České vysoké učení technické v Praze

Fakulta elektrotechnická

Bakalářský projekt

Syntéza generátoru testovacích vektorů pomocí genetických algoritmů

Pavel Galaktionov

Vedoucí práce: ing. Petr Fišer

Studijní program: Informatika a výpočetní technika

květen 2006

Poděkování

Rád bych na tomto místě poděkoval svým rodičům, kteří mi byli během celého mého studia neustálou oporou, zejména pak během práce na tomto textu i během vlastní implementace programu.

Děkuji svému vedoucímu bakalářského projektu, ing. Petru Fišerovi za písemné i ústní konzultace, za připomínky, návrhy i poskytnuté materiály.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu. Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb. , o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 16. května 2006 …….

Abstract

Main part of this work is about implementing genethic algorithm used to synthesis test vector generator of selected circuits. For each circuit we have a file that contains enter data. On each line of file is a vektor and its fault mask that represents errors which vector detects. Question is how to project test vector generator that with its vectors detects as many errors as possible.

Furthemore genetic algorithm is implemented grafic user interface that allows changing parameters of genetic algorithm during its work.

This thesis then compares results of genetic algorithm with results of randomly genera-ted vectors.

Anotace

Tato práce se zabývá implementací genetického algoritmu pro syntézu generátoru testovacích vektorů na vybraných typech obvodů. Pro každý obvod je k dispozici soubor se vstupními daty, který obsahuje vygenerované vektory a jim příslušné bitové masky, které vyjadřují, že příslušný vektor je schopen detekovat konkrétní poruchy. Účelem je navrhnout takový generátor vektorů, který by byl schopen svými vektory detekovat co nejvíce poruch. Kromě vlastního genetického algoritmu je implementováno i uživatelské rozhraní, pomocí něhož lze během vykonávání genetického algoritmu měnit parametry se kterými genetický algoritmus pracuje. Dále tato práce srovnává výsledky dosažené pomocí genetického algoitmu s výsledky, dosaženými bez použití genetického algoritmu prostým náhodným generováním vektorů.

Obsah

Seznam obrázků

Úvod

Ze složitosti současných VLSI (Very Large Scale Integration) zařízení, což jsou zařízení s vysokou hustotou integrace, kde na jeden čip připadá řádově několik desítek tisíc prvků vyplývá, že jednotlivé čipy již nejsou v rozumně krátké době otestovatelné standardními testy, které jsou prováděny při výrobě daného čipu, takzvanými ATE (Automated Test Equipment) testy. Protože se doba testů se velmi rychle prodlužovala, byly hledány nové techniky testování. Jednou z takovýchto technik je BIST (built-in self-test). Pomocí dané techniky je obvod schopen otestovat sám sebe bez nutnosti použití dalších ATE testů, či jiných prostředků. BIST napomáhá zkrátit dobu nutnou k otestování čipu a navíc paměťové nároky na takovýto test jsou menší, než při použití ATE testů. Technika BIST využívá testovacích vektorů, získaných buď deterministickými metodami, anebo, a to ve většině případů, získaných pomocí generátoru pseudonáhodných vektorů, který produkuje testovací vektory, jimiž se detekují, takzvané lehce detekovatelné poruchy, které tvoří až přes 90% všech chyb. Zbývající procenta chyb je nutno detekovat jinou metodou anebo testovacími vektory, které vygeneruje sama BIST technika.

1 Základní pojmy

Tato kapitola si klade za cíl seznámit ctenáře s pojmy, které se objevují v tomto textu.

1.1 Generátory testovacích vektorů

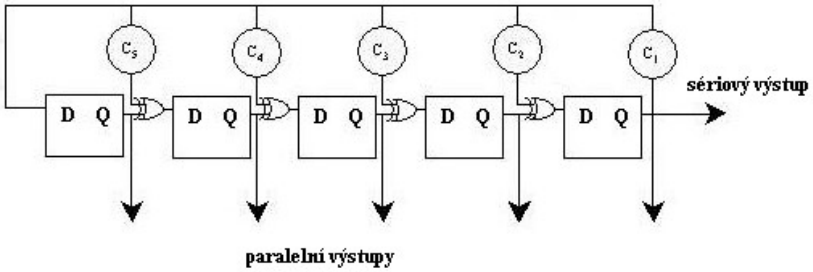
Generátory testovacích vektorů jsou takové struktury, které při své činnosti na výstupu produkují testovací vektory. Pod pojmem testovací vektor si lze představit libovolný řetězec určité délky, který je tvořen posloupností nul a jedniček. Nula reprezentuje logickou nulu, jednička logickou jedničku. Převedeno do výrazů Booleovy algebry, *1 = true (pravda), 0 = false (nepravda)*. Ve své práci se zabývám syntézou generátoru testovacích vektorů, který pokrývá co nejvíce poruch, k jejichž detekci slouží právě testovací vektory. Konkrétně to znamená, že pokud se v daném vektoru na určité pozici vyskytne jednička, znamená to, že vektor detekuje poruchu odpovídající pozici této jedničky. Dopodrobna je daná problematika rozebrána v kapitole 3.

Základní typy generátorů

Obecně lze říci, že generátor pseudonáhodných vektorů je jednoduchý obvod, který generuje kódová slova dle zadaného generujícího polynomu, který je blíže popsán v odstavci 1.2.2. Mezi základní typy generátorů testovacích vektorů řadíme Lineární zpětnovazební po-suvní registr (LZPR), viz 1.2.1 a celulární automat (CA), který je stručně popsán v článku 1.2.3.

1.2.1 Lineární zpětnovazební posuvní registr (LZPR)

Je nejběžněji používaný generátor. Jeho české pojmenování odpovídá anglickému ekvivalentu LFSR (Linear feedback shift registr) a pod tímto anglickým pojmenováním je také snáze vyhledatelný. Jedná se o lineární sekvenční obvod, který se skládá z klopných obvodů typu "D" a obvodů typu XOR generujících kódové slovo cyklického kódu, což je takový kód, pro který platí, že cyklickým posuvem kódového slova vznikne opět kódové slovo. Je zřejmé, že po určité periodě se začne generovat ta samá sekvence kódových slov. Kódovým slovem zde opět rozumíme binární vektor. U konkrétního cyklického kódu již musíme nutně rozlišovat kódová slova přípustná a kódová slova nepřípustná. Přípustná kódová slova jsou ta, která splňují podmínku definující cyklický kód. Naproti tomu kódová slova nepřípustná tuto podmínku, například důsledkem vzniku chyby, nesplňují. Schema LZPR je uvedeno na obrázku 1.1.



Obrázek 1.1: Schema lineárního zpětnovazebního registru

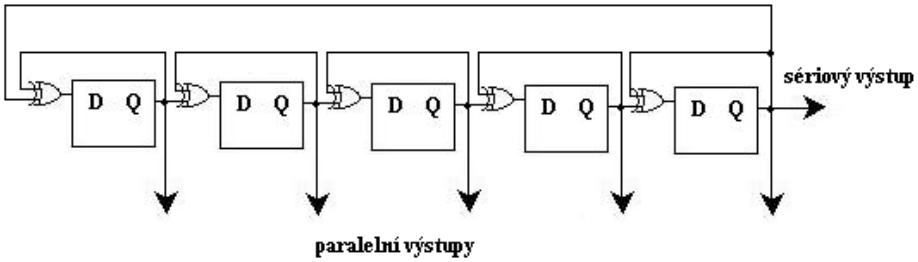
Koeficienty $C_5 - C_1$ odpovídají koeficientům v generujícím polynomu viz 1.2.2 a vyjadřují, zda v daném místě existuje (1), či neexistuje (0) spojení ze zpětné vazby do příslušného XORu.

1.2.2 Generující polynom

Sekvence kódových slov získaná pomocí LZPR může být popsána generujícím polynomem zapsaným v obecném tvaru $g(x) = x^n + c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + \ldots + c_1x^1 + 1$, kde *n* je počet paralelních výstupů. Generující polynom pro obrázek 1.1 by odpovídal zápisu $g(x) = x^6 + c_5x^5 + c_4x^4 + c_3x^3 + + c_2x^2 + c_1x + 1$. Počáteční nastavení registru, čili nastavení klopných obvodů se anglicky nazývá *seed*.

1.2.3 Celulární automat(CA)

Je sekvenční obvod podobný obvodu LZPR, uvedeného na obr. 1.1. Jeho perioda je ovšem často oproti LZPR kratší, nicméně kódová slova jím generovaná se jeví v mnoha případech vhodnější. Jelikož má práce je stavěna nad LZPR a ne nad CA, který zde byl uváděn spíše pro představu, uvedu na závěr této kapitoly již jen samotné schema celulárního automatu, které je znázorněno na obrázku 1.2.



Obrázek 1.2: Schema celulárního automatu

1.3 Genetické algoritmy (GA)

Genetika je věda, která se zabývá zkoumáním genetické informace živých organizmů. Každého jedince můžeme charakterizovat jeho zakódovanou genetickou informací, jež je uložena v chromozómech, a která určuje vlastnosti a chování a významným způsobem ovlivňuje životaschopnost daného jedince v dané populaci. Sama populace je tvořena určitým počtem jedinců. Genetické algoritmy jsou takové algoritmy, které napodobují přirozený výběr, který vidíme v přírodě. Slabší jedinci hynou a silnější přežívají a mají tak šanci předat svou genetickou informaci do nově zformované populace. Důležité je, že jedinci tvořící novou populaci nesou geny jedinců z populace předchozí, neboť každý jedinec v nové populaci vznikl, pokud neuvažujeme procento křížení rovno nule (viz.1.3.2), zkřížením dvou jedinců z populace předchozí, které považujeme za rodiče nově vzniklého jedince. V takto vzniklém jedinci se tedy mísí geny dvou rodičů a daný jedinec získává nové vlastnosti. Genetické algoritmy tedy pracují s populací jedinců, každého takového jedince nazýváme seed, kteří se v čase *t* vyvíjejí. Každý jedinec reprezentuje potenciální řešení daného problému a každé takovéto řešení je ohodnoceno ohodnocovací funkcí, takzvanou fitness, tak, aby se dala zjistit jeho úspěšnost. Poté je vytvořena nová populace pro čas *t + 1*. Nová populace je formována z nejlepších jedinců rodičovské populace, tzv. selekční krok. Někteří jedinci nové populace projdou transformací, tzv. generační krok, aby se vytvořila nová řešení. Nejčastěji se jedná o křížení(viz.1.3.2) a mutaci(viz.1.3.2). Cílem genetických algoritmů je tedy zvýšení kvality celé populace a nacházejí tak uplatnění při řešení optimalizačních úloh.

