $C = 2$, $C = 5$ nebo $C = 50$. Nebylo však zřejmé, podle čeho ji volit. Proto první, co bylo u tohoto kritéria třeba prozkoumat, byl **průběh závislosti délky výsledné posloupnosti na volbě konstanty** C, a to pro různé obvody. Závislost bude dále označována $L = f(C)$, kde L je délka výsledné posloupnosti.

Bylo tedy třeba zvolit rozsah konstanty C, který bude prozkoumán, dále skupinu obvodů, se nimiž se bude pracovat, a nakonec vybrat verze programu COMPAS, na kterých bude zkoumání provedeno. Rozsah konstanty C byl nejprve zvolen v intervalu 1 až 100 (pro obvod c7552 bylo třeba zvětšit rozsah konstanty C na interval 1 až 150). Dále byla na základě naměřených časů pro jednotlivé výpočty vybrána tato skupinu obvodů (aby výpočet doběhl v rozumném čase): c432, c880, c2670, c7552, s1196, s1238, s5378 a s9234. Obvody začínající písmenem c jsou obvody kombinační a obvody začínající písmenem s jsou obvody sekvenční. Výpočet byl proveden na čtyřech verzích SW COMPAS, a to na těchto: Stack, Stack.1, 3.2 a 3.2.1.

V tabulkách č. 4 a č. 5 jsou tyto výsledky uvedeny. V prvním sloupci je vždy název obvodu, ke kterému se výsledky vztahují, v druhém a třetím sloupci je délka posloupnosti pro konstanty $C \geq C_{ust}$. Ve čtvrtém a pátem sloupci jsou hodnoty konstanty C_{ust} pro jednotlivé verze SW COMPAS.

Grafy jednotlivých průběhů jsou na obrázcích 11 až 14. Na ose x jsou vyneseny konstanty C, na ose y délky výsledné posloupnosti.

Z těchto závislostí je patrné, že pokusy najít jednu univerzální konstantu C (při

obvod	délky posloupnosti		C_{ust}	
	verze 3.2	verze 3.2.1	verze 3.2	verze 3.2.1
c432	381	252	9	16
c880	500	570	14	11
c2670	4848	4238	43	38
c7552	7042	7267	118	118
s1196	1167	1167	7	7
s1238	1133	1133	8	8
s5378	2102	2103	21	18
s9234	11433	10976	39	42

Tabulka 4: Ustálení průběhů – verze 3.2 a 3.2.1

obvod	délky posloupnosti		C_{ust}	
	verze Stack	verze Stack.1	verze Stack	verze Stack.1
c432	235	225	8	9
c880	409	474	10	14
c2670	3944	3753	38	45
c7552	6405	6306	85	85
s1196	725	725	5	5
s1238	807	807	8	8
s5378	1967	1915	14	14
s9234	10038	10002	40	38

Tabulka 5: Ustálení průběhů – verze Stack a Stack.1

použití tohoto kritéria), která by dávala nejlepší výsledky přes všechny obvody, nemohly být úspěšné a to z toho důvodu, že tyto závislosti jsou z počátku chaotické, globální minima se pro každý obvod nacházejí v jiném místě závislosti a navíc jsou závislosti rozdílné pro každou verzi SW COMPAS. Není tedy vhodé použít toto kritérium s použitím pevné konstanty rovné malému číslu.

Zajímavým zjištěním bylo, že průběh závislosti se vždy od určité hodnoty konstanty C ustálil, u každého obvodu se tato konstanta C_{ust} liší.

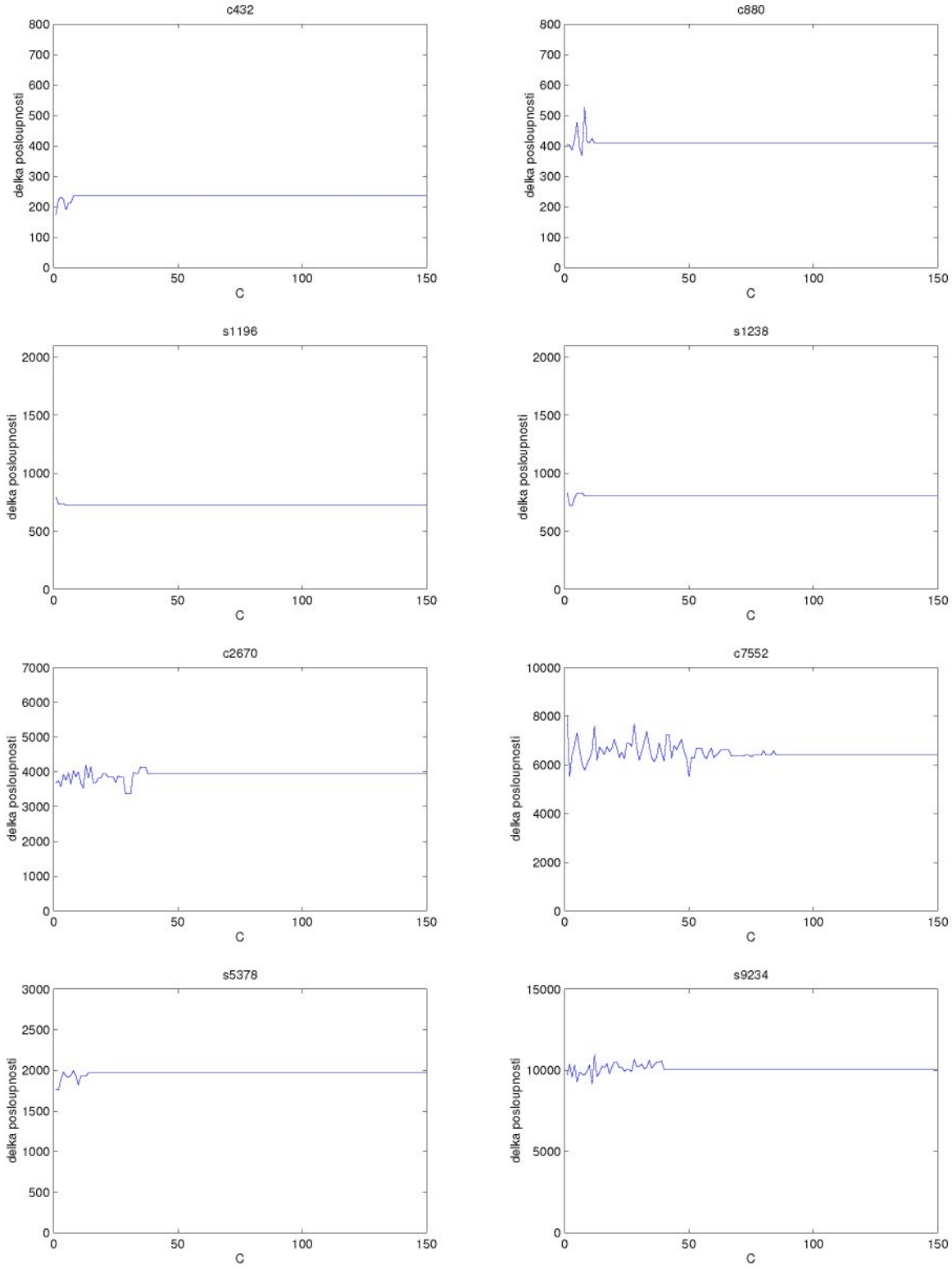
8.2 Kritérium 1.10

Dále probíhaly úvahy takto: co se stane, zjemníme-li volbu konstanty C (stále musíme mít na paměti, že můžeme volit pouze celá čísla). Byla tedy navržena úprava kritéria tak, aby byla konstanta C 10× „citlivější“. Kritérium potom vypadá takto (pracovně bylo nazváno **kritérium 1.10**):

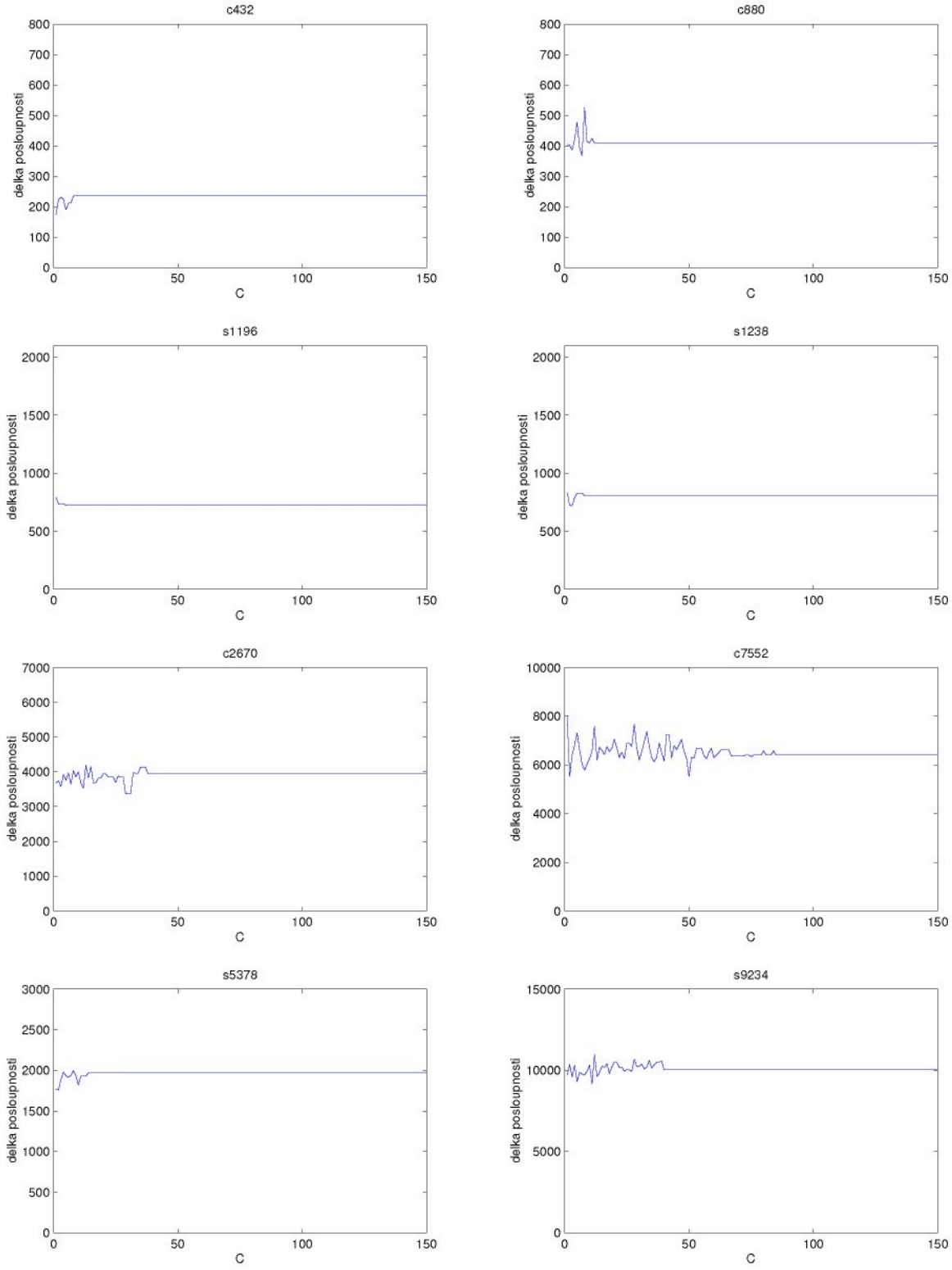
$$crit = (shift + no\_dontcares) \cdot C + 10 \cdot no\_globaldontcares$$

Volíme-li pro takto zvolené kritérium konstantu $C = 5$, dostaneme výsledky stejné, jako kdybychom použili původní kritérium 1 a zvolili konstantu $C = 0.5$ (což však není možné učinit). Cílem bylo zjistit, zda se průběh závislosti $L = f(C)$ pro menší C vyhladí, či nikoliv. Výsledné průběhy závislosti pro takto upravené kritérium jsou na obrázku č. 15 (průběhy pouze pro verzi 3.2).

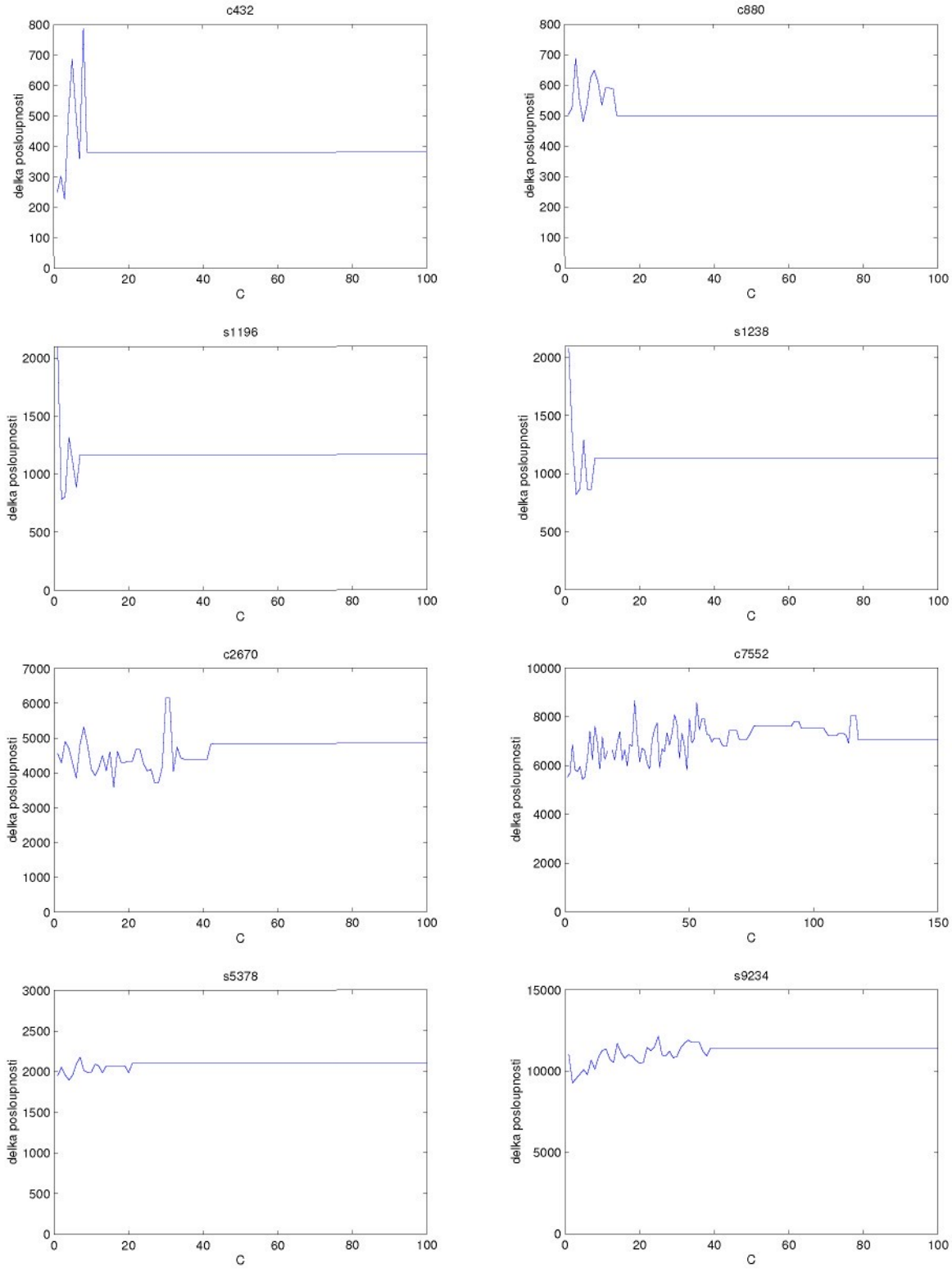
Z těchto průběhů bylo usouzeno, že „zjemněním“ možnosti volby konstanty C se průběh závislosti nijak zvlášt nezlepší, co se týče vyhlazení této závislosti. Proto s tímto kritériem nebylo dál pracováno..



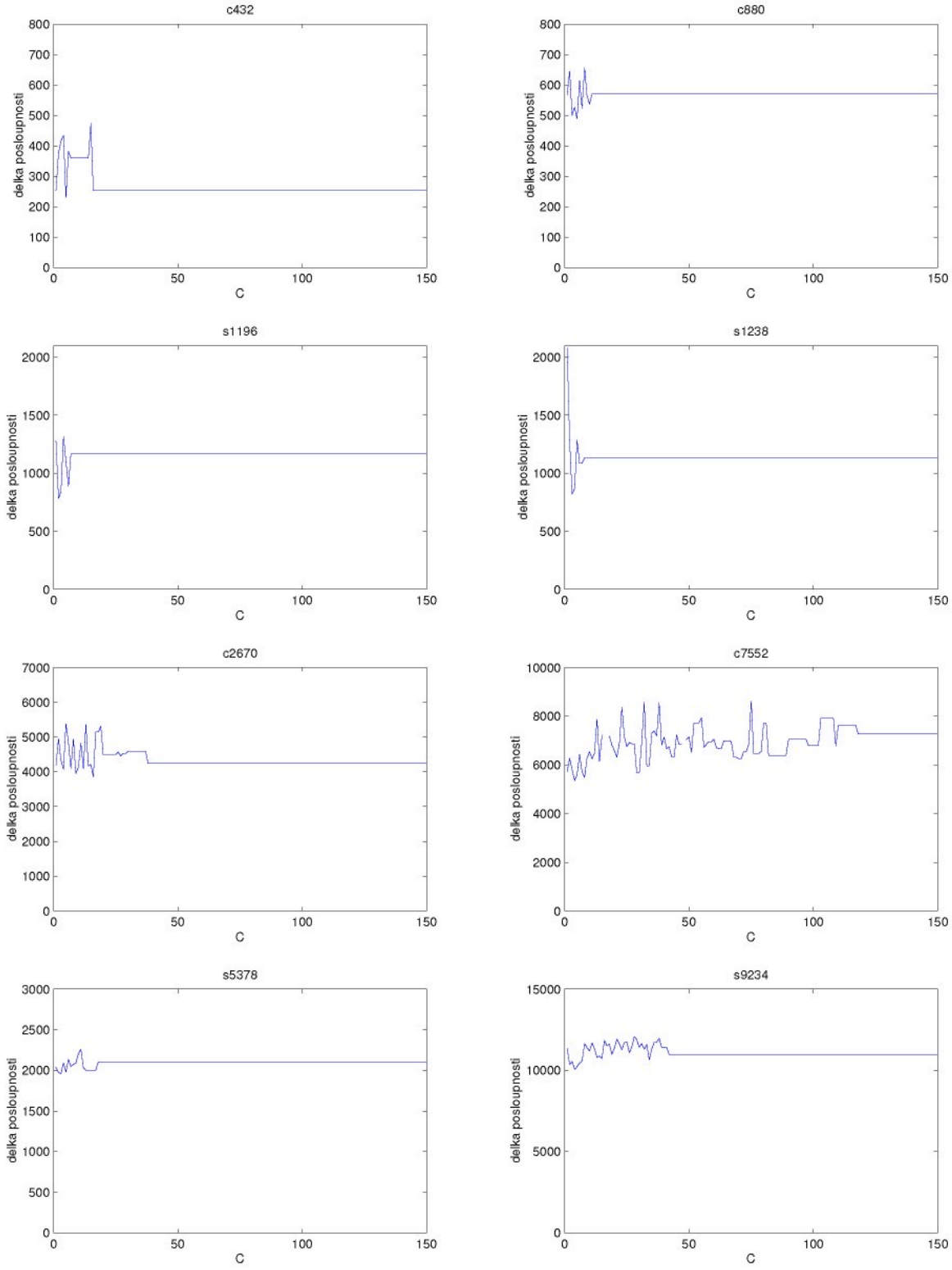
Obrázek 11: Průběhy závislosti $L = f(C)$, verze Stack, kritérium 1



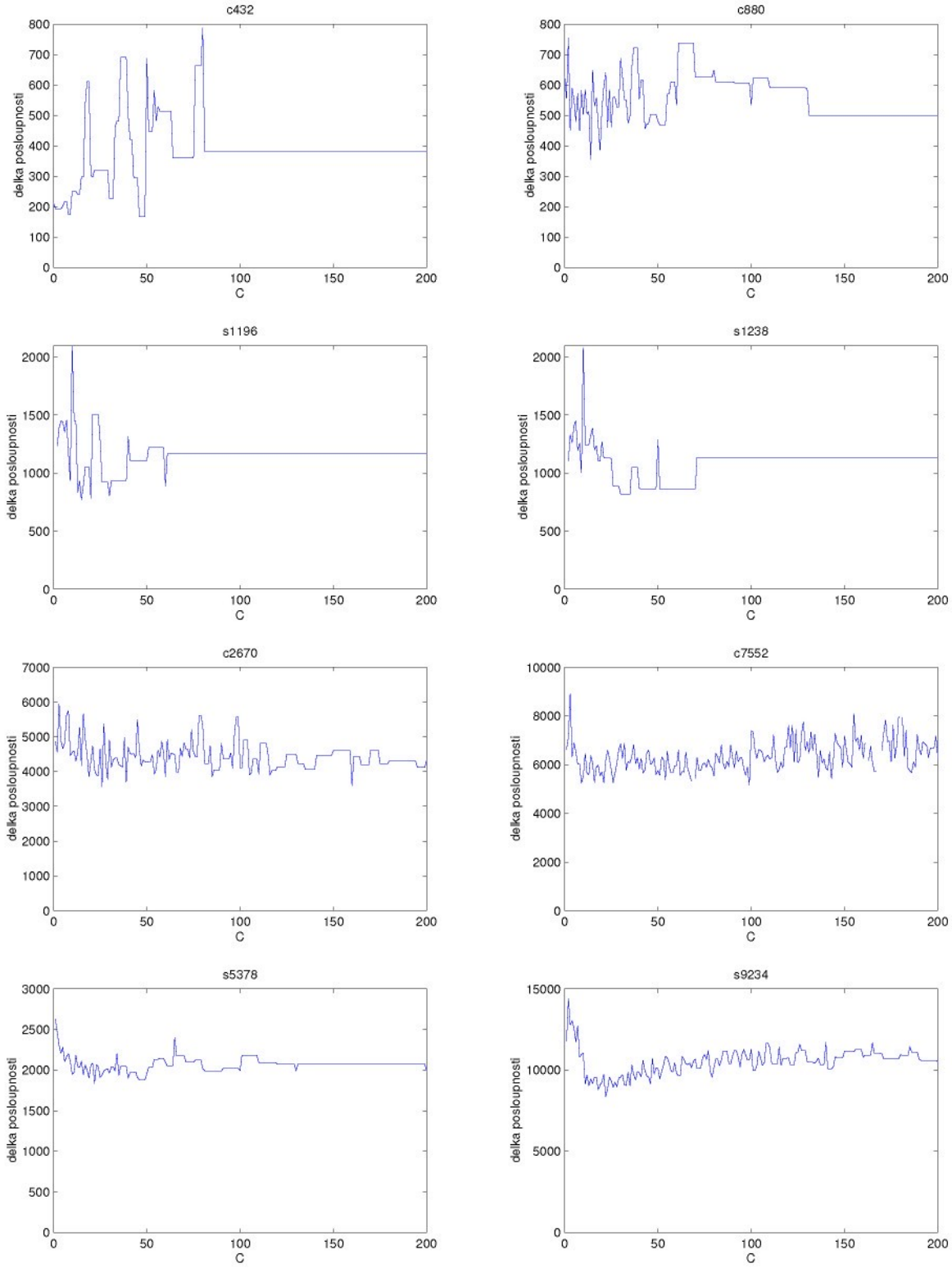
Obrázek 12: Průběhy závislosti $L = f(C)$, verze Stack.1, kritérium 1



Obrázek 13: Průběhy závislosti $L = f(C)$, verze 3.2, kritérium 1



Obrázek 14: Průběhy závislosti $L = f(C)$, verze 3.2.1, kritérium 1



Obrázek 15: Průběhy závislosti $L = f(C)$, verze 3.2, kritérium 1.10

8.3 Kritérium 1, ustálená oblast

8.3.1 Ověření ustálené oblasti

Pozornost tedy byla zaměřena na oblast průběhu $L = f(C)$, která je ustálená. Tato oblast je pro nás zajímavá, protože s použitím dostatečné velké konstanty C dostaneme stabilní výsledky přes všechny obvody a přitom výsledné délky posloupností jsou pro nás postačující.

Nejdříve bylo třeba ověřit, že i u větších obvodů se lze dostat do ustálené oblasti průběhu. Pro tyto obvody z časových důvodů nebylo možné provést zkoumání celého intervalu $C \in \langle 1, 150 \rangle$, navíc by se mohlo stát, že dosavadní maximální hodnota $C = 150$, nebude dostatečná. Proto byly vybrány dvě konstanty dostatečně velké, pro které měly být výsledné délky posloupností porovnány. Pokud by byly obě délky stejné, lze s velkou pravděpodobností prohlásit, že jsem se dostal do ustálené oblasti závislosti. Konstanty byly zvoleny takto: $C_A = 2500, C_B = 2600$. Ověření bylo provedeno pro skupinu dvaceti dvou obvodů ze sad ISCAS85 a ISCAS89.

Po zpracování výsledků jsme mohli konstatovat, že domněnka se potvrdila: délky posloupností byly pro obě konstanty stejné přes všechny obvody, a můžeme tedy tvrdit, že s největší pravděpodobností dochází k ustálení průběhů závislosti $L = f(C)$ i pro větší obvody. Délky posloupností nalezneme v tabulce č. 6. Z časových důvodů byla vynechána verze 3.2.1. Pokud dochází k ustálení závislosti ve verzi 3.2, velmi pravděpodobně je tomu tak i při použití verze 3.2.1. (viz verzi Stack versus Stack.1).

První otázkou, která vzápětí vyvstala, byla tato: Výsledná délka posloupnosti v ustálené oblasti se zvětšujícím se C nemění, váha části kritéria s *global_dontcares* je tedy malá; není možné, že tato část kritéria je zbytečná? Bylo tedy navrhženo kritérium 0, bez použití této části, které bylo třeba dále prozkoumat (viz kapitolu 9.1).

Druhou otázkou, kterou byla třeba zodpovědět, bylo, zda neexistuje závislost mezi fyzickými parametry obvodu a konstantou C, při které se průběh závislosti $L = f(C)$ ustálí.

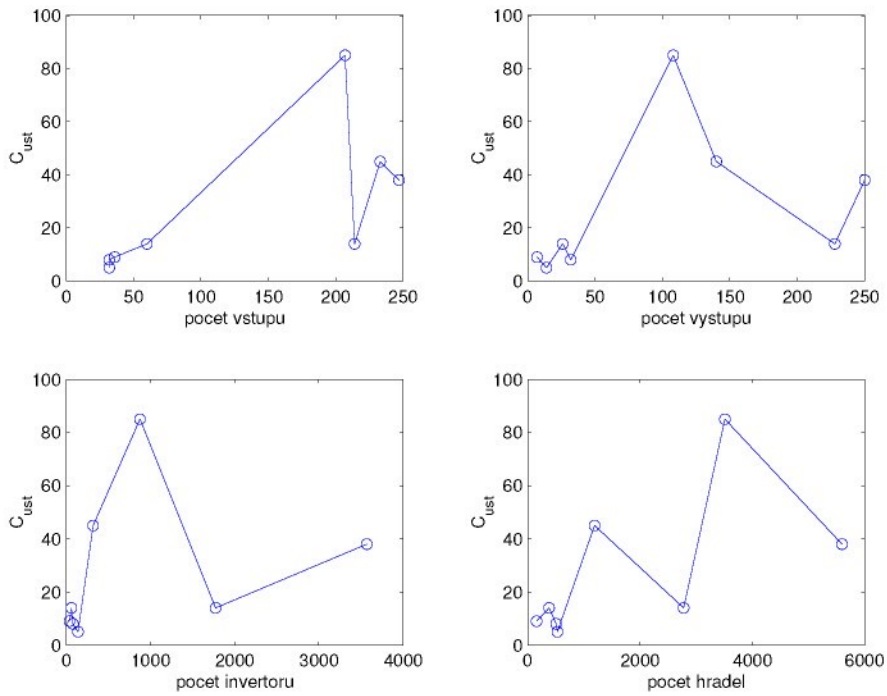
	délky posloupností		
	verze 3.2	verze stack	verze stack.1
c17	10	10	10
c432	381	235	225
c499	4566	302	367
c880	500	409	474
c1355	7906	1115	1115
c1908	2359	1106	940
c2670	4848	3944	3753
c3540	1061	792	806
c5315	1284	1204	1204
c6288	648	83	83
c7552	7042	6405	6306
s27	29	16	16
s1196	1167	725	725
s1238	1133	807	807
s1494	571	515	443
s5378	2102	1967	1915
s9234	11433	10038	10002
s13207	4125	4057	4057
s15850	7543	6580	6580
s35932	1944	1944	1944
s38417	22810	21021	20877
s38584	7828	6493	6493

Tabulka 6: Délky posloupností pro $C = 2500$ a $C = 2600$ (vycházejí stejně)

8.3.2 Ustálená oblast versus parametry obvodu

Pro zkoumání potenciální závislosti byly vybrány informace o těchto fyzických parametrech obvodů: počet vstupů, počet výstupů, počet invertorů a počet hradel celkem.

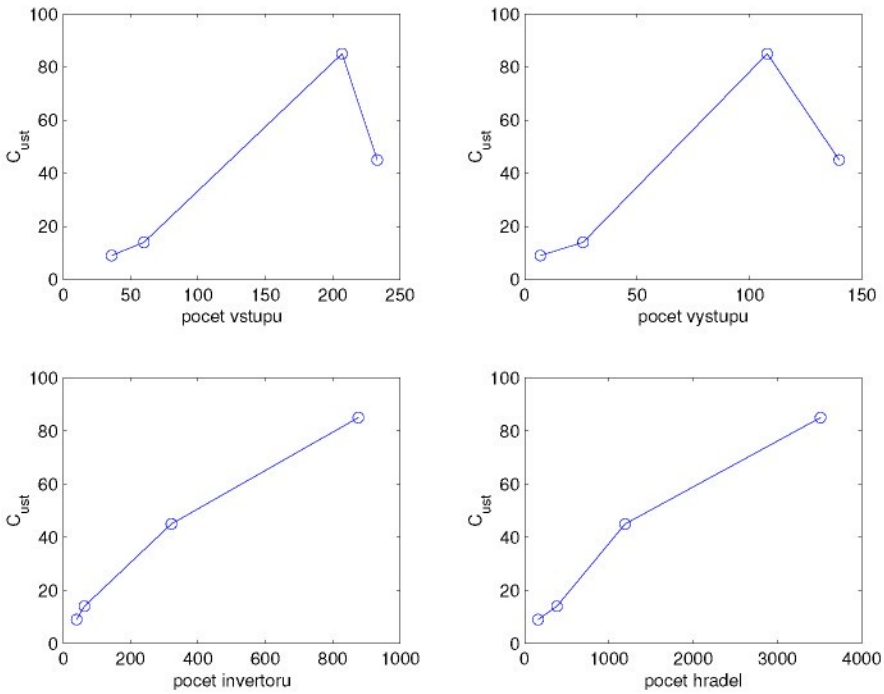
Nejprve byly vypracovány grafy potenciálních závislostí pro všechny obvody, pro které jsme měli k dispozici informaci, kde se závislost $L = f(C)$ ustaluje. Konkrétně šlo opět o obvody: c432, c880, c2670, c7552, s1196, s1238, s5378 a s9234, viz obrázek č. 16.



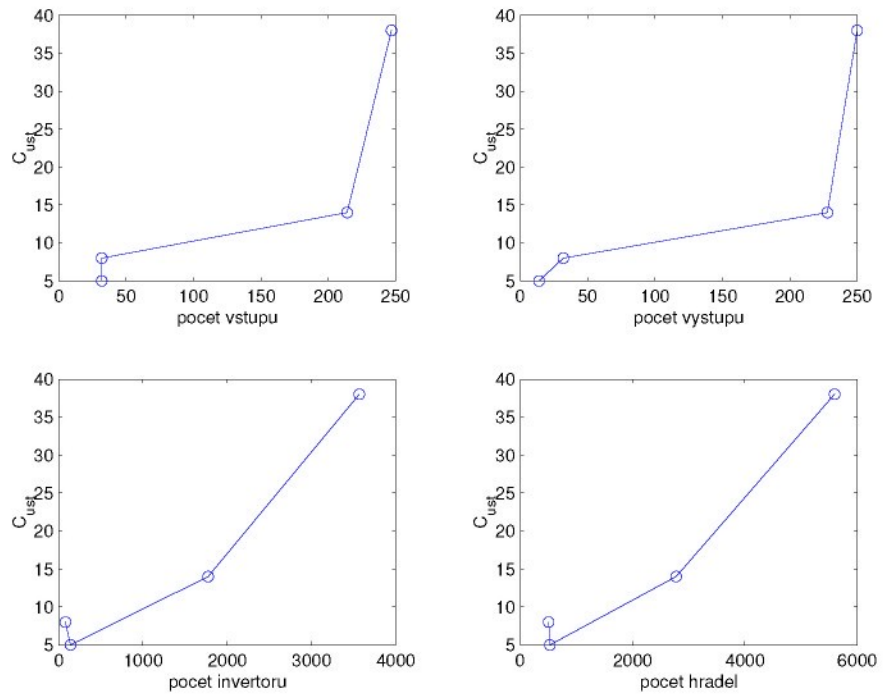
Obrázek 16: Potenciální závislosti, všechny obvody, verze Stack.1

Na ose x je vždy fyzický parametr obvodu a na ose y hodnota parametru C, kdy se závislost $L = f(C)$ ustálila. Byly použity hodnoty získané s použitím verze Stack.1 (hodnoty získané s použitím verze Stack jsou přitom velmi podobné).

Z těchto grafů však žádná závislost nevyplynula, tak byl další postup oddělit od sebe obvody kombinační a sekvenční (viz obrázky č. 17 a 18). V obou případech se objevily přibližně lineární závislosti. Bylo by tedy možné navrhnout kritérium, kde by



Obrázek 17: Potenciální závislosti, kombinační obvody, verze Stack.1



Obrázek 18: Potenciální závislosti, sekvenční obvody, verze Stack.1

se konstanta C volila podle některého z fyzických parametrů obvodu (viz kapitolu 8.4).

8.4 Kritérium 1.R

Než bylo možné navrhnout kritérium, kde by se konstanta C vypočítala z některého z fyzických parametrů obvodu, bylo potřeba vybrat konkrétní fyzický parametr obvodu. Jako nejvhodnější se jevil počet hradel, vzhledem k tomu, že je přímo ve vztahu se složitostí obvodu. Dále bylo třeba navrhnout způsob, jakým bude konstanta C z tohoto parametru vypočítána.

