Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Michal Konopa

Kvaziparalelní optimalizace simulovaných systémů pomocí genetických algoritmů

Katedra softwarového inženýrství
Vedoucí diplomové práce: Prof. PhDr. RNDr. Evžen Kindler, CSc.
Studijní program: Informatika, Softwarové systémy

Děkuji vedoucímu diplomové práce panu Prof. PhDr. RNDr. Evženu Kindlerovi, CSc., za cenné připomínky a rady při zpracování diplomové práce. Dále děkuji panu RNDr. Jiřímu Weinbergerovi, CSc. za poskytnutí optimalizátoru PAROPTMULTI a cenné rady při jeho používání.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 1.8. 2009 Michal Konopa

Obsah

Název práce: Kvaziparalelní optimalizace simulovaných systémů pomocí genetických algoritmů

Autor: Michal Konopa
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Prof. PhDr. RNDr. Evžen Kindler, CSc.
e-mail vedoucího: ekindler@centrum.cz

Abstrakt: Cílem této práce je v jazyku SIMULA sestavit třídy umožňující modelování sezení expertů, kteří mají své vlastní (počáteční) představy o tom, která konfigurace parametrů zkoumaného systému je optimální, a všichni tento systém (každý se svou variantou) simulují, avšak během simulace si vyměňují informace o chování svých modelů a podle toho se učí a své vlastní konfigurace modifikují. Učení expertů na základě informací, získaných od kolegů, se provede technikou genetických algoritmů. Výsledný optimalizátor bude vyzkoušen na konkrétních příkladech, jak abstraktních (čerpaných z čisté matematiky), tak konkrétních (např. optimalizace daného projektu). Práce navazuje na disertaci RNDr. Jiřího Weinbergera, CSc., který v 80.letech vytvořil podobný typ optimalizátoru, kdy ovšem nebylo nic konkrétního známo o genetických algoritmech, a tak "učení" expertů hledajících optimum bylo modelováno technikami, které se sice v praxi ukázaly jako velmi úspěšné, dnes by však zasluhovaly být nahrazeny (nebo - co do efektivnosti své práce - alespoň porovnány) s metodou genetických algoritmů. Ta se sice při optimalizaci systémů pomocí algoritmicky řízených pokusů s jejich modely úspěšně používá, avšak nikdy s průběžným uplatněním při běhu těchto modelů. Paralelní běh více simulačních modelů na jednom monoprocesorovém počítači vyžaduje speciální programovací prostředek, jímž je právě Simula (jazyk objektově, blokově a procesově orientovaný).

Klíčová slova: simulace, optimalizace, genetické algoritmy

Title: Quasi-parallel optimizing of simulated systems by means of genetic algorithms

Author: Michal Konopa
Department: Department of Software Engineering
Supervisor: Prof. PhDr. RNDr. Evžen Kindler, CSc.
Supervisor’s e-mail address: ekindler@centrum.cz

Abstract: This work goals are SIMULA classes to model experts sessions so that each expert has his own start ideas the optimal system’s parameters. All the experts are simulating the system (each expert with his own parameters), but during the simulation they are mutually exchanging information about behavior of theirs models and – in accordance with this information – they are learning, changing their own system parameters. The learning process is performed by means of genetic algorithms. The resulting optimizer is tested on concrete examples, both from the mathematical theory and real practice (e.g. optimizing of a given project). This work reassumes the dissertation of RNDR. Jiří Weinberger. CSc., who made similar type of optimizer in the 80es, when nothing concrete was known on genetic algorithms and so the experts learning scheme was modeled by techniques, which had great success in solution of real problems afterwards; but nowadays it is worth to replace them by genetic algorithms, or at least to compare both methods. Genetic algorithms are successfully used in systems optimization, where the model run is algorithmic-managed, but never in the way of directly changing the models during their running times. Parallel run of multiple simulating models on the computer equipped with only one processor requires special programming mean, which is just SIMULA (object, block and process oriented language).