1.3.1 Obecné schema genetického algoritmu

Ačkoliv je zřejmé, že pro řešený problém bude existovat mnoho různých algoritmů, které se mohou lišit v mnoha ohledech, jako např. v rozdílné reprezentaci řešení, v rozdílném nas-

tavení počáteční populace, či v rozdílných způsobech implementace, lze obecně říci, že každý genetický algoritmus se musí skládat z následujících pěti fází:

genetické reprezentace potenciálního řešení daného problému
  inicializace počáteční populace
ohodnocovací funkce (tzv. fitness)
genetických operátorů (selekce, křížení a mutace)
parametrů ovlivňujících genetický algoritmus

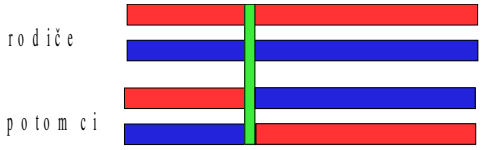
Jelikož při použití genetických algoritmů nepracujeme s živými jedinci, kteří mají svoji genetickou informaci uloženu v genech a posléze v chromozomech, ale s modely těchto jedinců, musíme zvolit vhodnou datovou strukturu, která by co nejlépe vyhovovala řešení našeho problému. Například můžeme použít binární řetězec, řetězec obsahující pouze nuly a jedničky, kde přítomnost jedničky na daném místě bude symbolizovat, že daný gen je přítomen, zatímco nula bude symbolizovat nepřítomnost daného genu. Fáze 1. tedy říká, že je nutné zvolit vhodnou reprezentaci. Fáze 2. říká, že před vlastním spuštěním genetického algoritmu potřebujeme nastavit počáteční, čili výchozí populaci, nad níž bude genetický algoritmus spuštěn. Záleží jen na nás, zda ji chceme nastavit přímo, s našimi parametry, nebo zda ji necháme vygenerovat náhodně. Samozřejmě reprezentace oné populace musí být opět taková, aby s ní algoritmus uměl správně pracovat. Aby daný genetický algoritmus byl schopen rozhodnout, které z potenciálních řešení je lepší a které hor- ší, přiřadíme ke každému řešení ohodnocovací funkci, dle jejíž hodnoty lze kvalitu řešení určit. Ohodnocovací funkce je tedy optimalizačním kritériem pro daný problém. To je podstatou fáze 3. Jelikož fáze 4. tvoří samu podstatu genetického algoritmu, jsou ony genetické operátory dopodrobna popsány v odstavci 1.3.2. A konečně fáze 5. říká, že průběh genetického algoritmu můžeme ovlivnit vlastním nastavením proměnných algoritmu. Jedná se zejména o parametr křížení, kde nastavujeme kolik potomků, které budou tvořit novou populaci, má vzniknout křížením a dále pak parametr mutace, který udává kolik potomků vzniklých křížením má ještě podstoupit mutaci. Pod pojmem 'kolik' můžeme chápat jednak přesné číslo, ale také, a to většinou tak chápeme a zadáváme, procentuální možnost provedení dané operace nad jedinci populace. Jako další parametry můžeme zmínit velikost počáteční populace, čili počet jedinců, nebo to, zda chceme používat elitismus (viz.1.3.5), či ne. Obecné doporučení nastavení parametrů je následující: pravděpodobnost křížení 80-90[%], pravděpodobnost mutace 1-2[%], velikost počáteční populace stovky jedinců a je doporučeno používat elitismus.

1.3.2 Fáze genetického algoritmu

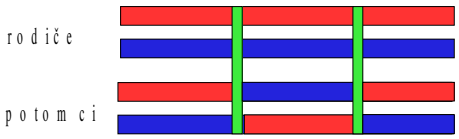
Samotné provádění genetického algoritmu, čili průběh po prvotní inicializaci vstupní populace, si můžeme popsat ve třech fázích, které se stále opakují a to tak dlouho, dokud není splněna podmínka ukončení běhu algoritmu. Onou podmínkou může být třeba počet iterací, nebo dosažení určité kvality populace, nebo jen dosažení určité hodnoty ohodnocovací funkce třeba jen pro jednoho jedince, a další. Jedná se tedy o fázi selekční, fázi křížení a fázi mutace. Při fázi selekční dochází ke kopírování jedinců z aktuální populace do populace nové. Jedinci s nejvyšší hodnotou ohodnocovací funkce mají největší šanci dostat se do nové populace, naopak jedinci s nízkou hodnotou ohodnocovací funkce mají šanci nejmenší. Existují dvě základní metody pro výběr daného jedince. Jedná se o metodu ruletového výběru (viz 1.3.3) a metodu turnaje (viz 1.3.4.) Samostatnou metodou je pak použití elitismu (viz 1.3.5). Při fázi křížení se provádí taková operace, kdy ze dvou rodičů vznikne potomek, který bude mít od každého z obou rodičů určitý počet genů, které spojením vytvoří novou genetickou výbavu

daného potomka. Samo křížení tedy probíhá následovně. Z dané populace jsou vybráni dva rodiče, pro názornost uvažujeme opět, že každý jedinec, tím pádem i rodič, je reprezentován binárním řetězcem. Čím větší má daný rodič hodnotu ohodnocovací funkce, tím pravděpodobněji bude vybrán. Poté, co jsou vybráni dva rodiče, vybereme místo v řetězci každého z nich, kde dojde k roztržení daného řetězce, ono místo bude místem křížení. První potomek těchto rodičů vznikne tak, že od prvího rodiče vezmeme první část roztrženého řetězce, od druhého rodiče část druhou, a tyto části spojíme. Druhý potomek vznikne spojením druhé části řetězce prvního rodiče s první částí řetězce rodiče druhého. Tito potomci jsou pak vloženi do populace na místa svých rodičů. Popsaný typ křížení je takzvané jednobodové křížení. Je samozřejmé, že při řešení konkrétního problémů můžeme využít i křížení, které se k danému problému hodí více, třeba křížení vícebodové, nebo křížení typu "cut and splice", při kterém dochází k prohození genetické informace obou rodičů za místem křížení, případně ještě potomky můžeme před vložením do populace na místa svých rodičů zmutovat. To již záleží na konkrétním řešeném problému.

Graficky si můžeme křížení ukázat na následujících obrázcích 1.5 a1.6. První rodič je označen červeně, druhý rodič modře, zelená čára znázorňuje místo, případně místa křížení.



Obrázek 1.5: Jednobodové křížení



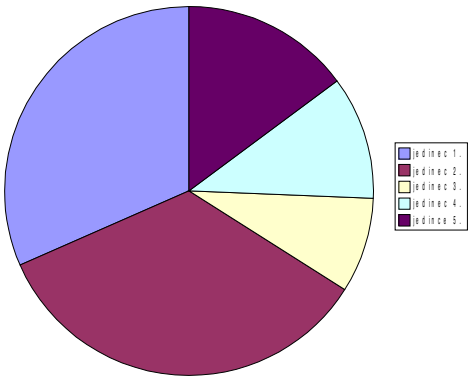
Obrázek 1.6: Dvoubodové křížení

Kromě zde již zmiňovaných typů křížení existují i další, např. křížení typu uniform crossover, nebo n-bodové křížení, případně jejich další modifikace.

Během fáze mutace dochází k náhodné změně genu, nebo genů. Realizujeme ji tak, že s pravděpodobností mutace vybereme jedince, kterého opět budeme pro jednoduchost reprezentovat binárním řetězcem, a na náhodně zvolené pozici, či pozicích v daném řetězci provedeme změnu. V binárním řetězci to znamená, že pokud je na dané pozici nula, zaměníme ji za jedničku a naopak. Mějme například takovýto řetězec 101011100011, o kterém předpokládáme, že byl vybrán pro operaci mutace a že jsme nastavili podmínku tak, že k mutaci má dojít na třech pozicích. Proveďme tedy mutaci na pozici 1, 5 a 10. Po provedení získáme řetězec 001001100111. Zvýrazněná místa představují pozice, kde došlo k mutaci.

1.3.3 Metoda ruletového výběru

Daná metoda je založena na podobnosti s klasickou herní ruletou. Mějme pomyslné ruletové kolo, skládající se místo z čísel, z kruhových výseků, jejichž počet odpovídá počtu jedinců v dané populaci. Jednotlivé kruhové výseky nejsou stejně velké, ale jejich velikost je úměrná hodnotě ohodnocovací funkce daného jedince, čili jedinci s vyšší hodnotou ohodnocovací funkce bude odpovídat větší kruhová výseč. Z onoho schematu na obrázku 1.4 tedy vyplývá, že jedinci, kteří na pomyslném kole zaujímají největší díl, budou nejčastěji vybíráni, a naopak jedinci, zaujímající díl nejmenší budou vybíráni nejméně často.



Obrázek 1.4: Schema ruletového výběru

1.3.4 Metoda turnaje

Jedná se o nejrychlejší metodu výběru. Z dané populace vždy vybereme dva a více jedinců, srovnáme hodnoty jejich ohodnocovacích funkcí a do nové populace umístíme jedince s vyšší hodnotou oné funkce. Jak je patrno, nejsilnější jedinec bude vždy vybrán a včleněn do nové populace, zatímco nejslabší jedinec nebude vybrán nikdy.

1.3.5 Metoda elitismu

Daná metoda je založena na tom, že z dané populace vybereme jedince, může jich být i více, s nejvyšší hodnotou ohodnocovací funkce a tento jedinec, případně jedinci, se nezúčatní žádné metody výběru a jsou přímo včleněni do nové populace. Může nastat případ, že takovýchto jedinců se stejnou, nejvyšší hodnotou ohodnocovací funkce bude vzhledem k velikost populaci značné množství, v nejhorším případě budou vybráni všichni. Je tedy vhodné aby do nové populace bylo přímo včleněno jen určité množství takovýchto jedinců. Vezmeme-li v úvahu nejhorší případ, a to ten, že budou vybráni všichni, neprovede se žádná operace křížení ani mutace a algoritmus uvázne na mrtvém bodě. Nebude docházet k žádnému zlepšení životaschopnosti dané populace, ani jedinců, neboť při další iteraci se provede opět to samé – včlenění všech jedinců do další populace. Toto se bude cyklicky opakovat.

2 Popis problému

2.1 Zadaný problém

Mějme sadu testovaných obvodů a ke každému z těchto obvodů mějme soubor, který obsahuje určitý počet testovacích vektorů. Ke každému vektoru přísluší bitová maska, která určuje kolik a jaké poruchy je ten daný testovací vektor schopen detekovat. V tomto konkrétním

případě počet poruch udává počet jedniček v bitové masce a typ poruchy, její číslo, odpovídá pozici jedničky v oné masce. Struktura vstupních dat je blíže popsána v odstavci 2.2.3. Pro různé obvody budeme mít různé soubory, takže počet vektorů, jejich délka i délka bitové masky je závislá na typu obvodu.

2.2 Cíle práce

Cílem této práce je vytvořit program, který pomocí genetického algoritmu bude zpracovávat vstupní testovací vektory pro daný obvod, což konkrétně znamená, že se snažím generovat takový seed, pomocí kterého jsem schopen pokrýt co nejvíce poruch. Srovnat výsledky několika testů pro jeden obvod při použití různých parametrů, neboť pro různé parametry, byly popsány v první kapitole, se výsledky mohou značně měnit. Program modifikovat tak, aby bylo možné srovnání výsledků získaných při použití genetického algoritmu s výsledky získanými náhodným generováním seedů. Pro oba dva způsoby získaná data zobrazit graficky a provést srovnání výsledků.