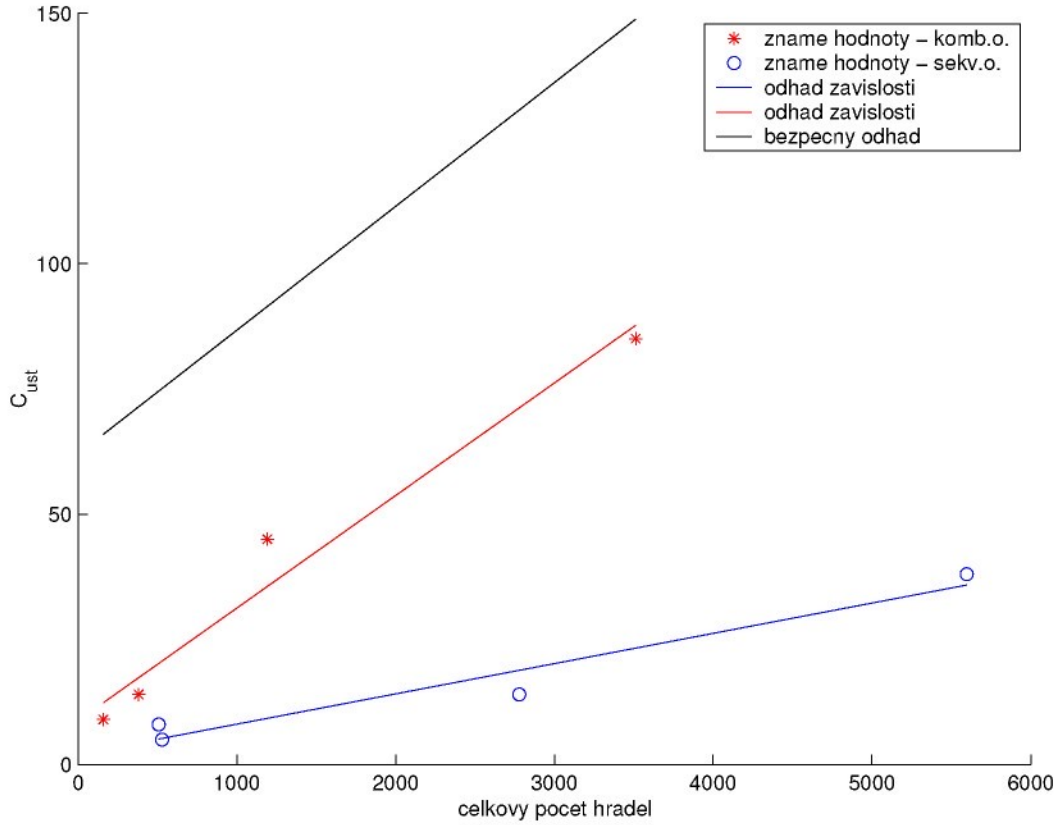
Nejprve byly nalezené závislosti pomocí metody nejmenších čtverců proložil přímkami zvlášť pro kombinační a zvlášť pro sekvenční obvody. Dostali jsme tedy dvě odhadnuté závislosti, jednu pro kombinační a druhou pro sekvenční obvody. Jelikož není vhodné zatěžovat uživatele SW COMPAS tím, aby zadával, zda zadaný obvod je kombinační či sekvenční, bylo rozhodnuto použít závislost odpovídající **kombinačním** obvodům, a to z toho důvodu, že pokud z počtu hradel libovolného sekvenčního obvodu vypočítáme dle této závislosti konstantu C, vždy se s použitím této konstanty dostaneme do ustálené oblasti průběhu $L = f(C)$. Navíc byly upraveny oba koeficienty určující rovnici přímky tak, aby odhadnutá závislost byla ještě bezpečnější, abychom měli větší jistotu, že jsme se do ustálené oblasti dostali. Koeficient určující směrnici přímky byl tedy zvětšen o 10% a koeficient určující posunutí byl zvětšen o pětinásobek největší odchylky odhadu lineární závislosti od skutečných hodnot (viz obrázek č. 19). Takto byl získán **bezpečný odhad** této závislosti. Výsledné koeficienty jsou tedy $k = 0.025$, $q = 62$. Konstanta C se potom vypočítá takto:

$$C = k \cdot no\_gates + q,$$

kde konstanty k a q mají uvedené hodnoty a parametr *no_gates* odpovídá celkovému počtu hradel obvodu.

Kritérium s použitím těchto koeficientů má tedy tvar

$$crit = (shift + no\_dontcares) \cdot (k \cdot no\_gates + q) + no\_globaldontcares$$



Obrázek 19: Závislosti C_{ust} na celkovém počtu hradel

Toto kritérium bylo označeno jako kritérium 1.R.

Nejprve bylo třeba ověřit, zda s takto navrženým kritériem se dostaneme do ustálenené oblasti závislosti $L = f(C)$. Byl tedy proveden test tohoto kritéria pro verze Stack, Stack.1 a verzi 3.2.

Výsledky jsou v tabulce č. 7. Ve druhém sloupci jsou hodnoty konstanty C vypočítané dle navrženého odhadu. V dalších sloupcích jsou výsledné délky posloupností. Srovnáme-li tyto hodnoty s hodnotami z tabulky č. 6, zjistíme, že výsledné délky jsou úplně shodné, můžeme tedy prohlásit, že jsme se takto zvolenou konstantou C ve všech případech dostali úspěšně do ustálené oblasti závislosti $L = f(C)$.

Dále bylo třeba ještě navrhnout způsob, jak bude SW COMPAS zjišťovat počet hradel obvodu. To se ukázalo jako velmi jednoduché, protože soubor, v němž je defi-

		délky posloupností		
	C	verze 3.2	verze stack	verze stack.1
c17	62	10	10	10
c432	65	381	235	225
c499	66	4566	302	367
c880	70	500	409	474
c1355	74	7906	1115	1115
c1908	81	2359	1106	940
c2670	88	4848	3944	3753
c3540	98	1061	792	806
c5315	112	1284	1204	1204
c6288	115	648	83	83
c7552	139	7042	6405	6306
s27	62	29	16	16
s1196	73	1167	725	725
s1238	73	1133	807	807
s1494	76	571	515	443
s5378	123	2102	1967	1915
s9234	185	11433	10038	10002
s13207	236	4125	4057	4057
s15850	276	7543	6580	6580
s35932	415	1944	1944	1944
s38417	549	22810	21021	20877
s38584	485	7828	6493	6493

Tabulka 7: Délky posloupností pro kritérium 1.R

nována struktura obvodu (`název_obvodu.bench`), má formát textového souboru, kde například hradlo AND je definováno takovýmto způsobem:

```
425 = AND(404, 405)
```

Stačí tedy spočítat počet rovnítek v definičním souboru a dostaneme počet hradel příslušného obvodu. Pro všechny obvody ze sad ISCAS85 a ISCAS89 byl tento postup ověřen jako funkční.

9 Kritérium bez parametrů

Při zkoumání kritéria 1 se ukázalo, že průběhy závislosti délky výsledné posloupnosti L na konstantě C se od určitých hodnot C ustalují (platí pro všechny obvody). Proto byl důvod se domnívat, že druhá část tohoto kritéria by mohla být zbytečná.

9.1 Kritérium 0

Za předpokladu, že konstanta C je velká, tedy váha části kritéria s *global_dontcares* je malá, vznikla otázka, zda tato část není zbytečná. Proto bylo navrženo kritérim č. 0 tak, že byla vypuštěna druhá část kritéria č. 1, má tedy tento tvar:

$$crit = shift + no\_dontcares$$

Dále bylo třeba zjistit, jaké budou výsledné délky posloupností s použitím tohoto kritéria. Po změně kritéria v SW COMPAS byly tyto posloupnosti s jeho pomocí vytvořeny, výsledné délky jsou v tabulce č. 8.

Jelikož výsledky s použitím tohoto kritéria byly dobré a zároveň nebylo třeba volit žádnou konstantu, bylo rozhodnuto je použít při normalizaci délek výsledných posloupností. Tuto normalizaci bylo nutno provést ve chvíli, kdy jsme chtěli programově porovnávat nárůst či zkrácení délky výsledné posloupnosti napříč obvody, jmenovitě při výpočtu účelové funkce v programu POET (viz kapitolu 6.3).

9.2 Srovnání s jednoparametrovými kritérii

Ve chvíli, kdy byly k dispozici výsledky tohoto kritéria, kritéria 1.R a hodnoty původních výsledných posloupností s použitím kritéria 1 a konstanty $C = 2$ (původní kritérium), bylo možné provést srovnání těchto hodnot (viz tabulku č. 9).

Grafické zpracování těchto hodnot je uvedeno na obrázcích č. 20 a 21. Všechny výsledné délky posloupností jsou vyneseny v poměru ke kritériu 0. Červené sloupce

obvod	výsledné délky		obvod	výsledné délky	
	verze Stack	verze Stack.1		verze Stack	verze Stack.1
c17	13	13	s27	16	16
c432	199	210	s1196	726	726
c499	311	259	s1238	780	780
c880	394	437	s1494	563	486
c1355	1115	1115	s5378	1956	2047
c1908	1176	1178	s9234	10306	10871
c2670	4019	4104	s13207	4335	4082
c3540	768	703	s15850	7037	7037
c5315	1031	1031	s35932	1937	1937
c6288	84	84	s38417	22255	22411
c7552	6623	6488	s38584	6783	6783

Tabulka 8: Výsledné délky, kritérium 0

vyjadřují výsledky s použitím kritéria 1.R a zelené odpovídají původnímu kritériu ($C = 2$).

Pro obě verze jsme došli k výsledkům, kdy 13 obvodů mělo kratší výsledné délky s použitím kritéria 1.R a 7 obvodů mělo kratší výsledné délky s použitím kritéria 0 (nebyly to však vždy pro obě verze obvody stejné), zbylé dva obvody měly stejné výsledné délky posloupností pro obě kritéria.

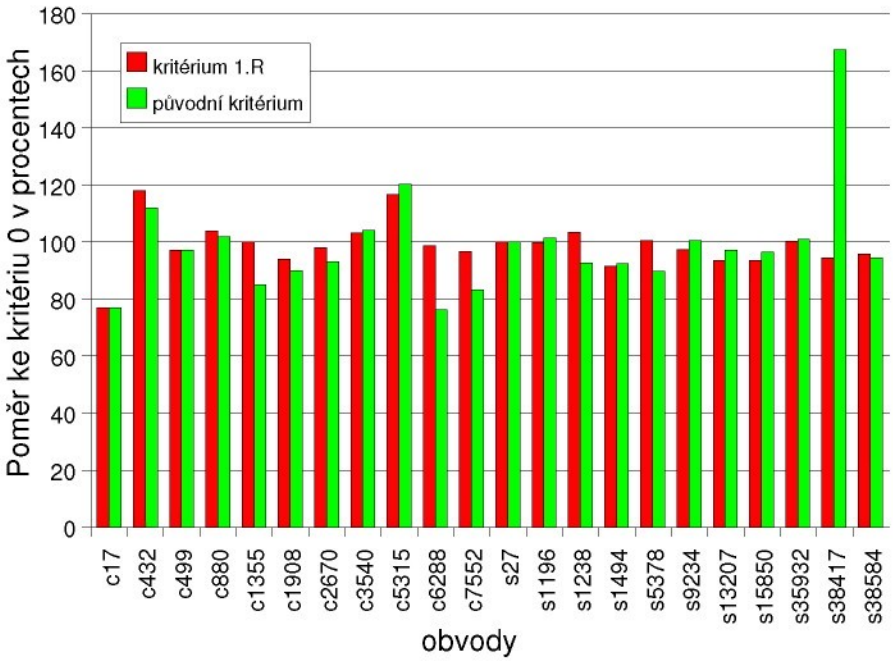
Použití kritéria 1.R je tedy celkově lepší než použití kritéria 0, není tomu tak ale pro všechny obvody.

Pro srovnání jsou též uvedeny hodnoty získané pomocí původního kritéria 1 s použitím konstanty $C = 2$.

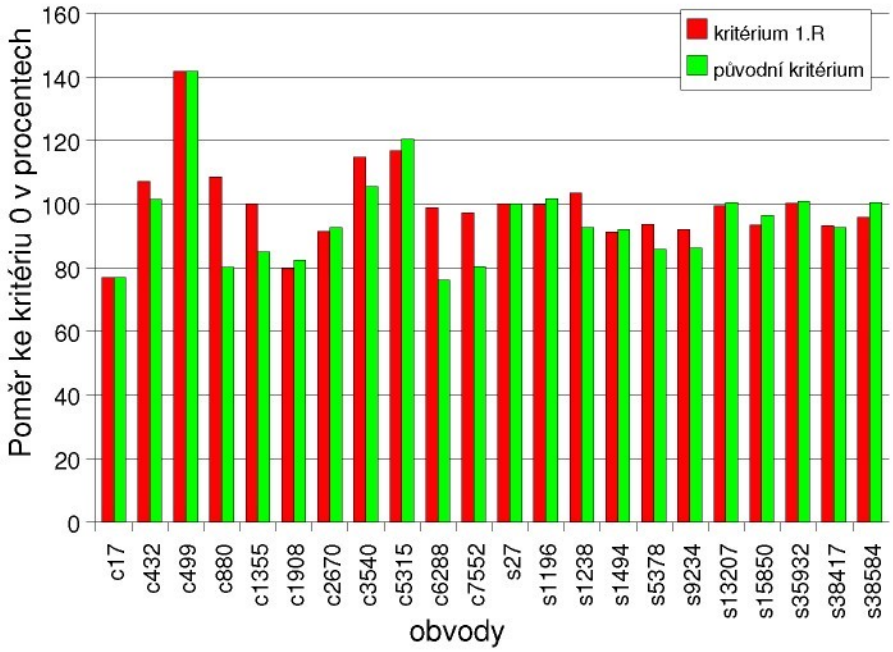
Srovnáme-li stejným způsobem výsledky kritéria 1.R proti výsledkům kritéria 1 s konstantou $C = 2$, dostaneme poměr 9:10 obvodů ve prospěch kritéria 1 (všimněme si však velmi špatného výsledku pro obvod s38417!). Použití kritéria 1.R se tedy zdá

	délky posloupností					
	verze Stack			verze Stack.1		
obvod	krit. č. 1.R	krit. č. 0	pův. krit.	krit. č. 1.R	krit. č. 0	pův. krit.
c17	10	13	10	10	13	10
c432	235	199	223	225	210	213
c499	302	311	302	367	259	367
c880	409	394	401	474	437	350
c1355	1115	1115	947	1115	1115	947
c1908	1106	1176	1058	940	1178	970
c2670	3944	4019	3743	3753	4104	3801
c3540	792	768	800	806	703	742
c5315	1204	1031	1241	1204	1031	1241
c6288	83	84	64	83	84	64
c7552	6405	6623	5518	6306	6488	5211
s27	16	16	16	16	16	16
s1196	725	726	736	725	726	738
s1238	807	780	723	807	780	723
s1494	515	563	520	443	486	447
s5378	1967	1956	1757	1915	2047	1757
s9234	10038	10306	10366	10002	10871	9371
s13207	4057	4335	4209	4057	4082	4101
s15850	6580	7037	6791	6580	7037	6791
s35932	1944	1937	1955	1944	1937	1955
s38417	21021	22255	37290	20877	22411	20752
s38584	6493	6783	6410	6493	6783	6811

Tabulka 9: Srovnání délek posloupností



Obrázek 20: Srovnání výsledků kritérií, verze Stack



Obrázek 21: Srovnání výsledků kritérií, verze Stack.1

přibližně stejně vhodné jako použití kritéria 1 a konstanty $C = 2$.

Musíme však vzít v úvahu, že při použití kritéria 1.R se dají očekávat stabilnější výsledky, s použitím kritéria 1 a konstanty $C = 2$ můžeme očekávat výrazné odchylky od jeho výsledků, změníme-li obvod, nebo verzi softwaru. Důvodem je chaotická závislost délky výsledné posloupnosti na hodnotě konstanty C z počátku této závislosti. Nemůžeme tedy říci, že s použitím kritéria 1 dostaneme stabilní výsledky (viz délku posloupnosti zmíněného obvodu s38417). Z těchto dvou kritérií je tedy lepší doporučit kritérium 1.R.

10 Dvouparametrová kritéria

Při zkoumání jednoparametrových kritérií vyvstala otázka, zda by nebylo lepší první dvě části zahrnuté do jedné váhové konstanty (*shift* a *no_dontcare*) od sebe oddělit. Pokud to provedeme, dostaneme kritérium se dvěma konstantami C_1 a C_2.

Vzhledem k tomu, že toto kritérium obsahovalo dvě konstanty (dva volitelné parametry), nebylo už možné zkoumat průběhy závislostí přímo jako u jednoparametrových kritérií. Byl tedy zvolen způsob optimalizace těchto parametrů pomocí genetických algoritmů (za tím účelem vznikl program POET). Cílem bylo nalézt hodnoty parametrů C_1 a C_2 tak, aby byly použitelné napříč obvody.

Navržen byl tento postup práce:

1. Vypočítáme pomocí SW COMPAS trénovací data pro několik menších obvodů, na kterých bude provedeno vyladění parametrů genetického algoritmu.
2. Provedeme optimalizaci parametrů C_1 a C_2 pomocí GA.
3. Získané kandidáty na optimální parametry ověříme na dalších větších obvodech.

Obvody, pro které byla trénovací data získávána, musely být vybrány tak, aby byly co největší a zároveň výpočet mohl doběhnout v rozumném čase. Dále bylo rozhodnuto, že polovina obvodů bude kombinačních a polovina sekvenčních, a to proto, aby skupina obvodů reprezentovala oba dva typy. Nakonec byly zvoleny tyto obvody: c432, c880, s1196 a s1238. Dále bylo potřeba zvolit rozsah konstant C_1 a C_2, ten byl zvolen pro obě konstanty stejný: 0 až 255. Pro každý obvod bylo tedy získáno 65536 různých hodnot odpovídajících všem kombinacím těchto dvou konstant[7]. Pro výpočet byla použita verze Stack a kritérium 2.B (viz kapitolu č. 10.2).

Po získání těchto dat bylo přistoupeno k vyladění parametrů genetického algoritmu. Také bylo třeba zjistit, který ze tří zvolených genetických algoritmů (Simple, Deme a Steady-State) bude dávat nejlepší výsledky.

[7] výpočet těchto dat trval zhruba tři týdny

Pro standardní jednoduchý GA byly laděny tyto parametry: velikost populace, pravděpodobnost křížení, pravděpodobnost mutace, počet generací, po kterých se má algoritmus ukončit při nenalezení žádného nového kandidáta, a případné použití elitismu.

Pro algoritmus s více populacemi ještě přibyly tyto parametry: počet paralelních populací, počet jedinců migrujících mezi populacemi.

Pro algoritmus s postupnou výměnou populace byl navíc kromě parametrů jednoduchého GA ještě navíc parametr určující, kolik jedinců se má každou generaci obměnit (respektive jaké procento jedinců).

Byly přitom sledovány tyto ukazatele: průběh počtu celkem nalezených kandidátů během evolučního procesu, vývoj diverzity populace, dosažená minimální hodnota účelové funkce a počet spuštění SW COMPAS. Tento počet byl omezen na 5500 po přibližném odhadu času (maximální doba běhu byla určena přibližně na dva dny, viz kapitolu 10.1), jak dlouho algoritmus poběží. To přibližně odpovídá 8.5% prohledávaného prostoru.

Program POET byl navržen tak, že si tato trénovací data načetl, a SW COMPAS se tedy nemusel vůbec spouštět. Proto doba trvání programu byla v řádu sekund. Bylo možné tedy jeden či více parametrů GA změnit, program spustit znovu a sledovat, jaký měla změna parametrů GA vliv na evoluční proces.

Jelikož genetické algoritmy jsou založeny na náhodných dějích, bylo též třeba program POET při každé změně parametrů spustit vícekrát, aby vliv na průběh evoluce byl průkazný. Počet spuštění byl zvolen roven deseti.

Jako první byly laděny parametry jednoduchého GA (Simple), druhým byl algoritmus s paralelními populacemi (Deme GA) a jako poslední algoritmus s částečnou obměnou populace (Steady-state GA).[8]

Příklad průběhů evoluce pro Simple GA je na obrázku č. 22, příklad průběhů evoluce pro Deme GA na obrázku č. 23 a příklad průběhů evoluce pro Steaty-state GA

[8] pro každý GA bylo vyzkoušeno v průměru přes padesát kombinací hodnot parametrů

na obrázku č. 24.

Hlavní sledovanou hodnotou byl minimální počet kandidátů, který byl program POET schopen za svůj běh najít (v grafech označen jako min(candidates)), dále průběhy narůstání počtu kandidátů a průběhy diverzity populace (ta se pohybuje v rozmezí hodnot 0 a 1, kde 0 znamená, že všichni jedinci v populaci jsou identičtí, hodnota 1 značí maximální diverzitu populace). Na těchto grafech už jsou průběhy pro optimalizované GA, byly to tedy nejlepší možné výsledky, kterých se na testovacích datech podařilo dosáhnout.

Jak je vidět z těchto grafů, nejvíce kandidátů byl schopen najít Deme GA, pak srovnatelný počet kandidátů dokázal najít Steady-state GA a nejmenší počet kandidátů našel Simple GA (jedná se vždy o minimum z počtu kandidátů nalezených během deseti běhů programu POET). Bylo tedy třeba rozhodnout, zda použít Deme GA nebo Steady-state GA. Nakonec na základě průběhů diverzity populace bylo rozhodnuto ve prospěch algoritmu s postupnou výměnou populace (Steady-state GA). Ten dokázal udržet diverzitu populace na rozumné výši, což přispívalo k odolnosti proti sklouznutí do lokálního extrému.

Pro optimalizaci byl tedy vybrán algoritmus s postupnou výměnou populace s těmito parametry:

Velikost populace (*popsize*): 170 jedinců

Procento obměny populace (*repper*): 24%, tedy 40 jedinců (*repnum*)

Pravděpodobnost křížení (P_x): 0.9

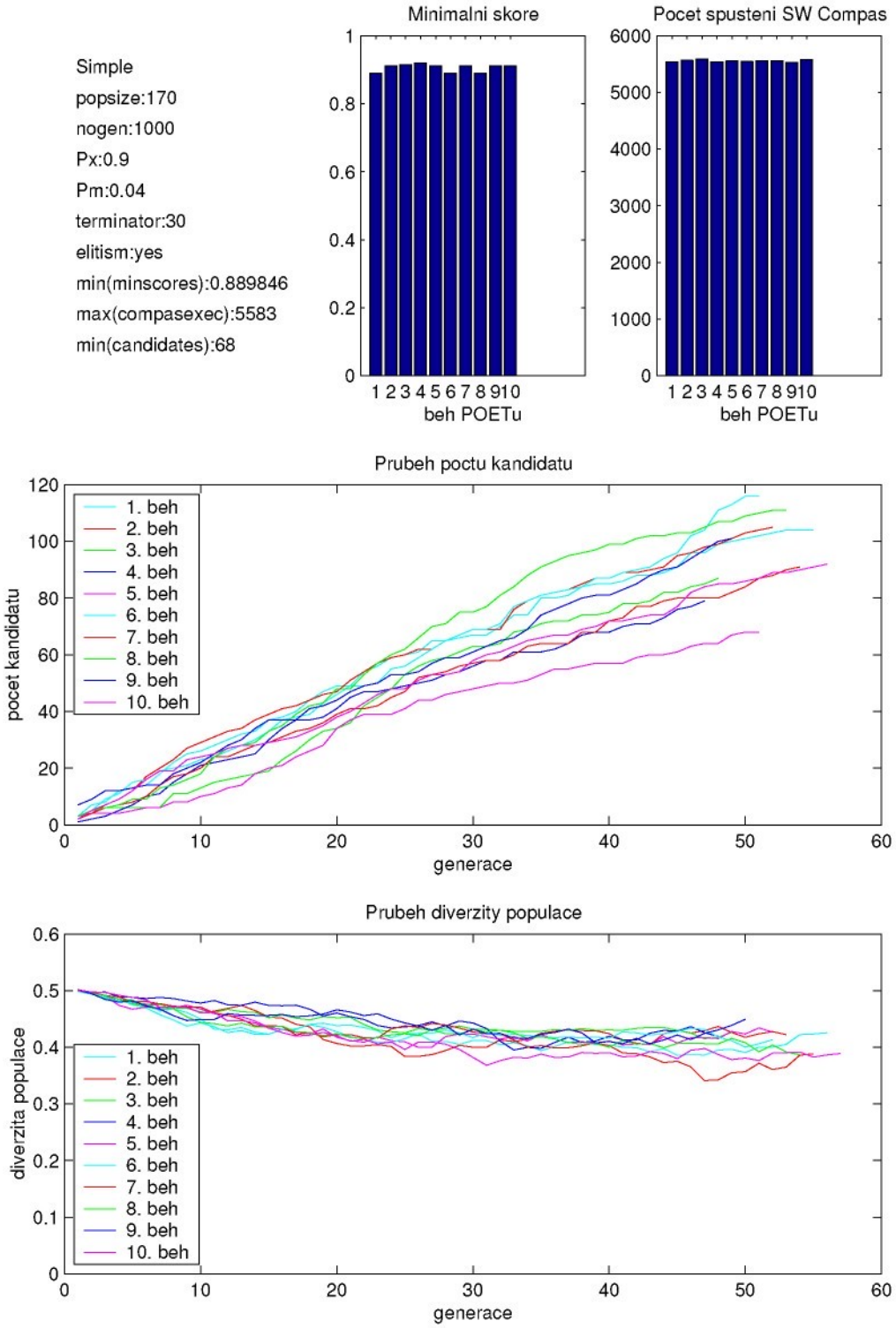
Pravděpodobnost mutace (P_m): 0.08

10.1 Kritérium 2.C

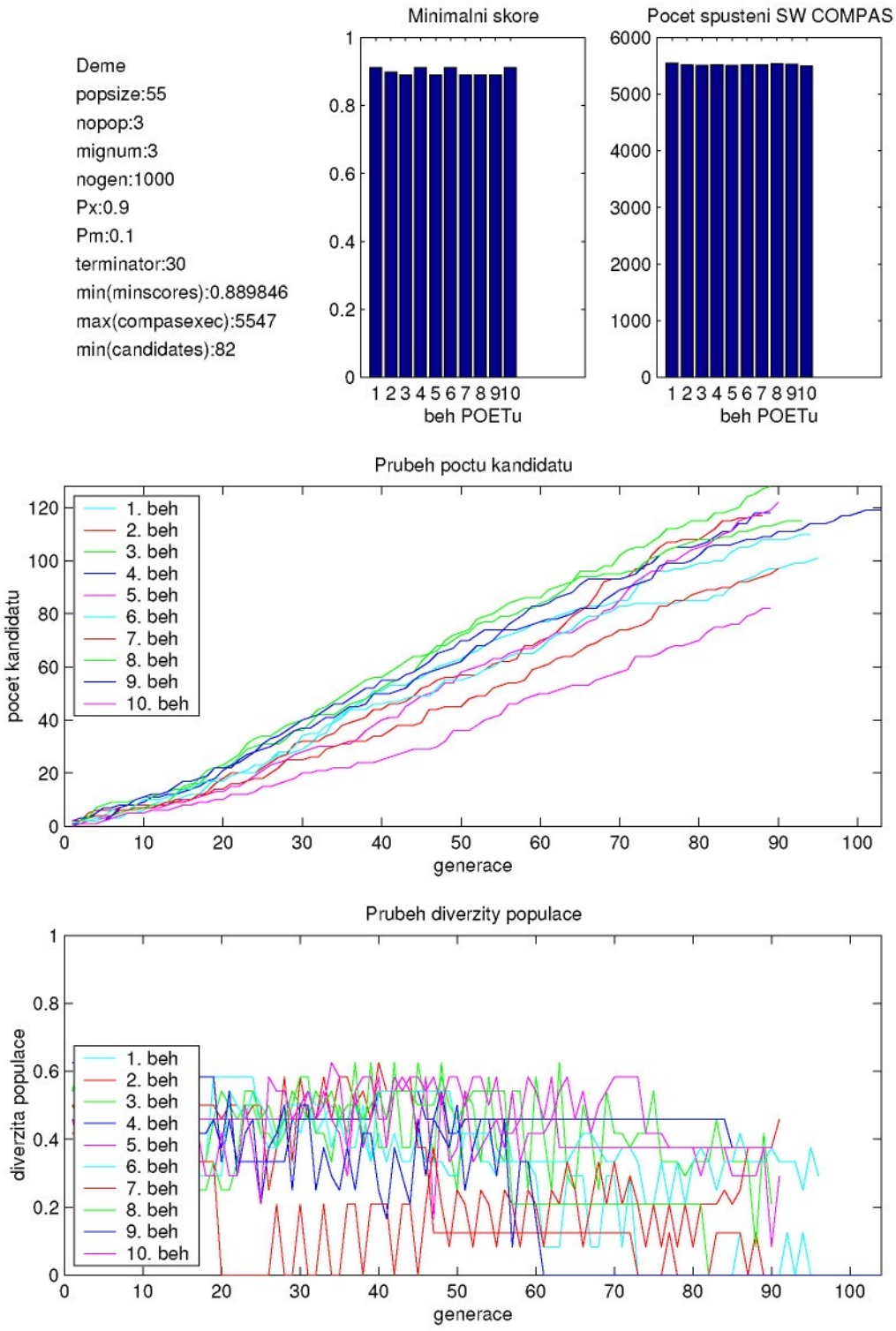
Kritérium 2.C je odvozeno od kritéria 1 a má takovýto tvar:

$$crit = C_1 \cdot shift + C_2 \cdot no\_dontcares + no\_globaldontcares$$

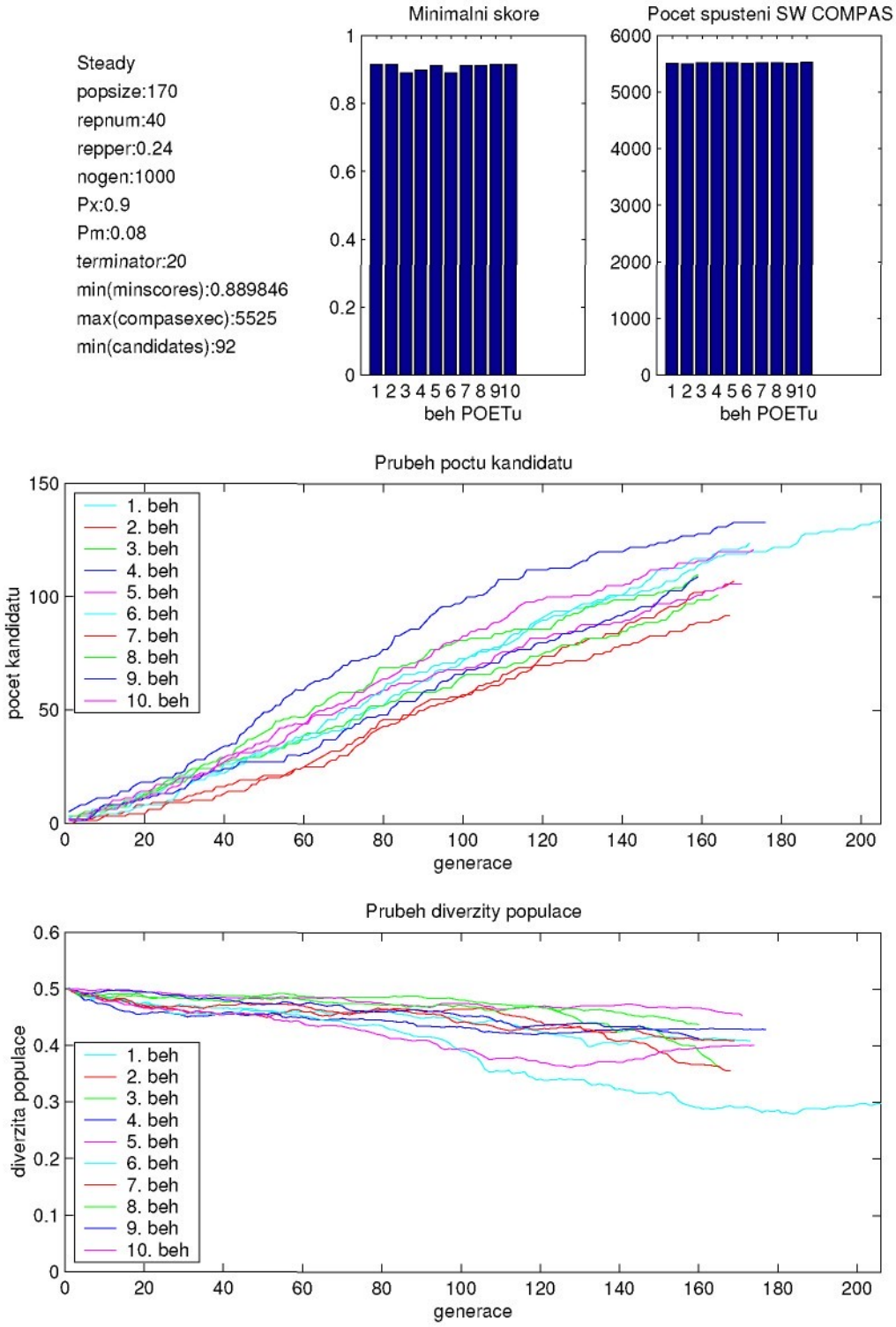
Pro vyhledávání kandidátů byly zvoleny tyto obvody: c1355, c3540, s1238 a s1494.



Obrázek 22: Grafický přehled sledovaných hodnot, Simple GA



Obrázek 23: Grafický přehled sledovaných hodnot, Deme GA



Obrázek 24: Grafický přehled sledovaných hodnot, Steady-state GA

Z verzí SW COMPAS byly vybrány Stack a Stack.1. Jak bylo řečeno, maximální počet spuštění SW COMPAS byl stanoven na 5500, aby algoritmus doběhl přibližně do dvou dnů. Tato doba byla určena z průměrných časů jednotlivých spuštění SW COMPAS pro tyto obvody a byla ponechána dostatečná rezerva, aby algoritmus do této doby určitě skončil.

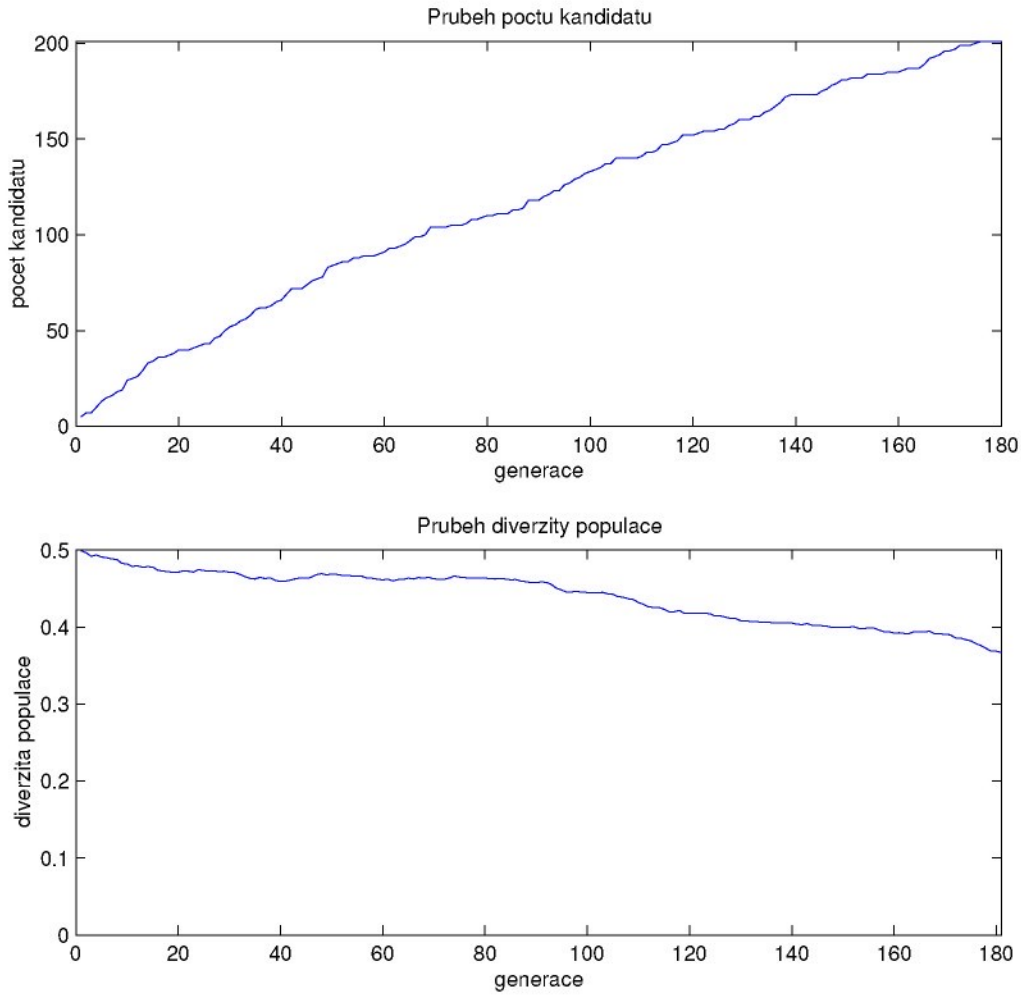
Průběh evoluce pro verzi Stack je na obrázku č. 25, podařilo se tedy najít 201 kandidátů. Při použití verze Stack.1 program našel 25 kandidátů. Rozdíly v nalezeném počtu kandidátů jsou způsobeny zejména rozdílnými výsledky jednotlivých verzí SW COMPAS při použití kritéria 0, z jehož hodnot se při optimalizaci vycházelo. Pro optimalizaci parametrů kritéria, použitého v konkrétní verzi, bylo samozřejmě využito výsledků kritéria 0, vytvořených pomocí verze stejné.