Keywords: simulation, optimization, genetic algorithms

1 Úvod

1.1 Optimalizace simulovaných systémů

Předmětem zájmu **simulace** a **modelování** je jistý **systém**. Systém lze chápat jako abstrakci určitého objektu reálného světa. Podle charakteru profese, která abstrakci na daném objektu zavádí, dostává systém i svůj přívlastek, např.: televizní přijímač je obvykle chápán jako elektronický systém – majitel si jej kupuje pro jeho elektronické vlastnosti, avšak bytový architekt ho tak chápat nemusí. Na jednom objektu lze tedy zavést více různých abstrakcí.

Abstrakce může nebo nemusí zanedbat význam času. Systém, v němž se význam času zanedbává, se nazývá **statický systém** (**static system**). Ve velké většině oborů, které berou v potaz význam času, se téměř vždy čas chápe „newtonovsky“, tj. má smysl mluvit o tom, že 2 události nastaly v systému současně nebo jedna nastala dříve než druhá. Systém, který význam času nezanedbává a kde je čas chápán „newtonovsky“, se v modelování a simulaci nazývá **dynamickým systémem** (**dynamic system**). Simulace se jinými než dynamickými systémy nezabývá.

V praxi zkoumání dynamických systémů se samozřejmě nepostupuje tím způsobem, že by se simulační experimenty odehrávaly vždy přímo se systémem, který je primárním předmětem onoho zkoumání (např. projektovaná továrna). Pro zkoumaný dynamický systém je vytvořen (vymodelován) jeho **simulující systém** (**simulátor**), který bude předmětem experimentování. Podstatou simulace, jakožto výzkumné techniky, je experimentování se simulujícím systémem, s cílem získat informace o systému simulovaném. V současnosti jsou simulátory realizovány téměř vždy formě počítačových programů.

Optimalizace daného systému je proces hledání takových hodnot volitelných parametrů tohoto systému, které povedou k „co nejlepším“ hodnotám uživatelem definovaných výstupních charakteristik tohoto systému. Z hlediska optimalizace má tedy smysl studovat pouze takové systémy, jejichž chování je ovlivnitelné nastavením hodnot volitelných parametrů tohoto systému (nelze např. „optimalizovat“ počasí – lze jej pouze s jistou přesností předpovědět). Další nezbytnou podmínkou je schopnost získat hodnoty výstupních veličin daného systému. Uživatel si vybere jistou podmnožinu těchto veličin, podle nichž bude daný systém optimalizován.

1.2 Cíle práce

Primárním cílem této práce bylo vyvinout takový optimalizátor simulovaných systémů, který hledá optimum metodou genetických algoritmů, přičemž se tato metoda uplatňuje přímo za běhu simulátorů optimalizovaných systémů. Metoda optimalizace pomocí genetických algoritmů se sice v simulační optimalizaci používá, ale nikdy ne tak, aby se uplatnila přímo za běhu simulátorů. Tuto práci lze tedy s ohledem na výše uvedenou skutečnost pojmout jako průkopnickou. Přímo navazuje na disertační práci RNDr. Jiřího Weinbergera, CSc., který vytvořil podobný typ optimalizátoru s tím rozdílem, že místo metod genetických algoritmů použil jiných technik, které se v praxi ukázaly být velmi úspěšné. Jeho optimalizátor, který implementuje tyto techniky, byl nazván **PAROPTMULTI**.

Název optimalizátoru, který je vyvíjen v této práci, je nazván **GA_PAROPTMULTI**. Podobně, jako je tomu v případě PAROPTMULTI, bude též GA_PAROPTMULTI

implementován ve formě počítačového programu, jehož zdrojový kód bude společně s uživatelskou dokumentací umístěn na CD připojeném k této práci.

Dalším cílem této práce je porovnat kvality výstupních řešení obou optimalizátorů. Porovnání bude provedeno jak na čistě abstraktních, tak na konkrétních příkladech.