2.2.1 Program

Mějme vstupní soubor, obsahující data, na něž chceme aplikovat program, nejprve s použitím genetického algoritmu, poté bez použití genetického algoritmu. Aby byl zaručen řádný průběh programu, musí být program schopen daná data správně analyzovat a další parametry, zadané uživatelem, musí být také zadány v požadovaném (správném) tvaru. Proto další nutností je korektní ošetření případného chybového vstupu. Pokud tato situace nastane, musí být uživatel o chybovém vstupu informován a vykonávání programu nesmí být zahájeno až do doby, dokud nebudou veškeré chyby odstraněny. Protože se výsledky algoritmu mění po každé iteraci, je vhodné, aby byly průběžně zobrazovány a dále, pro případnou pozdější kontrolu, ukládány. Program tedy musí zaručovat korektní zobrazování výsledků i jejich zápis do výstupního souboru. Obě tyto operace jsou výstupem tohoto programu. Aby bylo možné při běhu genetického algoritmu určité parametry měnit, a to již bez možnosti vzniku chybového vstupu, je vytvořeno uživatelské rozhraní. V něm jsou zobrazeny jak aktuální hodnoty parametrů s možností jejich změn, tak také grafický výstup celého programu. Při druhém způsobu zpracování, tedy bez použití genetického algoritmu, se během vykonávání programu žádné parametry nemění.

2.2.2 Vstupní data

Abychom měli genetický algoritmus na čem testovat, potřebujeme vstupní soubor, který bude daná testovací data obsahovat. Pokud bychom neměli již vygenerovaná data, jako v mém případě, bylo by nutné sestavit další program, který by nám data do souboru vygene-roval. Struktura uložených dat v souboru musí samozřejmě odpovídat struktuře, pro níž je daný genetický algoritmus navržen. Struktura mnou zpracovávaných souborů je popsána v následujícím odstavci 2.2.3.

2.2.3 Struktura souboru vstupních dat

Jeden řádek vstupního souboru odpovídá jednomu vektoru a jeho bitové masce. Vektor je řetězec znaků, skládající se z nul, jedniček a “x“. Znak “x“ odpovídá neurčenému stavu, což znamená, že na daném místě může být jak jednička, tak nula. To je důležité při testování na shodnost vektorů, což je popsáno v odstavci 2.3.4 a také v kapitole čtvrté, v odstavci.4.1.5. Bitová maska je řetězec, skládající se pouze z nul a jedniček. Například jeden řádek souboru, který odpovídá testovanému obvodu c17 by mohl vypadat následovně: 00xxx 10000000010000000000.

2.2.4 Měření a zpracování získaných dat

Jelikož provádění genetického algoritmu se odvíjí od nastavených parametrů, mezi které patří již zmiňovaná pravděpodobnost mutace, pravděpodobnost křížení, použití elitismu, ale i další jako počáteční nastavení LZPR, počet generovaných seedů, počet vektorů vygenerovaných během LZPR, či počet provedených iterací, veškeré tyto parametry jsou dopodrobna rozebrány v článku 2.2, je možné nad daným souborem testovacích dat provést nespočet měření, kde při každém měření mohu změnit jeden, či více parametrů. Získané výsledky se mohou značně lišit, takže pro názornost je vhodné je vynést do grafu, kde ovšem musí být uvedeny i všechny parametry, pro něž bylo výsledků dosaženo. V případě, že budu měnit parametry za běhu programu, bude pochopitelně těžší tyto změny ke grafu zapsat, nicméně určitě by to bylo vhodné. Při druhém způsobu zpracování, bez použití genetického algoritmu, se nastavují pouze počáteční LZPR a počet vektorů vygenerovaných během LZPR. Aby bylo možné srovnání výsledků prvního a druhého způsobu zpracování, musí být odpovídající parametry nastaveny stejně.

2.3 Realizace genetického algoritmu (GA)

V této části se budu zabývat konkrétním popisem funkčnosti genetického algoritmu tak, jak jsem jej aplikoval ve své práci. Konkrétním z toho důvodu, že problematika použití genetických algoritmů je značně rozsáhlá a není tedy možné popsat všechny aspekty jejich použití přímo, neboť na konkrétní problém se musí aplikovat konkrétní postup. Dále se tedy zabývám jen popisem mnou použitého postupu při řešení problému *Syntézy generátorů testovacích vektorů pomocí genetických algoritmů*.

2.3.1 Nastavení LZPR

Před samotným spuštěním genetického algoritmu musíme určit a zadat počáteční nastavení LZPR. Nastavení může být určeno v podstatě dvěma způsoby. Zaprvé generujícím polynomem, který byl blíže popsán a vysvětlen v odstavci 1.2.1 a za druhé přímo řetězcem, kde jednotlivé znaky na určitých pozicích v řetězci odpovídají právě nastavení obvodu. Použití prvního způsobu by bylo v mém případě náročnější, neboť bych musel daný generující polynom převést na odpovídající binární řetězec, což by vyžadovalo naprogramovat dodatečné metody, takže jsem zvolil pouze možnost druhou a zadávám tedy počáteční LZPR rovnou binárně. Délka LZPR musí být shodná s délkou vektorů v testovaném souboru.

2.3.2 Generování seedů

Poté, co máme nastaveno LZPR, můžeme vygenerovat seedy. Seedem nazýváme každého jedince v populaci a je to opět binární vektor, tedy skládá zase pouze z nul a jedniček. Délka každého seedu musí být opět shodná s délkou LZPR. Chceme-li tedy například populaci o mohutnosti n, vygenerujeme n různých seedů. Abychom mohli určit ohodnocovací funkci fitness každému seedu, musíme určit, kolik poruch v bitové masce daný seed pokrývá. K tomu slouží vektory, které od daného seedu získáme následujícím postupem, viz 2.3.3.

2.3.3 Generování vektorů seedu

Mějme seed a chceme k němu vygenerovat příslušný počet vektorů, který zadáváme jako jeden z parametrů. Postupujeme tak, že daný seed "vložíme" na vstup do obvodu LZPR, jehož nastavení je pevně dáno řetězcem nul a jedniček, jak bylo vysvětleno ve 2.3.1 a provedeme krok LZPR. Tím získáme jeden vektor. Pro takto získaný vektor opět provedeme krok LZPR. Pokud tedy chceme od daného seedu vygenerovat *n* vektorů, provedeme *n-krát* krok LZPR. Celý postup provádíme pro každý z námi vygenerovaných seedů, čímž pro každý z nich získáme sadu testovacích vektorů, pomocí nichž budeme posléze určovat fitness onoho seedu.

2.3.4 Test na shodu vektorů

Pro každý testovací vektor, vygenerovaným LZPR s daným seedem, který jsme získali postupem uvedeným v předchozím odstavci, provádíme test na shodu s každým testovaným vektorem, čili vektorem ve vstupním souboru. Test na shodu spočívá v porovnání logických hodnot na konkrétní pozici v řetězci testovacího vektoru s logickou hodnotou na stejné pozici v řetězci vektoru testovaného. Pokud je výsledkem shoda, testuji další pozici, pokud neshoda, vektory jsou různé a jako testovaný vektor beru následující vektor v souboru a test opakuji. Takto by bylo možné vektory testovat v případě, že by byly vyloučeny takzvané neurčené stavy, stavy, kde na dané pozici v řetězci vektoru může být jak logická nula, tak logická jednička. V případě výskytů neurčených stavů, a přesně s takovými testovanými vektory pracuji, se musí test shody provádět následujícím způsobem. Mějme testovaný vektor, pro ilustraci opět postačí vektor obvodu c17, zadaný následovně 1x1xx. Vektory shodné s tímto vektorem jsou všechny ty, které na první a třetí pozici mají logickou jedničku, zatímco na zbývajících pozicích mohou mít jak logickou jedničku, tak logickou nulu. Test shody při použití neurčených stavů tedy srovnává pouze konkrétní logické hodnoty na příslušné pozici testovaného vektoru s logickými hodnotami na stejné pozici vektoru testovacího a hodnoty na zbývajících pozicích vektoru testovacího nebere v úvahu. Je samozřejmé, že testovací vektor se může shodovat s více vektory testovanými.

2.3.5 Určení fitness seedu

Na počátku genetického algoritmu, má každý seed fitness rovnou nule. Pokud se testovací vektor shoduje s testovaným vektorem v souboru, viz 2.3.4, je k danému seedu přidán odkaz na bitovou masku, příslušející onomu testovanému vektoru. Pokud dojde k další shodě mezi vektory, další získaná bitová maska se logickou operací OR srovná s již existující bitovou maskou, na níž seed odkazuje. Takto se postupuje pro každý z vektorů, vygenerovaných LZPR s daným seedem. Po provedení všech testů na shodu danému seedu přísluší jedna bitová maska. Počet jedniček bitové masky je roven hodnotě ohodnocovací funkce, tedy fitness. Výše uvedeným postupem získáme fitness každého seedu a jsme schopni použít metodu ruletového kola, viz 1.3.3.

2.3.6 Vytvoření nové populace

Při vytváření nové populace hrají klíčovou roli hodnoty parametrů, o nichž byla řeč v úvodní kapitole. Pokud používáme elitismus (viz 1.3.5), jsou elitní seedy vloženy přímo do nové populace a zbývající počet jedinců vznikne tak, že metodou ruletového výběru (viz 1.3.3) vybereme rodiče, se zadanou pravděpodobností provedeme křížení a mutaci (viz 1.3.2) a takto vzniklé jedince umístíme do nové populace. Ruletový výběr, křížení a mutaci provádíme tak dlouho, dokud není populace úplná, to znamená dokud nová populace nemá stejně jedinců jako populace předchozí. Pokud elitismus nepoužíváme, ačkoli doporučeno je jej pou-

žívat, je vytvoření nové populace velmi podobné jako při jeho používání s tím rozdílem, že do populace nejsou vloženy elitní seedy, takže výběr z rulety, křížení a mutace se bude provádět právě tolikrát, kolik má mít populace jedinců. Když je nová populace kompletní, stará zaniká a nad novou spouštíme znovu genetický algoritmus.

2.3.7 Ukončení genetického algoritmu

Protože asi i nechceme, aby genetický algoritmus běžel "věčně", můžeme jeho ukončení ovlivnit příslušnými podmínkami. Může to být například po určitém počtu vykonaných iterací, po dosažení požadované kvality celé populace (pochopitelně vyjádřeno fitness), dosažení požadované fitness jednoho jedince a řady dalších, v neposlední řadě prostým zavřením aplikace.

2.4 Modifikovaný program

Protože dalším z cílů práce bylo porovnat výsledky dosažené pomocí genetického algoritmu s výsledky dosaženými bez genetického algoritmu, bylo nutné vytvořit program modifikovaný, který by genetický algoritmus nepoužíval. Jelikož se jedná o program jednodušší a navíc využívající vybrané postupy, použité při realizaci genetického algoritmu, popíši jeho činnost ve stručnosti. Aby se daly porovnat výsledky získané s použitím genetického algoritmu s výsledky získanými bez něj, je nutné, aby v obou případech byly příslušné parametry nastaveny stejně. Konkrétně jde o LZPR a o počet vektorů vygenerovaných během LZPR. Dalším parametrem, který se zde uplatňuje navíc je doba, po kterou má program běžet. Musí být shodná s dobou trvání genetického algoritmu. Ostatní parametry genetického algoritmu se zde neuplatní. Program tedy dělá pouze to, že po danou dobu generuje seedy, pro každý provede n-krát krok LZPR a určí seedu fitness. Vše zcela shodně jako při použití genetického algoritmu. Výstupem bude jeden seed s nejvyšší fitness. Po uplynutí dané doby program končí.