Z kombinací konstant C_1 a C_2, které algoritmus za svůj běh prošel, bylo možné vytvořit graf, ze kterého bude patrná závislost hodnot účelové funkce na kombinacích těchto konstant (verze Stack viz obrázek č. 26 a verze Stack.1 viz obrázek č. 27). Hodnoty, které algoritmus neprošel, jsou interpolovány z hodnot známých. Grafy tedy nejsou tolik detailní, jako by byly při použití všech kombinací konstant. Při hodnocení těchto výsledků je tedy nutné mít na paměti, že se jedná pouze o přibližné zobrazení těchto závislostí.

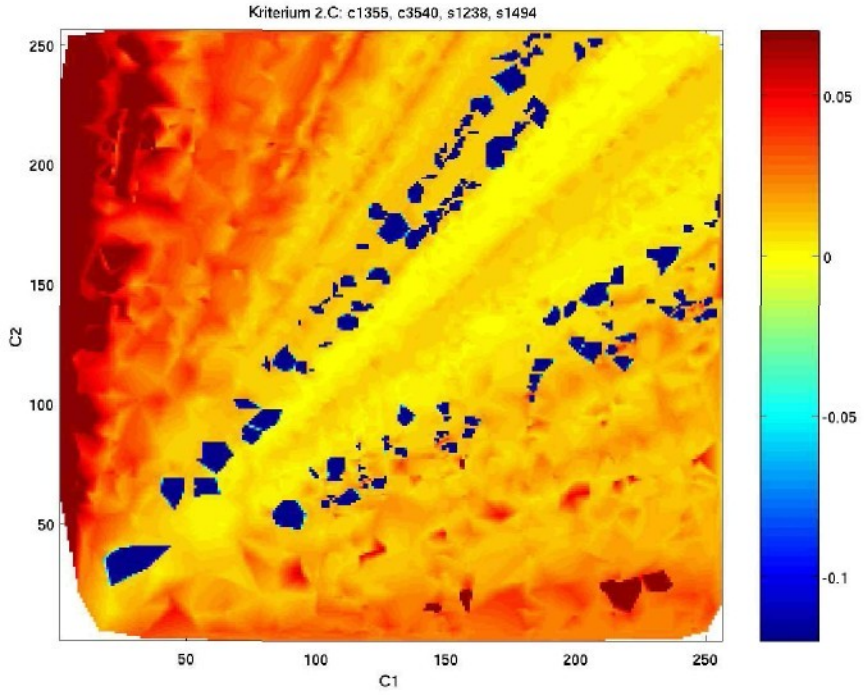
Na ose x je vynesen parametr C_1, na ose y parametr C_2. Barevnou škálou jsou označeny výsledné délky posloupností, respektive vyjádření těchto délek pomocí účelové funkce (zde však bez použitého posunutí do kladných čísel). Hodnoty záporné značí, že všechny výsledné délky pro tuto kombinaci konstant byly menší než délky s použitím kritéria 0, tato hodnota je tedy jejich průměrem. Hodnoty kladné potom značí průměrné odchylky směrem nahoru od délek výsledných posloupností s použitím kritéria 0. Jedná se přitom o výsledné délky normalizované (viz účelovou funkci, kapitola 6.3).

Při pohledu na tyto grafy si můžeme všimnout paprskovitého charakteru závislosti, který naznačuje, že hlavním faktorem, který ovlivňuje tuto závislost, je poměr konstant C_1 a C_2.

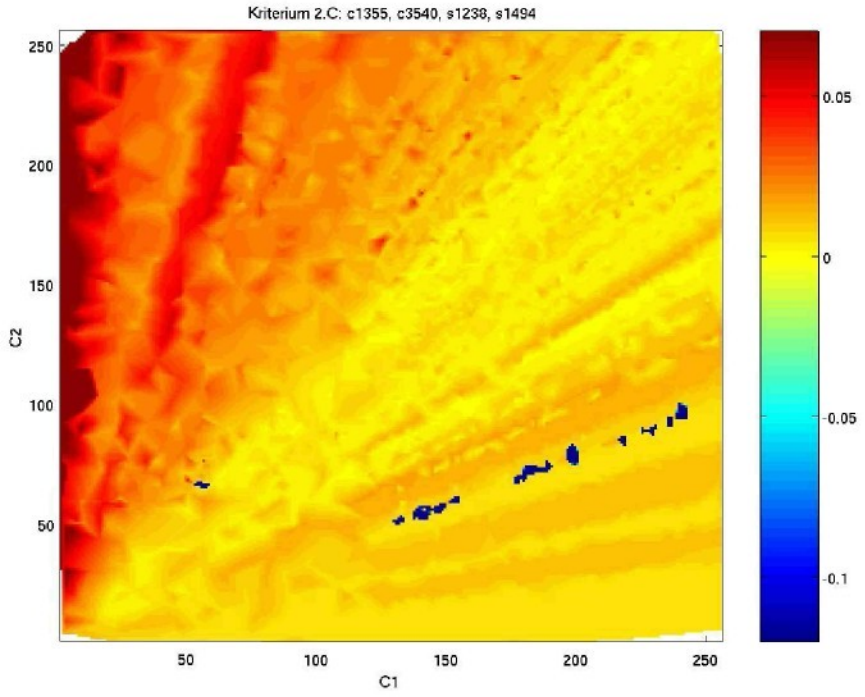
Steady
popsize:170
repnum:40
repper:0.24
nogen:1000
Px:0.9
Pm:0.08
terminator:20
candidates:201
compasexec:5514
minscore:8.86288



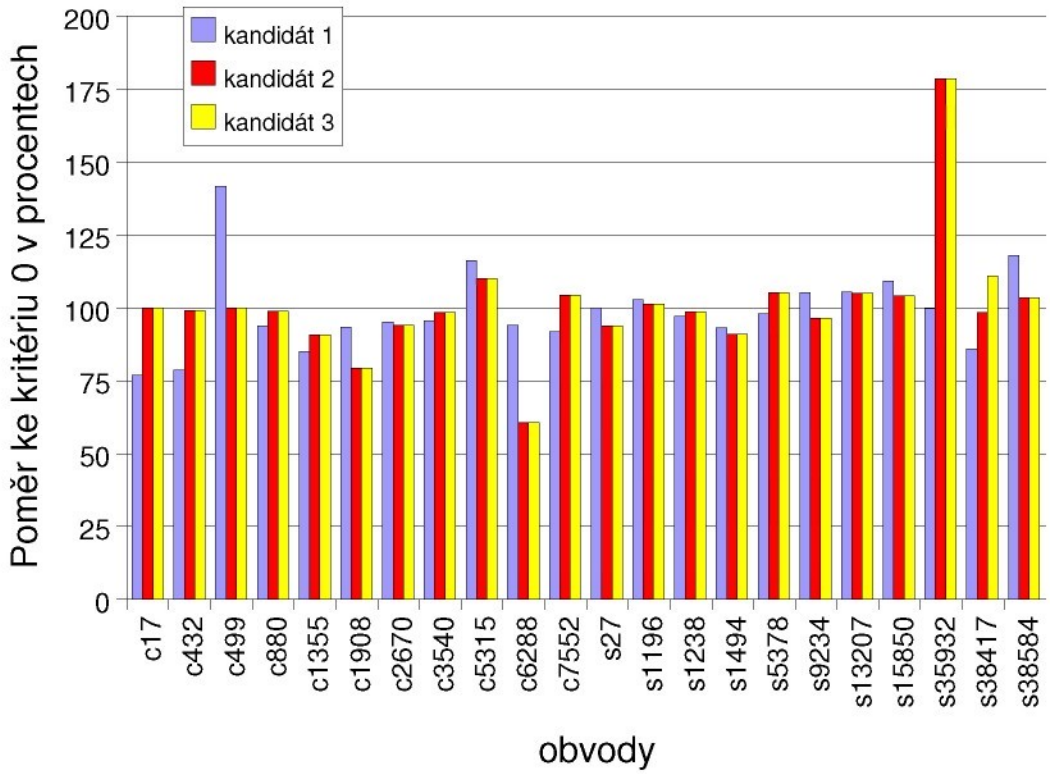
Obrázek 25: Průběh evoluce, kritérium 2.C, Stack



Obrázek 26: Grafické znázornění hodnot účelové funkce, kritérium 2.C, verze Stack



Obrázek 27: Grafické znázornění hodnot účelové funkce, kritérium 2.C, verze Stack.1



Obrázek 28: Porovnání výsledků kandidátů

Pro několik nalezených kandidátů (s použitím verze Stack.1) bylo provedeno ověření, zda s těmito kombinacemi konstant C_1 a C_2 dostaneme dobré výsledky napříč obvody. Výsledky s použitím třech vybraných kandidátů, tedy kombinací konstant C_1 a C_2, jsou uvedeny v tabulce č. 10.

Jak je ale patrné z grafu na obrázku č. 28, vytvořeného na základě těchto výsledků, použitím tohoto kritéria s těmito kombinacemi konstant C_1 a C_2 nezískáme lepší výsledky než s použitím kritéria 0 přes všechny obvody. Horší výsledky než s použitím kritéria 0 jsme dostali pro polovina obvodů. Nedá se tedy o těchto konstantách říci, že jsou univerzální. Navíc pro obvod s35932 dostaneme výsledky výrazně horší než s použitím kritéria 0.

Zhodnocení tohoto kritéria je uvedeno v kapitole č. 10.3.

obvody	kritérium 0	$C_1 = 56, C_2 = 66$	$C_1 = 235, C_2 = 92$	$C_1 = 130, C_2 = 50$
c17	13	10	13	13
c432	210	165	208	208
c499	259	367	259	259
c880	437	410	432	432
c1355	1115	947	1011	1011
c1908	1178	1099	934	934
c2670	4104	3907	3860	3860
c3540	703	672	692	692
c5315	1031	1198	1134	1134
c6288	84	79	51	51
c7552	6488	5953	6768	6768
s27	16	16	15	15
s1196	726	747	735	735
s1238	780	758	769	769
s1494	486	452	442	442
s5378	2047	2006	2153	2153
s9234	10871	11448	10491	10491
s13207	4082	4305	4285	4285
s15850	7037	7688	7326	7326
s35932	1937	1929	3458	3458
s38417	22411	19256	22039	24871
s38584	6783	7997	7013	7013

Tabulka 10: Porovnání vybraných kandidátů a výsledků s použitím kritéria 0

10.2 Kritérium 2.B

Trénovací data byla tedy získána s použitím tohoto kritéria, které bylo prvním dvouparametrovým kritériem. Jelikož bylo zkoumáno přibližně ve stejné době jako kritérium 1.10, bylo navrženo obdobně, a to v tomto tvaru:

$$crit = C_1 \cdot shift + C_2 \cdot no\_dontcares + 10 \cdot no\_globaldontcares$$

Jedná se opět v podstatě o zjemnění kritéria 2.C. Při použití rozsahu konstant 0 až 255 to odpovídá desetkrát jemnější oblasti kritéria 2.C s rozsahem konstant 0 až 25. Zobrazíme-li jeho výsledky opět graficky, jde o čtvercovou výseč oblasti kritéria 2.C od počátku ke konstantám $C_1 = 25$ a $C_2 = 25$.

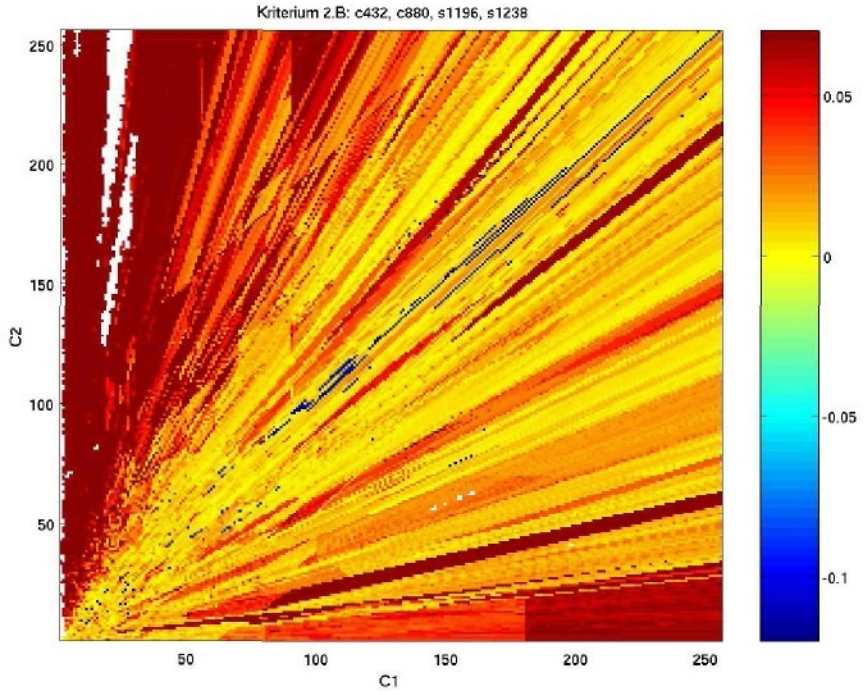
Grafické zobrazení závislosti účelové funkce na kombinaci konstant C_1 a C_2 (pro skupinu obvodů c432, c880, s1196 a s2138 a verzi Stack) vytvořené z trénovacích dat je na obrázku č. 29 (zde se jedná o detailní obraz této závislosti). Bílá místa v grafu odpovídají kombinacím konstant takovým, při jejichž použití pro jeden ze skupiny obvodů SW COMPAS vygeneroval posloupnost, která byla delší než pětinásobek délky posloupnosti s použitím kritéria 0. Při překročení této délky byl SW COMPAS ukončen. Toto opatření bylo zvoleno z toho důvodu, aby algoritmus nebyl „zdržován“ zbytečnými výpočty (v okamžiku, kdy je tato posloupnost významně delší než posloupnost s použitím kritéria 0, je zřejmé, že tato kombinace konstant nebude optimální).

Pro srovnání byla provedena i optimalizace pomocí GA pro stejnou skupinu obvodů a verzi Stack.1. Bylo nalezeno 343 kandidátů. Grafické znázornění přibližné závislosti účelové funkce na kombinaci konstant C_1, C_2 je na obrázku č. 30.

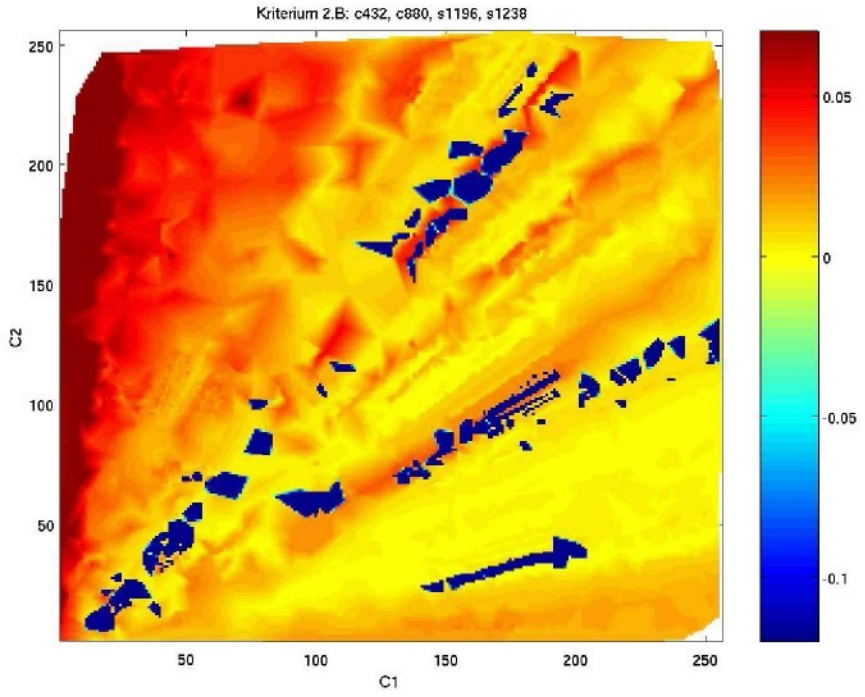
Pro optimalizaci pomocí GA byla zvolena opět tato skupina obvodů: c1355, c3540, s1238 a s1494. Byly použity verze Stack a Stack.1.

Zde se podařilo pomocí GA nalézt 249 kandidátů při použití verze Stack a 28 kandidátů při použití verze Stack.1.

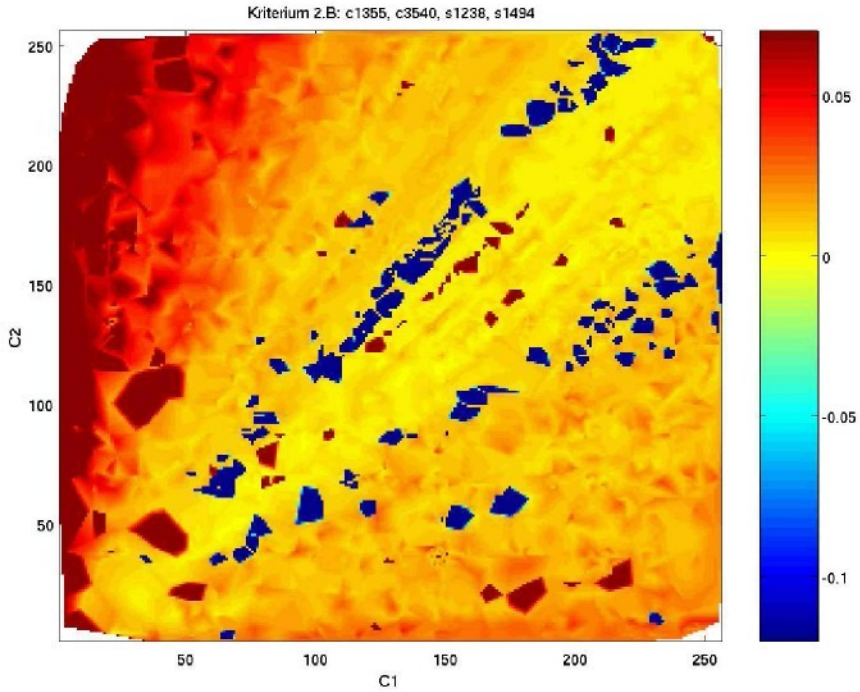
Grafické zobrazení přibližné závislosti účelové funkce na kombinaci konstant C_1 a C_2 je na obrázcích č. 31 a 32.



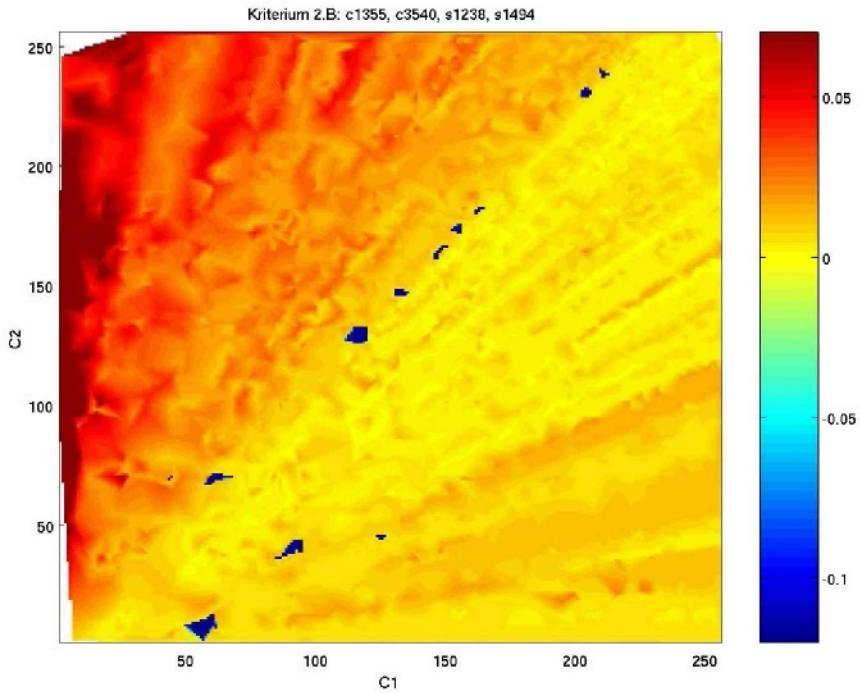
Obrázek 29: Detailní zobrazení závislosti, kritérium 2.B, verze Stack, trénovací obvody



Obrázek 30: Zobrazení přibližné závislosti, kritérium 2.B, verze Stack, trénovací obvody



Obrázek 31: Grafické znázornění hodnot účelové funkce, kritérium 2.B, verze Stack



Obrázek 32: Grafické znázornění hodnot účelové funkce, kritérium 2.B, verze Stack.1

10.3 Zhodnocení kritérií 2.C a 2.B

V první řadě si musíme všimnout faktu, že i když se nám podařilo nalézt kandidáty na kombinace konstant C_1 a C_2, nedostali jsme s jejich použitím vždy kratší délky výsledných testovacích posloupností pro všechny obvody, než jsou délky posloupností s použitím kritéria 0. Pro jedenáct z dvaceti dvou obvodů jsme dostali výsledné posloupnosti delší, přitom v nejhorším případě došlo ke zhoršení přibližně o 80%.

Dalším důležitým poznatkem je, že oblasti roviny C_1, C_2, kde se kandidáti nacházejí, jsou vždy v jiných místech této roviny, jak pro dvě testované skupiny obvodů, tak i pro různé verze SW COMPAS.

Není tedy možné pro tato kritéria určit univerzální kombinaci konstant C_1 a C_2.

Přesnější umístění kandidátů bychom dostali s použitím více obvodů ze sad ISCAS85 a ISCAS89, v tuto dobu však není možné provést vyhledávání kandidátů způsobem uvedeným v této práci, protože doba běhu algoritmu (za použití stejných HW prostředků) by byla přes sto dní (odhadnuto z průměrných délek doby běhu SW COMPAS pro všechny obvody použité v této práci). Urychlit toto hledání by bylo možné s použitím verze SW COMPAS s integrovaným simulátorem poruch (která však ještě není v tuto dobu k dispozici), nebo paralelizací genetického algoritmu.

Prohlédneme-li si tedy znovu grafy znázorňující závislosti účelové funkce na kombinacích konstant C_1 a C_2, všimněme si paprskovitého charakteru této závislosti a barevných škál v jednotlivých směrech. Nejčastěji se barvy odpovídající menším hodnotám účelové funkce vyskytují v oblasti diagonály směrem od počátku k vyšším hodnotám C_1 a C_2. Relativně dobré výsledky tedy můžeme získat použitím konstant C_1 a C_2, které mají stejnou hodnotu. Kritériu 2.C se stejnými hodnotami konstant C_1 a C_2 je však ekvivalentní kritérium 1 s jednou konstantou C.

Výběr kritéria je uveden v kapitole č. 11.

11 Výběr kritéria

Cílem této práce bylo nalézt takové kritérium, s jehož použitím budou výsledné posloupnosti (testovací sekvence, které jsou výstupem SW COMPAS) co nejkratší pro různé obvody. Toto kritérium mělo být současně použitelné i na neznámé obvody, mělo by být tedy univerzální a výsledky s jeho použitím stabilní i pro jiné obvody než obvody trénované.

Dvouparametrová kritéria v té podobě, v jaké byla zkoumána, není možné doporučit (viz kapitolu č. 10.3). Můžeme se také domnívat (z poznatků uvedených v kapitole č. 8.4), že optimální volba konstant C_1 a C_2 by mohla záviset na fyzických parametrech obvodu (v tom případě by nebylo možné najít optimální kombinaci konstant stejnou pro všechny obvody).

I když s použitím kritéria 1 a konstanty $C = 2$ jsme dostali lepší výsledky než s použitím kritéria 1.R, a to v poměru 10 ku 9 obvodům (z celkem 22 obvodů, pro dva obvody byly výsledky kritérií shodné), není možné kritérium 1 s použitím konstanty $C = 2$ nebo jiných menších konstant doporučit z důvodu nesplnění podmínky stability (viz kapitolu č. 9.2).

Kritérium 0 a kritérium 1.R doporučit lze, s jejich použitím jsou výsledky stabilní a dá se předpokládat, že tato kritéria budou univerzálně použitelná i pro jiné obvody než obvody v této práci používané. Vzhledem k délkám výsledných posloupností, které s použitím těchto dvou kritérií byly dosaženy, se však jako lepší jeví kritérium 1.R (viz kapitolu č. 9.2).

Jako nejvhodnější kritérium tedy bylo vybráno kritérium 1.R – kritérium s jednou konstantou, automaticky volenou podle počtu hradel obvodu. Vzhledem k tomu, že ne vždy je délka posloupnosti s použitím tohoto kritéria nejkratší možná, bylo by vhodné uživateli umožnit změnu hodnoty této konstanty, bude-li chtít.

12 Použité prostředky

12.1 Programové prostředky

12.1.1 COMPAS, Hope, Atalanta

COMPAS - Compressed Test Pattern Sequencer je software pro kompresi testovacích vektorů pro obvody se sériovým diagnostickým přístupem.

Tento software je stále vyvíjen na katedře KES pod vedením Prof. Ing. Ondřeje Nováka, CSc. Ve svých pracích se jím v poslední době zabývali Ing. Miroslav Holubec (diplomová práce, viz [6]), Ing. Jiří Jeníček (diplomová práce, viz [9]) a Ing. Jiří Zahrádka (diplomová práce, viz [14]). Další práce viz [10], [11], [15].

Program `Hope` je simulátor poruch synchronních sekvenčních číslicových obvodů. Jeho autory jsou Hyung K. Lee a Dr. Dong S. Ha z Virginia Polytechnic Institute & State University, [7]. Nekomprimované testy byly generovány programem `Atalanta`, což je generátor testů od stejných autorů. Tyto programy je možné volně používat pro nekomerční účely.

12.1.2 GAlib

V programu `POET` je použita knihovna komponent pro genetické algoritmy `GAlib`, napsaná v jazyce C++, autorem této knihovny je Matthew Wall z Mechanical Engineering Department, Messachusetts Institute of Technology. Tuto knihovnu je možné volně používat a modifikovat pro nekomerční účely. Také dle jejích licenčních podmínek lze její zdrojové kódy včetně dokumentace distribuovat, modifikace této knihovny lze distribuovat spolu s původní verzí knihovny. [1]

12.1.3 g++, Matlab, Open Office

Pro překlad zdrojových kódů v jazyce C++ byl použit kompilátor `g++`, který podléhá licenci GNU [2].

Předzpracování výstupních dat z programu COMPAS a `POET` bylo provedeno pomocí mnou napsaných skriptů pro interpretr `bash` a program `Matlab`. Grafy byly vytvořeny pomocí programu `Matlab` a balíku `Open Office`.

12.2 Hardwarové prostředky

Uvedené programy byly spouštěny na počítačích s těmito konfiguracemi:

- AMD Athlon XP 2500+, 1.8GHz, 768 MB RAM, operační systémy MS Windows XP, Linux Fedora Core 3, Linux Mandrake 9.2
- Intel Pentium 4, 2GHz, 256 MB RAM, OS MS Windows XP
- Intel Pentium 4, 3.2GHz, dvouprocesorový, 512 MB RAM, OS Red Hat Enterprise Linux

13 Závěr

Cílem diplomové práce bylo prozkoumat možnosti změn hodnot parametrů, které určují pořadí kódování testovacích vektorů systémem COMPAS, a dále navrhnout kritérium určující toto pořadí tak, aby při použití tohoto kritéria v softwaru COMPAS byly výsledné testovací sekvence pro různé obvody co nejkratší, a přitom aby výsledné kritérium s jeho parametry bylo použitelné i pro další neznámé obvody.

Bylo nutné se seznámit s problematikou generování testovacích sekvencí a výsledky práce týmu, zabývajícího se optimalizací generátoru těchto posloupností COMPAS. K dispozici bylo několik verzí softwaru COMPAS, obsahujících původní kritérium s jednou volitelnou konstantou, nebylo však úplně zřejmé, jakým způsobem její hodnotu volit. Dále bylo vycházeno z testovacích dat, která byla systémem COMPAS komprimována (kompresní poměr byl přitom ovlivňován změnami použitého kritéria a jeho parametrů).

Podařilo se prozkoumat navržená jednoparametrová kritéria a kritérium bez parametrů. Navržená dvouparametrová kritéria byla prozkoumána částečně, nebylo možné je prozkoumat zcela vyčerpávajícím způsobem, a to z důvodu velké výpočetní náročnosti softwaru COMPAS. Bylo doporučeno konkrétní kritérium pro použití v softwaru COMPAS.

V první části je stručný úvod do problematiky testování číslicových obvodů, dále popis softwaru COMPAS a úvod do genetických algoritmů.

V druhé části následuje stručný popis programu POET a nástin implementace jeho důležitých částí z pohledu genetických algoritmů. Dále je v této části uvedeno srovnání vybraných verzí softwaru COMPAS, analýza jednotlivých kritérií a návrh kritéria pro budoucí použití v tomto softwaru.

Pro optimalizaci parametrů kritérií s dvěma konstantami byl vytvořen program POET, čítající přibližně 1500 řádků v jazyce C++, což představuje asi 90kB kódu. V práci bylo popsáno šest kritérií, z toho pět nově navržených bylo prozkoumáno

a nejlepší z nich doporučeno – kritérium s automatickou volbou konstanty. Efektivita doporučeného kritéria byla ověřována na obvodech z benchmarkových sad ISCAS85 a ISCAS89.

Porovnají-li se výsledky dosažené s použitím tohoto doporučeného kritéria s výsledky dosaženými s použitím původního kritéria a konstanty $C = 2$, použití doporučeného kritéria má za následek zkrácení výsledných testovacích sekvencí u devíti z dvaceti dvou použitých obvodů. Prodloužení těchto sekvencí nastalo u deseti obvodů. K největšímu zkrácení testovací sekvence došlo u obvodu s38417 a to na 56% původní délky (při použití softwaru COMPAS ve verzi Stack). K největšímu prodloužení došlo u obvodu c880 a to na 135% délky původní testovací sekvence (při použití softwaru COMPAS ve verzi Stack.1). Toto kritérium však bylo doporučeno zejména na základě předpokladu jeho použitelnosti pro neznámé obvody. Navíc je možné, že nelze nalézt kritérium, s jehož použitím by všechny výsledné testovací sekvence byly kratší než s použitím kritéria původního.

Problém, kterému bylo nutné čelit, byla zejména omezená rychlost softwaru COMPAS, která ztěžovala možnosti optimalizace. Řešení tohoto problému nebylo předmětem této práce, zrychlováním softwaru COMPAS se zabývají další členové týmu a je pravděpodobné, že v blízké budoucnosti bude v této oblasti dosaženo pokroku, což by usnadnilo další optimalizaci kritérií.

Práce přispívá k lepšímu porozumění návrhu kritérií a jejich parametrů použitých v softwaru COMPAS. Zde navržené kritérium bude též možné použít v současných a budoucích verzích software COMPAS. Mimo jiné také práce ukazuje možnosti použití genetických algoritmů v problémech optimalizace.

Ukázalo se, že problematika zkoumání kritérií je velmi komplexní, je tedy možné dále v této práci pokračovat, zejména v oblasti dvouparametrových kritérií a závislostí délek výsledných posloupností na fyzických parametrech obvodů.