1.2.1 Stručné shrnutí hlavních cílů

1. Vyvinout metodu optimalizace simulovaných systémů, založenou na principu genetických algoritmů.
2. Implementovat tuto metodu do formy optimalizátoru GA_PAROPTMULTI.
3. Porovnat kvalitu výsledných řešení PAROPTMULTI a GA_PAROPTMULTI.

1.3 Struktura práce

Druhá kapitola seznamuje čtenáře se základní terminologií v oblasti simulace a modelování. Jsou zde vysvětleny pojmy jako: systém, model, simulace, které budou hojně používány v kapitolách následujících. Samostatnou podkapitolu tvoří definice pojmu „diskrétní simulační model“, který je pro svou důležitost uveden zvlášť. Konec kapitoly je věnován popisu výhod a rizik simulačního modelování.

Třetí kapitola popisuje základní principy fungování genetických algoritmů. Podrobně seznamuje čtenáře s pojmy gen, chromozom, populace. Dále je zde popsáno obecné schéma genetického algoritmu. Tato kapitola se nijak nezabývá teorií schémat, o které se předpokládá, že zdůvodňuje, proč genetické algoritmy úspěšně hledají řešení.

Čtvrtá kapitola podrobně rozebírá a formalizuje problém optimalizace simulátoru. Pro tyto účely jsou zde zavedeny nezbytné formalismy, definující pojmy jako: simulátor, parametry simulátoru, vektorové kritérium apod. Důraz byl kladen na také přesné vymezení optimalizačního kritéria.

Pátá kapitola dopodrobna popisuje optimalizátor PAROPTMULTI, který byl vyvinut RNDr. Jiřím Weinbergerem, CSc., a jehož výkonné jádro tvoří algoritmus tzv. alternujících heuristik, který zde bude také podrobně popsán.

Šestá kapitola detailně popisuje návrh optimalizační metody s diskusí různých alternativ řešení. Pozornost je věnována hlavně podkladovému genetickému algoritmu a jeho vztahu k dekompozici simulárního času.

Sedmá kapitola navazuje na kapitolu předchozí a popisuje implementaci navržené optimalizační metody do optimalizátoru GA_PAROPTMULTI. Nejprve je popsáno rozdělení optimalizátoru do hlavních modulů. Následuje detailní popis každého modulu.

Náplní osmé kapitoly je testování a porovnání obou optimalizátorů na uměle vytvořených instancích. Zdůrazňuje výhody uměle vytvořených instancí ve smyslu možnosti uživatelské volby jejich klíčových definovaných vlastností. Pojem testovací instance je zde přesně vymezen a formalizován. Práce jde ještě dále a definuje pojem třídy testovacích instancí a pojem systému tříd. Pojmem třídy testovacích instancí lze, vágně řečeno, označit množinu testovacích instancí jistých, přesně vymezených vlastností. Oba optimalizátory byly testovány a porovnávány, ve smyslu kvality svých výsledných řešení, na přesně vymezených třídách. Nezbytnou částí je i podrobný popis metrik kvalit výsledných řešení. Konec této kapitoly je tvořen tabulkami výsledků a interpretací jejich hodnot.

Devátá kapitola se též věnuje testování a porovnávání obou optimalizátorů, ale již na reálných simulačních modelech, přičemž jejich simulované systémy jsou systémy hromadné obsluhy. Na počátku je popsána definice a klasifikace systémů hromadné obsluhy. Následuje popis testovaných systémů a popis vlastních testů. Konec již tradičně patří shrnutí výsledků a tabulkám.

Závěr je věnován celkovému shrnutí výsledků práce. Jsou zde naznačeny možné směry budoucího vývoje optimalizátoru GA_PAROPTMULTI s ohledem na jeho výkonné jádro. Dále je zde naznačena i potřeba další práce v oblasti problematiky systémů tříd testovacích instancí.