3 Analýza

V této kapitole uvádím důvody proč jsem použil níže uvedené metody a postupy při realizaci genetického algoritmu i s odůvodněním volby hodnot nastavených parametrů. Dále se zabývám analýzou konkrétních obvodů pro různě nastavené parametry a srovnáním dosažených výsledku s a bez použití genetického algoritmu.

3.1 Metoda ruletového výběru

Ačkoli nepatří mezi nejrychlejší metody výběru, používám právě ji při volbě jedinců, kteří se stanou rodiči budoucích potomků. Zvolena byla proto, že na rozdíl od metody turnaje při této metodě má každý jedinec šanci být vybrán a je tu tedy větší možnost proměnlivosti životaschopnosti celé populace. Na druhou stranu zde stoprocentně neplatí to, že silnější přežije a jeho geny budou v potomcích nové populace. Pokud bych chtěl toto zaručit, použil bych pro výběr jedince metodu turnaje.

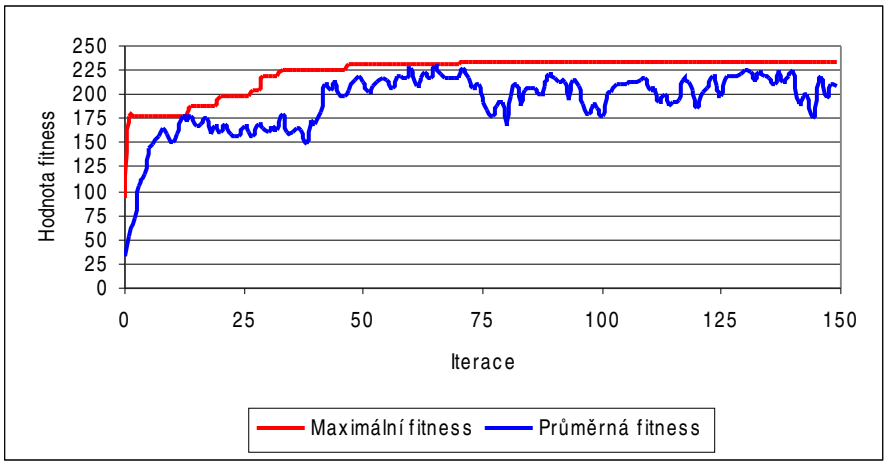
3.2 Použití elitismu

Při realizaci genetických algoritmů je obecné doporučení elitismus používat a z tohoto jsem vycházel i ve své práci. Navíc konkrétním důvodem je to, že cílem mé práce je generovat takový seed, který by pokryl co nejvíce poruch a jelikož fitness daného seedu je rovna počtu poruch, které seed pokrývá, je zřejmé, že daný elitní seed, případně seedy, s nejvyšší hodnotou fitness, budou požadovaná řešení. Pokud bych elitismus nepoužíval, může nastat situace, kdy v jedné iteraci algoritmu získám nejlepší řešení a v další iteraci o toto řešení přijdu a nalezené nejlepší řešení dané iterace už nemusí být stejně dobré jako řešení získané v iteraci předchozí. Pro ukázku jsem zvolil dva obvody, c432 a s349 na jejichž grafech je znázorněn vývoj populace včetně hodnoty nejlepšího řešení, v prvém případě za použití elitismu, ve druhém případě bez něj. Pochopitelně v prvém i druhém případě zůstávají zbývající parametry nastaveny na stejné hodnoty.

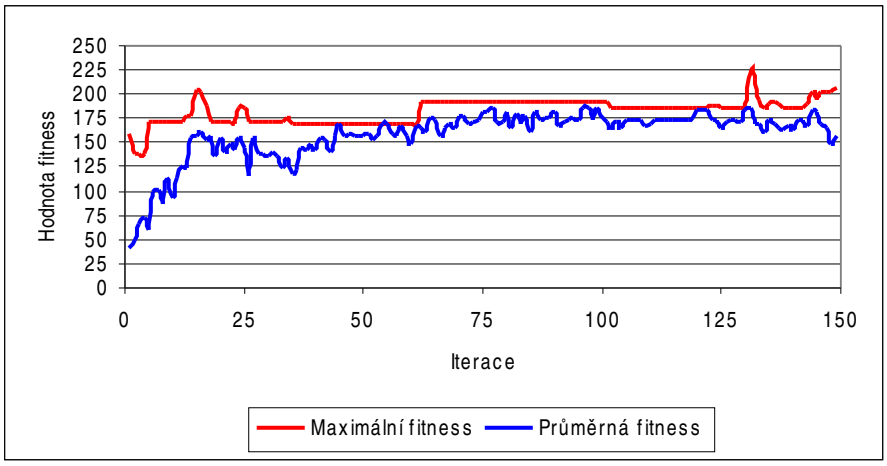
Grafy jsou uvedeny na obrázcích 3.1a, 3.1b, 3.2a a 3.2b.

3.2.1 Výsledky pro obvod c432

Výsledky znázorňují vývoj populace při použití elitismu a bez použití elitismu.
Parametry obvodu**:** LFSR 10000000000000000000000000000000001, počet seedů 15, počet vektorů od jednoho seedu 20, počet provedených iterací 150, pravděpodobnost křížení 80 %, pravděpodobnost mutace 0%.



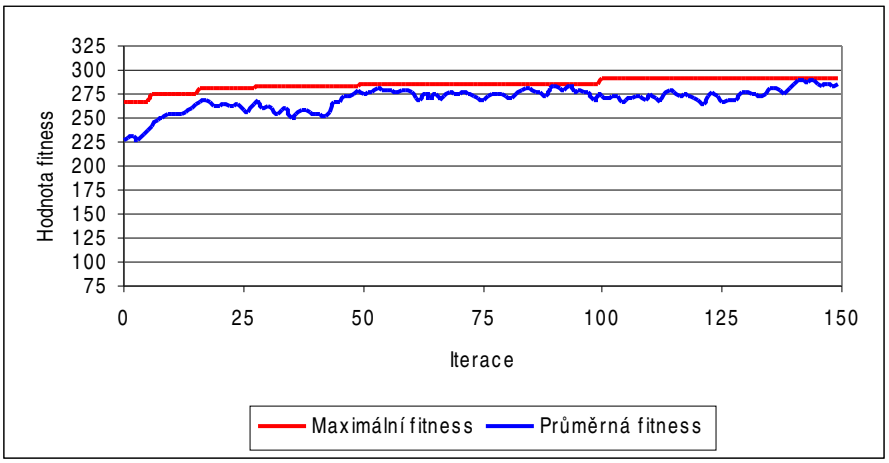
Obrázek 3.1a: Vývoj populace pro obvod c432 při použití elitismu (výsledky získané ze 150 iterací)



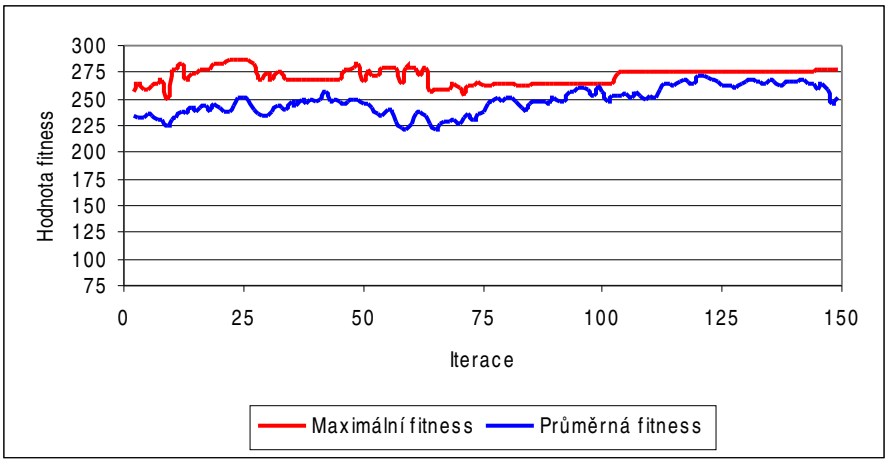
Obrázek 3.1b: Vývoj populace pro obvod c432 bez použití elitismu (výsledky získané ze 150 iterací)

3.2.2 Výsledky pro obvod s349

Výsledky znázorňují vývoj populace při použití elitismu a bez použití elitismu Parametry obvodu**:** LFSR 10000000000000000000001, počet seedů 20, počet vektorů od jednoho seedu 20, počet provedených iterací 150, pravděpodobnost křížení 80 %, pravděpodobnost mutace 0%.



Obrázek 3.2a: Vývoj populace pro obvod s349 při použití elitismu (výsledky získané ze 150 iterací)



Obrázek 3.2b: Vývoj populace pro obvod s349 bez použití elitismu (výsledky získané ze 150 iterací)

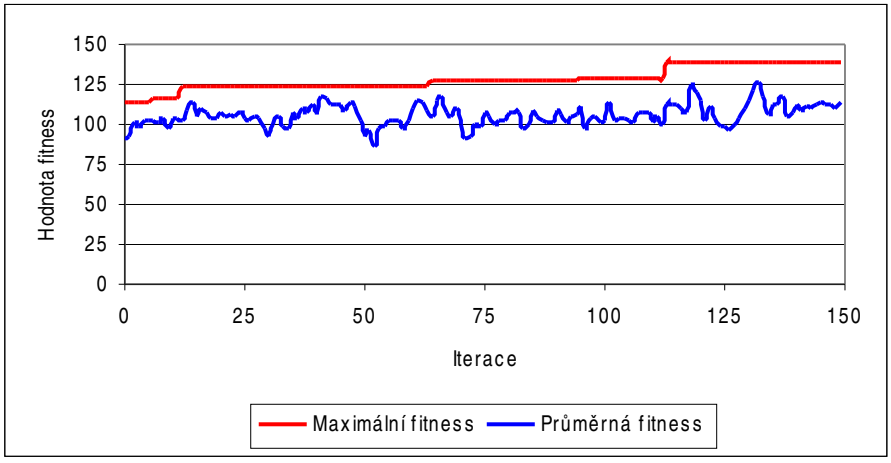
3.3 Parametry křížení a mutace

Při volbě hodnot těchto důležitých parametrů genetického algoritmu jsem opět vycházel z obecných doporučení. Při použití příliš vysoké pravděpodobnosti mutace by byli vznikající jedinci zatíženi značnými genovými změnami, což by již tak věrně nenapodobovalo přirozený vývoj v přírodě z něhož genetické algoritmy vychází, takže pravděpodobnost mutace nastavuji maximálně na jedno procento. Naproti tomu pravděpodobnost křížení se má pohybovat kolem 80% a více, aby docházelo k přirozené obměně jedinců v populaci a populace se mohla

vyvíjet. Výsledky pro různé hodnoty výše zmiňovaných parametrů demonstruji graficky na obrázcích 3.3, 3.4, 3.5, 3.6, 3.7 a 3.8. Zbývající parametry zůstávají pro každé měření stejné.