Reference

[1] GAlib homepage: `http://lancet.mit.edu/ga/`

[2] GNU/GPL licence: `http://www.gnu.org/licenses/gpl.cs.html`

[3] Herout, P.: *Učebnice jazyka C, 3. upravené vydání*, Kopp 1996, 269 stran

[4] Herout, P.: *Učebnice jazyka C, 2. díl, 3. upravené vydání*, Kopp 1996, 236 stran

[5] Hlavička J.: *Diagnostika a spolehlivost*, vydavatelství ČVUT 1998, 153 stran

[6] Holubec, M.: *Komprese vektorů pro obvody se sériovým diagnostickým přístupem*, diplomová práce, TU Liberec 2004, 64 stran

[7] Hope & Atalanta homepage: `http://www.ee.vt.edu/ha/cadtools/cadtools.html`

[8] Jaderný, J.: *Evoluční a genetické algoritmy*, ročníkový projekt, TU Liberec 2004, 43 stran

[9] Jeníček, J.: *Víceprocesorový algoritmus setřídění testovacích vektorů pro obvody se sériovým diagnostickým přístupem*, diplomová práce, TU Liberec 2005, 86 stran

[10] Novák, O. – Nosek, J.: *Test Pattern Decompression Using a Scan Chain*, proc. of 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 6 stran

[11] Novák, O. – Zahrádka, J.: *COMPAS – Compressed Test Patern Sekquencer for Scan Based Circuits*, Lecture Notes in Computer Science 3463, Springer 2005, str. 406–414

[12] Mařík, V. – Štěpánková, O – Lažanský, J.: *Umělá inteligence (3)*, Academia 2001, str. 117–159

[13] Virius, M.: *Od C k C++*, Kopp 2002, 227 stran

[14] Zahrádka, J.: *Optimalizace vestavěných generátorů testovacích vektorů*, diplomová práce, TU Liberec 2002

[15] Zahrádka, J. – Holubec, M. – Novák, O.: *COMPAS – System for Finding of a Compress Test Pattern Sequence*, proc. of ECMS Liberec 2003

Seznam obrázků

1 Schéma testu 21
2 Porovnání různých metod komprese testovacích vektorů 23
3 Výběr dalšího bitu testovací posloupnosti 25
4 Výběr dalšího bitu testovací posloupnosti – výpočet hodnoty kritéria . 28
5 Průběh algoritmu 34
6 Křížení 37
7 Mutace 37
8 Porovnání časů procentuelně, verze 3.2 vzhledem k verzi Stack 51
9 Porovnání časů procentuelně, Linux vzhledem k Windows 51
10 Porovnání délek procentuelně, verze 3.2 vzhledem k verzi Stack 52
11 Průběhy závislosti $L = f(C)$, verze Stack, kritérium 1 56
12 Průběhy závislosti $L = f(C)$, verze Stack.1, kritérium 1 57
13 Průběhy závislosti $L = f(C)$, verze 3.2, kritérium 1 58
14 Průběhy závislosti $L = f(C)$, verze 3.2.1, kritérium 1 59
15 Průběhy závislosti $L = f(C)$, verze 3.2, kritérium 1.10 60
16 Potenciální závislosti, všechny obvody, verze Stack.1 63
17 Potenciální závislosti, kombinační obvody, verze Stack.1 64
18 Potenciální závislosti, sekvenční obvody, verze Stack.1 64
19 Závislosti C_{ust} na celkovém počtu hradel 66
20 Srovnání výsledků kritérií, verze Stack 72
21 Srovnání výsledků kritérií, verze Stack.1 72
22 Grafický přehled sledovaných hodnot, Simple GA 77
23 Grafický přehled sledovaných hodnot, Deme GA 78
24 Grafický přehled sledovaných hodnot, Steady-state GA 79
25 Průběh evoluce, kritérium 2.C, Stack 81
26 Grafické znázornění hodnot účelové funkce, kritérium 2.C, verze Stack . 82

27 Grafické znázornění hodnot účelové funkce, kritérium 2.C, verze Stack.1 82
28 Porovnání výsledků kandidátů 83
29 Detailní zobrazení závislosti, kritérium 2.B, verze Stack, trénovací obvody 86
30 Zobrazení přibližné závislosti, kritérium 2.B, verze Stack, trénovací obvody 86
31 Grafické znázornění hodnot účelové funkce, kritérium 2.B, verze Stack . 87
32 Grafické znázornění hodnot účelové funkce, kritérium 2.B, verze Stack.1 87

Seznam tabulek

1 Porovnání s jinými metodami komprese testovacích vektorů 22
2 Porovnání časů, verze 3.2 a Stack 50
3 Výsledné délky posloupností, verze 3.2 a Stack 50
4 Ustálení průběhů – verze 3.2 a 3.2.1 54
5 Ustálení průběhů – verze Stack a Stack.1 54
6 Délky posloupností pro $C = 2500$ a $C = 2600$ (vycházejí stejně) 62
7 Délky posloupností pro kritérium 1.R 67
8 Výsledné délky, kritérium 0 70
9 Srovnání délek posloupností 71
10 Porovnání vybraných kandidátů a výsledků s použitím kritéria 0 84
11 Orientační časy, kritérium 1.R, verze Stack 100

Část III

Přílohy

A Přehled všech kritérií

A.1 Původní kritérium

Bylo nazváno kritérium 1, má tento tvar:

$$crit = (shift + no\_dontcares) \cdot C + no\_globaldontcares$$

Viz kapitoly č. 3.6.1, 3.6.2, 8.1 a 9.2.

A.2 Kritérium 1.10

Toto kritérium vypadá takto:

$$crit = (shift + no\_dontcares) \cdot C + 10 \cdot no\_globaldontcares$$

Viz kapitolu č. 8.2.

A.3 Kritérium 1.R

Toto kritérium má tvar:

$$crit = (shift + no\_dontcares) \cdot (k \cdot no\_gates + q) + no\_globaldontcares$$

Viz kapitolu č. 8.4 a 9.2.

A.4 Kritérium 0

Má tento tvar:

$$crit = shift + no\_dontcares$$

Viz kapitolu č. 9.1 a 9.2.

A.5 Kritérium 2.B

Má tento tvar:

$$crit = C_1 \cdot shift + C_2 \cdot no\_dontcares + 10 \cdot no\_globaldontcares$$

Viz kapitolu č. 10.2 a 10.3.

A.6 Kritérium 2.B

Toto kritérium má tvar:

$$crit = C_1 \cdot shift + C_2 \cdot no\_dontcares + no\_globaldontcares$$

Viz kapitolu č. 10.1 a 10.3.

B Doby běhu SW COMPAS

Zde jsou uvedenu orientační doby běhu SW COMPAS ve verzi Stack s použitým kritérium 1.R. Časy byly měřeny na na počítači s touto konfigurací: AMD Athlon XP 2500+, 1.8GHz, 768 MB RAM, OS Linux Fedora Core 3.

obvod	časy [H:MM:SS]	obvod	časy [H:MM:SS]
c17	0:00:01	s27	0:00:01
c432	0:00:01	s1196	0:00:03
c499	0:00:01	s1238	0:00:03
c880	0:00:02	s1494	0:00:03
c1355	0:00:04	s5378	0:00:31
c1908	0:00:06	s9234	0:06:21
c2670	0:00:31	s13207	0:05:0
c3540	0:00:06	s15850	0:08:53
c5315	0:00:17	s35932	0:08:38
c6288	0:00:01	s38417	1:18:25
c7552	0:02:08	s38584	0:25:11

Tabulka 11: Orientační časy, kritérium 1.R, verze Stack

C Implementace účelové funkce

Zde je uveden výpis implementace účelové funkce, její popis je v kapitole č. 6.3.

```
float objective(GAGenome &g)
{
  GABin2DecGenome &genome=(GABin2DecGenome &) g;
  float score=0.0,sum1,sum2,tmp;
  int i,ii,j,lessthanone;
  unsigned int l;
  long int r;
  bool isone;
  bool savedata;
  lessthanone=0;
  sum1=0.0;
  sum2=0.0;

  float p[MAX_N_OF_CIRCUITS];
  C_container *cc;
  C_TYPE C[C_CNT];
  for(ii=0;ii<C_CNT;ii++) C[ii]=(C_TYPE) genome.phenotype(ii);

  savedata=false;

  for(j=0;j<ncircuits;j++)
    {

      if (!conf.isdataset(j,C[0],C[1]))
        {
          compasrun++;
          savedata=true;
          strcpy(circuitname,conf.getname(j));
          r=run_compas(C,conf.getmaxlength(j));
          switch(r)
            {
            case 0:
              l=analyze_output(conf.getnobitsfname(j));
              conf.setnumber(j,C[0],C[1],l);
              break;
            case 255:
              //NaN
```

```
                l=0;
                conf.setNaN(j,C[0],C[1]);
                break;
              default:
                //Err
                l=0;
                conf.setErr(j,C[0],C[1]);
              }
          }
        else l=getlength(j,C);

        if (l==0) l=conf.getmaxlength(j);

        p[j]=float(l) / float(conf.getnormalizator(j));
        isone=false;
        if (p[j] <= 1 )
          {
            lessthanone++;
            sum1+=p[j];
            p[j]=1.0;
            isone=true;
          }
        if ((p[j]==1.0) || isone) tmp=0;
        else
          {
            tmp=(1.0 - p[j]);
            tmp=tmp*tmp;
          }

        sum2+=tmp;
      }
    if (lessthanone==ncircuits)
        score = SHIFT - (float(ncircuits) / sum1);
    else score = SHIFT + sum2/float(ncircuits);

    if (score<SHIFT)
    {
      if (!isinlist(C))
        {
        cc=new C_container(C);
```

```
        Clist.insert(Clist.begin(),*cc);
        ngen_inserted=gahelp->generation();
        candidates++;
        }
    }
    if ((savedata) && (savetemp)) conf.saveonedata(C[0],C[1]);

    return score;
}
```

D Přehled parametrů programu POET

Tyto parametry je možné programu POET při jeho spuštění předat příkazovou řádkou:

`compasmax N` – počet spuštění SW COMPAS, při jehož překročení bude program POET ukončen

`savetemp N` – pokud $N = 1$, program ukládá kombinace konstant, které prošel, do souboru `název_obvodu.probe`.

`ngen N` – maximální počet generací

`nconv N` – počet generací, po...

`pcross N` – pravděpodobnost křížení

`pmut N` – pravděpodobnost mutace

`popsize N` – velikost populace

`npop N` – počet populací

`prepl N` – procento obměny populace

`nrepl N` – počet obměňovaných jedinců

`el N` – při zadání hodnoty $N = 1$ je použit elitismus

`pmig N` – procento jedinců z populace migrujících

`nmig N` – počet migrujících jedinců z populace

E Obsah přiloženého CD

Na CD jsou tyto adresáře: Diplomová práce, Zdrojové kódy, Program POET.

E.1 Adresář Diplomová práce

V tomto adresáři se nachází elektronická verze této diplomové práce.

E.2 Adresář Zdrojové kody

V tomto adresáři se nacházejí zdrojové kódy programu POET (adresář POET) a části knihovny GAlib (jeho podadresář GA). Celé zdrojové kódy knihovny GAlib včetně dokumentace jsou v podadresáři GAlib.

E.3 Adresář Program POET

V tomto adresáři se nacházejí spustitelné verze programů pro OS Linux, a to program POET (verze se Simple GA, Deme GA a Steady-state GA), SW COMPAS ve verzi Stack s použitím kritéria 2.C a simulátor poruch Hope. V podadresáři Testdata jsou soubory se strukturami obvodů použitých v této práci (`*.bench` a soubory s nekomprimovanými testy `*.test`). Pro každé spuštění programu POET, COMPAS nebo Hope, je třeba mít tyto soubory ve stejném adresáři, jako jsou tyto programy.

GAlib: A C++ Library of Genetic Algorithm Components

version 2.4

Documentation Revision B

August 1996

Matthew Wall

Mechanical Engineering Department

Massachusetts Institute of Technology

http://lancet.mit.edu/ga/
galib-request@mit.edu

GAlib is a C++ library of genetic algorithm objects. The library includes tools for using genetic algorithms to do optimization in any C++ program using any representation and any genetic operators. This documentation includes an extensive overview of how to implement a genetic algorithm, the programming interface for GAlib classes, and examples illustrating customizations to the GAlib classes.

This work was supported by the Leaders for Manufacturing Program

GAlib: A C++ Library of Genetic Algorithm Components i

Licensing and Copyright Issues 1

GAlib For-profit User/Distributor License Agreement 1
GAlib Not-for-profit User License Agreement 1
The GNU portions of the GAlib distribution 1
The standard MIT copyright notice and disclaimer 2

Features 3

General Features 3
Algorithms, Parameters, and Statistics 3
Genomes and Operators 3

Overview 5

The Genetic Algorithm 6
Defining a Representation 7
The Genome Operators 7
The Population Object 8
Objective Functions and Fitness Scaling 8
So what does it look like in C++? 9
What can the operators do? 10
How do I define my own operators? 11
What about deriving my own genome class? 13

Class Hierarchy 15

GAlib Class Hierarchy - Pictorial 15
GAlib Class Hierarchy - Outline 16

Programming Interface 17

Global Typedefs and Enumerations 17
Global Variables and Global Constants 17
Function Prototypes 17
Parameter Names and Command-Line Options 18
Error Handling 20
Random Number Functions 21

GAGeneticAlgorithm 22
GADemeGA 27
GAIncrementalGA 29
GASimpleGA 31
GASteadyStateGA 32
Terminators 34
Replacement Schemes 35

GAGenome 36
GA1DArrayGenome<T> 39
GA1DArrayAlleleGenome<T> 41
GA2DArrayGenome<T> 42
GA2DArrayAlleleGenome<T> 44
GA3DArrayGenome<T> 45
GA3DArrayAlleleGenome<T> 47
GA1DBinaryStringGenome 48
GA2DBinaryStringGenome 50
GA3DBinaryStringGenome 52
GABin2DecGenome 54
GAListGenome<T> 56
GARealGenome 57
GAStringGenome 58
GATreeGenome<T> 59

GAEvalData 60
GABin2DecPhenotype 61
GAAlleleSet<T> 62
GAAlleleSetArray<T> 64

GAParameter and GAParameterList 65
GAStatistics 67
GAPopulation 70
GAScalingScheme 75
GASelectionScheme 77

GAArray<T> 79
GABinaryString 81
GAList<T> and GAListIter<T> 82
GATree<T> and GATreeIter<T> 85

Customizing GAlib **89**

Deriving your own genome class 89
Genome Initialization 91
Genome Mutation 91
Genome Crossover 92
Genome Comparison 92
Genome Evaluation 93
Population Initialization 93
Population Evaluation 93
Scaling Scheme 93
Selection Scheme 94
Genetic Algorithm 96
Termination Function 96

Descriptions of the Examples **98**

Licensing and Copyright Issues

The GAlib source code is not in the public domain, but it is available at no cost for non-profit purposes. If you would like to use GAlib for commercial purposes, for-profit single user and distributor licenses are available. All of GAlib (source and documentation) is protected by the Berne Convention. You may copy and modify GAlib, but by doing so you agree to the terms of the not-for-profit license.

GAlib For-profit User/Distributor License Agreement

Please contact the MIT Technology Licensing Office at 617.253.6966 or tlo@mit.edu.

GAlib Not-for-profit User License Agreement

1. You may copy and distribute copies of the source code for GAlib in any medium provided that you consipicuously and appropriately give credit to the author and keep intact all copyright and disclaimer notices in the library.

2. You may modify your copy (copies) of GAlib or any portion thereof, but you may not distribute modified versions of GAlib. You may distribute patches to the original GAlib as separate files along with the original GAlib.

3. You may not charge anything for copies of GAlib beyond a fair estimate of the cost of media and computer/network time required to make and distribute the copies.

4. Incorporation of GAlib or any portion thereof into commercial software, distribution of GAlib for-profit, or use of GAlib for other for-profit purposes requires a special agreement with the the MIT technology licensing office (TLO).

5. Any publications of work based upon experiments that use GAlib must include a suitable acknowledgement of GAlib. A suggested acknowledgement is: "The software for this work used the GAlib genetic algorithm package, written by Matthew Wall at the Massachusetts Institute of Technology."

6. The author of GAlib and MIT assume absolutely no responsibility for the use or misuse of GAlib. In no event shall the author of GAlib or MIT be liable for any damages resulting from use or performance of GAlib.

The GNU portions of the GAlib distribution

The portions of GAlib (see below) that contain code from the GNU g++ library are covered under the terms of the GNU Public License. As such they are freely available and do not fall under the terms of the GAlib licensing conditions above. The portions of GAlib that are based upon GNU code are all in the 'gnu' directory in the examples directory (in GAlib release 2.3.2 and later).

The standard MIT copyright notice and disclaimer

As a work developed using MIT resources and MIT funding, the GAlib source code copyright is owned by the Massachusetts Institute of Technology. All rights are reserved.

Features

General Features

- Many examples are included illustrating the use of various GAlib features, class derivations, parallelization, deterministic crowding, travelling salesman, DeJong, and Royal Road problems.
- The library has been used on various DOS/Windows, Windows NT/95, MacOS, and UNIX configurations. GAlib compiles without warnings on most major compilers.
- Templates are used in some genome classes, but GAlib can be used without templates if your compiler does not understand them.
- Four random number generators are included with the library. You can select the one most appropriate for your system, or use your own.

Algorithms, Parameters, and Statistics

- GAlib can be used with PVM (parallel virtual machine) to evolve populations and/or individuals in parallel on multiple CPUs.
- Genetic algorithm parameters can be configured from file, command-line, and/or code.
- Overlapping (steady-state GA) and non-overlapping (simple GA) populations are supported. You can also specify the amount of overlap (% replacement). The distribution includes examples of other derived genetic algorithms such as a genetic algorithm with sub-populations and another that uses deterministic crowding.
- New genetic algorithms can be quickly tested by deriving from the base genetic algorithm classes in the library. In many cases you need only overide one virtual function.
- Built-in termination methods include convergence and number-of-generations. The termination method can be customized for any existing genetic algorithm class or for new classes you derive.
- Speciation can be done with either DeJong-style crowding (using a replacement strategy) or Goldberg-style sharing (using fitness scaling).
- Elitism is optional for non-overlapping genetic algorithms.
- Built-in replacement strategies (for overlapping populations) include replace parent, replace random, replace worst. The replacement operator can be customized.
- Built-in selection methods include rank, roulette wheel, tournament, stochastic remainder sampling, stochastic uniform sampling, and deterministic sampling. The selection operator can be customized.
- "on-line" and "off-line" statistics are recorded as well as max, min, mean, standard deviation, and diversity. You can specify which statistics should be recorded and how often they should be flushed to file.

Genomes and Operators

- Chromosomes can be built from any C++ data type. You can use the types built-in to the library (bit-string, array, list, tree) or derive a chromosome based on your own objects.

Features: Genomes and Operators

- Built-in chromosome types include real number arrays, list, tree, 1D, 2D, and 3D arrays, 1D, 2D, and 3D binary string. The binary strings, strings, and arrays can be variable length. The lists and trees can contain any object in their nodes. The array can contain any object in each element.
- All chromosome initialization, mutation, crossover, and comparison methods can be customized.
- Built-in initialization operators include uniform random, order-based random, and initialize-to-zero.
- Built-in mutation operators include random flip, random swap, Gaussian, destructive, swap subtree, swap node.
- Built-in crossover operators include partial match, ordered, cycle, single point, two point, even, odd, uniform, node- and subtree-single point.
- Dominance and Diploidy are not explicitly built in to the library, but any of the genome classes in the library can easily be extended to become diploid chromosomes.
- Objective function
- Objective functions can be population- or individual-based.
- If the built-in genomes adequately represent your problem, a user-specified objective function is the only problem-specific code that must be written.

Overview

This document outlines the contents of the library and presents some of the design philosophy behind the implementation. Some source code samples are provided at the end of the page to illustrate basic program structure, operator capabilities, operator customization, and derivation of new genome classes.

When you use the library you will work primarily with two classes: a genome and a genetic algorithm. Each genome instance represents a single solution to your problem. The genetic algorithm object defines how the evolution should take place. The genetic algorithm uses an objective function (defined by you) to determine how 'fit' each genome is for survival. It uses the genome operators (built into the genome) and selection/replacement strategies (built into the genetic algorithm) to generate new individuals.

There are three things you must do to solve a problem using a genetic algorithm:

1. Define a representation
2. Define the genetic operators
3. Define the objective function

GAlib helps you with the first two items by providing many examples and pieces from which you can build your representation and operators. In many cases you can use the built-in representations and operators with little or no modification. The objective function is completely up to you. Once you have a representation, operators, and objective measure, you can apply any genetic algorithm to find better solutions to your problem.

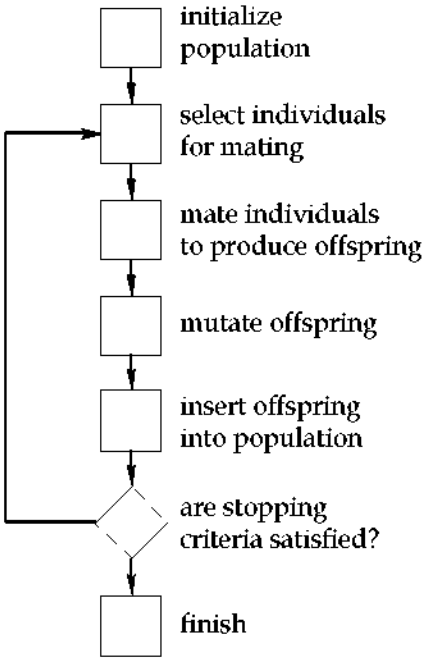
When you use a genetic algorithm to solve an optimization problem, you must be able to represent a single solution to your problem in a single data structure. The genetic algorithm will create a population of solutions based on a sample data structure that you provide. The genetic algorithm then operates on the population to evolve the best solution. In GAlib, the sample data structure is called a GAGenome (some people refer to it as a chromosome). The library contains four types of genomes: GAListGenome, GATreeGenome, GAArrayGenome, and GABinaryStringGenome. These classes are derived from the base GAGenome class and a data structure class as indicated by their names. For example, the GAListGenome is derived from the GAList class as well as the GAGenome class. Use a data structure that works with your problem definition. For example, if you are trying to optimize a function that depends on 5 real numbers, then use as your genome a 1-dimensional array of floats with 5 elements.

There are many different types of genetic algorithms. GAlib includes three basic types: 'simple', 'steady-state', and 'incremental'. These algorithms differ in the way that they create new individuals and replace old individuals during the course of an evolution.

GAlib provides two primary mechanisms for extending the capabilities of built-in objects. First of all (and most preferred, from a C++ point of view), you can derive your own classes and define new member functions. If you need to make only minor adjustments to the behavior of a GAlib class, in most cases you can define a single function and tell the existing GAlib class to use it instead of the default.

Genetic algorithms, when properly implemented, are capable of both exploration (broad search) and exploitation (local search) of the search space. The type of behavior you'll get depends on how the operators work and on the 'shape' of the search space.

The Genetic Algorithm



The genetic algorithm object determines which individuals should survive, which should reproduce, and which should die. It also records statistics and decides how long the evolution should continue. Typically a genetic algorithm has no obvious stopping criterion. You must tell the algorithm when to stop. Often the number-of-generations is used as a stopping measure, but you can use goodness-of-best-solution, convergence-of-population, or any problem-specific criterion if you prefer.

The library contains four flavors of genetic algorithms. The first is the standard 'simple genetic algorithm' described by Goldberg in his book. This algorithm uses non-overlapping populations and optional elitism. Each generation the algorithm creates an entirely new population of individuals. The second is a 'steady-state genetic algorithm' that uses overlapping populations. In this variation, you can specify how much of the population should be replaced in each generation. The third variation is the 'incremental genetic algorithm', in which each generation consists of only one or two children. The incremental genetic algorithms allow custom replacement methods to define how the new generation should be integrated into the population. So, for example, a newly generated child could replace its parent, replace a random individual in the population, or replace an individual that is most like it. The fourth type is the 'deme' genetic algorithm. This algorithm evolves multiple populations in parallel using a steady-state algorithm. Each generation the algorithm migrates some of the individuals from each population to one of the other populations.

In addition to the basic built-in types, GAlib defines the components you'll need to derive your own genetic algorithm classes. The examples include a few of these derivations including (1) a genetic algorithm that uses multiple populations and 'migration' between populations on multiple CPUs, and (2) a genetic algorithm that does 'deterministic crowding' to maintain different species of individuals during the evolution.

The base genetic algorithm class contains operators and data common to most flavors of genetic algorithms. When you derive your own genetic algorithm you can use these member data and functions to keep track of statistics and monitor performance.

The genetic algorithm contains the statistics, replacement strategy, and parameters for running the algorithm. the population object, a container for genomes, also contains some statistics as well as selection and scaling operators. A typical genetic algorithm will run forever. The library has built in functions for specifying when the algorithm should terminate. These include terminate-upon-generation, in which you specify a certain number of generations for which the algorithm should run, and terminate-upon-convergence, in which you specify a value to which the best-of-generation score should converge. You can customize the termination function to use your own stopping criterion.

The number of function evaluations is a good way to compare different genetic algorithms with various other search methods. The GAlib genetic algorithms keep track of both the number of genome evaluations and population evaluations.

Defining a Representation

Use a data structure that is appropriate for your problem. If you are optimizing a function of real numbers, use real numbers in your genome. If a solution to your problem can be represented with some imaginary numbers and some integer values, define a genome with these characteristics.

Defining an appropriate representation is part of the art of using genetic algorithms (and at this point, it is still an art, not a science). Use a representation that is minimal but completely expressive. Your representation should be able to represent any solution to your problem, but if at all possible you should design it so that it cannot represent infeasible solutions to your problem. Remember that if the genome can represent infeasible solutions then the objective function must be designed to give partial credit to infeasibles.

The representation should not contain information beyond that needed to represent a solution to the problem. Although there may be merit in using a representation that contains 'extra' genetic material, unless properly implemented (in concert with the objective function and in full consideration of the type and characteristics of the search space), this tends to increase the size of the search space and thus hinder the performance of the genetic algorithm.

The number of possible representations is endless. You may choose a purely numeric representation such as an array of real numbers. These could be implemented as real numbers, or, in the Goldberg-style of a string of bits that map to real numbers (beware that using real numbers directly far out-performs the binary-to-decimal representation for most problems, especially when you use reasonable crossover operators). Your problem may depend on a sequence of items, in which case an order-based representation (either list or array) may be more appropriate. In many of these cases, you must choose operators that maintain the integrity of the sequence; crossover must generate reordered lists without duplicating any element in the list. Other problems lend themselves to a tree structure. Here you may want to represent solutions explicitly as trees and perform the genetic operations on the trees directly. Alternatively, many people encode trees into an array or parsable string, then operate on the string. Some problems include a mix of continuous and discrete elements, in which case you may need to create a new structure to hold the mix of information. In these cases you must define genetic operators that respect the structure of the solution. For example, a solution with both integer and floating parts might use a crossover that crosses integer parts with integer parts and floating parts with floating parts, but never mixes floating parts with integer parts.

Whichever representation you choose, be sure to pick operators that are appropriate for your representation.

The Genome Operators

Each genome has three primary operators: initialization, mutation, and crossover. With these operators you can bias an initial population, define a mutation or crossover specific to your problem's representation, or evolve parts of the genetic algorithm as your population evolves. GAlib comes with these operators pre-defined for each genome type, but you can customize any of them.

The initialization operator determines how the genome is initialized. It is called when you initialize a population or the genetic algorithm. This operator does not actually create new genomes, rather it 'stuffs' the genomes with the primordial genetic material from which all solutions will evolve. The population object has its own initialization operator. By default this simply calls the initialization operators of the genomes in the population, but you can customize it to do whatever you want.

The mutation operator defines the procedure for mutating each genome. Mutation means different things for different data types. For example, a typical mutator for a binary string genome flips the bits in the string with a given probability. A typical mutator for a tree, on the other hand, would swap subtrees with a given probability. In general, you should define a mutation that can do both exploration

and exploitation; mutation should be able to introduce new genetic material as well as modify existing material. You may want to define multiple types of mutation for a single problem.

The crossover operator defines the procedure for generating a child from two parent genomes. Like the mutation operator, crossover is specific to the data type. Unlike mutation, however, crossover involves multiple genomes. In GAlib, each genome 'knows' its preferred method of mating (the default crossover method) but it is incapable of performing crossover itself. Each genetic algorithm 'knows' how to get the default crossover method from its genomes then use that method to peform the mating. With this model it is possible to derive new genetic algorithm classes that use mating methods other than the defaults defined for a genome.

Each of these methods can be customized so that it is specific not only to the data type, but also to the problem type. This is one way you can put some problem-specific 'intelligence' into the genetic algorithm (I won't go into a discussion about whether or not this is a good thing to do...)

In addition to the three primary operators, each genome must also contain an objective function and may also contain a comparator. The objective function is used to evaluate the genome. The comparator (often referred to as a 'distance function') is used to determine how different one genome is from another. Every genetic algorithm requires that an objective function is defined - this is how the genetic algorithm determines which individuals are better than others. Some genetic algorithms require a comparator.

The library has some basic data types built in, but if you already have an array or list object, for example, then you can quickly build a genome from it by multiply inheriting from your object and the genome object. You can then use this new object directly in the GAlib genetic algorithm objects.

In general, a genetic algorithm does not need to know about the contents of the data structures on which it is operating. The library reflects this generality. You can mix and match genome types with genetic algorithms. The

genetic algorithm knows how to clone genomes in order to create populations, initialize genomes to start a run, cross genomes to generate children, and mutate genomes. All of these operations are performed via the genome member functions.

The Population Object

The population object is a container for genomes. Each population object has its own initializer (the default simply calls the initializer for each individual in the population) and evaluator (the default simply calls the evaluator for each individual in the population). It also keeps track of the best, average, deviation, etc for the population. Diversity can be recorded as well, but since diversity calculations often require a great deal of additional compuation, the default is to not record diversity.

The selection method is also defined in the population object. This method is used by the genetic algorithms to choose which individuals should mate.

Each population object has a scaling scheme object associated with it. The scaling scheme object converts the objective score of each genome to a fitness score that the genetic algorithm uses for selection. It also caches fitness information for use later on by the selection schemes.

Objective Functions and Fitness Scaling

Genetic algorithms are often more attractive than gradient search methods because they do not require compilicated differential equations or a smooth search space. The genetic algorithm needs only a single measure of how good a single individual is compared to the other individuals. The objective function provides this measure; given a single solution to a problem, how good is it?

It is important to note the distinction between fitness and objective scores. The objective score is the value returned by your objective function; it is the raw performance evaluation of a genome. The fitness score, on the other hand, is a possibly-transformed rating used by the genetic algorithm to determine the fitness of individuals for mating. The fitness score is typically obtained by a linear scaling of the raw objective scores (but you can define any mapping you want, or no transformation at all). For example, if you use linear scaling then the fitness scores are derived from the objective scores using the fitness proportional scaling technique described in Goldberg's book. The genetic algorithm uses the fitness scores, not the objective scores, to do selection.

You can evaluate the individuals in a population using an individual-based evaluation function (which you define), or a population-based evaluator (also which you define). If you use an individual-based objective, then the function is assigned to each genome. A population-based objective function can make use of individual objective functions, or it can set the individual scores itself.

So what does it look like in C++?

A typical optimization program has the following form. This example creates a one-dimensional binary string genome with the default operators then uses a simple genetic algorithm to do the evolution.

```
float Objective(GAGenome&);

main(){
  GA1DBinaryStringGenome genome(length, Objective);    // create a genome
  GASimpleGA ga(genome);          // create the genetic algorithm
  ga.evolve();                          // do the evolution
  cout << ga.statistics() << endl;      // print out the results
}

float Objective(GAGenome&) {
  // your objective function goes here
}
```

You can very easily change the behaviour of the genetic algorithm by setting various parameters. Some of the more common ones are set like this:

```
ga.populationSize(popsize);
ga.nGenerations(ngen);
ga.pMutation(pmut);
ga.pCrossover(pcross);
GASigmaTruncationScaling sigmaTruncation;
ga.scaling(sigmaTruncation);
```

Alternatively you can have GAlib read the genetic algorithm options from a file or from the command line. This snippet creates a genetic algorithm, reads the parameters from a file, reads parameters (if any) from the command line, performs the evolution, then prints out the statistics from the run.

```
GASteadyStateGA ga(genome);
ga.parameters("settings.txt");
ga.parameters(argc, argv);
ga.evolve();
cout << ga.statistics() << endl;
```

A typical (albeit simple) objective function looks like this (this one gives a higher score to a binary string genome that contains all 1s):

```
float
Objective(GAGenome & g) {
  GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)g;
  float score=0.0;
  for(int i=0; i<genome.length(); i++)
    score += genome.gene(i);
  return score;
```

```
}
```

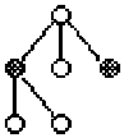
You can define the objective function as a static member of a derived class, or just define a function and use it with the existing GAlib genome classes.

When you write an objective function, you must first cast the generic genome into the type of genome that your objective function is expecting. From that point on you can work with the specific genome type. Each objective function returns a single value that represents the objective score of the genome that was passed to the objective function.

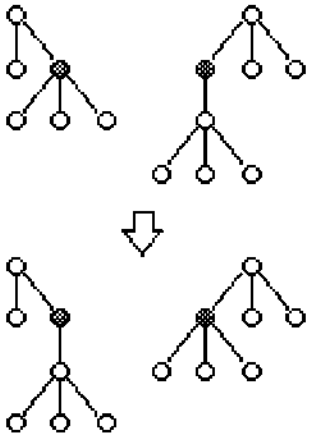
Please see the examples for more samples of the library in action. And see the programming interface page for a complete list of member functions and built-in operators.

What can the operators do?

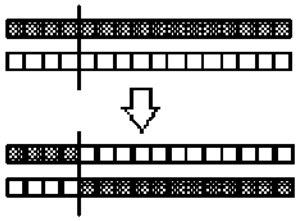
Here are some examples of the types of mutation and crossover that can be done using GAlib. Traditional crossover generates two children from two parents, and mutation is typically applied to a single individual. However, many other types of crossover and mutation are possible, such as crossover using three or more parents, asexual crossover or population-based mutation. The following examples illustrate some of the standard, sexual crossover and individual mutation methods in GAlib.



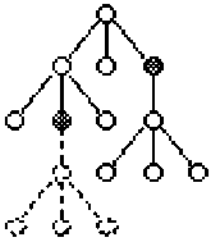
tree node swap mutation



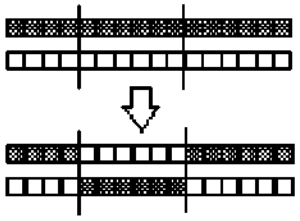
tree one point crossover



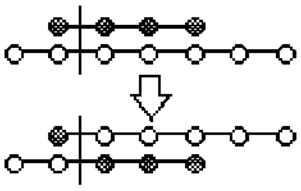
array one point crossover



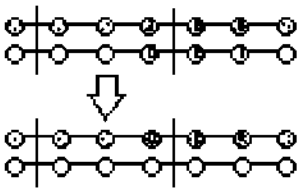
tree sub-tree swap mutation



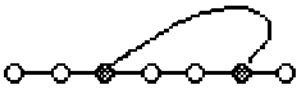
array two point crossover



list one point crossover

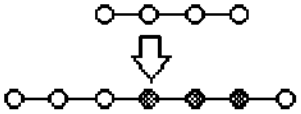


list order crossover

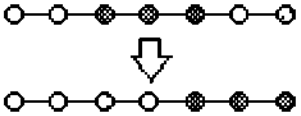


list node swap mutation

list destructive mutation



list generative mutation



list sequence swap mutation

How do I define my own operators?

Defining the operators is only as difficult as figuring out the algorithm you want to implement. As far as the actual implementation goes, there's not much to it. To assign an operator to a genome, just use the appropriate member function. For example, the following code snippet assigns 'MyInitializer' as the initialization function and 'MyCrossover' as the crossover function for a binary string genome.

```
GA1DBinaryStringGenome genome(20);
genome.initializer(MyInitializer);
genome.crossover(MyCrossover);
```

If you do this to the first genome (the one you use to create the genetic algorithm) then all of the ones that the GA clones will have the same operators defined for them.

When you derive your own genome class you will typically hard-code the operators into the genome like this:

```
class MyGenome : public GAGenome {
public:
  static void RandomInitializer(GAGenome&);
  static int JuggleCrossover(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
  static int KillerMutate(GAGenome&, float);
  static float ElementComparator(const GAGenome&, const GAGenome&);
  static float ThresholdObjective(GAGenome&);

public:
  MyGenome() {
    initializer(RandomInitializer);
    crossover(JuggleCrossover);
    mutator(KillerMutate);
    comparator(ElementComparator);
    evaluator(ThresholdObjective);
  }

  // remainder of class definition here

};
```

Notice how easy it becomes to change operators. You can very easily define a multitude of operators for a single representation and experiment with them to see which performs better.

Why are the genome operators GAlib not member functions? The primary reason is so that you do not have to derive a new class in order to change the behaviour of one of the built-in genome types. In addition, the use of function pointers rather than member functions lets us change operators at run-time (unlike member functions or templatized classes). And they are faster than virtual functions (OK, so this the virtual/non-virtual component is a pretty small fraction of actual execution time compared to most objective functions...). On the down side, they permit you to make some ugly mistakes by improperly casting.