2 Simulace a modelování

Tato kapitola seznamuje čtenáře se základní terminologií v oblasti simulace a modelování. Jsou zde vysvětleny pojmy jako: systém, model, simulace, které budou hojně používány v kapitolách následujících. Na konci kapitoly jsou rozebrány výhody a rizika simulačního modelování – viz [2]. Čtenáře, který má hlubší zájem o problematiku simulace a modelování, lze odkázat na zdroj [1], který byl hlavním informačním zdrojem této kapitoly.

2.1 Definice základních pojmů

2.1.1 Systém

V simulaci a modelování se studuje určitý **objekt** reálného světa. Tento objekt buď skutečně existuje (např. konkrétní továrna), nebo o kterém se uvažuje, že by existovat mohl (např. výrobní provoz, který by měl být realizován). Objekt, ve své úplné složitosti, není možno zcela a beze zbytku lidskými rozumovými schopnostmi pochopit a intelektuálně zvládnout. Z tohoto důvodu zavádí různé obory vědy na objektu vhodně zvolené **abstrakce**, které zanedbávají jeho jisté aspekty, což posléze umožňuje danému oboru objekt racionálně uchopit. To mimo jiné znamená, že pracovníci daného odvětví budou moci o objektu racionálně komunikovat.

Takováto abstrakce se v modelování a simulaci nazývá **systémem**. Podle charakteru profese, která abstrakci na daném objektu zavádí, dostává systém i svůj přívlastek, např.: televizní přijímač je obvykle chápán jako elektronický systém – majitel si jej kupuje pro jeho elektronické vlastnosti, avšak bytový architekt ho tak chápat nemusí. Na jednom objektu lze tedy zavést více různých abstrakcí.

Abstrakce může nebo nemusí zanedbat význam času. Systém, v němž se význam času zanedbává, se nazývá **statický systém** (**static system**). Ve velké většině oborů, které berou v potaz význam času, se téměř vždy čas chápe „newtonovsky“, tj. má smysl mluvit o tom, že 2 události nastaly v systému současně nebo jedna nastala dříve než druhá. Systém, který význam času nezanedbává a kde je čas chápán „newtonovsky“, se v modelování a simulaci nazývá **dynamickým systémem** (**dynamic system**). Simulace se jinými než dynamickými systémy nezabývá.

Množina okamžiků, v nichž dynamický systém existuje, se nazývá **časovou existencí** tohoto systému. V praxi nemá smysl mluvit o jiných druzích existence – dále se proto bude mluvit pouze o **existenci dynamického systému**.

Existence dynamického systému je dána rovněž abstrakcí. Např. počítač se při nějaké akci (třeba řízení výrobního procesu) chápe jako systém, který existuje pouze během této akce, přestože jako objekt existuje před ní a pravděpodobně i po ní. Množina takovýchto okamžiků nemusí být interval reálných čísel: např. pro makroekonoma mohou být důležitá jen data na koncích periodicky se opakujících účtovacích období a to, co se děje během těchto období, zanedbává – systém pro něj existuje pouze v konečné množině navzájem izolovaných časových okamžiků, kterým lze jednoznačně přiřadit polohu na časové ose pomocí reálných čísel. Existence dynamického systému může být v principu jakákoliv neprázdná množina reálných čísel. V praxi jde vždy o množinu „dostatečně velikou“, což je ovšem mlhavý, ale srozumitelný pojem.

V modelování a simulaci je systém chápan tak, že je složen z **prvků** (**elements**). Mezním případem je systém o jediném prvku, což je v simulaci poměrně vzácná praxe. Běžně se sytém skládá z více prvků: znalost jejich chování nám usnadňuje porozumět

chování celého systému. Prvky systému – prvky určité abstrakce na objektu, mohou odpovídat pozorovaným fyzickým složkám objektu, pozorovaným logickým složkám objektu (např. schopnost jisté součásti), simulace však neklade omezení na způsob, jak rozklad provést.