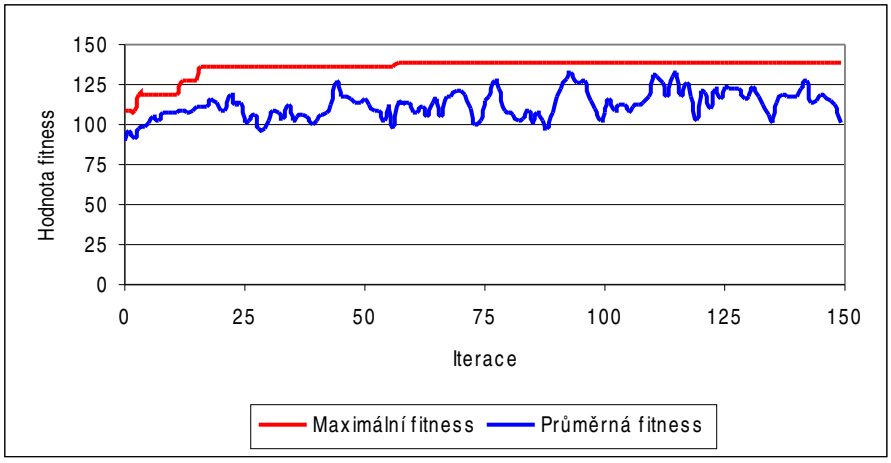
3.3.1 Měření první

Parametry, které jsou pro všechna měření stejné mají následující hodnoty: LZPR 1000111000100000011, počet seedů 10, počet vektorů od seedu 10, počet provedených iterací 150 a byl použit elitismus.
Parametry, které se mění jsou pravděpodobnost mutace a pravděpodobnost křížení**.**



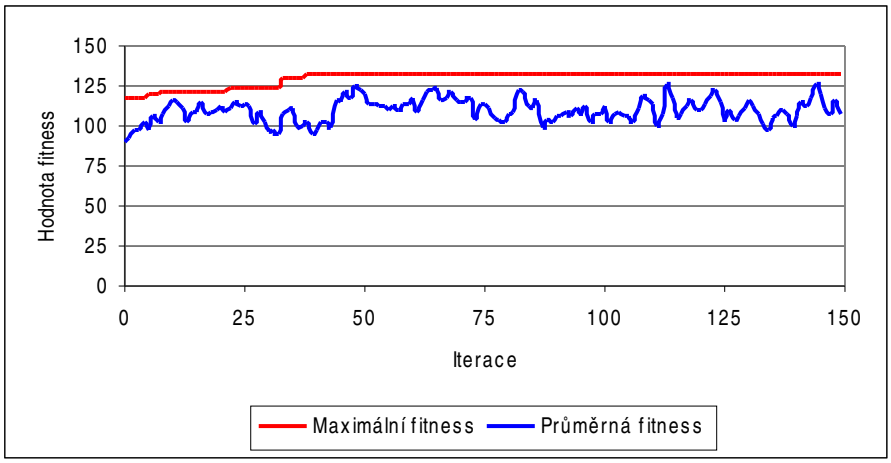
Obrázek 3.3: Vývoj populace pro obvod s208 dle nastavených parametrů (pravděpodobnost mutace 70%, pravděpodobnost křížení 30%)

3.3.2 Měření druhé



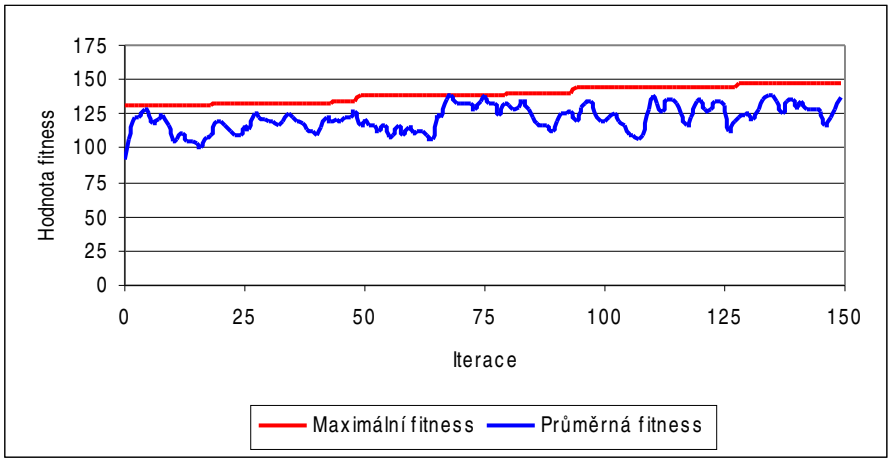
Obrázek 3.4: Vývoj populace pro obvod s208 dle nastavených parametrů (pravděpodobnost mutace 30%, pravděpodobnost křížení 70%)

3.3.3 Měření třetí



Obrázek 3.5: Vývoj populace pro obvod s208 dle nastavených parametrů (pravděpodobnost mutace 50%, pravděpodobnost křížení 50%)

3.3.4 Měření čtvrté



Obrázek 3.6: Vývoj populace pro obvod s208 dle nastavených parametrů (pravděpodobnost mutace 1%, pravděpodobnost křížení 90%)

3.4 Volba testovaných obvodů

Ačkoli jsem měl k dispozici data ještě pro další testované obvody, vybral jsem ze všech předložených obvodů jen několik konkrétních. Důvod je ten, že již i pro poměrně malé soubory vstupních dat daného obvodu, řádově stovky kB, bylo provádění testu časově dosti náročné. Proto jsem tedy test neprováděl pro obvody jejichž soubory dat velikostí přesáhly 1MB, ačkoli jsem měl i soubory velikosti stovek MB až 1GB.

4 Použité řešení

V této kapitole se zabývám jednak rozborem nutných funkcí, které musí být implementovány, aby byla zaručena správnost prováděného algoritmu a dále zde popisuji a zdůvodňuji volbu programovacího jazyka a vývojového prostředí.

4.1 Rozbor nutných funkcí

U nestandardních řešení je náznak implementace dané funkce uveden v pseudokódu.

Popisované funkce nejsou nutně uváděny v pořadí, v jakém jsou vykonávány programem.

4.1.1 Test platnosti vstupních dat

Pod funkcí *Test platnosti vstupních dat* spadá řada dílčích funkcí, z nichž každá testuje platnost, čili správnost zadání konkrétního parametru. Jelikož se jedná o celou řadu parametrů, například existence vstupního souboru, struktura dat v souboru, délka LZPR, počet seedů a řada dalších, nebudu zde každou dílčí funkci popisovat. Obecně tedy lze říci, že před spuštěním programu, ať už s použitím genetického algoritmu, či bez něj program provádí test na platnost každého z parametrů. Pokud by byť jediný byl neplatný, program nebude spuštěn.

4.1.2 Generování seedů

Dle příslušné délky LZPR funkce "*Generování seedů*" vygeneruje požadovaný počet

seedů, které budou tvořit počáteční populaci. Každý seed bude binární vektor, tedy vektor skládající se pouze z nul a jedniček, což je zajištěno metodou *nextInt(int počet)*, kde *počet* určuje interval použitých čísel o jedna zvětšený. V našem případě *nextInt(2)*. Takto získané číslo je metodou *concat(String)* připojeno ke stavájícímu řetězci. Takto se celý postup opakuje, dokud nezískáme seed požadované délky.

4.1.3 Načtení dat z konfiguračního souboru

Daná funkce se skládá z dalších dílčích funkcí, jež snad není nutné dopodrobna rozebírat, nicméně souhrnně tyto funkce dělají to, že načtou veškerá data, parametry programu, z konfiguračního souboru a předávají je příslušným dalším funkcím ke zpracování. Načítané parametry jsou: vstupní soubor, výstupní soubor, lfsr, počet požadovaných iterací, počet seedu na jednu populaci, počet vektorů od jednoho seedu, počet elit vstupujících do další populace a jemnost grafu (v počtech bodů na jednu iteraci). Vstupní soubor existovat musí, výstupní nikoli, ale jeho jméno zadáno být musí. Pokud neexistuje, je vytvořen, pokud existuje, jsou výsledky připisovány na jeho konec, není přepsán.

4.1.4 StepLFSR(*aktNastavení*)

Operace stepLFSR(*aktNastavení*) provede *XOR(aktNastavení[i], nastavení LZPR[i])*, kde *i* vyjadřuje aktuální pozici v řetězci. Výstupem z LZPR a zároveň výsledkem operace XOR bude následující vektor LZPR. Důležité je, že takto získaný vektor se stává při dalším volání stepLFSR(*aktNastavení*) aktNastavením. Provedením stepLFSR tedy vygeneruji nás-ledující vektor LFSR. Pokud chceme od daného seedu vygenerovat *n* vektorů, provedeme *n-krát* stepLFSR(*aktNastavení*). Získaný vektor si pokaždé uložíme a po *n* provedeních stepLFSR(*aktNastavení*) budeme mít sadu testovacích vektorů. S pomocí těchto vektorů budeme hledat fitness daného seedu. Pseudokód stepLFSR(*aktNastavení*) je uveden v algoritmu 4.1.

```
StepLFSR(aktNastavení) {//na počátku aktNastavení rovno seedu, který do LFSR vstupuje
while(početVektoruOdSeedu < požadovanýPočet){
aktuálníZnak = aktNastavení[n];
for (int n) {
výsledek[n] = xor(aktNastavení[n-1],nastaveníLZPR[n]);
}
výsledek[0] = aktuálniZnak;
aktNastavení = "";

for (int n) {
aktNastavení += výsledek[n];
}
poleVektoůOdSeedu[0][početVektoruOdSeedu] = aktNastavení;
početVektorůOdSeedu++;
}
```

```
//zde již aktuální nastavení rovno vektoru, který vznikl krokem LFSR
return aktNastavení;
}
```

Algoritmus 4.1: Pseudokód stepLFSR(aktNastavení)

4.1.5 Test na shodnost vektoru

Pro každý vektor vygenerovaným LZPR s daným seedem procházím soubor vstupních dat až do konce a testuji, zda se vektory v souboru shodují s vektory vygenerovanými LZPR; připomínám, že vektor je reprezentován jako vektor binární. Shodou rozumíme to, že na stejných pozicích v řetězci mají stejné znaky, případně že je na daném místě znak neurčeného stavu, který může být zastoupen buď nulou, nebo jedničkou. Například s vektorem 00xxx, pro obvod c17, budou shodné všechny vektory vygenerované LZPR s daným seedem začínající dvěmi nulami a pokračující libovolnou posloupností délky tři, kde na každé pozici může být jak nula, tak jednička. Pokud při testování narazím na první rozdílný znak, vektory jsou různé a začínám tedy s testem shody vektoru následujícího. Pokud se vektor vygenerovaný LZPR s daným seedem, dále jen vektor od LZPR, shoduje s vektorem v souboru, vezmu od vektoru v souboru bitovou masku a přiřadím ji jako výsledek celé operace k danému seedu. Při shodě dvou a více vektorů v souboru s vektorem od LZPR získám pochopitelně více bitových masek, které mezi sebou ORuji. Po dosažení konce souboru mám tedy pro první vektor od LZPR určenu bitovou masku. Pro další vektor od LZPR postupuji zcela obdobně, opět od začátku souboru až do jeho konce, s tím, že získanou bitovou masku ORuji s bitovou masku získanou k předchozímu vektoru. Tento postup opakuji tak dlouho, dokud sem nevyčerpal všechny vektory od LZPR. Ve finále tak získám jeden vektor bitové masky, který bude obsahovat určitý počet jedniček, a určitý počet nul. Počet jedniček v tomto vektoru je fitness, ohodnocovací funkce příslušného seedu, což je počet poruch, které jsem pokryl. Pseudokód *Test na shodu vektorů* je uveden v algoritmu 4.2.

```
otestujVektor(testovanýVektor, testovacíVektor) {
int i; stejný = true;
while(stejný && neníKonecVektoru) {
// x určuje neurčené stavy,
if ( (testovacíVektor[i] == 'x') || (testovanýVektor[i] == 'x') {
stejný = true;
}
else {
// pokud se vyskytl jiný, než neurčený stav
if(testovacíVektor[i]== testovanýVektor[i]) then stejný
}
}
```

```
return stejný;
}
```