The definition for the List1PtCrossover looks like this:

> *This crossover picks a single point in the parents then generates one or two children from the two halves of each parent.*

```
template <class T> int
OnePointCrossover(const GAGenome& p1, const GAGenome& p2, GAGenome* c1, GAGenome* c2){
  GAListGenome<T> &mom=(GAListGenome<T> &)p1;
  GAListGenome<T> &dad=(GAListGenome<T> &)p2;
  int nc=0;
  unsigned int a = GARandomInt(0, mom.size());
  unsigned int b = GARandomInt(0, dad.size()); GAList<T> * list;
```

```
// first do the sister...
  if(c1){
    GAListGenome<T> &sis=(GAListGenome<T> &)*c1;
    sis.GAList<T>::copy(mom);
    list = dad.GAList<T>::clone(b);
    if(a < mom.size()){
      T *site = sis.warp(a);
      while(sis.tail() != site)
        sis.destroy();      // delete the tail node
      sis.destroy();        // trash the trailing node (list[a])
    }
    else{
      sis.tail();   // move to the end of the list
    }
    sis.insert(list);       // stick the clone onto the end
    delete list;
    sis.warp(0);    // set iterator to head of list
    nc += 1;
  }

// ...now do the brother
  if(c2){
    GAListGenome<T> &bro=(GAListGenome<T> &)*c2;
    bro.GAList<T>::copy(dad);
    list = mom.GAList<T>::clone(a);
    if(b < dad.size()){
      T *site = bro.warp(b);
      while(bro.tail() != site)
        bro.destroy();      // delete the tail node
      bro.destroy();        // trash the trailing node (list[a])
    }
    else{
      bro.tail();   // move to the end of the list
    }
    bro.insert(list);       // stick the clone onto the end
    delete list;
    bro.warp(0);    // set iterator to head of list
    nc += 1;
  }
  return nc;
}
```

The definition for FlipMutator for 1DArrayAlleleGenomes looks like this:

This mutator flips the value of a single element of the array to any of the possible allele values.

```
int
FlipMutator(GAGenome & c, float pmut) {
  GA1DArrayAlleleGenome<T> &child=(GA1DArrayAlleleGenome<T> &)c;
  register int n, i;
  if(pmut <= 0.0) return(0);
  float nMut = pmut * (float)(child.length());
  if(nMut < 1.0){  // we have to do a flip test on each bit
    nMut = 0;
    for(i=child.length()-1; i>=0; i--){
      if(GAFlipCoin(pmut)){
        child.gene(i, child.alleleset().allele());
        nMut++;
      }
    }
  }
  else{     // only flip the number of bits we need to flip
    for(n=0; n<nMut; n++){
      i = GARandomInt(0, child.length()-1);
      child.gene(i, child.alleleset().allele());
```

```
    }
  }
  return((int)nMut);
}
```

And the definition for a typical initializer looks like this:

This initializer creates a tree of bounded random size and forkiness.

```
void
TreeInitializer(GAGenome & c) {
  GATreeGenome<Point> &tree=(GATreeGenome<Point> &)c;
  tree.root();
  tree.destroy();   // destroy any pre-existing tree
  Point p(0,0,0);
  tree.insert(p,GATreeBASE::ROOT);
  int n = GARandomInt(0,MAX_CHILDREN);  // limit number of children
  for(int i=0; i<n; i++)
    DoChild(tree, 0);
}

void
DoChild(GATreeGenome<Point> & tree, int depth) {
  if(depth >= MAX_DEPTH) return;         // limit depth of the tree
  int n = GARandomInt(0,MAX_CHILDREN);   // limit number of children
  Point p(GARandomFloat(0,25), GARandomFloat(0,25), GARandomFloat(0,25));
  tree.insert(p,GATreeBASE::BELOW);
  for(int i=0; i<n; i++)
  DoChild(tree, depth+1);
  tree.parent();          // move the iterator up one level
}
```

What about deriving my own genome class?

Here is the definition of a genome that contains an arbitrary number of lists. It could easily be modified to become a diploid genome. It is used in exactly the same way that the built-in genomes are used. For a simpler example, see the GNU example which integrates the GNU BitString object with GAlib to form a new genome class.

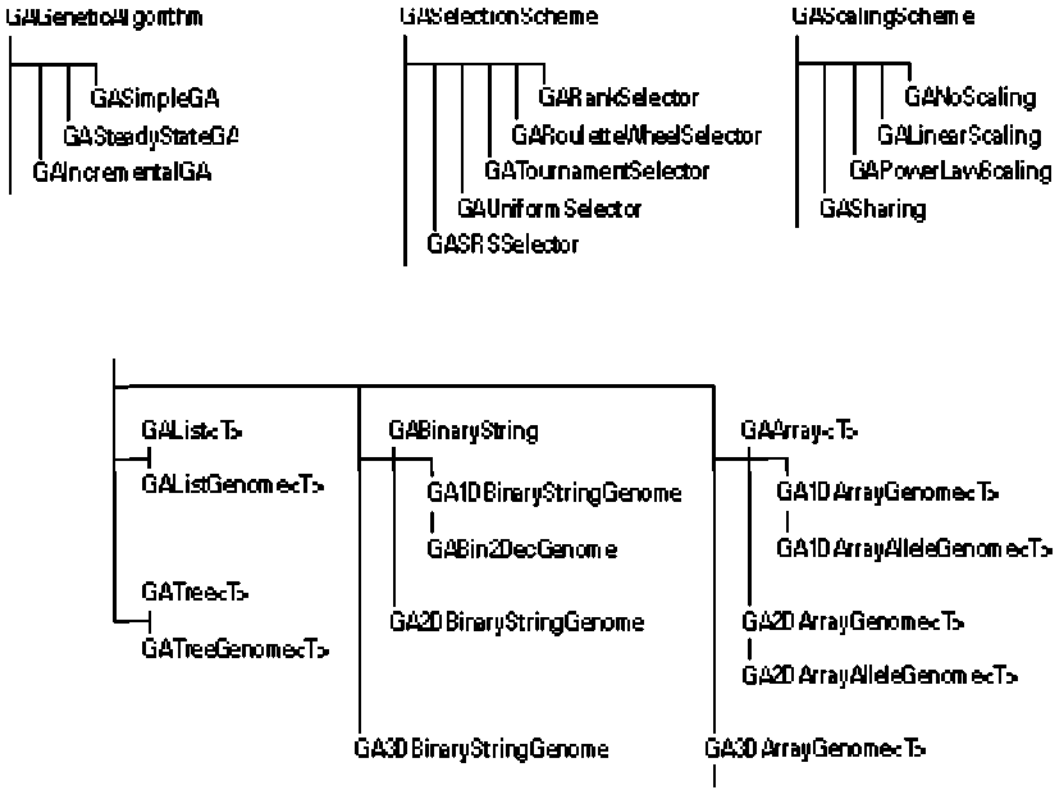
```
class RobotPathGenome : public GAGenome {
public:
  GADefineIdentity("RobotPathGenome", 251);
  static void Initializer(GAGenome&);
  static int Mutator(GAGenome&, float);
  static float Comparator(const GAGenome&, const GAGenome&);
  static float Evaluator(GAGenome&);
  static void PathInitializer(GAGenome&);

public:
  RobotPathGenome(int nrobots, int pathlength);
  RobotPathGenome(const RobotPathGenome & orig);
  RobotPathGenome& operator=(const GAGenome & arg);
  virtual ~RobotPathGenome();
  virtual GAGenome *clone(GAGenome::CloneMethod) const ;
  virtual void copy(const GAGenome & c);
  virtual int equal(const GAGenome& g) const ;
  virtual int read(istream & is);
  virtual int write(ostream & os) const ;
  GAListGenome<int> & path(int i){ return *list[i]; }
  int npaths() const { return n; }
  int length() const { return l; }

protected:
```

```
  int n, l;
  GAListGenome<int> **list;
};
```

Class Hierarchy

Here is an outline of the GAlib class hierarchy. The first section is a graphic map, the second section contains an outline of the hierarchy.

GAlib Class Hierarchy - Pictorial



GAlib Class Hierarchy - Outline

GAGeneticAlgorithm
- GASteadyStateGA (overlapping populations)
- GASimpleGA (non-overlapping populations)
- GAIncrementalGA (overlapping with custom replacement)
- GADemeGA (parallel populations with migration)

GAStatistics

GAParameterList

GAPopulation

GAScalingScheme
- GANoScaling
- GALinearScaling
- GASigmaTruncationScaling
- GAPowerLawScaling
- GASharing

GASelectionScheme
- GARankSelector
- GARouletteWheelSelector
- GATournamentSelector
- GAUniformSelector
- GASRSSelector
- GADSSelector

GAGenome
- GA1DBinaryStringGenome
  - GABin2DecGenome
- GA2DBinaryStringGenome
- GA3DBinaryStringGenome
- GA1DArrayGenome<>
  - GA1DArrayAlleleGenome<>
  - GAStringGenome (same as GA1DArrayAlleleGenome<char>)
  - GARealGenome (same as GA1DArrayAlleleGenome<float>)
- GA2DArrayGenome<>
  - GA2DArrayAlleleGenome<>
- GA3DArrayGenome<>
  - GA3DArrayAlleleGenome<>
- GATreeGenome<>
- GAListGenome<>

GAArray<>	GAAlleleSetArray<>	GAAlleleSet<>
GABinaryString	GABin2DecPhenotype	
GATree<>	GATreeIter<>	
GAList<>	GAListIter<>	

Programming Interface

This document describes the programming interface for the library. The section for each class contains a description of the object's purpose followed by the creator signature and member functions. There are also sections for library constants, typedefs, and function signatures.

Global Typedefs and Enumerations

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

Global Variables and Global Constants

```
char* gaErrMsg; // globally defined pointer to current error message
int gaDefScoreFrequency1 = 1;   // for non-overlapping populations
int gaDefScoreFrequency2 = 100; // for overlapping populations
float gaDefLinearScalingMultiplier = 1.2;
float gaDefSigmaTruncationMultiplier = 2.0;
float gaDefPowerScalingFactor = 1.0005;
float gaDefSharingCutoff = 1.0;
```

Function Prototypes

```
GABoolean (*GAGeneticAlgorithm::Terminator)(GAGeneticAlgorithm&)
GAGenome& (*GAIncrementalGA::ReplacementFunction)(GAGenome&, GAPopulation&)
void (*GAPopulation::Initializer)(GAPopulation &)
void (*GAPopulation::Evaluator)(GAPopulation &)
void (*GAGenome::Initializer)(GAGenome &)
float (*GAGenome::Evaluator)(GAGenome &)
int (*GAGenome::Mutator)(GAGenome &, float)
float (*GAGenome::Comparator)(const GAGenome &, const GAGenome&)
int (*GAGenome::SexualCrossover)(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*)
int (*GAGenome::AsexualCrossover)(const GAGenome&, GAGenome*)
int (*GABinaryEncoder)(float& value, GABit* bits, unsigned int nbits, float min, float
      max)
int (*GABinaryDecoder)(float& value, const GABit* bits, unsigned int nbits, float min,
      float max)
```

Parameter Names and Command-Line Options

Parameters may be specified using the full name strings (for example in parameter files), short name strings (for example on the command line), or explicit member functions (such as those of the genetic algorithm objects). All of the #defined names are simply the full names declared as #defined strings; you can use either the string (e.g. number_of_generations) or the #defined name (e.g. gaNnGenerations), but if you use the #defined name then the compiler will be able to catch your spelling mistakes.

When you specify GAlib arguments on the command line, they must be in name-value pairs. You can use either the long or short name. For example, if my program is called optimizer, the command line for running the program with a population size of 150, mutation rate of 10%, and score filename of evolve.txt would be:

```
optimizer popsize 150 pmut 0.1 sfile evolve.txt
```

#define name	full name short name	data type and default value
gaNminimaxi	minimaxi mm	int gaDefMiniMaxi = 1
gaNnGenerations	number_of_generations ngen	int gaDefNumGen = 250
gaNpConvergence	convergence_percentage pconv	float gaDefPConv = 0.99
gaNnConvergence	generations_to_convergence nconv	int gaDefNConv = 20
gaNpCrossover	crossover_probability pcross	float gaDefPCross = 0.9
gaNpMutation	mutation_probability pmut	float gaDefPMut = 0.01
gaNpopulationSize	population_size popsize	int gaDefPopSize = 30
gaNnPopulations	number_of_populations npop	int gaDefNPop = 10
gaNpReplacement	replacement_percentage prepl	float gaDefPRepl = 0.25
gaNnReplacement	replacement_number nrepl	int gaDefNRepl = 5
gaNnBestGenomes	number_of_best nbest	int gaDefNumBestGenomes = 1
gaNscoreFrequency	score_frequency sfreq	int gaDefScoreFrequency1 = 1
gaNflushFrequency	flush_frequency ffreq	int gaDefFlushFrequency = 0

#define name	full name short name	data type and default value
gaNscoreFilename	score_filename sfile	char* gaDefScoreFilename = "generations.dat"
gaNselectScores	select_scores sscores	int gaDefSelectScores = GAStatistics::Maximum
gaNelitism	elitism el	GABoolean gaDefElitism = gaTrue
gaNnOffspring	number_of_offspring noffspr	int gaDefNumOff = 2
gaNrecordDiversity	record_diversity recdiv	GABoolean gaDefDivFlag = gaFalse
gaNpMigration	migration_percentage pmig	float gaDefPMig = 0.1
gaNnMigration	migration_number nmig	int gaDefNMig = 5

Error Handling

Exceptions are not used in GAlib version 2.x. However, some GAlib functions return a status value to indicate whether or not their operation was successful. If a function returns an error status, it posts its error message on the global GAlib error pointer, a global string called gaErrMsg.

By default, GAlib error messages are sent immediately to the error stream. You can disable the immediate printing of error messages by passing gaFalse to the ::GAReportErrors function. Passing a value of gaTrue enables the behavior.

If you would like to redirect the error messages to a different stream, use the ::GASetErrorStream function to assign a new stream. The default stream is the system standard error stream, cerr.

Here are the error control functions and variables:

```
extern char gaErrMsg[];
void GAReportErrors(GABoolean flag);
void GASetErrorStream(ostream&);
```

Random Number Functions

GAlib includes the following functions for generating random numbers:

```
void GARandomSeed(unsigned s = 0)

int GARandomInt()
int GARandomInt(int low, int high)

double GARandomDouble()
double GARandomDouble(double low, double high)

float GARandomFloat()
float GARandomFloat(float low, float high)

int GARandomBit()

GABoolean GAFlipCoin(float p)

int GAGaussianInt(int stddev)
float GAGaussianFloat(float stddev)
double GAGaussianDouble(double stddev)
double GAUnitGaussian()
```

If you call it with no argument, the GARandomSeed function uses the current time multiplied by the process ID (on systems that have PIDs) as the seed for a psuedo-random number generator. On systems with no process IDs it uses only the time. You can specify your own random seed if you like by passing a value to this function. Once a seed has been specified, subsequent calls to GARandomSeed with the same value have no effect. Subsequent calls to GARandomSeed with a different value will re-initialize the random number generator using the new value.

The functions that take low and high as argument return a random number from low to high, inclusive. The functions that take no arguments return a value in the interval [0,1]. GAFlipCoin returns a boolean value based on a biased coin toss. If you give it a value of 1 it will return a 1, if you give it a value of 0.75 it will return a 1 with a 75% chance.

The GARandomBit function is the most efficient way to do unbiased coin tosses. It uses the random bit generator described in Numerical Recipes in C.

The Gaussian functions return a random number from a Gaussian distribution with deviation that you specify. The GAUnitGaussian function returns a number from a unit Gaussian distribution with mean 0 and deviation of 1.

GAlib uses a single random number generator for the entire library. You may not change the random number generator on the fly - it can be changed only when GAlib is compiled. See the config.h and random.h header files for details. By default, GAlib uses the ran2 generator described in Numerical Recipes in C.

GAGeneticAlgorithm

This is an abstract class that cannot be instantiated. Each genetic algorithm, when instantiated, will have default operators defined for it. See the documentation for the specific genetic algorithm type for details.

The base genetic algorithm class keeps track of evolution statistics such as number of mutations, number of crossovers, number of evaluations, best/mean/worst in each generation, and initial/current population statistics. It also defines the terminator, a member function that specifies the stopping criterion for the algorithm.

You can maximize or minimize by calling the appropriate member function. If you derive your own genetic algorithm, remember that users of your algorithm may need either type of optimization.

Statistics can be written to file each generation or periodically by specifying a flush frequency. Generational scores can be recorded each generation or less frequently by specifying a score frequency.

Parameters such as generations-to-completion, crossover probability and mutation probability can be set by member functions, command-line, or from file.

The evolve member function first calls initialize then calls the step member function until the done member function returns gaTrue. It calls the flushScores member as needed when the evolution is complete. If you evolve the genetic algorithm without using the evolve member function, be sure to call initialize before stepping through the evolution. You can use the step member function to evolve a single generation. You should call flushScores when the evolution is finished so that any buffered scores are flushed.

The names of the individual parameter member functions correspond to the #defined string names. You may set the parameters on a genetic algorithm one at a time (for example, using the nGenerations member function), using a parameter list (for example, using the parameters member function with a GAParameterList), by parsing the command line (for example, using the parameters member function with argc and argv), by name-value pairs (for example, using the set member function with a parameter name and value), or by reading a stream or file (for example, using the parameters member with a filename or stream).

see also: GAParameterList

see also: GAStatistics

see also: Terminators

class hierarchy

```
class GAGeneticAlgorithm : public GAID
```

typedefs and constants

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

member function index

```
static GAParameterList& registerDefaultParameters(GAParameterList&)
void * userData()
void * userData(void *)
void initialize(unsigned int seed=0)
void evolve(unsigned int seed=0) void step()
GABoolean done()
GAGeneticAlgorithm::Terminator terminator()
GAGeneticAlgorithm::Terminator terminator(GAGeneticAlgorithm::Terminator)
const GAStatistics & statistics() const
float convergence() const
```

```
int generation() const
void flushScores()
int minimaxi() const
int minimaxi(int)
int minimize()
int maximize()
int nGenerations() const
int nGenerations(unsigned int)
int nConvergence() const
int nConvergence(unsigned int)
float pConvergence() const
float pConvergence(float)
float pMutation() const
float pMutation(float) float pCrossover() const
float pCrossover(float)
GAGenome::SexualCrossover crossover(GAGenome::SexualCrossover func)
GAGenome::SexualCrossover sexual() const
GAGenome::AsexualCrossover crossover(GAGenome::AsexualCrossover func)
GAGenome::AsexualCrossover asexual() const
const GAPopulation & population() const
const GAPopulation & population(const GAPopulation&)
int populationSize() const
int populationSize(unsigned int n)
int nBestGenomes() const
int nBestGenomes(unsigned int n)
GAScalingScheme & scaling() const
GAScalingScheme & scaling(const GAScalingScheme&)
GASelectionScheme & selector() const
GASelectionScheme & selector(const GASelectionScheme& s)
void objectiveFunction(GAGenome::Evaluator)
void objectiveData(const GAEvalData&)
int scoreFrequency() const
int scoreFrequency(unsigned int frequency)
int flushFrequency() const
int flushFrequency(unsigned int frequency)
char* scoreFilename() const
char* scoreFilename(const char *filename)
int selectScores() const
int selectScores(GAStatistics::ScoreID which)
GABoolean recordDiversity() const
GABoolean recordDiversity(GABoolean flag)
const GAParameterList & parameters()
const GAParameterList & parameters(const GAParameterList &)
const GAParameterList & parameters(int& argc, char** argv, GABoolean flag = gaFalse)
const GAParameterList & parameters(const char* filename, GABoolean flag = gaFalse)
const GAParameterList & parameters(istream&, GABoolean flag = gaFalse);
int set(const char* s, int v)
int set(const char* s, unsigned int v)
int set(const char* s, char v)
int set(const char* s, const char* v)
int set(const char* s, const void* v)
int set(const char* s, double v);
int write(const char* filename)
int write(ostream&)
int read(const char* filename)
int read(ostream&)
```

member function descriptions

`convergence`

Returns the current convergence. The convergence is defined as the ratio of the Nth previous best-of-generation score to the current best-of-generation score.

`crossover`

Specify the mating method to use for evolution. This can be changed during the course of an evolution. This genetic algorithm uses only sexual crossover.

`done`

Returns gaTrue if the termination criteria have been met, returns gaFalse otherwise. This function simply calls the completion function that was specified using the terminator member function.

`evolve`

Initialize the genetic algorithm then evolve it until the termination criteria have been satisfied. This function first calls initialize then calls the step member function until the done member function returns gaTrue. It calls the flushScores member as needed when the evolution is complete. You may pass a seed to evolve if you want to specify your own random seed.

`flushFrequency`

Use this member function to specify how often the scores should be flushed to disk. A value of 0 means do not write to disk. A value of 100 means to flush the scores every 100 generations.

`flushScores`

Force the genetic algorithm to flush its generational data to disk. If you have specified a flushFrequency of 0 or specified a scoreFilename of nil then calling this function has no effect.

`generation`

Returns the current generation.

`initialize`

Initialize the genetic algorithm. If you specify a seed, this function calls GARandomSeed with that value. If you do not specify a seed, GAlib will choose one for you as described in the random functions section. It then initializes the population and does the first population evaluation.

`nBestGenomes`

Specify how many 'best' genomes to record. For example, if you specify 10, the genetic algorithm will keep the 10 best genomes that it ever encounters. Beware that if you specify a large number here the algorithm will slow down because it must compare the best of each generation with its current list of best individuals. The default is 1.

`nConvergence`

Set/Get the number of generations used for the convergence test.

`nGenerations`

Set/Get the number of generations.

`objectiveData`

Set the objective data member on all individuals used by the genetic algorithm. This can be changed during the course of an evolution.

`objectiveFunction`

Set the objective function on all individuals used by the genetic algorithm. This can be changed during the course of an evolution.

```
parameters
```

Returns a reference to a parameter list containing the current values of the genetic algorithm parameters.

```
parameters(GAParameterList&)
```

Set the parameters for the genetic algorithm. To use this member function you must create a parameter list (an array of name-value pairs) then pass it to the genetic algorithm.

```
parameters(int& argc, char** argv, GABoolean flag = gaFalse)
```

Set the parameters for the genetic algorithm. Use this member function to let the genetic algorithm parse your command line for arguments that GAlib understands. This method decrements argc and moves the pointers in argv appropriately to remove from the list the arguments that it understands. If you pass gaTrue as the third argument then the method will complain about any command-line arguments that are not recognized by this genetic algorithm.

```
parameters(char* filename, GABoolean flag = gaFalse)
parameters(istream&, GABoolean flag = gaFalse)
```

Set the parameters for the genetic algorithm. This version of the parameters member function will parse the specified file or stream for parameters that the genetic algorithm understands. If you pass gaTrue as the second argument then the method will complain about any parameters that are not recognized by this genetic algorithm.

```
pConvergence
```

Set/Get the convergence percentage. The convergence is defined as the ratio of the Nth previous best-of-generation score to the current best-of-generation score. N is defined by the nConvergence member function.

```
pCrossover
```

Set/Get the crossover probability.

```
pMutation
```

Set/Get the mutation probability.

```
population
```

Set/Get the population. Returns a reference to the current population.

```
populationSize
```

Set/Get the population size. This can be changed during the course of an evolution.

```
recordDiversity
```

Convenience function for specifying whether or not to calculate diversity. Since diversity calculations require comparison of each individual with every other, recording this statistic can be expensive. The default is gaFalse (diversity is not recorded).

```
registerDefaultParameters
```

Each genetic algorithm defines this member function to declare the parameters that work with it. Pass a parameter list to this function and this function will configure the list with the default parameter list and values for the genetic algorithm class from which you called it. This is a statically defined function, so invoke it using the class name of the genetic algorithm whose parameters you want to use, for example, GASimpleGA::registerDefaultParameters(list). The default parameters for the base genetic algorithm class are:

```
flushFrequency          pConvergence            scoreFilename
minimaxi                pCrossover              scoreFrequency
nBestGenomes            pMutation               selectScores
nGenerations            populationSize
nConvergence            recordDiversity
```

scaling

Set/Get the scaling scheme. The specified scaling scheme must be derived from the GAScalingScheme class. This can be changed during the course of an evolution.

scoreFilename

Specify the name of the file to which the scores should be recorded.

scoreFrequency

Specify how often the generational scores should be recorded. The default depends on the type of genetic algorithm that you are using. You can record mean, max, min, stddev, and diversity for every n generations.

selector

Set/Get the selection scheme for the genetic algorithm. The selector is used to pick individuals from a population before mating and mutation occur. This can be changed during the course of an evolution.

selectScores

This function is used to specify which scores should be saved to disk. The argument is the logical OR of the following values: Mean, Maximum, Minimum, Deviation, Diversity (all defined in the scope of the GAStatistics object). To record all of the scores, pass GAStatistics::AllScores. When written to file, the format is as follows:

```
generation TAB mean TAB max TAB min TAB deviation TAB diversity NEWLINE
```

set

Set individual parameters for the genetic algorithm. The

first argument should be the full- or short-name of the parameter you wish to set. The second argument is the value to which you would like to set the parameter.

statistics

Returns a reference to the statistics object in the genetic algorithm. The statistics object maintains information such as best, worst, mean, and standard deviation, and diversity of each generation as well as a separate population with the best individuals ever encountered by the genetic algorithm.

step

Evolve the genetic algorithm for one generation.

terminator

Set/Get the termination function. The genetic algorithm is complete when the completion function returns gaTrue. The function must have the proper signature.

userData

Set/Get the userData member of the genetic algorithm. This member is a generic pointer to any information that needs to be stored with the genetic algorithm.

GADemeGA

(parallel populations with migration)

This genetic algorithm has multiple, independent populations. It creates the populations by cloning the genome or population that you pass when you create it.

Each population evolves using a steady-state genetic algorithm, but each generation some individuals migrate from one population to another. The migration algorithm is deterministic stepping-stone; each population migrates a fixed number of its best individuals to its neighbor. The master population is updated each generation with best individual from each population.

If you want to experiment with other migration methods, derive a new class from this one and define a new migration operator. You can change the evolution behavior by defining a new step method in a derived class.

see also: GAGeneticAlgorithm

class hierarchy

```
class GADemeGA : public GAGeneticAlgorithm
```

typedefs and constants

```
enum { ALL= -1 };
```

constructors

```
GADemeGA(const GAGenome&)
GADemeGA(const GAPopulation&)
GADemeGA(const GADemeGA&)
```

member function index

```
static GAParameterList& registerDefaultParameters(GAParameterList&);
void migrate()
GADemeGA & operator++()
const GAPopulation& population(unsigned int i) const
const GAPopulation& population(int i, const GAPopulation&)
int populationSize(unsigned int i) const
int populationSize(int i, unsigned int n)
int nReplacement(unsigned int i) const
int nReplacement(int i, unsigned int n)
int nMigration() const
int nMigration(unsigned int i)
int nPopulations() const
int nPopulations(unsigned int i)
const GAStatistics& statistics() const
const GAStatistics& statistics(unsigned int i) const
```

member function descriptions

nMigration

Specify the number of individuals to migrate each generation. Each population will migrate this many of its best individuals to the next population (the stepping-stone migration model). The individuals replace the worst individuals in the receiving population.

nReplacement

Specify a number of individuals to replace each generation. When you specify a number of individuals to replace, the pReplacement value is set to 0. The first argument specifies which population should be modified. Use GADemeGA::ALL to apply to all populations.

`operator++`

The increment operator evolves the genetic algorithm's population by one generation by calling the step member function.

`pReplacement`

Specify a percentage of the population to replace each generation. When you specify a replacement percentage, the nReplacement value is set to 0. The first argument specifies which population should be modified. Use GADemeGA::ALL to apply to all populations.

`registerDefaultParameters`

This function adds parameters to the specified list that are of interest to this genetic algorithm. The default parameters for the deme genetic algorithm are the parameters for the base genetic algorithm class plus the following:

```
nMigration
nPopulations
```

GAIncrementalGA

(overlapping populations with 1 or 2 children per generation)

This genetic algorithm is similar to those based on the GENITOR model. It uses overlapping populations, but very little overlap (only one or two individuals get replaced each generation). The default replacement scheme is WORST. A replacement function is required only if you use CUSTOM or CROWDING as the replacement scheme. You can do DeJong-style crowding by specifying a distance function with the CROWDING option. (for best DeJong-style results, derive your own genetic algorithm)

You can specify the number of children that are generated in each 'generation' by using the nOffspring member function. Since this genetic algorithm is based on a two-parent crossover model, the number of offspring must be either 1 or 2. The default is 2.

Use the replacement method to specify which type of replacement the genetic algorithm should use. The replacement strategy determines how the new children will be inserted into the population. If you want the new child to replace one of its parents, use the Parent strategy. If you want the child to replace a random population member, use the Random strategy. If you want the child to replace the worst population member, use the Worst strategy.

If you specify CUSTOM or CROWDING you must also specify a replacement function with the proper signature. This function is used to pick which genome will be replaced. The first argument passed to the replacement function is the individual that is supposed to go into the population. The second argument is the population into which the individual is supposed to go. The replacement function should return a reference to the genome that the individual should replace. If no replacement should take place, the replacement function should return a reference to the individual.

The score frequency for this genetic algorithm defaults to 100 (it records the best-of-generation every 100th generation). The default scaling is Linear, the default selection is RouletteWheel.

see also: GAGeneticAlgorithm

class hierarchy

```
class GAIncrementalGA : public GAGeneticAlgorithm
```

typedefs and constants

```
GAGenome& (*GAIncrementalGA::ReplacementFunction)(GAGenome &, GAPopulation &)
enum ReplacementScheme {
  RANDOM = GAPopulation::RANDOM,
  BEST = GAPopulation::BEST,
  WORST = GAPopulation::WORST,
  CUSTOM = -30,
  CROWDING = -30,
  PARENT = -10
};
```

constructors

```
GAIncrementalGA(const GAGenome&)
GAIncrementalGA(const GAPopulation&)
GAIncrementalGA(const GAIncrementalGA&)
```

member function index

```
static GAParameterList& registerDefaultParameters(GAParameterList&)
GASteadyStateGA & operator++()
ReplacementScheme replacement()
ReplacementScheme replacement(ReplacementScheme, ReplacementFunction f = NULL)
int nOffspring() const
```

```
int nOffspring(unsigned int n)
```

member function descriptions

`nOffspring`

The incremental genetic algorithm can produce either one or two individuals each generation. Use this member function to specify how many individuals you would like.

`operator++`

The increment operator evolves the genetic algorithm's population by one generation by calling the step member function.

`registerDefaultParameters`

This function adds to the specified list parameters that are of interest to this genetic algorithm. The default parameters for the incremental genetic algorithm are the parameters for the base genetic algorithm class plus the following: `nOffspring`

`replacement`

Specify a replacement method. The scheme can be one of:

`GAIncrementalGA::RANDOM`	`GAIncrementalGA::BEST`	`GAIncrementalGA::CUSTOM`
`GAIncrementalGA::PARENT`	`GAIncrementalGA::WORST`	`GAIncrementalGA::CROWDING`

If you specify custom or crowding replacement then you must also specify a function. The replacement function takes two arguments: the individual to insert and the population into which it will be inserted. The replacement function should return a reference to the genome that should be replaced. If no replacement should take place, the replacement function should return a reference to the individual passed to it.

GASimpleGA

(non-overlapping populations)

This genetic algorithm is the 'simple' genetic algorithm that Goldberg describes in his book. It uses non-overlapping populations. When you create a simple genetic algorithm, you must specify either an individual or a population of individuals. The new genetic algorithm will clone the individual(s) that you specify to make its own population. You can change most of the genetic algorithm behaviors after creation and during the course of the evolution.

The simple genetic algorithm creates an initial population by cloning the individual or population you pass when you create it. Each generation the algorithm creates an entirely new population of individuals by selecting from the previous population then mating to produce the new offspring for the new population. This process continues until the stopping criteria are met (determined by the terminator).

Elitism is optional. By default, elitism is on, meaning that the best individual from each generation is carried over to the next generation. To turn off elitism, pass gaFalse to the elitist member function.

The score frequency for this genetic algorithm defaults to 1 (it records the best-of-generation every generation). The default scaling is Linear, the default selection is RouletteWheel.

class hierarchy

```
class GASimpleGA : public GAGeneticAlgorithm
```

constructors

```
GASimpleGA(const GAGenome&)
GASimpleGA(const GAPopulation&)
GASimpleGA(const GASimpleGA&)
```

member function index

```
static GAParameterList& registerDefaultParameters(GAParameterList&)
GASimpleGA & operator++()
GABoolean elitist() const
GABoolean elitist(GABoolean flag)
```

member function descriptions

`elitist`

Set/Get the elitism flag. If you specify gaTrue, the genetic algorithm will copy the best individual from the previous population into the current population if no individual in the current population is any better.

`operator++`

The increment operator evolves the genetic algorithm's population by one generation by calling the step member function.

`registerDefaultParameters`

This function adds to the specified list parameters that are of interest to this genetic algorithm. The default parameters for the simple genetic algorithm are the parameters for the base genetic algorithm class plus the following: `elitism`

GASteadyStateGA

(overlapping populations)

This genetic algorithm is similar to the algorithms described by DeJong. It uses overlapping populations with a user-specifiable amount of overlap. The algorithm creates a population of individuals by cloning the genome or population that you pass when you create it. Each generation the algorithm creates a temporary population of individuals, adds these to the previous population, then removes the worst individuals in order to return the population to its original size.

You can select the amount of overlap between generations by specifying the pReplacement parameter. This is the percentage of the population that should be replaced each generation. Newly generated offspring are added to the population, then the worst individuals are destroyed (so the new offspring may or may not make it into the population, depending on whether they are better than the worst in the population).

If you specify a replacement percentage, then that percentage of the population will be replaced each generation. Alternatively, you can specify a number of individuals (less than the number in the population) to replace each generation. You cannot specify both - in a parameter list containing both parameters, the latter is used.

The score frequency for this genetic algorithm defaults to 100 (it records the best-of-generation every 100th generation). The default scaling is Linear, the default selection is RouletteWheel.

see also: GAGeneticAlgorithm

class hierarchy

```
class GASteadyStateGA : public GAGeneticAlgorithm
```

constructors

```
GASteadyStateGA(const GAGenome&)
GASteadyStateGA(const GAPopulation&)
GASteadyStateGA(const GASteadyStateGA&)
```

member function index

```
static GAParameterList& registerDefaultParameters(GAParameterList&)
GASteadyStateGA & operator++()
float pReplacement() const
float pReplacement(float percentage)
int nReplacement() const
int nReplacement(unsigned int)
```

member function descriptions

`nReplacement`

Specify a number of individuals to replace each generation. When you specify a number of individuals to replace, the pReplacement value is set to 0.

`operator++`

The increment operator evolves the genetic algorithm's population by one generation by calling the step member function.

`pReplacement`

Specify a percentage of the population to replace each generation. When you specify a replacement percentage, the nReplacement value is set to 0.

`registerDefaultParameters`

This function adds to the specified list parameters that are of interest to this genetic algorithm. The default parameters for the steady-state genetic algorithm are the parameters for the base genetic algorithm class plus the following:

```
pReplacement
nReplacement
```

Terminators

Completion functions are used to determine whether or not a genetic algorithm is finished. The done member function simply calls the completion function to determine whether or not the genetic algorithm should continue. The predefined completion functions use generation and convergence to determine whether or not the genetic algorithm is finished.

The completion function returns gaTrue when the genetic algorithm should finish, and gaFalse when the genetic algorithm should continue.

In this context, convergence refers to the the similarity of the objective scores, not similarity of underlying genetic structure. The built-in convergence measures use the best-of-generation scores to determine whether or not the genetic algorithm has plateaued.

```
GABoolean GAGeneticAlgorithm::TerminateUponGeneration(GAGeneticAlgorithm &)
GABoolean GAGeneticAlgorithm::TerminateUponConvergence(GAGeneticAlgorithm &)
GABoolean GAGeneticAlgorithm::TerminateUponPopConvergence(GAGeneticAlgorithm &)
```

`TerminateUponGeneration`

This function compares the current generation to the specified number of generations. If the current generation is less than the requested number of generations, it returns gaFalse. Otherwise, it returns gaTrue.