V dynamickém systému se může počet jeho prvků během jeho existence měnit. Např. biologický systém se může rozrůstat, v technických a ekonomických aplikacích je ovšem častá taková praxe, kdy prvky mohou do systému „vstupovat“ a systém „opouštět“. Takové prvky se nazývají **transakcemi** (**transactions** nebo **temporary elements**). Příkladem transakcí mohou být zákazníci přicházející do obchodního domu nebo zakázky, přicházející do výrobního podniku. Prvky, které jsou v dynamickém systému během celé jeho existence, se nazývají **permanentní prvky** (**permanent elements**) nebo též **aktivity** (**activities**).

Prvky systému mají své vlastnosti, které se nazývají **atributy**. Příkladem může být teplota ingotu v systému oceláren (jde o tzv. **aritmetický** nebo reálný atribut, protože nabývá aritmetických hodnot, reálných čísel), funkčnost stroje ve výrobním systému (jde o tzv. **booleovský** atribut, protože nabývá booleovských hodnot „ano a „ne, konkrétněji řečeno „schopen pracovat a „v poruše), nebo jméno zákazníka banky (jde o tzv. **textový** atribut, neboť nabývá textových hodnot). Atributy tedy přiřazují prvkům určité hodnoty a ty se u prvků dynamického sytému mohou v čase měnit.

Dynamický systém je každém okamžiku své existence v určitém **stavu** (**state**). Stav dynamického systému v určitém čase *t* by měl být dán hodnotami všech atributů prvků přítomných v tomto čase v systému. Stav je ovšem ovlivněn i vzájemnými relacemi prvků v systému: např. zakázka *Z* zpracovávána na stroji *S1* definuje přirozeně jiný stav než ten, kdy je zakázka *Z* zpracovávána na stroji *S2*, přičemž $S1 \neq S2$.

V praxi simulace a dalších oblastí modelování se ovšem relace v matematickém slova smyslu a jak bylo též naznačeno na příkladu zakázky a strojů, nezavádějí. Jsou nahrazeny tzv. **referenčními atributy** (**pointers**), které přiřazují prvkům systému jiné prvky. Tedy relace, že zakázka *Z* je právě zpracovávána na stroji *S1*, může být reprezentována jako: hodnota referenčního atributu „zpracovávaná zakázka“ prvku *S1* je rovna *Z*. Atributy, které nejsou referenční, se nazývají **standardní atributy**. Standardní atributy přiřazují prvkům systému „Standardní“ hodnoty (reálná čísla, booleovské hodnoty, texty). Oproti tomu referenční atributy přiřazují prvkům jiné prvky téhož systému nebo nic (zakázka je již v systému ale není zpracovávána na žádném stroji).

Změna hodnoty referenčního atributu znamená změnu konfigurace (struktury) dynamického systému. **Konfigurací** lze tedy rozumět souhrn všech referenčních atributů prvků v systému.

2.1.2 Model

V modelování a simulaci je termín **model** použit pro vyjádření analogie mezi dvěma systémy. Vztah obou systémů – **modelovaného** a **modelujícího** je dán tím, že každému prvku *P* modelovaného systému je přiřazen prvek *Q* modelujícího systému, každému atributu *g* prvku *P* je přiřazen atribut *h* prvku *Q* a pro hodnoty atributů *g* a *h* je dána určitá relace. Charakter takové relace není obecně omezen, v případě aritmetických atributů může být takovou relací např. úměrnost, tolerance, kombinace obého.