Algoritmus 4.2: Test na shodu vektorů

Pro názornost uvádím příklad na test shody vektorů a určení bitové masky a fitness celého seedu. Jako referenční testované vektory mi opět poslouží vybrané vektory obvodu c17. Výsledky jsou přehledně zobrazeny v následující tabulce 4.3

vektor vzniklý krokem LZPR	vektor ve vstupním souboru	shoda	bitová maska vektoru	dílčí bitová maska
00110	00xxx	ano	10000000010000000000	10000000010000000000
00110	001xx	ano	10000100010000000000	10000100010000000000
00110	x0xx0	ano	00000000010000001000	00000000000000001000
			výsledná bitová maska	10000100010000001000

Tabulka 4.3: Test na shodu vektorů a získání bitové masky

4.1.6 Počítání fitness

Zde se jedná o čtyři funkce. Určení minimální, průměrné, maximální a celkové fitness. Důležité jsou především dvě z nich. Fitness maximální, pomocí níž se vybírají seedy do další populace, ať už se jedná o výběr elitních, nebo "jen" o výběr rodičovských seedů z nichž vzniknou potomci, kteří budou novou populaci, společně s elitními seedy tvořit a fitness celková. Pomocí celkové fitness se vytváří ruletové kolo z něhož jsou seedy vybírány. Průměrná fitness vyjadřuje aktuální průměrnou životaschopnost celé populace, čili součet fitness všech seedů podělený počtem seedů. Konečně minimální fitness, jak z pojmenování vyplývá, vyjadřuje aktuální hodnotu fitness seedů nejhorších.

4.1.7 Vytvoření a výběr z ruletového kola

Funkce *Vytvoření a výběr z ruletového kola* vytvoří pomyslné ruletové kolo, které je reprezentováno dvourozměrným polem, kde se každý záznam skládá z označení seedu a jeho fitness. Počínaje záznamem druhým se k fitness daného seedu připočítává fitness seedu předchozího. Tak nám vlastně každé políčko rulety bude vymezovat určitý interval. Celkový interval, který bude ruletové kolo vymezovat, bude v rozsahu od nuly do celkové fitness. Následný výběr se provádí tak, že je náhodně vygenerováno celé číslo v intervalu nula až celková fitness a dle toho do jakého políčka kola rulety spadá, čili do jakého intervalu, je z rulety vybrán

příslušný seed.

4.1.8 Křížení

Pomocí funkce jednobodového křížení vznikají z vybraných jedinců potomci, kteří pak tvoří, vedle případných elitních jedinců, novou populaci. Vždy ze dvou jedinců vzniknou dva potomci. Samotné křížení se koná tak, že vždy dva vybrané jedince, které můžeme považovat za rodiče budoucích potomků, se v určitém, náhodně zvoleném místě[2] roztrhnou, čímž vzniknou čtyři dílčí části. Tyto části se pak spojí tak, že první část prvního rodiče se spojí s druhou částí rodiče druhého a druhá část rodiče prvního se spojí s první částí rodiče druhého, jak je naznačeno na obrázku 1.5. Tak vzniknou dva noví potomci, kteří budou tvořit část nově vznikající populace.

4.1.9 Mutace

Funkce mutace provádí na náhodně vybrané pozici, či pozicích v řetězci potomka záměnu znaku za znak jiný. Konkrétně, je vybrán řetězec, na němž má být provedena mutace. Následně je náhodně zvolen počet míst a tomu odpovídající počet náhodných pozic v řetězci, na nichž má být mutace provedena. Poté jsou na vybraných pozicích zaměněny znaky za znaky jiné, v našem případě nuly za jedničky a naopak. Takto vzniklý mutant je vložen do nově vznikající populace.

4.1.10 Vytvoření nové populace

Tato funkce vytvoří novou populaci, na kterou je v další iteraci programu opět aplikován celý genetický algoritmus od začátku. Do nové populace náleží, již zmiňovaní elitní jedinci, potomci vzniklí křížením i mutanti. Po vytvoření nové populace stará populace definitivně zaniká.

4.1.11 Určení elitního jedince

Při používání genetického algoritmu tato funkce najde a vloží do nově vznikající populace požadovaný počet elitních jedinců. Pokud je aktuální počet menší, než požadovaný, je do nově vznikající populace vložen jen počet aktuální. Ve druhém případě, kdy genetický algoritmus použit není funkce vyhledá jen jednoho nejlepšího jedince ze všech.

4.2 Volba programovacího jazyka a prostředí

Jako vhodný programovací jazyk byla zvolena Java, verze 5.0. Důvodů byla celá řada, ale pro představu zmíním jen některé.

- Jedná se o objektově orientovaný jazyk jehož výhody oproti neobjektově orientovaným jazykům jsou zřejmé.
- Jedná se o velice rozšířený a známý jazyk, takže výsledný program bude pochopitelný, případně snáze upravitelný pro většinu uživatelů.
- Výsledný kód programu (bytecode) je na platformě nezávislý a je tedy možné daný

program používat na libovolném počítači. Jedinou podmínkou je, aby na daném počítači byla nainstalovaná Java verze 5.0, případně vyšší.
Umožňuje psát snadno čitelný kód.
Jsem s ní nejlépe obeznámen.

Programovacím prostředím byl zvolen JBuilder a to z důvodů:

Je mi nejvíce znám.
Umožňuje zvýrazňování syntaxe.
Podporuje automatické formátování kódu.
Umožňuje kontextové doplňování kódu.

4.2.1 Návrh koncepce řešení

Jádrem celé aplikace je balíček, který obsahuje veškeré potřebné třídy a metody. Nad tímto balíčkem bude celá aplikace implementována.

Jedná se o jedinou aplikaci, která se stará o celý běh programu. Je tedy navržena pro hromadné zpracování. Po spuštění načte data z konfiguračního souboru, provede test na případné chyby a posléze uživateli nabídne jednoduché uživatelské rozhranní, pomocí něhož se zadávají a případně i za běhu programu mění vybrané parametry. Pomocí tohoto grafického rozhranní se také celá aplikace ovládá.

Popis hlavnívh tříd a metod je uveden v příloze B.

5 Měření

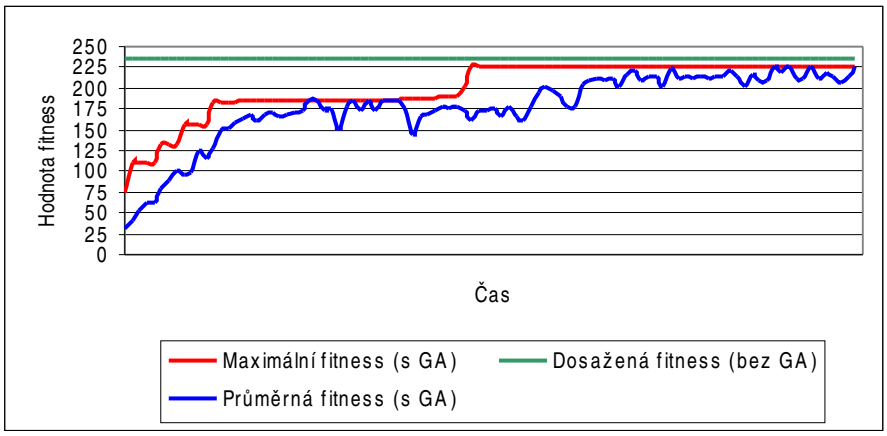
V této kapitole se zabývám srovnáním výsledků dosažených pro vybrané obvody za určitý čas. Zaprvé při použití genetického algoritmu a za druhé bez použití genetického algoritmu, tedy při náhodném generování seedů. V obou případech byla doba měření vždy shodná. Samotná měření probíhala tak, že jsem nejprve nad danými daty pustil genetický algoritmus s příslušnými parametry. Po určité době, po splnění omezující podmínky, kterou ve všech případech byl počet provedených iterací, se algoritmus zastavil. Následně jsem spustil náhodné

generování seedů. To probíhalo po stejnou dobu, po jakou běžel genetický algoritmus. Výsledkem běhu genetického algoritmu je jednak určení hodnoty průměrné fitness, ale také, a to je podstatou měření, určení vývoje hodnoty maximální fitness a konečná dosažená hodnota maximální firness. Výsledkem náhodného generování seedů je jeden seed s nejvyšší dosaženou hodnotou fitness. Obě takto získané hodnoty porovnám, čímž zjistím, zda mělo použití genetického algoritmu význam, či nikoli. Výsledky několika takto provedených měření jsou uvedeny níže v přehledné grafické formě a pro doplnění u každého měření ještě uvádím pro jaké hodnoty parametrů genetického algoritmu bylo měření prvním způsobem provedeno.

5.1 Měření první – obvod c432

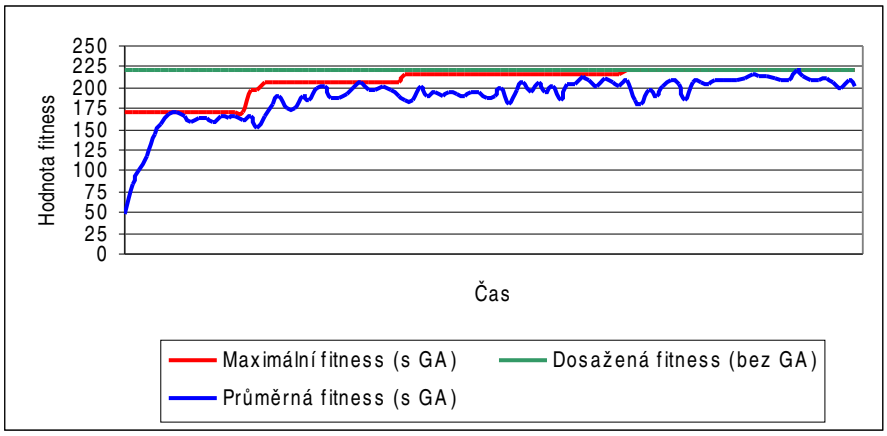
Nastavené LZPR 10000000000000000000000000000000001.

Parametry genetického algoritmu: počet seedů 15, pravděpodobnost křížení 80%, pravděpodobnost mutace 0%.