`TerminateUponConvergence`

This function compares the current convergence to the specified convergence value. If the current convergence is less than the requested convergence, it returns gaFalse. Otherwise, it returns gaTrue.

Convergence is a number between 0 and 1. A convergence of 1 means that the nth previous best-of-generation is equal to the current best-of-generation. When you use convergence as a stopping criterion you must specify the convergence percentage and you may specify the number of previous generations against which to compare. The genetic algorithm will always run at least this many generations.

`TerminateUponPopConvergence`

This function compares the population average to the score of the best individual in the population. If the population average is within pConvergence of the best individual's score, it returns gaTrue. Otherwise, it returns gaFalse.

For details about how to write your own termination function, see the customizations section.

Replacement Schemes

The replacement scheme is used by the incremental genetic algorithm to determine how a new individual should be inserted into a population. Valid replacement schemes include:

```
GAIncrementalGA::RANDOM          GAIncrementalGA::CUSTOM
GAIncrementalGA::BEST            GAIncrementalGA::CROWDING
GAIncrementalGA::WORST           GAIncrementalGA::PARENT
```

In general, replace worst produces the best results. Replace parent is useful for basic speciation, and custom replacement can be used when you want to do your own type of speciation.

If you specify CUSTOM or CROWDING replacement then you must also specify a replacement function. The replacement function takes as arguments an individual and the population into which the individual should be placed. It returns a reference to the genome that the individual should replace. If the individual should not be inserted into the population, the function should return a reference to the individual.

Any replacement function must have the following function prototype:

```
typedef GAGenome& (*GAIncrementalGA::ReplacementFunction)(GAGenome &, GAPopulation &);
```

The first argument is the genome that will be inserted into the population, the second argument is the population into which the genome should be inserted. The function should return a reference to the genome that will be replaced. If no replacement occurs, the function should return a reference to the original genome.

For details about how to write your own replacement function, see the customizations section.

GAGenome

The genome is a virtual base class and cannot be instantiated. It defines a number of constants and function prototypes specific to the genome and its derived classes.

The dimension is used to specify which dimension is being referred to in multi-dimensional genomes. The clone method specifies whether to clone the entire genome (a new genome with contents identical to the original will be created) or just the attributes of the genome (a new genome with identical characteristics will be created). In both cases the caller is responsible for deleting the memory allocated by the clone member function. The resize constants are used when specifying a resizable genome's resize behavior.

The genetic operators for genomes are functions that take generic genomes as their arguments. This makes it possible to define new behaviors for existing genome classes without deriving a new class.

class hierarchy

```
class GAGenome : public GAID
```

typedefs and constants

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

member function index

```
virtual void copy(const GAGenome & c)
virtual GAGenome * clone(CloneMethod flag = CONTENTS)
float score() const
float score(float s)
int nevals()
float evaluate(GABoolean flag = gaFalse) const
GAGenome::Evaluator evaluator() const
GAGenome::Evaluator evaluator(GAGenome::Evaluator func)
void initialize()
GAGenomeInitializer initializer() const
GAGenomeInitializer initializer(GAGenome::Initializer func)
int mutate(float pmutation)
GAGenome::Mutator mutator() const
GAGenome::Mutator mutator(GAGenome::Mutator func)
float compare(const GAGenome& g) const
GAGenome::Comparator comparator() const
GAGenome::Comparator comparator(GAGenome::Comparator c)
GAGenome::SexualCrossover crossover(GAGenome::SexualCrossover f)
GAGenome::SexualCrossover sexual()
GAGenome::AsexualCrossover crossover(GAGenome::AsexualCrossover f)
GAGenome::AsexualCrossover asexual()
GAGeneticAlgorithm * geneticAlgorithm() const
GAGeneticAlgorithm * geneticAlgorithm(GAGeneticAlgorithm &)
void * userData() const
void * userData(void * data)
GAEvalData * evalData() const
GAEvalData * evalData(void * data)
virtual int read(istream &)
virtual int write(ostream &) const
```

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

These operators call the corresponding virtual members so that they will work on any properly derived genome class.

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

member function descriptions

`clone`

This method allocates space for a new genome whereas the copy method uses the space of the genome to which it belongs.

`compare`

Compare two genomes. The comparison can be genotype- or phenotype-based. The comparison returns a value greater than or equal to 0. 0 means the two genomes are identical (no diversity). The exact meaning of the comparison is up to you.

`comparator`

Set/Get the comparison method. The comparator must have the correct signature.

`copy`

The copy member function is called by the base class' operator= and clone members. You can use it to copy the contents of a genome into an existing genome.

`crossover`

Each genome class can define its preferred mating method. Use this method to assign the preferred crossover for a genome instance.

`equal`
`notequal`

'equal' and 'notequal' are genome-specific. See the documentation for each genome class for specific details about what 'equal' means. For example, genomes that have identical contents but different allele sets may or may not be considered equal. By default, notequal just calls the equal function, but you can override this in derived classes if you need to optimize the comparison.

`evalData`

Set/Get the object used to store genome-specific evaluation data. Each genome owns its own evaluation data object; cloning a genome clones the evaluation data as well.

`evaluate`

Invoke the genome's evaluation function. If you call this member with gaTrue, the evaluation function is called no matter what (assuming one has been assigned to the genome). By default, the argument to this function is gaFalse, so the genome's evaluation function is called only if the state of the genome has not changed since the last time the evaluator was invoked.

`evaluator`

Set/Get the function used to evaluate the genome.

`geneticAlgorithm`

The member function returns a pointer to the genetic algorithm that 'owns' the genome. If this function returns nil then the genome has no genetic algorithm owner.

`initialize`

Calls the initialization function for the genome.

`initializer`

Set/Get the initialization method. The initializer must have the correct signature.

`mutate`

Calls the mutation method for the genome. The value is typically the mutation likliehood, but the exact interpretation of this value is up to the designer of the mutation method.

`mutator`

Set/Get the mutation method. The mutator must have the correct signature.

`nevals`

Returns the number of objective function evaluations since the genome was initialized.

```
operator==          operator!=
operator<<          operator>>
```

These methods call the associated virtual member functions. They can be used on any generic genome. If the derived class was properly defined, the appropriate derived functions will be called and the functions will operate on the derived classes rather than the base class.

`read`

Fill the genome with the data read from the specified stream. sexual

`asexual`

Returns a pointer to the preferred mating method for this genome. If this function returns nil, no mating method has been defined for the genome. The genetic algorithm object has ultimate control over the mating method that is actually used in the evolution.

`score`

Returns the objective score of the genome using the objective function assigned to the genome. If no objective function has been assigned and no score has been set, a score of 0 will be returned. If the score function is called with an argument, the genome's objective score is set to that value (useful for population-based objective functions in which the population object does the evaluations).

`userData`

Each genome contains a generic pointer to user-specifiable data. Use this member function to set/get that pointer. Notice that cloning a genome will cause the cloned genome to refer to the same user data pointer as the original; the user data is not cloned as well. So all genomes in a population refer to the same user data.

`write`

Send the contents of the genome to the specified stream.

GA1DArrayGenome<T>

The 1D array genome is a generic, resizable array of objects. It is a template class derived from the GAGenome class as well as the GAArray<> class.

Each element in the array is a gene. The values of the genes are determines by the type of the genome. For example, an array of ints may have integer values whereas an array of doubles may have floating point values.

see also: GAArray, GAGenome

class hierarchy

```
class GA1DArrayGenome<T> : public GAArray<T>, public GAGenome
```

constructors

```
GA1DArrayGenome(unsigned int length, GAGenome::Evaluator objective = NULL, void * userData
      = NULL)
GA1DArrayGenome(const GA1DArrayGenome<T> &)
```

member function index

```
const T & gene(unsigned int x=0) const
T & gene(unsigned int x=0)
T & gene(unsigned int x, const T& value) const
T & operator[](unsigned int x) const
T & operator[](unsigned int x)
int length() const
int length(int l)
int resize(int x)
int resizeBehaviour() const
int resizeBehaviour(unsigned int minx, unsigned int maxx)
void copy(const GA1DArrayGenome<T>& original, unsigned int dest, unsigned int src,
      unsigned int length)
void swap(unsigned int x1, unsigned int x2)
```

member function descriptions

`copy`

Copy the specified bits from the designated genome.

`gene`

Set/Get the specified element.

`length`

Set/Get the length.

`resize`

Set the length.

`resizeBehaviour`

Set/Get the resize behavior. The min value specifies the minimum allowable length, the max value specifies the maximum allowable length. If min and max are equal, the genome is not resizable.

Use the resizeBehaviour and resize member functions to control the size of the genome. The default behavior is fixed size. Using the resizeBehaviour method you can specify minimum and maximum values for the size of the genome. If you specify minimum and maximum as the same values then fixed

size is assumed. If you use the resize method to specify a size that is outside the bounds set earlier using resizeBehaviour, the bounds will be 'stretched' to accommodate the value you specify with resize. Conversely, if the values you specify with resizeBehaviour conflict with the genome's current size, the genome will be resized to accommodate the new values.

When resizeBehaviour is called with no arguments, it returns the maximum size if the genome is resizable, or GAGenome::FIXED_SIZE if the size is fixed.

swap

Swap the specified elements.

genetic operators for this class

```
GA1DArrayGenome<>::SwapMutator
GA1DArrayGenome<>::ElementComparator
GA1DArrayGenome<>::UniformCrossover
GA1DArrayGenome<>::EvenOddCrossover
GA1DArrayGenome<>::OnePointCrossover
GA1DArrayGenome<>::TwoPointCrossover
GA1DArrayGenome<>::PartialMatchCrossover
GA1DArrayGenome<>::OrderCrossover
GA1DArrayGenome<>::CycleCrossover
```

default genetic operators for this class

initialization:	`GAGenome::NoInitializer`
comparison:	`GA1DArrayGenome<>::ElementComparator`
mutation:	`GA1DArrayGenome<>::SwapMutator`
crossover:	`GA1DArrayGenome<>::OnePointCrossover`

GA1DArrayAlleleGenome<T>

The one-dimensional array allele genome is derived from the one-dimensional array genome class. It shares the same behaviors, but adds the features of allele sets. The value assumed by each element in an array allele genome depends upon the allele set specified for that element. In the simplest case, you can create a single allele set which defines the possible values for any element in the array. More complicated examples can have a different allele set for each element in the array.

If you create the genome with a single allele set, the genome will have a length that you specify and the allele set will be used for the mapping of each element. If you create the genome using an array of allele sets, the genome will have a length equal to the number of allele sets in the array and each element of the array will be mapped using the corresponding allele set.

When you define an allele set for an array genome, the genome makes its own copy. Subsequent clones of this genome will refer to the original genome's allele set (allele sets do reference counting).

see also: GAArray, GA1DArrayGenome, GAAlleleSet, GAAlleleSetArray

class hierarchy

```
class GA1DArrayAlleleGenome<T> : public GAArrayGenome<T>
```

constructors

```
GA1DArrayAlleleGenome(unsigned int length, const GAAlleleSet<T>& alleleset,
      GAGenome::Evaluator objective = NULL, void * userData = NULL)
GA1DArrayAlleleGenome(const GAAlleleSetArray<T>& allelesets, GAGenome::Evaluator objective
      = NULL, void * userData = NULL)
GA1DArrayAlleleGenome(const GA1DArrayAlleleGenome<T>&)
```

member function index

```
const GAAlleleSet<T>& alleleset(unsigned int i = 0) const
```

member function descriptions

`alleleset`

Returns a reference to the allele set for the specified gene. If the genome was created using a single allele set, the allele set will be the same for every gene. If the genome was created using an allele set array, each gene may have a different allele set.

genetic operators for this class

```
GA1DArrayAlleleGenome<>::UniformInitializer
GA1DArrayAlleleGenome<>::OrderedInitializer
GA1DArrayAlleleGenome<>::FlipMutator
```

default genetic operators for this class

initialization:	`GA1DArrayAlleleGenome<>::UniformInitializer`
comparison:	`GA1DArrayGenome<>::ElementComparator`
mutation:	`GA1DArrayAlleleGenome<>::FlipMutator`
crossover:	`GA1DArrayGenome<>::OnePointCrossover`

GA2DArrayGenome<T>

The two-dimensional array genome is a generic, resizable array of objects. It is a template class derived from the GAGenome class as well as the GAArray<> class.

Each element in the array is a gene. The values of the genes are determines by the type of the genome. For example, an array of ints may have integer values whereas an array of doubles may have floating point values.

see also: GAArray, GAGenome

class hierarchy

```
class GA2DArrayGenome<T> : public GAArray<T>, public GAGenome
```

constructors

```
GA2DArrayGenome(unsigned int width, unsigned int height, GAGenome::Evaluator objective =
       NULL, void * userData = NULL)
GA2DArrayGenome(const GA2DArrayGenome<T> &)
```

member function index

```
const T & gene(unsigned int x, unsigned int y) const
T & gene(unsigned int x, unsigned int y)
T & gene(unsigned int x, unsigned int y, const T& value)
int width() const
int width(int w)
int height() const
int height(int h)
int resize(int x, int y)
int resizeBehaviour(GADimension which) const
int resizeBehaviour(GADimension which, unsigned int min, unsigned int max)
int resizeBehaviour(unsigned int minx, unsigned int maxx, unsigned int miny, unsigned int
      maxy)
void copy(const GA2DArrayGenome<T>& original, unsigned int xdest, unsigned int ydest,
       unsigned int xsrc, unsigned int ysrc, unsigned int width, unsigned int height)
void swap(unsigned int x1, unsigned int y1, unsigned int x2, unsigned int y2)
```

member function descriptions

`copy`

Copy the specified bits from the designated genome.

`gene`

Set/Get the specified element.

`height`

Set/Get the height.

`resize`

Change the size to the specified dimensions.

`resizeBehaviour`

Set/Get the resize behavior. The min value specifies the minimum allowable length, the max value specifies the maximum allowable length. If min and max are equal, the genome is not resizable.

Use the resizeBehaviour and resize member functions to control the size of the genome. The default behavior is fixed size. Using the resizeBehaviour method you can specify minimum and maximum values for the size of the genome. If you specify minimum and maximum as the same values then fixed size is assumed. If you use the resize method to specify a size that is outside the bounds set earlier using resizeBehaviour, the bounds will be 'stretched' to accommodate the value you specify with resize. Conversely, if the values you specify with resizeBehaviour conflict with the genome's current size, the genome will be resized to accommodate the new values.

When resizeBehaviour is called with no arguments, it returns the maximum size if the genome is resizable, or GAGenome::FIXED_SIZE if the size is fixed.

The resizeBehaviour function works similarly to that of the 1D array genome. In this case, however, you must also specify for which dimension you are setting the resize behavior. When resizeBehaviour is called with no arguments, it returns the maximum size if the genome is resizable, or gaNoResize if the size is fixed.

`swap`

Swap the specified elements.

`width`

Set/Get the width.

genetic operators for this class

```
GA2DArrayGenome<>::SwapMutator
GA2DArrayGenome<>::ElementComparator
GA2DArrayGenome<>::UniformCrossover
GA2DArrayGenome<>::EvenOddCrossover
GA2DArrayGenome<>::OnePointCrossover
```

default genetic operators for this class

initialization:	`GAGenome::NoInitializer`
comparison:	`GA2DArrayGenome<>::ElementComparator`
mutation:	`GA2DArrayGenome<>::SwapMutator`
crossover:	`GA2DArrayGenome<>::OnePointCrossover`

GA2DArrayAlleleGenome<T>

The two-dimensional array allele genome is derived from the two-dimensional array genome class. It shares the same behaviors, but adds the features of allele sets. The value assumed by each element in an array allele genome depends upon the allele set specified for that element. In the simplest case, you can create a single allele set which defines the possible values for any element in the array. More complicated examples can have a different allele set for each element in the array.

The genome will have width and height that you specify and the allele set will be used for the mapping of each element. When you define an allele set for an array genome, the genome makes its own copy. Subsequent clones of this genome will refer to the original genome's allele set (allele sets do reference counting).

If you create a genome using an allele set array, the array of alleles will be mapped to the two dimensions in the order width-then-height.

see also: GAArray, GA2DArrayGenome, GAAlleleSet, GAAlleleSetArray

class hierarchy

```
class GA1DArrayAlleleGenome<T> : public GAArrayGenome<T>
```

constructors

```
GA2DArrayAlleleGenome(unsigned int width, unsigned int height, GAAlleleSet<T>& alleles,
      GAGenome::Evaluator objective = NULL, void * userData = NULL)
GA2DArrayAlleleGenome(unsigned int width, unsigned int height, GAAlleleSetArray<T>&
      allelesets, GAGenome::Evaluator objective = NULL, void * userData = NULL)
GA2DArrayAlleleGenome(const GA2DArrayAlleleGenome<T> &)
```

member function index

```
const GAAlleleSet<T>& alleleset(unsigned int i = 0, unsigned int j = 0) const
```

member function descriptions

`alleleset`

Returns a reference to the allele set for the specified gene. If the genome was created using a single allele set, the allele set will be the same for every gene. If the genome was created using an allele set array, each gene may have a different allele set.

genetic operators for this class

```
GA2DArrayAlleleGenome<>::UniformInitializer
GA2DArrayAlleleGenome<>::FlipMutator
```

default genetic operators for this class

initialization:	`GA2DArrayAlleleGenome<>::UniformInitializer`
comparison:	`GA2DArrayGenome<>::ElementComparator`
mutation:	`GA2DArrayAlleleGenome<>::FlipMutator`
crossover:	`GA2DArrayGenome<>::OnePointCrossover`

GA3DArrayGenome<T>

The three-dimensional array genome is a generic, resizable array of objects. It is a template class derived from the GAGenome class as well as the GAArray<> class.

Each element in the array is a gene. The values of the genes are determines by the type of the genome. For example, an array of ints may have integer values whereas an array of doubles may have floating point values.

see also: GAArray, GAGenome

class hierarchy

```
class GA3DArrayGenome<T> : public GAArray<T>, public GAGenome
```

constructors

```
GA3DArrayGenome(unsigned int width, unsigned int height, unsigned int depth,
       GAGenome::Evaluator objective = NULL, void * userData = NULL)
GA3DArrayGenome(const GA3DArrayGenome<T>&)
```

member function index

```
const T & gene(unsigned int x, unsigned int y, unsigned int z) const
T & gene(unsigned int x, unsigned int y, unsigned int z)
T & gene(unsigned int x, unsigned int y, unsigned int z, const T& value)
int width() const
int width(int w)
int height() const
int height(int h)
int depth() const
int depth(int d)
int resize(int x, int y, int z)
int resizeBehaviour(GADimension which) const
int resizeBehaviour(GADimension which, unsigned int min, unsigned int max)
int resizeBehaviour(unsigned int minx, unsigned int maxx, unsigned int miny, unsigned int
       maxy, unsigned int minz, unsigned int maxz)
void copy(const GA3DArrayGenome<T>& original, unsigned int xdest, unsigned int ydest,
       unsigned int zdest, unsigned int xsrc, unsigned int ysrc, unsigned int zsrc,
       unsigned int width, unsigned int height, unsigned int depth)
void swap(unsigned int x1, unsigned int y1, unsigned int z1, unsigned int x2, unsigned int
       y2, unsigned int z2)
```

member function descriptions

`copy`

Copy the specified bits from the designated genome.

`depth`

Set/Get the depth.

`gene`

Set/Get the specified element.

`height`

Set/Get the height.

`resize`

Change the size to the specified dimensions.

`resizeBehaviour`

Set/Get the resize behavior. The min value specifies the minimum allowable length, the max value specifies the maximum allowable length. If min and max are equal, the genome is not resizable.

Use the resizeBehaviour and resize member functions to control the size of the genome. The default behavior is fixed size. Using the resizeBehaviour method you can specify minimum and maximum values for the size of the genome. If you specify minimum and maximum as the same values then fixed size is assumed. If you use the resize method to specify a size that is outside the bounds set earlier using resizeBehaviour, the bounds will be 'stretched' to accommodate the value you specify with resize. Conversely, if the values you specify with resizeBehaviour conflict with the genome's current size, the genome will be resized to accommodate the new values.

When resizeBehaviour is called with no arguments, it returns the maximum size if the genome is resizable, or GAGenome::FIXED_SIZE if the size is fixed.

The resizeBehaviour function works similarly to that of the 1D array genome. In this case, however, you must also specify for which dimension you are setting the resize behavior. When resizeBehaviour is called with no arguments, it returns the maximum size if the genome is resizable, or gaNoResize if the size is fixed.

`swap`

Swap the specified elements.

`width`

Set/Get the width.

genetic operators for this class

```
GA3DArrayGenome<>::SwapMutator
GA3DArrayGenome<>::ElementComparator
GA3DArrayGenome<>::UniformCrossover
GA3DArrayGenome<>::EvenOddCrossover
GA3DArrayGenome<>::OnePointCrossover
```

default genetic operators

initialization:	`GAGenome::NoInitializer`
comparison:	`GA3DArrayGenome<>::ElementComparator`
mutation:	`GA3DArrayGenome<>::SwapMutator`
crossover:	`GA3DArrayGenome<>::OnePointCrossover`

GA3DArrayAlleleGenome<T>

The three-dimensional array allele genome is derived from the three-dimensional array genome class. It shares the same behaviors, but adds the features of allele sets. The value assumed by each element in an array allele genome depends upon the allele set specified for that element. In the simplest case, you can create a single allele set which defines the possible values for any element in the array. More complicated examples can have a different allele set for each element in the array.

The genome will have width, height, and depth that you specify and the allele set will be used for the mapping of each element. When you define an allele set for an array genome, the genome makes its own copy. Subsequent clones of this genome will refer to the original genome's allele set (allele sets do reference counting).

If you create a genome using an allele set array, the array of alleles will be mapped to the three dimensions in the order width-then-height-then-depth.

see also: GAArray, GA3DArrayGenome, GAAlleleSet, GAAlleleSetArray

class hierarchy

```
class GA1DArrayAlleleGenome<T> : public GAArrayGenome<T>
```

constructors

```
GA3DArrayAlleleGenome(unsigned int width, unsigned int height, unsigned int depth,
       GAAlleleSet<T>& alleles, GAGenome::Evaluator objective = NULL, void * userData =
       NULL)
GA3DArrayAlleleGenome(unsigned int width, unsigned int height, unsigned int depth,
       GAAlleleSet<T>& alleles, GAGenome::Evaluator objective = NULL, void * userData =
       NULL)
GA3DArrayAlleleGenome(const GA3DArrayAlleleGenome<T> &)
```

member function index

```
const GAAlleleSet<T>& alleleset(unsigned int i = 0, unsigned int j = 0, unsigned int k =
       0) const
```

member function descriptions

alleleset

Returns a reference to the allele set for the specified gene. If the genome was created using a single allele set, the allele set will be the same for every gene. If the genome was created using an allele set array, each gene may have a different allele set.

genetic operators for this class

```
GA3DArrayAlleleGenome<>::UniformInitializer
GA3DArrayAlleleGenome<>::FlipMutator
```

default genetic operators for this class

initialization:	GA3DArrayAlleleGenome<>::UniformInitializer
comparison:	GA3DArrayGenome<>::ElementComparator
mutation:	GA3DArrayAlleleGenome<>::FlipMutator
crossover:	GA3DArrayGenome<>::OnePointCrossover

GA1DBinaryStringGenome

The binary string genome is derived from the GABinaryString and GAGenome classes. It is a string of 1s and 0s whose length may be fixed or variable. The genes in this genomes are bits. The alleles for each bit are 0 and 1.

see also: GABinaryString

see also: GAGenome

class hierarchy

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

constructors

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

member function index

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

member function descriptions

`copy`

Copy the specified bits from the designated genome.

`gene`

Set/Get the specified bit.

`length`

Set/Get the length of the bit string.

`resize`

Set the length of the bit string.

`resizeBehaviour`

Set/Get the resize behavior. The min value specifies the minimum allowable length, the max value specifies the maximum allowable length. If min and max are equal, the genome is not resizable.

Use the resizeBehaviour and resize member functions to control the size of the genome. The default behavior is fixed size. Using the resizeBehaviour method you can specify minimum and maximum values for the size of the genome. If you specify minimum and maximum as the same values then fixed size is assumed. If you use the resize method to specify a size that is outside the bounds set earlier using resizeBehaviour, the bounds will be 'stretched' to accommodate the value you specify with resize.

Conversely, if the values you specify with resizeBehaviour conflict with the genome's current size, the genome will be resized to accommodate the new values.

When resizeBehaviour is called with no arguments, it returns the maximum size if the genome is resizable, or GAGenome::FIXED_SIZE if the size is fixed.

```
set
unset
```

Set/Unset the bits in the specified range. If you specify a range that is not represented by the genome, the range that you specified will be clipped to fit the genome.

genetic operators for this class

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

default genetic operators for this class

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

GA2DBinaryStringGenome

The binary string genome is derived from the GABinaryString and GAGenome classes. It is a matrix of 1s and 0s whose width and height may be fixed or variable. The genes in this genomes are bits. The alleles for each bit are 0 and 1.

see also: GABinaryString

see also: GAGenome

class hierarchy

```
class GA2DBinaryStringGenome : public GABinaryString, public GAGenome
```

constructors

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

member function index

```
short gene(unsigned int x, unsigned int y) const
short gene(unsigned int x, unsigned int y, const short value)
int width() const
int width(int w)
int height() const
int height(int h)
int resize(int x, int y)
int resizeBehaviour(GADimension which) const
int resizeBehaviour(GADimension which, unsigned int min, unsigned int max)
int resizeBehaviour(unsigned int minx, unsigned int maxx, unsigned int miny, unsigned int
       maxy)
void copy(const GA2DBinaryStringGenome &, unsigned int xdest, unsigned int ydest, unsigned
       int xsrc, unsigned int ysrc, unsigned int width, unsigned int height)
void set(unsigned int, unsigned int, unsigned int, unsigned int)
void unset(unsigned int, unsigned int, unsigned int, unsigned int)
```

member function descriptions

`copy`

Copy the specified bits from the designated genome. If you specify a range that is not represented by the genome, the range that you specified will be clipped to fit the genome.

`gene`

Set/Get the specified bit.

`height`

Set/Get the height of the bit string.

`resize`

Set the size of the genome to the specified dimensions.

`resizeBehaviour`

Set/Get the resize behavior. The min value specifies the minimum allowable length, the max value specifies the maximum allowable length. If min and max are equal, the genome is not resizable.

Use the resizeBehaviour and resize member functions to control the size of the genome. The default behavior is fixed size. Using the resizeBehaviour method you can specify minimum and maximum

values for the size of the genome. If you specify minimum and maximum as the same values then fixed size is assumed. If you use the resize method to specify a size that is outside the bounds set earlier using resizeBehaviour, the bounds will be 'stretched' to accommodate the value you specify with resize. Conversely, if the values you specify with resizeBehaviour conflict with the genome's current size, the genome will be resized to accommodate the new values.

When resizeBehaviour is called with no arguments, it returns the maximum size if the genome is resizable, or GAGenome::FIXED_SIZE if the size is fixed.

`set`
`unset`

Set/Unset the bits in the specified range. If you specify a range that is not represented by the genome, the range that you specified will be clipped to fit the genome.

`width`

Set/Get the width of the bit string.

genetic operators for this class

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

default genetic operators for this class

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

GA3DBinaryStringGenome

The binary string genome is derived from the GABinaryString and GAGenome classes. It is a three-dimensional block of 1s and 0s whose width, height, and depth can be fixed or variable. The genes in this genomes are bits. The alleles for each bit are 0 and 1.

see also: GABinaryString

see also: GAGenome

class hierarchy

```
class GA3DBinaryStringGenome : public GABinaryString, public GAGenome
```

constructors

```
GA3DBinaryStringGenome(unsigned int x, unsigned int y, unsigned int z, GAGenome::Evaluator
       objective = NULL, void * userData = NULL)
GA3DBinaryStringGenome(const GA3DBinaryStringGenome&)
```

member function index

```
short gene(unsigned int x, unsigned int y, unsigned int z) const
short gene(unsigned int x, unsigned int y, unsigned int z, short value)
int width() const
int width(int w)
int height() const
int height(int h)
int depth() const
int depth(int d)
int resize(int x, int y, int z)
int resizeBehaviour(GADimension which) const
int resizeBehaviour(GADimension which, unsigned int min, unsigned int max)
int resizeBehaviour(unsigned int minx, unsigned int maxx, unsigned int miny, unsigned int
       maxy, unsigned int minz, unsigned int maxz)
void copy(const GA3DBinaryStringGenome &, unsigned int xdest, unsigned int ydest, unsigned
       int zdest, unsigned int xsrc, unsigned int ysrc, unsigned int zsrc, unsigned int
       width, unsigned int height, unsigned int depth);
void set(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, unsigned
       int);
void unset(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, unsigned
       int);
```

member function descriptions

`copy`

Copy the specified bits from the designated genome. If you specify a range that is not represented by the genome, the range that you specified will be clipped to fit the genome.

`depth`

Set/Get the depth of the bit string.

`gene`

Set/Get the specified bit.

`height`

Set/Get the height of the bit string.

`resize`

Set the size of the genome to the specified dimensions.

`resizeBehaviour`

Set/Get the resize behavior. The min value specifies the minimum allowable length, the max value specifies the maximum allowable length. If min and max are equal, the genome is not resizable.

Use the resizeBehaviour and resize member functions to control the size of the genome. The default behavior is fixed size. Using the resizeBehaviour method you can specify minimum and maximum values for the size of the genome. If you specify minimum and maximum as the same values then fixed size is assumed. If you use the resize method to specify a size that is outside the bounds set earlier using resizeBehaviour, the bounds will be 'stretched' to accommodate the value you specify with resize. Conversely, if the values you specify with resizeBehaviour conflict with the genome's current size, the genome will be resized to accommodate the new values.

When resizeBehaviour is called with no arguments, it returns the maximum size if the genome is resizable, or GAGenome::FIXED_SIZE if the size is fixed.

`set`
`unset`

Set/Unset the bits in the specified range. If you specify a range that is not represented by the genome, the range that you specified will be clipped to fit the genome.

`width`

Set/Get the width of the bit string.

genetic operators for this class

```
GA3DBinaryStringGenome::UniformInitializer
GA3DBinaryStringGenome::SetInitializer
GA3DBinaryStringGenome::UnsetInitializer
GA3DBinaryStringGenome::FlipMutator
GA3DBinaryStringGenome::BitComparator
GA3DBinaryStringGenome::UniformCrossover
GA3DBinaryStringGenome::EvenOddCrossover
GA3DBinaryStringGenome::OnePointCrossover
```

default genetic operators for this class

initialization:	`GA3DBinaryStringGenome::UniformInitializer`
comparison:	`GA3DBinaryStringGenome::BitComparator`
mutation:	`GA3DBinaryStringGenome::FlipMutator`
crossover:	`GA3DBinaryStringGenome::OnePointCrossover`

GABin2DecGenome

This genome is an implementation of the traditional method for converting binary strings to decimal values. It contains a mechanism for customized encoding of the bit string; binary-to-decimal and one form of Gray coding are built in to the library. The default is binary-to-decimal mapping (counting in base 2). To use this genome, you must create a mapping of bits to decimal values by specifying how many bits will be used to represent what bounded numbers. The binary-to-decimal genome is derived from the 1DBinaryStringGenome class.

You must create a phenotype before you can instantiate this genome. The phenotype defines how bits should map into decimal values and vice versa. A single binary-to-decimal phenotype contains the number of bits used to represent the decimal value and the minimum and maximum decimal values to which the set of bits will map.

see also: GA1DBinaryStringGenome

see also: GABin2DecPhenotype

see also: GACrossover

class hierarchy

```
class GABin2DecGenome : public GA1DBinaryStringGenome
```

constructors

```
GABin2DecGenome(const GABin2DecPhenotype &, GAGenome::Evaluator objective = NULL, void *
      userData = NULL)
GABin2DecGenome(const GABin2DecGenome&)
```

member function index

```
const GABin2DecPhenotype& phenotypes(const GABin2DecPhenotype &)
const GABin2DecPhenotype& phenotypes() const
int nPhenotypes() const
float phenotype(unsigned int n) const
float phenotype(unsigned int n, float value)
void encoder(GABinaryEncoder)
void decoder(GABinaryDecoder)
```

member function descriptions

`encoder`
`decoder`

Use these member functions to set the encoder/decoder for the genome. These functions determine what method will be used for converting the binary bits to decimal numbers. The functions that you specify here must have the proper signature.

`nPhenotype`

Returns the number of phenotypes (i.e. the number of decimal values represented) in the genome.

`phenotypes`

Set/Get the mapping from binary to decimal numbers.

`phenotype`

Set/Get the specified phenotype.

default genetic operators for this class

initialization:	`GA1DBinaryStringGenome::UniformInitializer`
comparison:	`GA1DBinaryStringGenome::BitComparator`
mutation:	`GA1DBinaryStringGenome::FlipMutator`
crossover:	`GA1DBinaryStringGenome::OnePointCrossover`
de/encoding:	`GABinaryEncode/GABinaryDecode`

additional information

Conversion functions are defined for transforming strings of bits to decimal values and vice versa. The function prototypes for the encoding (decimal-to-binary) and decoding (binary-to-decimal) are defined as follows:

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

The library includes the following binary-to-decimal/decimal-to-binary converters:

`GABinaryEncode/GABinaryDecode`

Convert using a binary coding scheme.

`GAGrayEncode/GAGrayDecode`

Convert using a Gray coding scheme.

GAListGenome<T>

The list genome is a template class. It is derived from the GAGenome class as well as the GAList<> class. It can be used for order-based representations or variable length sequences as well as traditional applications of lists.

You must define an initialization operator for this class. The default initializer is NoInitializer - if you do not assign an initialization operator then you'll get errors about no initializer defined when you try to initialize the genome.

see also: GAList

see also: GAGenome

class hierarchy

```
class GAListGenome<T> : public GAList<T>, public GAGenome
```

constructors

```
GAListGenome(GAGenome::Evaluator objective = NULL, void * userData = NULL)
GAListGenome(const GAListGenome<T> &)
```

genetic operators for this class

```
GAListGenome<>::DestructiveMutator
GAListGenome<>::SwapMutator
GAListGenome<>::OnePointCrossover
GAListGenome<>::PartialMatchCrossover
GAListGenome<>::OrderCrossover
GAListGenome<>::CycleCrossover
```

default genetic operators for this class

initialization:	`GAGenome::NoInitializer`
comparison:	`GAGenome::NoComparator`
mutation:	`GAListGenome<>::SwapMutator`
crossover:	`GAListGenome<>::OnePointCrossover`

GARealGenome

The real number genome was designed to be used for applications whose representation requires an array of (possibly bounded) real number parameters. The elements of the array can assume bounded values, discretized bounded values, or enumerated values, depending on the type of allele set that is used to create the genome. You can mix the bounding within the genome by specifying an appropriate array of allele sets. The allele set defines the possible values that each element in the genome may assume.

The real number genome is a specialization of the array genome with alleles. The specialization is of type float. You must create an allele set or array of allele sets before you can instantiate this genome. If you create a real number genome using a single allele set, each element in the genome will use that allele set to determine its value. If you create a real number genome using an allele set array, the genome will have a length equal to the number of elements in the array and each element of the real number will be governed by the allele set corresponding to its location in the genome.