Jsou-li modelovaný i modelující systém statické, nazývá se takový model **statický**. V simulaci se však uplatní pouze tzv. **simulační modely**, které splňují následující požadavky:

1. Jejich modelované i modelující systémy jsou dynamické systémy
2. Existuje zobrazení τ existence modelovaného systému do existence modelujícího systému. Je-li tedy t_1 okamžik, v němž existuje modelovaný systém M_1, je mu přiřazen okamžik $\tau(t_1) = t_2$, v němž existuje modelující systém M_2, a tak je zobrazením τ přiřazen i stavu $S_1(t_1) = \sigma_1$ systému M_1 stav $S_2(t_2) = \sigma_2$ systému M_2.
3. Mezi stavy σ_1 a σ_2 jsou splněny požadavky na vztahy mezi prvky a jejich atributy, jak bylo výše popsáno pro modely obecně. Jako kdyby každému stavu σ_1 modelovaného systému odpovídal stav σ_2 modelujícího systému tak, že oba stavy jsou ve vztahu statického modelu.
4. Zobrazení τ je neklesající. Pokud nastane stav s modelovaného systému před stavem s^* téhož systému, pak stav, který odpovídá v modelujícím systému stavu s, nastane před stavem, který odpovídá stavu s^*, nebo mohou oba stavy nastat v modelujícím systému současně (v případě, kdy není modelující systém natolik přesný, aby dokázal zobrazit všechny detaily v modelovaném systému). Nikdy se však nesmí stát, aby bylo pořadí stavů v modelovaném systému a jim odpovídajících stavů v systému modelujícím přehozeno.

Model je tedy obecně složitá struktura, která váže dva systémy, jejich prvky a atributy, a v případě simulačních modelů i existence obou systémů. V běžné mluvě se však ustálila praxe, že pod slovem model se rozumí modelující systém. Je zřejmé, že toto není příliš výstižné a přesné, neboť opomíjí skutečnost, že model není pouze systém, ale že je obrazem určitého systému a že tento sytém zobrazuje určitým, přesně definovaným způsobem. Místo termínu „modelovaný systém“ se používá slovo **originál**.

V případě simulačního modelu se hovoří raději o systému **simulovaném simulujícím** než modelovaném a modelujícím. Existuje praxe, v níž se na místo termínu simulující systém používá termín **simulační model** nebo také **simulátor**. Termín simulátor nezavádí nepřesnost, a proto bude také v dalším textu použit.

2.1.3 Modelování

Podstatou modelování ve smyslu výzkumné techniky je náhrada zkoumaného systému jeho modelem (přesněji: systémem, který jej modeluje), jejímž cílem je získat pomocí pokusů s modelem informaci o původním zkoumaném systému.

Postupuje se tedy tak, že se vytvoří model, v němž je zkoumaný systém modelovaným systémem, ale experimentovat se bude s modelujícím systémem, přičemž cílem bude dozvědět se něco o modelovaném systému.

2.1.4 Simulace

Simulace je výzkumná technika, jejíž podstatou je náhrada zkoumaného dynamického systému jeho simulátorem s tím, že se simulátorem se experimentuje s cílem získat informace o zkoumaném dynamickém systému.

Je třeba zdůraznit, že aby šlo o simulaci, musí být cílem experimentů se simulátorem snaha dozvědět se něco o simulovaném systému. Pokud je simulátor realizován jako výpočet na číslicovém počítači, může se složkou simulace stát i experimentování se simulátorem, jehož cílem je získat informaci o něm samotném a ne o simulovaném systému: např. ladění příslušného programu nebo zjišťování správnosti použité numerické metody. Tento proces se **nazývá ověření správnosti modelu** neboli **verifikace modelu**.

Význam slovesa **simulovat** se tedy chápe jako dělat simulaci. Některá slovní spojení, která se občas objeví, je nejlépe ignorovat, neboť v dnešní době působí paradoxně, např. statická simulace. Doporučuje se rovněž vyhýbat se opačným výrazům, např. dynamická simulace.

2.1.5 Simulace na počítačích

Ve starších dobách byla simulace realizována na speciálních zařízeních, podle kterých potom dostávala příslušná simulace přívlastek: elektromechanická, hydrodynamická, mechanická atd. V dnešní době jsou již všechny tyto druhy simulací nahrazeny simulací, realizované na číslicovém počítači – **simulace číslicová** (**digital simulation**). Protože v dalším textu se bude hovořit pouze o ní, přívlastek „číslicová“ bude vynechán.