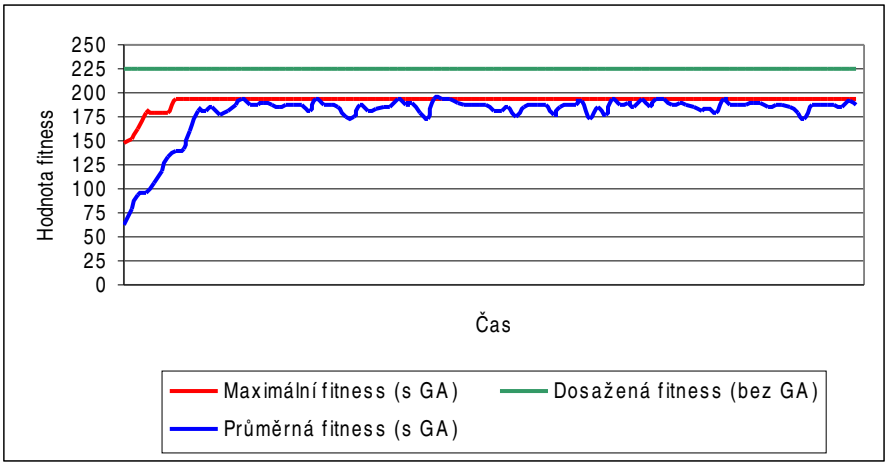
Obrázek 5.1: Srovnání výsledků pro obvod c432
(při 30 vektorech od seedu)

Parametry genetického algoritmu: počet seedů 15, pravděpodobnost křížení 80%, pravděpodobnost mutace 0%.



Obrázek 5.2: Srovnání výsledků pro obvod c432
(při 40 vektorech od seedu)

Parametry genetického algoritmu: počet seedů 20, pravděpodobnost křížení 80%, pravděpodobnost mutace 0%.

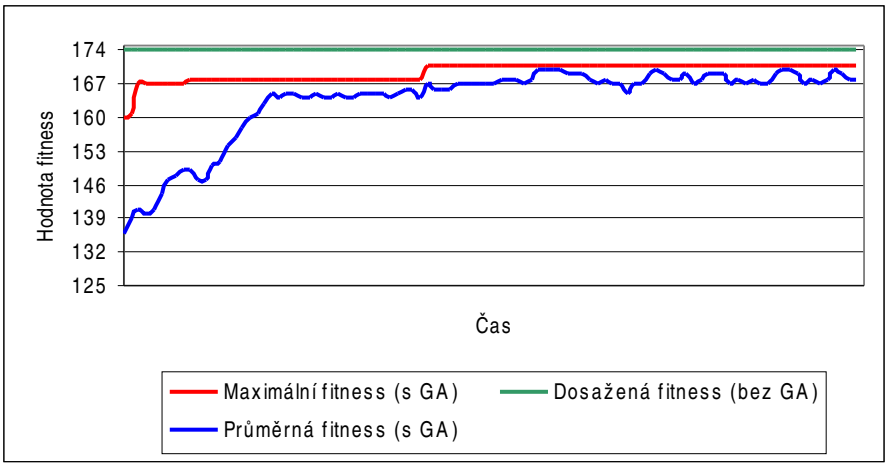


Obrázek 5.3: Srovnání výsledků pro obvod c432
(při 45 vektorech od seedu)

5.2 Měření druhé – obvod s208

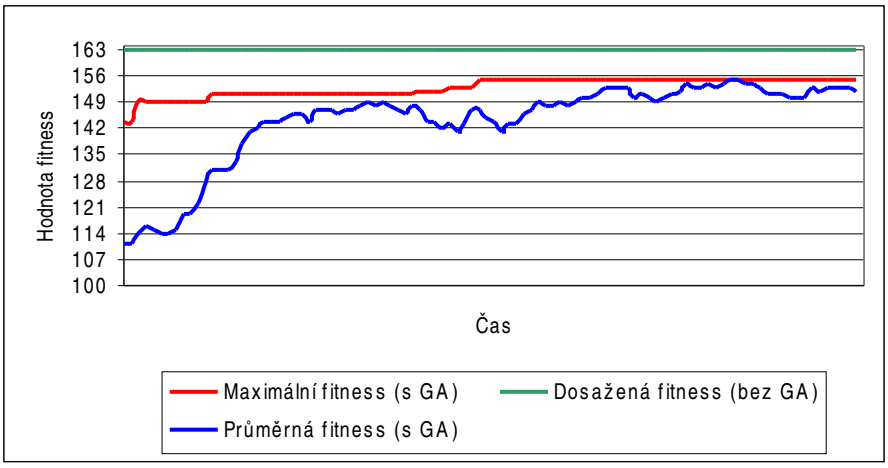
Nastavené LZPR 1000101010001001001.

Parametry genetického algoritmu: počet seedů 50, pravděpodobnost křížení 80%, pravděpodobnost mutace 0%.



Obrázek 5.4: Srovnání výsledků pro obvod s208
(při 80 vektorech od seedu)

Parametry genetického algoritmu: počet seedů 50, pravděpodobnost křížení 80%, pravděpodobnost mutace 0%.

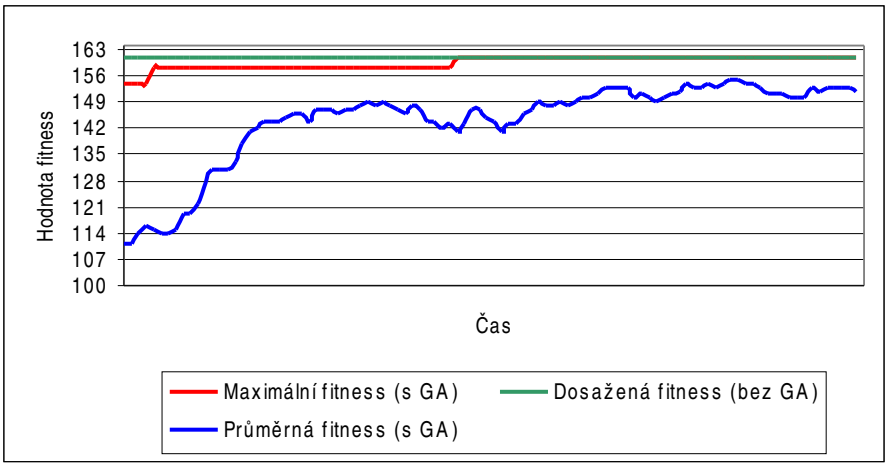


Obrázek 5.5: Srovnání výsledků pro obvod s208
(při 80 vektorech od seedu)

5.3 Měření třetí – obvod s208.1

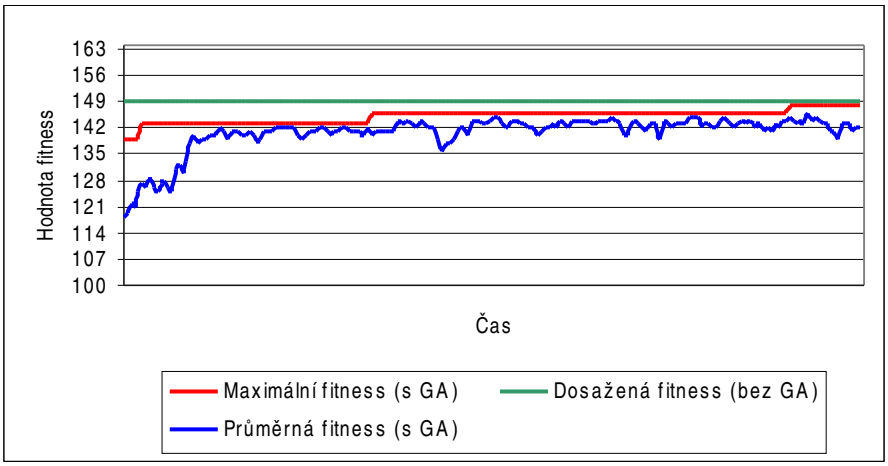
Nastavené LZPR 100010000000001001.

Parametry genetického algoritmu: počet seedů 15, pravděpodobnost křížení 80%, pravděpodobnost mutace 0%.



Obrázek 5.6: Srovnání výsledků pro obvod s208.1
(při 35 vektorech od seedu)

Parametry genetického algoritmu: počet seedů 30, pravděpodobnost křížení 80%, pravděpodobnost mutace 0%.

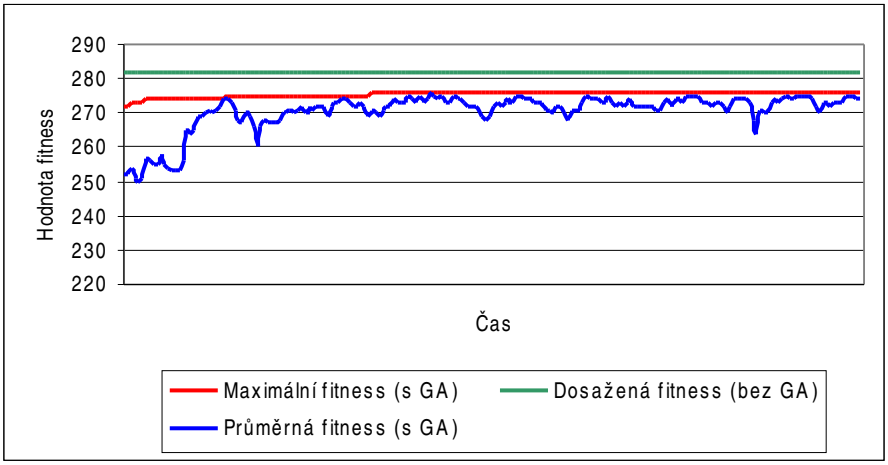


Obrázek 5.7: Srovnání výsledků pro obvod s208.1
(při 50 vektorech od seedu)

5.4 Měření čtvrté – obvod s298

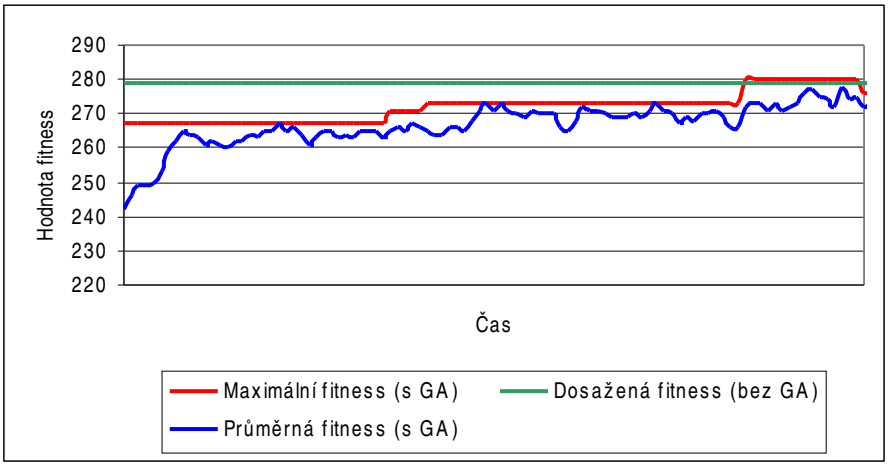
Nastavené LZPR 11000110000110011.

Parametry genetického algoritmu: počet seedů 20, pravděpodobnost křížení 80%, pravděpodobnost mutace 0%.



Obrázek 5.8: Srovnání výsledků pro obvod s298
(při 30 vektorech od seedu)

Parametry genetického algoritmu: počet seedů 30, pravděpodobnost křížení 80%, pravděpodobnost mutace 0%.