To use the real genome in your code, you must include the real genome header file in each of your files that uses the real genome. You must also include the real genome source file (it contains template specialization code) in one (and only one) of your source files. Including the real genome source file will force the compiler to use the real specializations. If you do not include the real genome source file you will get the generic array routines instead (and some of the allele methods will not work as expected).

see also: GA1DArrayAlleleGenome, GAAlleleSet, GAAlleleSetArray

class hierarchy

```
typedef GAAlleleSet<float>              GARealAlleleSet
typedef GAAlleleSetCore<float>          GARealAlleleSetCore
typedef GAAlleleSetArray<float>         GARealAlleleSetArray
typedef GA1DArrayAlleleGenome<float>    GARealGenome
```

constructors

```
GARealGenome(unsigned int length, const GARealAlleleSet &, GAGenome::Evaluator objective =
       NULL, void * userData = NULL)
GARealGenome(const GARealAlleleSetArray &, GAGenome::Evaluator objective = NULL, void *
       userData = NULL)
GARealGenome(const GARealGenome&)
```

genetic operators for this class

```
GARealGenome::UniformInitializer
GARealGenome::OrderedInitializer
GARealGenome::FlipMutator
GARealGenome::SwapMutator
GARealGaussianMutator
GARealGenome::UniformCrossover
GARealGenome::EvenOddCrossover
GARealGenome::OnePointCrossover
GARealGenome::TwoPointCrossover
GARealGenome::PartialMatchCrossover
GARealGenome::OrderCrossover
GARealGenome::CycleCrossover
```

default genetic operators for this class

initialization:	`GARealGenome::UniformInitializer`
comparison:	`GARealGenome::ElementComparator`
mutation:	`GARealGaussianMutator`
crossover:	`GARealGenome::UniformCrossover`

GAStringGenome

The string genome can be used for order-based applications, variable length string applications, or non-binary allele set alphabets. The allele set defines the possible values that each element in the string may assume.

The string genome is a specialization of the array genome with alleles. The specialization is of type char. You must create an allele set or array of allele sets before you can instantiate this genome.

If you create a string genome using a single allele set, each element in the genome will use that allele set to determine its value. If you create a string genome using an allele set array, the string will have a length equal to the number of elements in the array and each element of the string will be governed by the allele set corresponding to its location in the string.

To use the string genome in your code, you must include the string genome header file in each of your files that uses the string genome. You must also include the string genome source file (it contains template specialization code) in one (and only one) of your source files. Including the string genome source file will force the compiler to use the string specializations. If you do not include the string genome source file you will get the generic array routines instead (and some of the allele methods will not work as expected).

see also: GA1DArrayAlleleGenome, GAAlleleSet, GAAlleleSetArray

class hierarchy

```
typedef GAAlleleSet<char>          GAStringAlleleSet
typedef GAAlleleSetCore<char>            GAStringAlleleSetCore
typedef GAAlleleSetArray<char>           GAStringAlleleSetArray
typedef GA1DArrayAlleleGenome<char>      GAStringGenome
```

constructors

```
GAStringGenome(unsigned int length, const GAStringAlleleSet &, GAGenome::Evaluator
       objective = NULL, void * userData = NULL)
GAStringGenome(const GAStringAlleleSetArray &, GAGenome::Evaluator objective = NULL, void
       * userData = NULL)
GAStringGenome(const GAStringGenome&)
```

genetic operators for this class

```
GAStringGenome::UniformInitializer
GAStringGenome::OrderedInitializer
GAStringGenome::FlipMutator
GAStringGenome::SwapMutator
GAStringGenome::UniformCrossover
GAStringGenome::EvenOddCrossover
GAStringGenome::OnePointCrossover
GAStringGenome::TwoPointCrossover
GAStringGenome::PartialMatchCrossover
GAStringGenome::OrderCrossover
GAStringGenome::CycleCrossover
```

default genetic operators for this class

initialization:	`GAStringGenome::UniformInitializer`
comparison:	`GAStringGenome::ElementComparator`
mutation:	`GAStringGenome::FlipMutator`
crossover:	`GAStringGenome::UniformCrossover`

GATreeGenome<T>

The tree genome is a template class. It is derived from the GAGenome class as well as the GATree<> class. The tree genome can be used for direct manipulation of tree objects. It can be used to represent binary trees as well as non-binary trees.

You must define an initialization operator for this class. The default initializer is NoInitializer - if you do not assign an initialization operator then you'll get errors about no initializer defined when you try to initialize the genome.

see also: GATree

see also: GAGenome

class hierarchy

```
class GATreeGenome<T> : public GATree<T>, public GAGenome
```

constructors

```
GATreeGenome(GAGenome::Evaluator objective = NULL, void * userData = NULL)
GATreeGenome(const GATreeGenome<T> &)
```

genetic operators for this class

```
GATreeGenome<>::DestructiveMutator
GATreeGenome<>::SwapSubtreeMutator
GATreeGenome<>::SwapNodeMutator
GATreeGenome<>::OnePointCrossover
```

default genetic operators for this class

initialization:	`GAGenome::NoInitializer`
comparison:	`GAGenome::NoComparator`
mutation:	`GATreeGenome<>::SwapSubtreeMutator`
crossover:	`GATreeGenome<>::OnePointCrossover`

GAEvalData

The evaluation data object is a generic base class for genome- and/or population-specific data. Whereas the userData member of the genome is shared by all genomes in a population, the evalData member is unique to each genome. The base class defines the copy/clone interface for the evaluation data object. Your derived classes should use this mechanism. Any derived class must define a clone and copy member function. These will be called by the base class when the evaluation data is cloned/copied by the genomes/populations.

class hierarchy

```
class GAEvalData : public GAID
```

constructors

```
GAEvalData()
GAEvalData(const GAEvalData&)
```

member functions

```
GAEvalData* clone() const
void copy(const GAEvalData&)
```

GABin2DecPhenotype

The binary-to-decimal phenotype defines the mapping from binary string to decimal values. A mapping for a single binary-to-decimal conversion consists of a range of decimal values and a number of bits. For example, a map of 8 bits and range of [0,255] would use 8 bits to represent the numbers from 0 to 255, inclusive. This object does reference counting in order to minimize the memory overhead imposed by instantiating binary-to-decimal mappings.

constructors

```
GABin2DecPhenotype()
GABin2DecPhenotype(const GABin2DecPhenotype&)
```

member function index

```
void add(unsigned int nbits, float min, float max)
void remove(unsigned int which)
int size() const
int nPhenotypes() const
float min(unsigned int which) const
float max(unsigned int which) const
int length(unsigned int which) const
int offset(unsigned int which) const
void link(GABin2DecPhenotype&)
void unlink()
```

member function descriptions

add

Create a mapping that tells the phenotype that nbits should be used to represent a floating point number from min to max, inclusive.

link
unlink

The phenotype object does reference counting to reduce the number of instantiated objects. Use the link member to make a phenotype object refer to another. Use the unlink member to remove the connection. When you unlink, the phenotype makes its own copy of the mapping information.

length

Returns the number of bits required for the specified mapping.

max
min

Returns the maximum/minimum decimal value for the specified mapping.

offset

Returns the offset (in bits) for the specified mapping.

remove

Removes a single binary-to-decimal from the phenotype.

size

Returns the number of bits that the set of mappings requires for converting a decimal value to binary and back again.

GAAlleleSet<T>

The allele set class is a container for the different values that a gene may assume. It can contain objects of any type as long as the object has the =, ==, and != operators defined.

Allele sets may be enumerated, bounded, or bounded with discretization. For example, an integer allele set may be defined as {1,3,5,2,99,-53} (an enumerated set). A bounded float set may be defined such as [2,743) (the set of numbers from 2, inclusive, to 743, exclusive). A bounded, discretized set may defined such as [4.5,7.05](0.05) (the set of numbers from 4.5 to 7.5, inclusive, with increment of 0.05).

If you call the allele member function with no argument, the allele set picks randomly from the alleles it contains and returns one of them.

constructors

```
GAAlleleSet()
GAAlleleSet(unsigned int n, const T a[])
GAAlleleSet(const T& lower, const T& upper, GAAllele::BoundType
       lowerbound=GAAllele::INCLUSIVE, GAAllele::BoundType upperbound=GAAllele::INCLUSIVE)
GAAlleleSet(const T& lower, const T& upper, const T& increment, GAAllele::BoundType
       lowerbound=GAAllele::INCLUSIVE, GAAllele::BoundType upperbound=GAAllele::INCLUSIVE)
GAAlleleSet(const GAAlleleSet<T>& set)
```

member function index

```
GAAlleleSet<T> * clone() const
T add(const T& allele)
T remove(T& allele)
T allele() const
T allele(unsigned int i)
int size() const
T lower() const
T upper() const
T inc() const
GAAllele::BoundType lowerBoundType() const
GAAllele::BoundType upperBoundType() const
GAAllele::Type type() const
void link(GAAlleleSet<T>&) void unlink()
```

member function descriptions

`add`
`remove`

Add/Remove the indicated allele from the set. This method works only for enumerated allele sets. Both functions return zero if the operation was successful, non-zero status otherwise.

`lower`
`upper`

Returns the lower/upper bounds on the allele set. If the allele set is enumerated, lower returns the first element of the set and upper returns the last element of the set.

`inc`

Returns the increment of the allele set. If the set is not discretized, the first element or lower bounds of the set is returned.

`lowerBoundType`
`upperBoundType`

Returns GAAllele::INCLUSIVE or GAAllele::EXCLUSIVE to indicate the type of bound on the limits of the allele set. If no bounds have been defined, these method return GAAllele::NONE.

`link`
`unlink`

The alleleset object does reference counting to reduce the number of instantiated objects. Use the link member to make an alleleset object refer to the data in another. Use the unlink member to remove the connection. When you unlink, the alleleset makes its own copy of the set data.

`size`

Returns the number of elements in the allele set. This member is meaningful only for the enumerated allele set.

`type`

Returns GAAllele::ENUMERATED, GAAllele::BOUNDED, or GAAllele::DISCRETIZED to indicate the type of allele set that has been defined. The type of the allele set is specified by the creator used to instantiate the allele set.

GAAlleleSetArray<T>

The GAAlleleSetArray is a container object with an array of allele sets.

constructors

```
GAAlleleSetArray()
GAAlleleSetArray(const GAAlleleSet<T>&)
GAAlleleSetArray(const GAAlleleSetArray<T>&)
```

member function index

```
int size() const
const GAAlleleSet<T>& set(unsigned int i) const
int add(const GAAlleleSet<T>& s)
int add(unsigned int n, const T a[])
int add(const T& lower, const T& upper, GAAllele::BoundType lb=GAAllele::INCLUSIVE,
      GAAllele::BoundType ub=GAAllele::INCLUSIVE)
int add(const T& lower, const T& upper, const T& increment, GAAllele::BoundType
      lb=GAAllele::INCLUSIVE, GAAllele::BoundType ub=GAAllele::INCLUSIVE)
int remove(unsigned int)
```

member function descriptions

`add`

Use the add members to append an allele set to the end of the array. Each of the overloaded add members invokes a corresponding allele set creator, so you can use the appropriate add member for your particular allele set application.

`remove`

Remove the indicated allele set from the array. Returns zero if successful, non-zero otherwise.

`size`

Returns the number of allele sets in the array.

GAParameter and GAParameterList

The parameter list object contains information about how genetic algorithms should behave. Each parameter list contains an array of parameters. Each parameter is a name-value pair, where the name is a string (e.g. "number_of_generations") and the value is an int, float, double, char, string, boolean, or pointer.

Each parameter is uniquely identified by a pair of names: the full name and the short name. Associated with the names is a value. Each parameter also has a type from the enumerated list of types shown above. The GAParameter object automatically does type coercion of the pointer that is passed to it based upon the type that is passed to it upon its creation. The type cannot be changed once the parameter has been created.

typedefs and constants

```
enum GAParameter::Type {BOOLEAN, CHAR, STRING, INT, FLOAT, DOUBLE, POINTER};
```

constructors

```
GAParameter(const char* fn, const char* sn, Type tp, const void* v)
GAParameter(const GAParameter& orig)
```

member function index

```
void copy(const GAParameter&)
char* fullname() const
char* shrtname() const
const void* value() const
const void* value(const void* v) Type type() const
```

constructors

```
GAParameterList()
GAParameterList(const GAParameterList&)
```

member function index

```
int size() const
int get(const char*, void*) const
int set(const char*, const void*)
int set(const char* s, int v)
int set(const char* s, unsigned int v)
int set(const char* s, char v)
int set(const char* s, char* v)
int set(const char* s, double v)
int add(const char*, const char*, GAParameter::Type, const void*)
int remove();
GAParameter& operator[](unsigned int i) const
GAParameter& next()
GAParameter& prev()
GAParameter& current() const
GAParameter& first()
GAParameter& last()
GAParameter* operator()(const char* name)
int parse(int& argc, char **argv, GABoolean flag = gaFalse)
int write(const char* filename) const
int write(ostream& os) const
int read(const char* filename)
int read(istream& is)
ostream& operator<<(ostream& os, const GAParameterList& plist)
istream& operator>>(istream& is, GAParameterList& plist)
```

member function descriptions

`add`

Add a parameter with specified name, type, and default value to the parameter list. This becomes the current parameter.

`current`

Return a reference to the current parameter in the list.

`first`

Return a reference to the first parameter in the list. This becomes the current parameter.

`get`

Fills the contents of the space pointed to by ptr with the current value of the named parameter. Returns 0 if the parameter was found, non-zero otherwise.

`last`

Return a reference to the last parameter in the list. This becomes the current parameter.

`next`

Return a reference to the next parameter in the list. This becomes the current parameter.

`parse`

Parse an argument list (in command-line format) for recognized name-value pairs. If you pass gaTrue as the third argument then this method will post warnings about names that it does not recognize.

`prev`

Return a reference to the next parameter in the list. This becomes the current parameter.

`read`

Read a parameter list from the specified file or stream. set

Set the named parameter to the specified value. Returns 0 if the paramter was found and successfully set, non-zero otherwise. You can use either the full or short name to specify a parameter.

`size`

Returns the number of parameters in the parameter list.

`remove`

Remove the current parameter from the parameter list.

`write`

Write the parameter list to the specified file or stream.

GAStatistics

The statistics object contains information about the current state of the genetic algorithm objects. Every genetic algorithm contains a statistics object.

The statistics object defines the following enumerated constants for use by the selectScores member. They can be bitwise-ORed to specify desired combinations of components. Use the class name to refer to the values, for example GAStatistics::Mean | GAStatistics::Deviation

typedefs and constants

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

constructors

```
GAStatistics()
GAStatistics(const GAStatistics&)
```

member function index

```
void copy(const GAStatistics &);
float online() const
float offline() const
float initial(ScoreID w=Maximum) const
float current(ScoreID w=Maximum) const
float worstEver() const
float bestEver() const
int generation() const
float convergence() const
int selections() const
int crossovers() const
int mutations() const
int replacements() const
int nConvergence(unsigned int)
int nConvergence() const
int nBestGenomes(const GAGenome&, unsigned int)
int nBestGenomes() const
int scoreFrequency(unsigned int x)
int scoreFrequency() const
int flushFrequency(unsigned int x)
int flushFrequency() const
char* scoreFilename(const char *filename)
char* scoreFilename() const
int selectScores(int whichScores)
int selectScores() const
GABoolean recordDiversity(GABoolean flag)
GABoolean recordDiversity() const
void flushScores()
void update(const GAPopulation& pop)
void reset(const GAPopulation& pop)
const GAPopulation& bestPopulation() const
const GAGenome& bestIndividual(unsigned int n=0) const
int scores(const char* filename, ScoreID which=NoScores)
int scores(ostream& os, ScoreID which=NoScores)
int write(const char* filename) const
int write(ostream& os) const;
ostream& operator<<(ostream&, const GAStatistics&)
```

member function descriptions

`bestEver`

Returns the score of the best individual ever encountered.

`bestIndividual`

This function returns a reference to the best individual encountered by the genetic algorithm.

`bestPopulation`

This function returns a reference to a population containing the best individuals encountered by the genetic algorithm. The size of this population is specified using the nBestGenomes member function.

`convergence`

Returns the current convergence. Here convergence means the ratio of the nth previous best-of-generation to the current best-of-generation.

`crossovers`

Returns the number of crossovers that have occurred since initialization.

`current`

Returns the specified score from the current population.

`flushFrequency`

Set/Get the frequency at which the generational scores should be flushed to disk. A score frequency of 100 means that at every 100th recorded score the scores buffer will be appended to the scores file.

`flushScores`

Force a flush of the scores buffer to the score file.

`generation`

Returns the current generation number.

`initial`

Returns the specified score from the initial population.

`mutations`

Returns the number of mutations that have occurred since initialization.

`nBestGenomes`

Set/Get the number of unique best genomes to keep.

`nConvergence`

Set/Get the number of generations to use for the convergence measure. A value of 10 means the best-of-generation from 10 generations previous will be used for the convergence test.

`offline`

Returns the average of the best-of-generation scores.

`online`

Returns the average of all scores.

`recordDiversity`

This boolean option determines whether or not the diversity of the population will be calculated each generation. By default, this option is set to false.

`replacements`

Returns the number of replacements that have occurred since initialization.

`reset`

Reset the contents of the statistics object using the contents of the specified population.

`scoreFilename`

Set the name of the file to which the scores should be output. If the filename is set to nil, the scores will not be written to disk. The default filename is "generations.dat".

`scoreFrequency`

Set/Get the frequency at which the generational scores should be recorded. A score frequency of 1 means the scores will be recorded each generation. The default depends on the type of genetic algorithm that is being used.

`scores`

Print the generational scores to the specified stream. Output is tab-delimited with each line containing the generation number and the specified scores. You can specify which score you would like by logically ORing one of the score identifiers listed above. The order of the tab-delimited scores is as follows:

```
generation TAB  mean TAB  max TAB min TAB deviation TAB diversity NEWLINE
```

`selections`

Returns the number of selections that have occurred since initialization.

`selectScores`

This function is used to specify which scores should be saved to disk. The argument is the logical OR of the following values: Mean, Maximum, Minimum, Deviation, Diversity (all defined in the scope of the GAStatistics object). To record all of the scores, pass GAStatistics::AllScores.

`update`

Update the contents of the statistics object to reflect the state of the specified population.

`worstEver`

Returns the score of the worst individual ever encountered.

GAPopulation

The population object is a container for the genomes. It also contains population statistics such as average, maximum, and minimum genome objective scores. Each population contains a scaling object that is used to determine the fitness of its genomes. The population also contains a function used for selecting individuals from the population.

Whenever possible, the population caches the statistics. This means that the first call to one of the statistics members will be slower than subsequent calls.

You can customize the initialization, evaluation, and sort methods. Use the appropriate member function. Your customized functions must have the appropriate signature.

The default scaling scheme is linear scaling. The default evaluator invokes the objective function for each genome. The default selector is roulette wheel and uses the scaled (fitness) scores for its selections.

typedefs and constants

```
void (*GAPopulation::Initializer)(GAPopulation &)
void (*GAPopulation::Evaluator)(GAPopulation &)
enum SortBasis { RAW, SCALED };
enum SortOrder { LOW_IS_BEST, HIGH_IS_BEST };
enum Replacement { BEST = -1, WORST = -2, RANDOM = -3 };
```

constructors

```
GAPopulation()
GAPopulation(const GAGenome&, unsigned int popsize = gaDefPopSize)
GAPopulation(const GAPopulation&)
```

member function index

```
GAPopulation * clone() const
void copy(const GAPopulation&)
int size(unsigned int popsize)
int size() const
float sum() const
float ave() const
float var() const
float dev() const
float max() const
float min() const
float div() const
float div(unsigned int i, unsigned int j) const
float fitsum() const
float fitave() const
float fitmax() const
float fitmin() const
float fitvar() const
float fitdev() const
float psum(unsigned int i) const
int nevals() const
void touch()
void statistics(GABoolean flag = gaFalse) const
void diversity(GABoolean flag = gaFalse) const
void prepselect(GABoolean flag = gaFalse) const
GAGenome& select()
GASelectionScheme& selector() const
GASelectionScheme& selector(const GASelectionScheme&)
void scale(GABoolean flag = gaFalse) const
GAScalingScheme& scaling() const
GAScalingScheme& scaling(const GAScalingScheme&)
```

```
void sort(GABoolean flag = gaFalse, SortBasis basis = RAW) const
SortOrder order() const
SortOrder order(SortOrder flag)
void evaluate(GABoolean flag = gaFalse) const
GAPopulation::Evaluator evaluator(GAPopulation::Evaluator func)
GAPopulation::Evaluator evaluator(GAPopulation::Evaluator func)
void initialize()
GAPopulation::Initializer initializer(GAPopulation::Initializer func)
GAPopulation::Initializer initializer(GAPopulation::Initializer func)
GAGeneticAlgorithm * geneticAlgorithm() const
GAGeneticAlgorithm * geneticAlgorithm(GA&)
void * userData() const
void * userData(void * u)
GAEvalData * evalData() const
GAEvalData * evalData(const GAEvalData&)
GAGenome& individual(unsigned int x, SortBasis basis = RAW) const
GAGenome& best(unsigned int i = 0, SortBasis basis = RAW) const
GAGenome& worst(unsigned int i = 0, SortBasis basis = RAW) const
GAGenome * add(GAGenome *)
GAGenome * add(const GAGenome&)
GAGenome * remove(unsigned int i, SortBasis basis = RAW)
GAGenome * remove(GAGenome *)
GAGenome * replace(GAGenome *, int which = gaPopReplaceRandom, SortBasis basis = RAW)
GAGenome * replace(GAGenome *, GAGenome *)
void destroy(int w = WORST, SortBasis basis = RAW)
virtual void read(istream &)
virtual void write(ostream &) const
ostream& operator<<(ostream &, const GAPopulation &)
istream& operator>>(istream &, GAPopulation &)
```

member function descriptions

`add`

Add the specified individual to the population. If you call this method with a reference to a genome, the population will clone the genome. If you call this method with a pointer to a genome, the population will use the genome pointed to by the pointer. From then on the population is responsible for deleting the genome.

`ave`

Returns the average of the objective scores.

`best`

Returns a reference to the best individual in the population. Use the SortBasis flag to specify whether you want the best in terms of raw objective score or scaled (fitness) score.

`destroy`

Remove the specified individual from the population and free the memory used by that individual. Use the SortBasis flag to specify whether to use raw objective score or scaled (fitness) score when determining which genome to destroy.

`dev`

Returns the standard deviation of the objective scores.

`div`

Returns the diversity of the population. Diversity is a number between 0 and 1 where 0 indicates that each individual is completely different than every other individual. If you specify two indices, this

member function returns the diversity of the specified individuals (it invokes the comparison function for those individuals).

`evalData`

Set/Get the evaluation data for the population. This object is unrelated to any evaluation data objects used by the genomes in the population.

`evaluate`

Evaluate the population using the method set by the evaluator function. The default evaluator simply calls the evaluate member of each genome in the population. If you call this function with gaTrue then the population performs the evaluation even if it has already cached the evaluation results.

`evaluator`

Specifies which function to use to evaluate the population. The specified function must have the proper signature.

`fitave`

Returns the average of the fitness scores.

`fitdev`

Returns the standard deviation of the fitness scores.

`fitmax`

Returns the maximum fitness score.

`fitmin`

Returns the minimum fitness score.

`fitsum`

Returns the sum of the fitness scores.

`fitvar`

Returns the variance of the fitness scores.

`geneticAlgorithm`

Set/Get the genetic algorithm that 'owns' this population. A return value of nil indicates that the population is owned by no genetic algorithm.

`individual`

Returns a reference to the specified individual. Indices for individuals in the population start at 0 and go to size()-1. the 0th individual is the best individual when the population has been sorted. Use the SortBasis flag to specify whether you want the ith individual based upon the raw objective score or scaled (fitness) score.

`initialize`

Initialize the population using the method set by initializer. The default initializer simply calls the initialize method of each genome in the population.

`initializer`

Specifies which function to use to initialize the population. The specified function must have the proper signature.

`max`

Returns the maximum objective score in the population.

`min`

Returns the minimum objective score in the population.

`order`

Set/Get the sort order. A population may be sorted in two ways, highest-score-is-best or lowest-score-is-best.

`prepselect`

The function calls the selector's update method. It is typically called by the population before it does a selection.

`psum`

Returns the partial sum of the ith fitness score in the array of (sorted) fitness scores.

`remove`

Remove the specified individual from the population. The genome to be replaced can be specified by either an index or by pointer. This function returns a pointer to the genome that was removed from the population. The caller is responsible for the memory used by the returned genome. Use the SortBasis flag to specify whether to use raw objective score or scaled (fitness) score when determining which genome to remove.

`replace`

Replace the specified individual with the first argument. The genome to be replaced can be specified by either an index or by pointer. This function returns a pointer to the genome that was replaced. If no genome was replaced or the specified index or pointer is bogus, it returns nil. Use the SortBasis flag to specify whether to use raw objective score or scaled (fitness) score when determining which genome to replace.

`scale`

Scale the raw (objective) scores in the population using the scaling method. If you call this function with gaTrue then the scaled scores are recalculated even if the population has already cached them.

`scaling`

Set/Get the scaling method for this population.

`select`

Returns a reference to a genome from the population using the selection scheme associated with the population.

`selector`

Set/Get the selection method for this population.

`size`

Set/Get the number of individuals in the population. If you resize to a larger size, the new individuals will be initialized but not evaluated. If you resize to a smaller size, the best individuals will be kept.

`sort`

Sort the individuals in the population. Individuals may be sorted based upon their raw or scaled scores.

`statistics`

Calculate the population statistics. This method is automatically invoked whenever any of the population statistics are requested. If you call this function with gaTrue then the statistics are recalculated even if the population has already cached them.

`sum`

Returns the sum of the objective scores.

`touch`

The population object remembers its state so that it does not execute the evaluate or sort methods unless its state has been changed. If you want to force the population to execute any of its methods the next time they are invoked, invoke this method.

`userData`

Set/Get the user data pointer for the population. You can use the user data member to store a pointer to any object.

`var`

Returns the variance of the objective scores.

`worst`

Returns a reference to the worst individual in the population. Use the SortBasis flag to specify whether you want the worst in terms of raw objective score or scaled (fitness) score.

GAScalingScheme

The scaling object is embedded in the population object. The base scaling object is not instantiable. This object keeps track of the fitness scores (not the objective scores) of each individual in the population. The genomes that it returns are the genomes in the population to which it is linked; it does not make its own copies.

For details about how to write your own scaling scheme, see the customizations section.

constructors

```
GAScalingScheme()
GAScalingScheme(const GAScalingScheme& s)
```

member function index

```
virtual GAScalingScheme * clone() const
virtual void copy(const GAScalingScheme &)
virtual void evaluate(const GAPopulation & p)
```

built-in scaling schemes

GAlib contains a number of instantiable scaling objects derived from the base class. Here are the constructors for these scaling schemes:

```
GANoScaling()
```

The fitness scores are identical to the objective scores. No scaling takes place.

```
GALinearScaling(float c = gaDefLinearScalingMultiplier)
```

The fitness scores are derived from the objective scores using the linear scaling method described in Goldberg's book. You can specify the scaling coefficient. Negative objective scores are not allowed with this method. Objective scores are converted to fitness scores using the relation

$$f = a \bullet obj + b$$

where a and b are calculated based upon the objective scores of the individuals in the population as described in Goldberg's book.

```
GASigmaTruncationScaling(float c = gaDefSigmaTruncationMultiplier)
```

Use this scaling method if your objective scores will be negative. It scales based on the variation from the population average and truncates arbitrarily at 0. The mapping from objective to fitness score for each individual is given by

$$f = obj - (obj\_ave - c \bullet obj\_dev)$$

```
GAPowerLawScaling(int k = gaDefPowerScalingFactor)
```

Power law scaling maps objective scores to fitness scores using an exponential relationship defined as

$$f = obj^k$$

```
GASharing(GAGenome::Comparator func = 0,
          float cutoff = gaDefSharingCutoff, float alpha = 1)
```

This scaling method is used to do speciation. The fitness score is derived from its objective score by comparing the individual against the other individuals in the population. If there are other similar individuals then the fitness is derated. The distance function is used to specify how similar to each other

two individuals are. A distance function must return a value of 0 or higher, where 0 means that the two individuals are identical (no diversity). For a given individual,

$$f = \frac{obj}{\sum_{j=0}^{n} s(d_j)}$$

$$s(d_j) = \begin{cases} d_j < \sigma \rightarrow 1 - \left(\frac{d_j}{\sigma}\right)^{\alpha} \\ d_j \geq \sigma \rightarrow 0 \end{cases}$$

d_j = distance between current individual and individual j

n = number of individuals in the population

The default sharing object uses the triangular sharing function described in Goldberg's book. You can specify the cutoff value (sigma in Goldberg's book) using the sigma member function. The curvature of the sharing function is controlled by the alpha value. When alpha is 1.0 the sharing function is a straight line (triangular sharing). If you specify a comparator, that function will be used as the distance function for all comparisons. If you do not specify a comparator, the sharing object will use the default comparator of each genome.

Notice that the sharing scaling differs depending on whether the objective is to maximized or minimized. If the goal is to maximize the objective score, the raw scores will be divided by the sharing factor. If the goal is to minimize the objective score, the raw scores will be multiplied by the sharing factor. You can explicitly tell the sharing object to do minimize- or maximize-based scaling by using the minimaxi member function. By default, it uses the min/max settings of the genetic algorithm that is using it (based on information in the population with which the sharing object is associated). If the scaling object is associated with a population that has been created independently of any genetic algorithm object, the sharing object will use the population's order to decide whether to multiply or divide to do its scaling.

GASelectionScheme

Selection schemes are used to pick genomes from a population for mating. The GASelectionScheme object defines the basic selector behavior. It is an abstract class and cannot be instantiated. Each selector object may be linked to a population from which it will make its selections. The select member returns a reference to a single genome. A selector may operate on the scaled objective scores or the raw objective scores. Default behavior is to operate on the scaled (fitness) scores.

For details about how to write your own selection scheme, see the customizations section.

typedefs and constants

```
enum { RAW, SCALED };
```

constructors

```
GASelectionScheme(int which = FITNESS)
GASelectionScheme(const GASelectionScheme&)
```

member function index

```
virtual GASelectionScheme* clone() const;
virtual void copy(const GASelectionScheme& orig)
virtual void assign(GAPopulation& pop)
virtual void update()
virtual GAGenome& select() const;
```

built-in selection schemes

GAlib contains a number of instantiable scaling objects derived from the base class. Here are the constructors for these scaling schemes:

```
GARankSelector(int w=GASelectionScheme::SCALED)
```

The rank selector picks the best member of the population every time.

```
GARouletteWheelSelector(int w=GASelectionScheme::SCALED)
```

This selection method picks an individual based on the magnitude of the fitness score relative to the rest of the population. The higher the score, the more likely an individual will be selected. Any individual has a probability p of being chosen where p is equal to the fitness of the individual divided by the sum of the fitnesses of each individual in the population.

```
GATournamentSelector(int w=GASelectionScheme::SCALED)
```

The tournament selector uses the roulette wheel method to select two individuals then picks the one with the higher score. The tournament selector typically chooses higher valued individuals more often than the RouletteWheelSelector.

```
GADSSelector(int w=GASelectionScheme::SCALED)
```

The deterministic sampling selector (DS) uses a two-staged selection procedure. In the first stage, each individual's expected representation is calculated. A temporary population is filled using the individuals with the highest expected numbers. Any remaining positions are filled by first sorting the original individuals according to the decimal part of their expected representation, then selecting those highest in the list. The second stage of selection is uniform random selection from the temporary population.

```
GASRSSelector(int w=GASelectionScheme::SCALED)
```

The stochastic remainder sampling selector (SRS) uses a two-staged selection procedure. In the first stage, each individual's expected representation is calculated. A temporary population is filled using the individuals with the highest expected numbers. Any fractional expected representations are used to give the individual more likeliehood of filling a space. For example, an individual with e of 1.4 will have 1 position then a 40% chance of a second position. The second stage of selection is uniform random selection from the temporary population.

```
GAUniformSelector(int w=GASelectionScheme::SCALED)
```

The stochastic uniform sampling selector picks randomly from the population. Any individual in the population has a probability p of being chosen where p is equal to 1 divided by the population size.

GAArray<T>

The GAArray<T> object is defined for your convenience so that you do not have to create your own array object. It is a template-ized container class whose elements can contain objects of any type. The 1-, 2-, and 3-dimensional arrays used in GAlib are all based upon this single-dimensional array object. This object is defined in the file arraytmpl.h.

The squares are elements in the array. Arrays are 1 dimensional, but derived classes can have 2 or more dimensions. Each element contains a user-specified object.

Any object in the array must have the following methods defined and publicly available:

copy constructor	operator ==
operator =	operator !=

The elements in an array are indexed starting with 0 (the first element in the array is element number 0). The last element in array with n elements is element n-1.

constructors

```
GAArray(unsigned int)
GAArray(const GAArray<T>&)
```

member function index

```
GAArray<T> & operator=(const GAArray<T>& orig)
GAArray<T> & operator=(const T array [])
GAArray<T> * clone()
const T & operator[](unsigned int i)
const T & operator[](unsigned int i)
void copy(const GAArray<T>& orig)
void copy(const GAArray<T>& orig, unsigned int dest, unsigned int src, unsigned int
       length)
void move(unsigned int dest, unsigned int src, unsigned int length)
void swap(unsigned int i, unsigned int j)
int size() const
int size(unsigned int n)
int equal(const GAArray<T>& b, unsigned int dest, unsigned int src, unsigned int length)
       const
int operator==(const GAArray<T>& a, const GAArray<T>& b)
int operator!=(const GAArray<T>& a, const GAArray<T>& b)
```

member function descriptions

`clone`

Return a pointer to an exact duplicate of the original array. The caller is responsible for the memory allocated by the call to this function.

`copy`

Duplicate the specified array or part of the specified array. If duplicating a part of the specified array, length elements starting at position src in the original are copied into position dest in the copy. If there is not enough space in the copy, the extra elements are not copied.

`equal`

Return 1 if the specified portion of the two arrays is identical, return 0 otherwise.

`move`

Move the number of elements specified with length from position src to position dest.

`operator[]`

Return a reference to the contents of the ith element of the array.

`size`

Return the number of elements in the array.

`swap`

Swap the contents of element i with the contents of element j.

GABinaryString

The binary string object is a simple implementation of a string of bits. Each bit is represented by a single word of memory (no fancy bit-munging happens here). The binary string class defines the following member functions. Binary strings are resizable.

constructors

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

member function index

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

member function descriptions

`copy`

Makes an exact copy of the specified string. If invoked with a range of bits then copies the specified range of bits.

`bit`

Set/Get the specified bit.

`equal`

Returns 1 if the specified range of bits are equal, 0 otherwise.

`move`

Move length bits starting at src to dest.

`set/unset`

Set/Unset length bits starting at a

`size`
`resize`

Set/Get the length of the bit string.

`randomize`

Set to random values length bits starting at a

GAList<T> and GAListIter<T>

The GAList<T> object is defined for your convenience so that you do not have to create your own list object. It is a template-ized container class whose nodes can contain objects of any type. The GAList<T> object is circular and doubly-linked. A list iterator object is also defined to be used when moving around the list to keep track of the current, next, previous, or whichever node. Iterators do not change the state of the list.

The circles are nodes in the list. Each node contains a user-specified object; the initialization method determines the size of the list and the contents of each node. The list is circular and doubly linked.

The template-ized GAList<T> is derived from a generic list base class called GAListBASE. The template list is defined in listtmpl.h, the list base class is defined in listbase.h

Any object used in the nodes must have the following methods defined and publicly available:

copy constructor	operator ==
operator =	operator !=

Each list object contains an iterator. The list's traversal member functions (next, prev, etc) simply call the member functions on the internal iterator. You can also instantiate iterators external to the list object so that you can traverse the list without modifying its state.