Přesnější vyjádření druhu simulace podle výše zmíněných pravidel se dnes již nepoužívá – nepíše se např. o simulaci pentiové. Pokud je však známo, že jde o simulaci číslicovou, může se vyjádřit přívlastkem charakter simulovaného systému. O **spojité simulaci** (**continuous simulation** nebo **continuous system simulation**, tj. simulace spojitých systémů) se mluví v případě, že se jeho atributy mění spojitě v čase. Naopak, pokud v simulovaném systému nenastávají spojité změny v čase, mluví se o **diskrétní simulaci** (v anglické literatuře se používá téměř výhradně termín **discrete event simulation**, tj. simulace diskrétních událostí). A konečně, pokud simulovaný systém vykazuje jak vlastnosti typické pro spojité systémy, tak vlastnosti diskrétního charakteru, mluví se o **kombinované diskrétně spojité simulaci**, častěji zkráceně o **kombinované simulaci** (**combined simulation**).

Jelikož je systém definován na určitém objektu pomocí jisté abstrakce, je logické, že na jednom a tomtéž objektu může být definován jak spojitý, tak diskrétní, případně i kombinovaný systém. I když simulujeme spojitý systém na číslicovém počítači, kde za tímto účelem vzniká jakýsi diskrétní dynamický systém, vzniklý z modelovaného spojitého dynamického systému diskretizací, jde stále o spojitou simulaci, protože přívlastek reflektuje simulovaný systém a nikoliv simulátor.

2.1.6 Termíny používané při číslicové simulaci

Program, který řídí výpočet při číslicové simulaci, se nazývá **simulačním programem**. Dosud neexistuje shoda v tom, zda se tímto programem myslí zdrojový kód napsaný autorem nebo již do strojového kódu přeložený program. V praxi však kvůli této nejednoznačnosti nedochází k velkým nedorozuměním a tak v dalším textu bude toto sousloví používáno jak pro text, který napíše autor simulačního modelu v programovacím jazyku, tak pro strojový kód, který z něho vznikne následnou kompilací.

Pokus se simulačním modelem se nazývá **simulační pokus** (**simulation experiment**). Když na počítači běží simulační pokus, je vhodné evidovat čas, který by dané fázi výpočtu odpovídal v simulovaném systému. Nechť je tento čas uložen v proměnné *time*. Hodnota *T* proměnné *time* vyjadřuje situaci, jako by se simulovaný systém právě nacházel v čase *T*. Vzhledem k tomu, že v simulačním modelu nesmí být pořadí odpovídajících si stavů v simulovaném a simulujícím systému přehozeno, nesmí hodnota proměnné *time* nikdy klesnout, musí občas povyrůst. Mezinárodní autority v oboru simulací doporučily, aby se tyto hodnoty nazývaly **simulárním časem**

(**simular time**), avšak odborná veřejnost používá ne zcela přesný, ale všeobecně rozšířený termín **simulovaný čas** (**simulated time**).

Posloupnost podobných simulačních pokusů, které mají stejný účel, se nazývá **simulační studie** (**simulation study**). V dnešní době je již obvykle realizována jako jeden výpočet (task). Před každým pokusem se simulovaný čas vrátí zpět a změní se některé hodnoty způsobem, který nemá v simulovaném systému smysl – např. se vyprázdní fronty a vynulují některé součty. Jeden běh – ve smyslu použití zkompilovaného programu, odpovídá nejčastěji jedné simulační studii, obecně tedy několika simulačním pokusům.