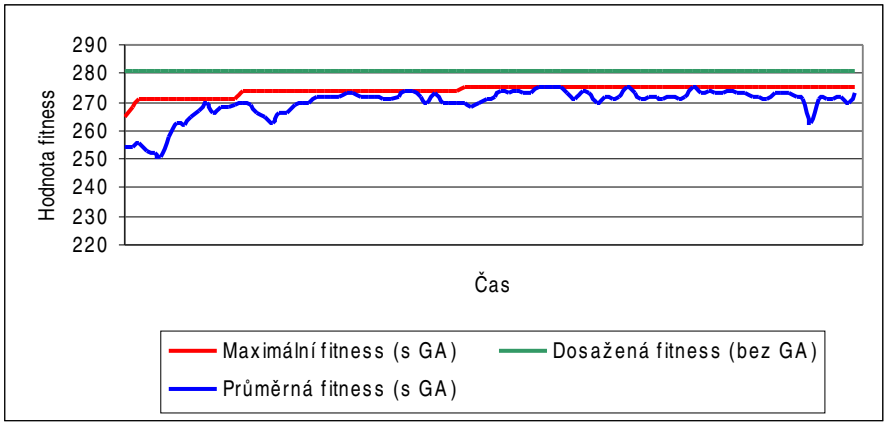


Obrázek 5.9: Srovnání výsledků pro obvod s298
(při 40 vektorech od seedu)

5.5 Měření páté – obvod s344

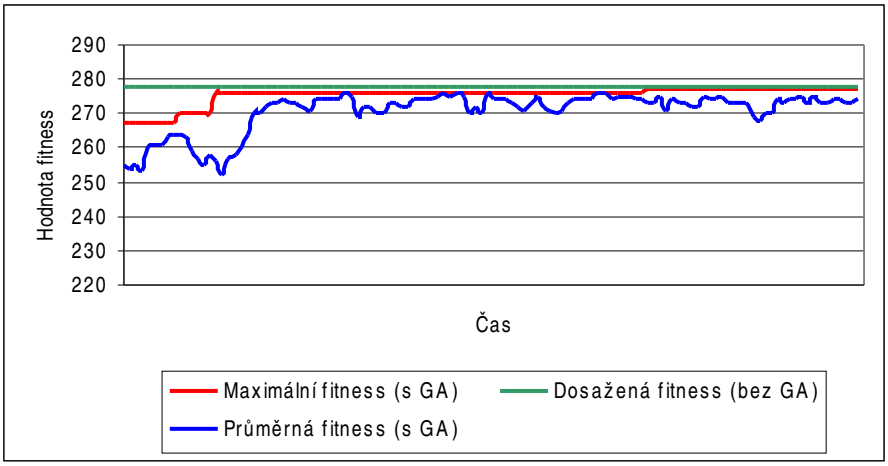
Nastavené LZPR 10000000000000001.

Parametry genetického algoritmu: počet seedů 20, pravděpodobnost křížení 80%, pravděpodobnost mutace 1%.



Obrázek 5.10: Srovnání výsledků pro obvod s344
(při 40 vektorech od seedu)

Parametry genetického algoritmu: počet seedů 20, pravděpodobnost křížení 80%, pravděpodobnost mutace 1%.



Obrázek 5.10: Srovnání výsledků pro obvod s344

(při 50 vektorech od seedu)

6 Závěr

Podařilo se mi implementovat program realizující genetický algoritmus, i program lehce modifikovaný, který genetický algoritmus nevyužíval. Z výsledků testů provedených programem realizující genetický algoritmus i z výsledků testů provedených bez genetického algoritmu a jejich vzájemným porovnáním vyplývá následující závěr.

Ačkoli předpokladem bylo, že výsledky dosažené pomocí genetického algoritmu budou lepší, čili že hodnota ohodnocovací funkce fitness bude ve srovnání s výslednou hodnotou fitness při náhodném generováním vektorů vyšší, z naměřených dat a posléze z grafů vyplývá, že použití genetického algoritmu na syntézu generátoru testovacích vektorů se jeví jako neúčelné.

Byla provedena řada testů, z nichž některé byly prezentovány v kapitole 3. a kapitole 5. Z testů provedených ve třetí kapitole vyplývá, že při použití elitismu v genetických algoritmech je za stejný čas a při stejně zvolených parametrech dosahováno výsledků značně lepších oproti genetickému algoritmu, který elitismus nepoužívá, jak je patrno z grafů na obrázcích 3.1a, 3.1b a 3.2a a 3.2b.

Z testů a následných grafů v kapitole 5. vyplývá, že výsledky dosažené za použití genetického algoritmu jsou téměř shodné s výsledky dosaženými při náhodném generování vektorů. Dokonce v několika případech byla hodnota fitness při náhodném generování vektorů vyšší. Nejvyššího rozdílu hodnot dosahovala u testovaného obvodu c432, kde fitness dosažená bez enetického algoritmu byla o 13% lepší, než fitness dosažená při jeho použití.

Nicméně pro korektnější posouzení zda je, či není účelné na daný problém genetický algoritmus používat, by bylo vhodné udělat řádově stovky testů, což ovšem vyžaduje výkonější výpočetní aparát, než mi byl k dispozici. Případně další možností by bylo upravit celý algoritmus tak, aby mohl běžet paralelně na několika počítačích.

Výběr testovaných obvodů byl tedy volen s ohledem na časovou I paměťovou náročnost testů.

Použitá literatura

[1]http://cs.felk.cvut.cz/~xobitko/ga/

[2]http://cs.felk.cvut.cz/~fiserp/papers/eci04.pdf

[3] http://cs.felk.cvut.cz/~fiserp/papers/dsd05.pdf

[4]http://service.felk.cvut.cz/courses/36NLP/slides/nlp-bk2.pdf

[5] http://www.acken.com/project/ECEN5223/LFSRtype2WithCs.jpg

[6]http://en.wikipedia.org/wiki/Genetic_algorithm

[7]http://service.felk.cvut.cz/courses/36PAA/

Příloha A

Seznam použitých zkratek

LZPR – lineární zpětnovazební posuvný registr

GA – genetický algoritmus

CA – celulární automat

Příloha B

Popis hlavních tříd a metod

Ačkoli v programu využívám ještě další třídy a metody, považuji zde za důležité zmínit jen ty, bez nichž by realizace běhu genetického algoritmu byla neuskutečnitelná.

1. Třída GenAlgus

Třída, která realizuje celý genetický algoritmus. Využívá následující metody.

int minFitnes(), int maxFitness(), int prumerFitness(), int celkovaFitnes()

String[][]ruleta() – metoda realizující ruletové kolo, návratem je datová struktura dvourozměrné pole, v němž jsou uloženy jedinci a každému jedinci je algoritmem přidělen interval hodnot, do nějž pokud padne náhodně vygenerované číslo, je jedinec vybrán jako rodič

String[] elitismus() – vrátí pole o určitém počtu nejlepších jedinců v populaci

String[][] vyberZRulety() – návratovou hodnotou je datová struktura dvouroz-měrného pole o dvou prvcích, kde prvky jsou jedinci, kteří budou bráni jako rodi-če budoucích potomků

String[] krizeni() – metoda provede křížení dvou rodičů, vybraných předchozí metodou, a vzniklé potomky uloží do pole

String mutace() – metoda vybere jedince, náhodně zmutuje geny a vrátí jedince zpět do populace

String[] novaPopulace() – metoda vytvoří na základě elitismu, křížení a mutace novou populaci, po jejímž vytvoření stará definitivně zaniká; nová populace je opět uložena v poli o n prvcích

2. Třída GenerovaniSeedu

Třída využívá *java.util.Random* a obsahuje pouze dvě metody:

String[] generujSeedy() – tato metoda je využívána při použití genetického algoritmu; návratovou hodnotou je datová struktura pole o velikosti *n,* kde *n* je počet seedů, každý prvek pole odpovídá jednomu seedu

String[] genSeedyBezGA() – tato metoda je použita v případě, že nepoužívám genetický algoritmus, ale modifikovaný program; návratovou hodnotou je datová struktura pole o velikosti jedna, kde je uložen právě vygenerovaný seed

3. Třída LFSRStep

Pomocí třech metod získám ke každému seedu příslušný počet vektorů. Jedná se o metody:

String stepLFSR(String aktNastavení) – tato metoda, využívající ještě metodu XOR(String A, String B) vrátí vektor, který vznikl ”průchodem” přes LZPR a uloží jej do pole k příslušnému seedu

String XOR(String A, String B) – provede logickou operaci xor se dvěma vektory a vrátí výsledek této operace jako výsledný vektor

String[][] vektoryOdSeedu(String danýSeed) – do pole uloží daný seed a k němu příslušné vektory

4. Třída NeGenAlgus

Tuto třídu a metodu v ní implementovanou využívá při použití modifikovaného programu, bez použití genetického algoritmu. Jedná se o metodu:

int maxFitko() – tato metoda vrátí nejvyšší dosaženou fitness, kterou poté srovnávám s fitness dosaženou pomocí GA

5. Třída PraceSeSouborem

Jedná se o třídu, která nad daným souborem vstupních dat provádí genetický algoritmus, čili volá metody jiných, výše zmiňovaných tříd, s příslušnými parametry. Krom toho jsou zde implementovány další metody:

boolean otevriSoubor() – metoda se pokusí otevřít vstupní soubor, pokud se zadaří vrátí *true*, jinak *false* a program končí chybovým výstupem

String nactiRadku() – metoda načte jednu řádku souboru a předá ji jako parametr dalším metodám ke zpracování

int zacatekVektoru() – tato metoda z načtené řádky určí začátek vektoru v souboru

int konecVektoru() – z načtené řádky určí konec a délku vektoru v souboru

int zacBitMasky() – z načtené řádky určí začátek a délku bitové masky¨

void prectiData() – každý vektor testovací vzniklý krokem LZPR s příslušným seedem testuje na shodu testovanývh vektorů v souboru, pokud se shodují, seedu jemuž testovací vektor náleží, přiřadí bitovou masku testovaného vektoru, pokud se shoduje více vektorů, jsou bitové masky ORovány; po provedení testu shody na všechny testovací vektory se všemi vektory testovanými, seedu náleží jedna bitová

maska

String[] retezecFitness(String A, String B) – metoda provede výše zmiňovanou logickou operaci OR dvou vektorů a výsledkem bude jeden vektor, bitová maska

String[][] spoctiFitness() – z výsledné bitové masky spočte fitness, což je počet jedniček v ní obsažených a do pole uloží seed s jeho příslušnou fitnes

void zavriSoubor() – po ukončení algoritmu je vstupní soubor uzavřen

6. Třída SasBezGA

Pomocí této třídy je realizován program běžící bez použití genetického algoritmu, čili program modifikovaný. V této třídě není implementována žádná nová metoda, neboť třída využívá metod zmiňovaných dříve. Některé jsou mírně upraveny.

7. Třída SouborASeedy

Tato třída dělá to, že v cyklu provádí celý genetický algoritmus se všemi jeho metodami a to tak dlouho, dokud nebylo provedeno požadované množství iterací. V jedné itaraci je tedy vykonán celý genetický algoritmus s tím, že daná iterace končí vytvořením nové populace, která se stává vstupní populaci iterace následující. Třída neobsahuje žádnou novou metodu, ale navíc využívá proměnnou, která měří čas který byl potřebný pro provedení všech iterací. S tímto parametrem operuje třída předchozí, neboť po stejnou dobu bude běžet program modifikovaný.

8. Třída TestVektoru

V této třídě je implementovány následující metoda:

boolean shodaVektoru(String A, String B) – metoda provede porovnání na shodu dvou vektorů, pokud jsou stejné vrátí *true*, pokud ne, *false*;

Příloha C

Obsah přiloženého CD

Na přiloženém CD se nachází všechny zdrojové kódy programu