The list base class defines constants for specifying where insertions should take place (these are relative to the node to which the iterator is currently pointing).

Nodes in the list are numbered from 0 to 1 less than the list size. The head node is node 0.

When you do an insertion, the list makes a copy of the specified object (allocating space for it in the process). The internal iterator is left pointing to the node which was just inserted. The insertion function uses the copy constructor member to do this, so the objects in your list must have a copy constructor defined. The new node is inserted relative to the current location of the list's internal iterator. Use the where flag to determine whether the new node will be inserted before or after the current node, or if the new node should become the head node of the list.

The remove member returns a pointer to the object that was in the specified node. You are responsible for deallocating the memory for this object! The destroy member deallocates the memory used by the object in the current node. In both cases the iterator is left pointing to the node previous to the one that was deleted.

All of the list traversal functions (prev, next, current, etc) return a pointer to the contents of the node on which they are operating. You should test the pointer to see if it is NULL before you dereference it. When you call any of the traversal functions, the list's internal iterator is left pointing to the node to which traversal function moved. You can create additional iterators (external to the list) to keep track of multiple positions in the list.

typedefs and constants

```
GAListBASE::HEAD
GAListBASE::TAIL
GAListBASE::BEFORE
GAListBASE::AFTER
```

constructors

```
GAListIter(const GAList<T> &)
```

member function index

```
T * current()
T * head()
T * tail()
T * next()
T * prev()
T * warp(const GAList<T>& t)
T * warp(const GAListIter<T>& i)
T * warp(unsigned int i)
```

constructors

```
GAList()
GAList(const T& t)
GAList(const GAList<T>& orig)
```

member function index

```
GAList<T> * clone()
void copy(const GAList<T>& orig)
void destroy()
void swap(unsigned int, unsigned int)
T * remove()
void insert(GAList<T> * t, GAListBASE::Location where=AFTER)
void insert(const T& t, GAListBASE::Location where=AFTER)
T * current()
T * head()
T * tail()
T * next()
T * prev()
T * warp(unsigned int i)
T * warp(const GAListIter<T>& i)
T * operator[](unsigned int i)
int size() const
```

member function descriptions

These functions change the state of the list.

`clone`

Return a pointer to an exact duplicate of the original list. The caller is responsible for the memory allocated by the call to this function.

`copy`

Duplicate the specified list.

`destroy`

Destroy the current node in the list. This function uses the location of the internal iterator to determine which node should be destroyed. If the head node is destroyed, the next node in the list becomes the head node.

`insert`

Add a node or list to the list. The insertion is made relative to the location of the internal iterator. The where flag specifies whether the insertion should be made before or after the current node.

`remove`

Returns a pointer to the contents of the current node and removes the current node from the list. The iterator moves to the previous node. The caller is responsible for the memory used by the contents.

`swap`

Swap the positions of the two specified nodes. The internal iterator is not affected. If the iterator was pointing to one of the nodes before the swap it will still point to that node after the swap, even if that node was swapped.

> *These functions do not change the contents of the list, but they change the state of the list's internal iterator (when invoked on a list object).*

`current`

Returns a pointer to the contents of the current node.

`head`

Returns a pointer to the contents of the first node in the list.

`next`

Returns a pointer to the contents of the next node.

`operator[]`

Returns a pointer to the contents of the ith node in the list (same as warp).

`prev`

Returns a pointer to the contents of the previous node.

`tail`

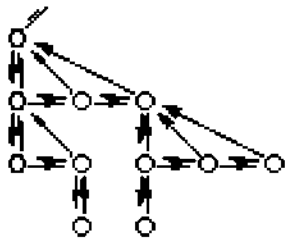
Returns a pointer to the contents of the last node in the list.

`warp`

Returns a pointer to the contents of the ith node in the list, or a pointer to the element in the list pointed to by the specified iterator. The head node is number 0.

GATree<T> and GATreeIter<T>

The GATree<T> object is defined for your convenience so that you do not have to create your own tree object. It is a template-ized container class whose nodes can contain objects of any type. Each level in the GATree<T> object is a circular and doubly-linked list. The eldest child of a level is the head of the linked list, each child in a level points to its parent, and the parent of those children points to the eldest child. Any tree can have only one root node. Any node can have any number of children. A tree iterator is also defined to be used when moving around the list to keep track of the current, next, parent, or whichever node. Iterators do not change the state of the tree.



The circles are nodes in the tree. Each node contains a user-specified object; the initialization method determines the tree topology and the contents of each node. Each tree contains one (and only one) root node. Each level in the tree is a circular, doubly linked list. The head of each list is called the 'eldest' child, each node in a level has a link to its parent, and each parent has a link to the eldest of its children (if it has any children).

The template-ized GATree<T> is derived from a generic tree base class called GATreeBASE. The template tree is defined in treetmpl.h, the tree base class is defined in treebase.h

Any object used in the nodes have the following methods defined and publicly available:

copy constructor	operator ==
operator =	operator !=

Each tree object contains an iterator. The tree's traversal member functions (next, prev, etc) simply call the member functions on the internal iterator. You can also instantiate iterators external to the tree object so that you can traverse the tree without modifying its contents.

The tree base class defines constants for specifying where insertions should occur.

Nodes in a tree are numbered starting at 0 then increasing in a depth-first traversal of the tree. The root node is node 0. A tree can have only one root node, but any node in the tree can have any number of children.

When you do an insertion, the tree makes a copy of the specified object (allocating space for it in the process). The internal iterator is left pointing to the node which was just inserted. The insertion function uses the copy constructor member to do this, so the objects in your tree must have a copy constructor defined. The new node is inserted relative to the current location of the tree's internal iterator. Use the where flag to determine whether the new node will be inserted before, after, or below the current node, or if the new node should become the root node of the tree.

The remove member returns a pointer to a tree. The root node of this tree is the node at which the iterator was pointing. You are responsible for deallocating the memory for this tree! The destroy member deallocates the memory used by the object in the current node and completely destroys any subtree hanging on that node. In both cases, the iterator is left pointing to the elder child or parent of the node that was removed/destroyed.

All of the tree traversal functions (prev, next, current, etc) return a pointer to the contents of the node on which they are operating. You should test the pointer to see if it is NULL before you dereference it. Also, the iterator is left pointing to the node to which you traverse with each traversal function. You can create additional iterators (external to the tree) to keep track of multiple positions in the tree.

typedefs and constants

```
GATreeBASE::ROOT
GATreeBASE::BEFORE
GATreeBASE::AFTER
GATreeBASE::BELOW
```

constructors

```
GATreeIter(const GATree<T>& t)
```

member function index

```
T * current()
T * root()
T * next()
T * prev()
T * parent()
T * child()
T * eldest()
T * youngest()
T * warp(const GATree<T>& t)
T * warp(const GATreeIter<T>& i)
T * warp(unsigned int i)
int size()
int depth()
int nchildren()
int nsiblings()
```

constructors

```
GATree()
GATree(const T& t)
GATree(const GATree<T>& orig)
```

member function index

```
GATree<T> * clone()
void copy(const GATree<T>& orig)
void destroy()
void swaptree(GATree<T> * t)
void swaptree(unsigned int, unsigned int)
void swap(unsigned int, unsigned int)
GATree<T> * remove()
void insert(GATree<T> * t, GATreeBASE::Location where=BELOW)
void insert(const T& t, GATreeBASE::Location where=BELOW)
T * current()
T * root()
T * next()
T * prev()
T * parent()
T * child()
T * eldest()
T * youngest()
T * warp(unsigned int i)
T * warp(const GATreeIter<T>& i)
int ancestral(unsigned int i, unsigned int j) const
int size()
int depth()
int nchildren()
int nsiblings()
```

member function descriptions

These functions change the state of the tree.

`clone`

Return a pointer to an exact duplicate of the original tree. The caller is responsible for the memory allocated by the call to this function.

`copy`

Duplicate the specified tree.

`destroy`

Destroy the current node in the tree. If the node has children, the entire sub-tree connected to the node is destroyed as well. This function uses the location of the internal iterator to determine which node should be destroyed. If the root node is destroyed, the entire contents of the tree will be destroyed, but the tree object itself will not be deleted.

`insert`

Add a node or tree to the tree. The insertion is made relative to the location of the internal iterator. The where flag specifies whether the insertion should be made before, after, or below the current node.

`remove`

Returns a pointer to a new tree object whose root node is the (formerly) current node of the original tree. Any subtree connected to the node stays with the node. The iterator moves to the previous node in the current generation, or the parent node if no elder sibling exists. The caller is responsible for the memory used by the new tree.

`swap`

Swap the contents of the two specified nodes. Sub-trees connected to either node are not affected; only the specified nodes are swapped.

`swaptree`

Swap the contents of the two specified nodes as well as any sub-trees connected to the specified nodes.

> *These functions do not change the contents of the tree, but they change the state of the tree's internal iterator (when invoked on a tree object).*

`ancestral`

Returns 1 if one of the two specified nodes is the ancestor of the other, returns 0 otherwise.

`child`

Returns a pointer to the contents of the eldest child of the current node. If the current node has no children, this function returns NULL.

`current`

Returns a pointer to the contents of the current node.

`depth`

Returns the number of generations (the depth) of the tree. When called as the member function of a tree iterator, this function returns the depth of the subtree connected to the iterator's current node.

`eldest`

Returns a pointer to the contents of the eldest node in the current generation. The eldest node is the node pointed to by the 'child' function in the node's parent.

`nchildren`

Returns the number of children of the node to which the iterator is pointing.

`next`

Returns a pointer to the contents of the next node in the current generation.

`nsiblings`

Returns the number of nodes in the level of the tree as the node to which the iterator is pointing.

`parent`

Returns a pointer to the contents of the parent of the current node. If the current node is the root node, this function returns NULL.

`prev`

Returns a pointer to the contents of the previous node in the current generation.

`root`

Returns a pointer to the contents of the root node of the tree.

`size`

Returns the number of nodes in the tree. When called as the member function of a tree iterator, this function returns the size of the subtree connected to the iterator's current node.

`warp`

Returns a pointer to the contents of the ith node in the tree, or a pointer to the element in the tree pointed to by the specified iterator. The head node is number 0 then the count increases as a depth-first traversal of the tree.

`youngest`

Returns a pointer to the contents of the youngest node in the current generation.

Customizing GAlib

This document describes how to extend GAlib's capabilities by defining your own genomes and genetic operators. The best way to customize the behavior of an object is to derive a new class. If you do not want to do that much work, GAlib is designed to let you replace behaviors of existing objects by defining new functions.

Deriving your own genome class

You can create your own genome class by multiply-inheriting from the base genome class and your own data type. For example, if you have already have an object defined, say MyObject, then you would derive a new genome class called MyGenome, whose class definition looks like this:

```
// Class definition for the new genome object, including statically
// defined declarations for default evaluation, initialization,
// mutation, and comparison methods for this genome class.

class MyGenome : public MyObject, public GAGenome {
public:
  GADefineIdentity("MyGenome", 201);
  static void Init(GAGenome&);
  static int Mutate(GAGenome&, float);
  static float Compare(const GAGenome&, const GAGenome&);
  static float Evaluate(GAGenome&);
  static int Cross(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);

public:
  MyGenome() : GAGenome(Init, Mutate, Compare) {
    evaluator(Evaluate);
    crossover(Cross);
  }
  MyGenome(const MyGenome& orig) { copy(orig); }
  virtual ~MyGenome() {}
  MyGenome& operator=(const GAGenome& orig){
    if(&orig != this) copy(orig);
    return *this;
  }
  virtual GAGenome* clone(CloneMethod) const
    {return new MyGenome(*this);}
  virtual void copy(const GAGenome& orig) {
    GAGenome::copy(orig);  // this copies all of the base genome parts
    // copy any parts of MyObject here
    // copy any parts of MyGenome here
  }
  // any data/member functions specific to this new class
};

void
MyGenome::Init(GAGenome&){
  // your initializer here
}

int
MyGenome::Mutate(GAGenome&, float){
  // your mutator here
}

float
MyGenome::Compare(const GAGenome&, const GAGenome&){
```

```
  // your comparison here
}

float
MyGenome::Evaluate(GAGenome&){
  // your evaluation here
}

int
MyGenome::Cross(const GAGenome& mom, const GAGenome& dad, GAGenome* sis, GAGenome* bro){
  // your crossover here
}
```

By convention, one of the arguments to a derived genome constructor is the objective function. Alternatively (as illustrated in this example), you can hard code a default objective function into your genome - just call the evaluator member somewhere in your constructor and pass the function you want used as the default.

Once you have defined your genome class, you should define the intialization, mutation, comparison, and crossover operators for it. The comparison operator is optional, but if you do not define it you will not be able to use the diversity measures in the genetic algorithms and/or populations.

Note that the genetic comparator is not necessarily the same as the boolean operator== and operator!= comparators. The genetic comparator returns 0 if the two individuals are the same, -1 if the comparison fails for some reason, and a real number greater than 0 indicating the degree of difference if the individuals are not identical but can be compared. It may be based on genotype or phenotype. The boolean comparators, on the other hand, indicate only whether or not two individuals are identical. In most cases, the boolean comparator can simply call the genetic comparator, but in some cases it is more efficient to define different operators (the boolean comparators are called much more often than the genetic comparators, especially if no diversity is being measured).

To work properly with the GAlib, you must define the following:

```
MyGenome( -default-args-for-your-genome-constructor )
MyGenome(const MyGenome&)
virtual GAGenome* clone(GAGenome::CloneMethod) const
```

If your genome adds any non-trivial member data, you must define these:

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

To enable streams-based reading and writing of your genome, you should define these:

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

When you derive a genome, don't forget to use the _evaluated flag to indicate when the state of the genome has changed and an evaluation is needed. If a member function changes the state of your genome, that member function should set the _evaluated flag to gaFalse. GAlib uses the _evaluated flag to control its caching of the genome scores, so setting and unsetting the flag is critical. If your member functions do not unset the flag when they modify the contents of the genome, then the genome will not be (re)evaluated.

Assign a default crossover, mutation, initialization, and comparison method so that users don't have to assign one unless they want to.

It is a good idea to define an identity for your genome (especially if you will be using it in an environment with multiple genome types running around). Use the DefineIdentity macro (defined in id.h) to do this in your class definition. The DefineIdentity macro sets a class ID number and the name

that will be used in error messages for the class. You can use any number above 200 for the ID, but be sure to use a different number for each of your classes.

When run-time type information (RTTI) has stabilized across compilers, GAlib will probably use that instead of the Define/Declare identity macros.

Genome Initialization

The initializer takes a single argument: the genome to be initialized. The genome has already been allocated; the intializer only needs to populate it with appropriate contents.

Here is the implementation of an initializer for the GATreeGenome<int> class.

```
void
TreeInitializer(GAGenome & c) {
  GATreeGenome<int> &child=(GATreeGenome<int> &)c;
// destroy any pre-existing tree
  child.root();
  child.destroy();
// Create a new tree with depth of 'depth' and each eldest node
// containing 'n' children (the other siblings have none).
  int depth=2, n=3, count=0;
  child.insert(count++,GATreeBASE::ROOT);
  for(int i=0; i<depth; i++){ child.eldest();
    child.insert(count++);
  for(int j=0; j<n; j++) child.insert(count++,GATreeBASE::AFTER);
  }
}
```

Genome Mutation

The genome mutator takes two arguments: the genome that will receive the mutation(s) and a mutation probability. The exact meaning of the mutation probability is up to the designer of the mutation operator. The mutator should return the number of mutations that occured.

Most genetic algorithms invoke the mutation method on each newly generated offspring. So your mutation operator should base its actions on the value of the mutation probability. For example, an array of floats could flip a pmut-biased coin for each element in the array. If the coin toss returns true, the element gets a Gaussian mutation. If it returns false, the element is left unchanged. Alternatively, a single biased coin toss could be used to determine whether or not the entire genome should be mutated.

Here is an implementation of the flip mutator for the GA1DBinaryString class. This mutator flips a biased coin for each bit in the string.

```
int
GA1DBinStrFlipMutator(GAGenome & c, float pmut) {
  GA1DBinaryStringGenome &child=(GA1DBinaryStringGenome &)c;
  if(pmut <= 0.0) return(0);
  int nMut=0;
  for(int i=child.length()-1; i>=0; i--){
    if(GAFlipCoin(pmut)){
      child.gene(i, ((child.gene(i) == 0) ? 1 : 0));
      nMut++;
    }
  }
  return nMut;
}
```

Genome Crossover

The crossover method is used by the genetic algorithm to mate individuals from the population to form new offspring. Each genome should define a default crossover method for the genetic algorithms to use. The sexual and asexual member functions return a pointer to the preferred sexual and asexual mating methods, respectively. The crossover member function is used to change the preferred mating method. The genome does not have a member function to invoke the crossover; only the genetic algorithm can actually perform the crossover.

Some genetic algorithms use sexual mating, others use asexual mating. If possible, define both so that your genome will work with either kind of genetic algorithm. If your derived class does not define a cross method, an error message will be posted whenever crossover is attempted.

Sexual crossover takes four arguments: two parents and two children. If one child is nil, the operator should be able to generate a single child. The genomes have already been allocated, so the crossover operator should simply modify the contents of the child genome as appropriate. The crossover function should return the number of crossovers that occurred. Your crossover function should be able to operate on one or two children, so be sure to test the child pointers to see if the genetic algorithm is asking you to create one or two children.

Here is an implementation of the two-parent/one-or-two-child single point crossover operator for fixed-length genomes of the GA1DBinaryStringGenome class.

```
int
SinglePointCrossover(const GAGenome& p1, const GAGenome& p2, GAGenome* c1, GAGenome* c2){
GA1DBinaryStringGenome &mom=(GA1DBinaryStringGenome &)p1;
GA1DBinaryStringGenome &dad=(GA1DBinaryStringGenome &)p2;
  int n=0;
  unsigned int site = GARandomInt(0, mom.length());
  unsigned int len = mom.length() - site;
  if(c1){
    GA1DBinaryStringGenome &sis=(GA1DBinaryStringGenome &)*c1;
    sis.copy(mom, 0, 0, site);
    sis.copy(dad, site, site, len);
    n++;
  }
  if(c2){
    GA1DBinaryStringGenome &bro=(GA1DBinaryStringGenome &)*c2;
    bro.copy(dad, 0, 0, site);
    bro.copy(mom, site, site, len);
    n++;
  }
  return n;
}
```

Genome Comparison

The comparison method is used for diversity calculations. It compares two genomes and returns a number that is greater than or equal to zero. A value of 0 means that the two genomes are identical (no diversity). There is no maximum value for the return value from the comparator. A value of -1 indicates that the diversity could not be calculated.

Here is the comparator for the binary string genomes. It simply counts up the number of bits that both genomes share. In this example, we return a -1 if the genomes are not the same length.

```
float
GA1DBinStrComparator(const GAGenome& a, const GAGenome& b){   GA1DBinaryStringGenome
       &sis=(GA1DBinaryStringGenome &)a;       GA1DBinaryStringGenome
       &bro=(GA1DBinaryStringGenome &)b;
  if(sis.length() != bro.length()) return -1;
  float count = 0.0;
```

```
  for(int i=sis.length()-1; i>=0; i--)
    count += ((sis.gene(i) == bro.gene(i)) ? 0 : 1);
  return count/sis.length();
}
```

Genome Evaluation

The genome evaluator is the objective function for your problem. It takes a single genome as its argument. The evaluator returns a number that indicates how good or bad the genome is. You must cast the generic genome to the genome type that you are using. If your objective function works with different genome types, then use the genome object's className and/or classID member functions to determine the genome class before you do the casts.

Here is a simple evaluation function for a real number genome with a single element. The function tries to maximize a sinusoidal.

```
float
Objective(GAGenome& g){
  GARealGenome& genome = (GARealGenome &)g;
  return 1 + sin(genome.gene(0)*2*M_PI);
}
```

Population Initialization

This method is invoked when the population is initialized.

Here is an implemenation that invokes the initializer for each genome in the population.

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

Population Evaluation

This method is invoked when the population is evaluated. If your objective is population-based, you can use this method to set the score for each genome rather than invoking an evaluator for each genome.

Here is an implementation that invokes the evaluation method for each genome in the population.

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

Scaling Scheme

The scaling object does the transformation from raw (objective) scores to scaled (fitness) scores. The most important member function you will have to define for a new scaling object is the evaluate member function. This function calculates the fitness scores based on the objective scores in the population that is passed to it.

The GAScalingScheme class is a pure virtual (abstract) class and cannot be instantiated. To make your derived class non-virtual, you must define the clone and evaluate functions. You should also define the copy method if your derived class introduces any additional data members that require non-trivial copy.

The scaling class is polymorphic, so you should define the object's identity using the GADefineIdentity macro. This macro sets a class ID number and the name that will be used in error messages for the class. You can use any number above 200 for the ID, but be sure to use a different number for each of your objects.

Here is an implementation of sigma truncation scaling.

```
class SigmaTruncationScaling : public GAScalingScheme {
public:
  GADefineIdentity("SigmaTruncationScaling", 286);
  SigmaTruncationScaling(float m=gaDefSigmaTruncationMultiplier) :
    c(m) {}
  SigmaTruncationScaling(const SigmaTruncationScaling & arg)
    {copy(arg);}
  SigmaTruncationScaling & operator=(const GAScalingScheme & arg)
    { copy(arg); return *this; }
  virtual ~SigmaTruncationScaling() {}
  virtual GAScalingScheme * clone() const
    { return new SigmaTruncationScaling(*this); }
  virtual void evaluate(const GAPopulation & p);
  virtual void copy(const GAScalingScheme & arg){
    if(&arg != this && sameClass(arg)){
      GAScalingScheme::copy(arg);
      c=((SigmaTruncationScaling&)arg).c;
    }
  }
  float multiplier(float fm) { return c=fm; }
  float multiplier() const { return c; }

protected:
  float c;                        // std deviation multiplier
};

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

Selection Scheme

The selection object is used to pick individuals from the population. Before a selection occurs, the update method is called. You can use this method to do any pre-selection data transformations for your selection scheme. When a selection is requested, the select method is called. The select method should return a reference to a single individual from the population.

A selector may make its selections based either on the scaled (fitness) scores or on the raw (objective) scores of the individuals in the population. Note also that a population may be sorted either low-to-high or high-to-low, depending on which sort order was chosen. Your selector should be able to handle either order (this way it will work with genetic algorithms that maximize or minimize).

The selection scheme class is polymorphic, so you should define the object's identity using the GADefineIdentity macro. This macro sets a class ID number and the name that will be used in error messages for the class. You can use any number above 200 for the ID, but be sure to use a different number for each of your objects.

Here is an implementation of a tournament selector. It is based on the roulette wheel selector and shares some of the roulette wheel selector's functionality. In particular, this tournament selector uses the roulette wheel selector's update method, so it does not define its own. The select method does two fitness-proportionate selections then returns the individual with better score.

```
class TournamentSelector : public GARouletteWheelSelector {
public:
  GADefineIdentity("TournamentSelector", 255);
  TournamentSelector(int w=GASelectionScheme::FITNESS) :
    GARouletteWheelSelector(w) {}
  TournamentSelector(const TournamentSelector& orig) { copy(orig); }
  TournamentSelector& operator=(const GASelectionScheme& orig)
    { if(&orig != this) copy(orig); return *this; }
  virtual ~TournamentSelector() {}
  virtual GASelectionScheme* clone() const
    { return new TournamentSelector; }
  virtual GAGenome& select() const;
};

GAGenome &
TournamentSelector::select() const {
  int picked=0;
  float cutoff;
  int i, upper, lower;
  cutoff = GARandomFloat();
  lower = 0; upper = pop->size()-1;
  while(upper >= lower){
    i = lower + (upper-lower)/2;
    if(psum[i] > cutoff)
      upper = i-1;
    else
      lower = i+1;
  }
  lower = Min(pop->size()-1, lower);
  lower = Max(0, lower);
  picked = lower;
  cutoff = GARandomFloat();
  lower = 0; upper = pop->size()-1;
  while(upper >= lower){
    i = lower + (upper-lower)/2;
    if(psum[i] > cutoff)
      upper = i-1;
    else
      lower = i+1;
  }
  lower = Min(pop->size()-1, lower);
  lower = Max(0, lower);
  GAPopulation::SortBasis basis =
      (which == FITNESS ? GAPopulation::SCALED : GAPopulation::RAW);
  if(pop->order() == GAPopulation::LOW_IS_BEST){
    if(pop->individual(lower,basis).score() <
       pop->individual(picked,basis).score())
      picked = lower;
  }
  else{
    if(pop->individual(lower,basis).score() >
       pop->individual(picked,basis).score())
      picked = lower;
  }
  return pop->individual(picked,basis);
}
```

Genetic Algorithm

Here is a sample derived class that does restricted mating. In this example, one of the parents is selected as usual. The second individual is select as the first, but it is used only if it is similar to the first individual. If not, we make another selection. If enough selections fail, we take what we can get.

```
class RestrictedMatingGA : public GASteadyStateGA {
public:
  GADefineIdentity("RestrictedMatingGA", 288);
  RestrictedMatingGA(const GAGenome& g) : GASteadyStateGA(g) {}
  virtual ~RestrictedMatingGA() {}
  virtual void step();
  RestrictedMatingGA & operator++() { step(); return *this; }
};

void
RestrictedMatingGA::step() {
  int i, k;
  for(i=0; i<tmpPop->size(); i++){
    mom = &(pop->select());
    k=0;
    do {
      k++; dad = &(pop->select());
    } while(mom->compare(*dad) < THRESHOLD && k<pop->size());
    stats.numsel += 2;
    if(GAFlipCoin(pCrossover()))
      stats.numcro += (*scross)(*mom, *dad, &tmpPop->individual(i), 0);
    else
      tmpPop->individual(i).copy(*mom);
    stats.nummut += tmpPop->individual(i).mutate(pMutation());
  }
  for(i=0; i<tmpPop->size(); i++)
    pop->add(tmpPop->individual(i));
  pop->evaluate();          // get info about current pop for next
  time pop->scale();                // remind the population to do its scaling
  for(i=0; i<tmpPop->size(); I++)
    pop->destroy(GAPopulation::WORST, GAPopulation::SCALED);
  stats.update(*pop);               // update the statistics by one generation
}
```

Termination Function

The termination function determines when the genetic algorithm should stop evolving. It takes a genetic algorithm as its argument and returns gaTrue if the genetic algorithm should stop or gaFalse if the algorithm should continue.

Here are three examples of termination functions. The first compares the current generation to the desired number of generations. If the current generation is less than the desired number of generations, it returns gaFalse to signify that the GA is not yet complete.

```
GABoolean
GATerminateUponGeneration(GAGeneticAlgorithm & ga){
  return(ga.generation() < ga.nGenerations() ? gaFalse : gaTrue);
}
```

The second example compares the average score in the current population with the score of the best individual in the current population. If the ratio of these exceeds a specified threshhold, it returns gaTrue to signify that the GA should stop. Basically this means that the entire population has converged to a 'good' score.

```
// stop when pop average is 95% of best
const float desiredRatio = 0.95;
```

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

The third uses the population diversity as the criterion for stopping. If the diversity drops below a specified threshhold, the genetic algorithm will stop.

```
// stop when population diversity is below this
const float thresh = 0.01;

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

A faster method of doing a nearly equivalent termination is to use the population's standard deviation as the stopping criterion (this method does not require comparisons of each individual). Notice that this judges diversity based upon the genome scores rather than their actual genetic diversity.

```
// stop when population deviation is below this
const float thresh = 0.01;

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

Descriptions of the Examples

Each of the examples contains comments in the source files with complete description about what is going on. Here is a short summary of what each one of the examples does:

ex1

Fill a 2DBinaryStringGenome with alternating 0s and 1s using a SimpleGA.

ex2

Generate a sequence of random numbers, then use a Bin2DecChromosome and SimpleGA to try and match the sequence. This example shows how to use the user-data member of genomes in objective functions.

ex3

Read a 2D pattern from a data file then try to match the pattern using a 2DBinaryStringGenome and a SimpleGA. This example also shows how to use the GAParametes object for setting genetic algorithm parameters and reading command-line arguments.

ex4

Fill a 3DBinaryStringChromosome with alternating 0s and 1s using a SteadyStateGA. This example uses many member functions of the genetic algorithm to control which statistics are recorded and dumped to file.

ex5

This example shows how to build a composite genome (a cell?) using a 2DBinaryStringGenome and a Bin2DecGenome. The composite genome uses behaviors that are defined in each of the genomes that it contains. The objective is to match a pattern and sequence of numbers.

ex6

Grow a GATreeGenome using a SteadyStateGA. This example illustrates the use of specialized methods to override the default initialization method and to specialize the output from a tree. It also shows how to use templatized genome classes. Finally, it shows the use of the parameters object to set default values then allow these to be modified from the command line. The objective function in this example tries to grow the tree as large as possible.

ex7

Identical in function to example 3, this example shows how to use the increment operator (++), completion measure, and other member functions of the GA. It uses a GA with overlapping populations rather than the non-overlapping GA in example 3 and illustrates the use of many of the GA member functions. It also illustrates the use of the parameter list for reading settings from a file, and shows how to stuff a genome with data from an input stream.

ex8

Grow a GAListGenome using a GA with overlapping populations. This shows how to randomly initialize a list of integers, how to use the sigma truncation scaling object to handle objective scores that may be positive or negative, and the 'set' member of the genetic algorithm for controlling statistics and other genetic algorithm parameters.

ex9

Find the maximum value of a continuous function in two variables. This example uses a GABin2DecGenome and simple GA. It also illustrates how to use the GASigmaTruncationScaling object (rather than the default linear scaling). Sigma truncation is particularly useful for objective functions that return negative values.

ex10

Find the maximum value of a continuous, periodic function. This example illustrates the use of sharing to do speciation. It defines a sample distance function (one that does the distance measure based on the genotype, the other based on phenotype). It uses a binary- to-decimal genome to represent the function values.

ex11

Generate a sequence of descending numbers using an order-based list. This example illustrates the use of a GAListGenome as an order-based chromosome. It contains a custom initializer and shows how to use this custom initializer in the List genome.

ex12

Alphabetize a sequence of characters. Similar to example 11, this example illustrates the use of the GAStringGenome (rather than a list) as an order-based chromosome.

ex13

This program runs a GA-within-GA. The outer level GA tries to match the pattern read in from a file. The inner GA tries to match a sequence of randomly generated numbers (the sequence is generated at the beginning of the program's execution). The inner level GA is run only when the outer GA reaches a threshhold objective score.

ex14

Another illustration of how to use composite chromosomes. In this example, the composite chromosome contains a user-specifiable number of lists. Each list behaves differently and is not affected by mutations, crossovers, or initializations of the other lists.

ex15

The completion function of a GA determines when it is "done". This example uses the convergence to tell when the GA has reached the optimum (the default completion measure is number-of-generations). It uses a binary-to-decimal genome and tries to match a sequence of randomly generated numbers.

ex16

Tree chromosomes can contain any kind of object in the nodes. This example shows how to put a point object into the nodes of a tree to represent a 3D plant. The objective function tries to maximize the size of the plant.

ex17

Array chromsomes can be used when you need tri-valued alleles. This example uses a 2D array with trinary alleles.

ex18

This example compares the performance of three different genetic algorithms. The genome and objective function are those used in example 3, but this example lets you specify which type of GA you want to use to solve the problem. You can use steady state, simple, or incremental just by specifying

one of them on the command line. The example saves the generational data to file so that you can then plot the convergence data to see how the performance of each genetic algorithm compares to the others.

ex19

The 5 DeJong test problems.

ex20

Holland's royal road function. This example computes Holland's 1993 ICGA version of the Royal Road problem. Holland posed this problem as a challenge to test the performance of genetic algorithms and challenged other GA users to match or beat his performance.

ex21

This example illustrates various uses of the allele set in array genomes. The allele set may be an enumerated list of items or a bounded range of continuous values, or a bounded set of discrete values. This example shows how each of these may be used in combination with a real number genome.

ex22

This example shows how to derive a new genetic algorithm class in order to customize the replacement method. Here we derive a new type of steady-state genetic algorithm in which speciation is done more effectively by not only scaling fitness values but also by controlling the way new individuals are inserted into the population.

ex23

The genetic algorithm object can either maximize or minimize your objective function. This example shows how to use the minimize abilities of the genetic algorithm. It uses a real number genome with one element to find the maximum or minimum of a sinusoid.

ex24

This example shows how to restricted mating using a custom genetic algorithm and custom selection scheme. The restricted mating in the genetic algorithm tries to pick individuals that are similar (based upon their comparator). The selector chooses only the upper half of the population (so it cannot choose very bad individuals, unlike the roulette wheel selector, for example).

ex25

Multiple populations on a single CPU. This example uses the genetic algorithm class called a 'DemeGA'. The genetic algorithm controls the migration behavior for moving individuals between populations. In this example, the island model is used with a stepping-stone migration behavior in which the best individuals from each population migrate to their nearest neighboring population. You can easily modify both the migration algorithm and the population behaviors by deriving a new class from the DemeGA.

ex26

Travelling Salesperson Problem. Although genetic algorithms are not the best way to solve the TSP, we include an example of how it can be done. This example uses an order-based list as the genome to figure out the shortest path that connects a bunch of towns such that each town is visited exactly once. It uses the edge recombination crossover operator (you can try it with the partial match crossover as well to see how poorly PMX does on this particular problem).

ex27

Deterministic crowding. Although the algorithms built-in to GAlib allow you to do quite a bit of customization, sometimes you'll want to derive your own class so that you can really tweak the way the

algorithm works. This example shows one way of implementing the deterministic crowding method by deriving an entirely new genetic algorithm class.

`graphic`[1]

You can learn a great deal by watching the genetic algorithm evolve. This example has a simple X windows interface that lets you start, stop, restart, and incrementally evolve a population of indivdiuals. You can see the evolution in action, so it becomes very obvious if your operators are not working correctly or if the algorithm is converging prematurely.

The directory contains two different examples. In the first you can choose between 3 different genetic algorithms, 2 different genomes (real or binary-to-decimal), and 4 different functions. In the second you can watch a population of routes evolve for the travelling salesman problem. Both programs use X resources as well as command-line arguments to control their behavior. You can also use a standard GAlib settings file. These programs will compile using either the Motif or the athena widget set.

`gnu`[1]

This directory contains the code for an example that uses the BitString object from the GNU class library. The example illustrates how to incorporate an existing object (in this case the BitString) into a GAlib Genome type. The gnu directory contains the source code needed for the BitString object (taken from the GNU library) plus the two files (bitstr.h and bitstr.C) needed to define the new genome type and the example file that runs the GA (gnuex.C).

`pvmind`[1]

This directory contains code that illustrates how to use GAlib with PVM in a master-slave configuration wherein the master process is the genetic algorithm with a single population and each slave process is a genome evaluator. The master sends individual genomes to the slave processes to be evaluated then the slaves return the evaluations.

`pvmpop`[1]

This directory contains code that illustrates a PVM implementation of parallel populations. The master process initiates a cluster of slaves each of which contains a single population. The master process harvests individuals from all of the distributed populations. With a few modifications you can also use this example with the deme GA from example 25 (it uses migration to distribute diversity between pops).

`randtest`

Use this program to verify that the random number generator is generating suitably random numbers on your machine. This is by no means a comprehensive random number testor, but it will give you some idea of how well GAlib's random number generator is working.

---

[1] These examples are included only in the UNIX distribution.