Simulační krok (simulation step) je takový úsek výpočtu, během něhož se nemění hodnota simulovaného času. Simulační studie se tedy skládá ze simulačních pokusů a ty se skládají ze simulačních kroků. Na začátku každého simulačního pokusu se simulovaný čas vrátí na svou výchozí hodnotu (obvykle hodnotu nula) a s výjimkou prvního simulačního kroku každého simulačního pokusu, se zvětší hodnota simulovaného času o určitý nezáporný přírůstek. Pokud je tento přírůstek pro celý simulační pokus stejně velký, potom se mluví o **ekvidistantním** simulovaném čase, v ostatních případech se mluví o **neekvidistantním** simulovaném čase.

2.2 Diskrétní simulační modely

Diskrétní simulační modely jsou charakterizovány tím, že všechny stavové proměnné nabývají pouze diskrétních hodnot a v průběhu času se mění skokem. Nejčastějším případem diskrétních modelů jsou aplikace teorie hromadné obsluhy.

Pro mnohé diskrétní simulační modely (včetně těch využívaných v oblasti hromadné obsluhy) jsou charakteristické následující rysy:

- proměnný počet prvků systému (požadavků)
- reprezentace front pomocí spojových seznamů
- vysoký stupeň paralelnosti výpočtu
- velké nároky na řízení programu (vyplývá z paralelnosti),
- velké nároky na paměť (velký počet prvků – požadavků)

2.3 Výhody simulačního experimentování

- Simulacemi lze řešit takové problémy, které nejsou řešitelné jinými objektivními metodami (např. analytickými). Modelovaný systém bývá totiž většinou velmi složitý, obsahuje mnohodimensionální vektory vstupů a výstupů, které jsou mezi sebou velmi komplikovaně vázány a často ani nejsou známy struktury těchto vazeb.
- Simulační modely mohou být velmi komplexní bez újmy na své platnosti. Lze jimi zachytit velmi rozsáhlé modelované systémy, a to s podrobností, která je omezena prakticky pouze stupněm našich kvantifikovatelných znalostí.
- Mohou dobře reprezentovat stochastické vlastnosti reality. Narozdíl od jiných disciplín operačního výzkumu (např. teorie front), se může v simulacích pracovat s libovolným rozdělením pravděpodobnosti. Náhodnou veličinu lze zadat např. empiricky zjištěným histogramem a tímto způsobem je možno v modelu zobrazit náhodný charakter objektu mnohem přesněji, než kdyby se

chování objektu nahrazovalo jistým teoretickým rozdělením, které by do modelu mohlo zanést další nepřesnosti.

- Lze snadno modelovat časový průběh operací, a to ve zvolených časových bodech.
- Experimentováním je možno vyšetřit široké pásmo alternativ modelovaného systému, přičemž přejít z jedné varianty na druhou je snadné. Problémově orientované jazyky, sestavené na základě moderních univerzálních jazyků, umožňují často i snadné zásahy do struktury modelu, který je reprezentován počítačovým programem.
- Lze modelovat alternativy, které by jinak nebylo možné uskutečnit na reálném objektu, např. pro jejich nákladnost, délku trvání, nebezpečnost apod.
- Chod experimentu, délka simulovaného období a opakovatelnost je plně pod kontrolou experimentátora. Lze tedy snadno vyhovět některým požadavkům, které např. vyžadují určité metody matematické statistiky.
- Simulační model se relativně snadno udržuje i uchovává, takže je dle potřeby k dispozici pro možné další experimentování.
- Existuje celá řada programovacích simulačních jazyků, které usnadňují realizaci simulačních modelů

2.4 Rizika simulačního experimentování

- Oproti analytickým metodám je simulační řešení často značně nákladné a vyžaduje dostatek času jak programátora, tak počítače.
- Je třeba pečlivě sledovat, zda při experimentování nedošlo k porušení platnosti modelu. Potom by totiž obdržené informace vedly k chybným závěrům.
- Dobrá simulace vyžaduje zkušeného analytika, který se dostatečně hluboko vyzná nejenom v programování, stochastice apod., ale dobře chápe a rozumí struktuře i funkci simulačního modelu.

3 Genetické algoritmy

Tato kapitola poskytuje pouze stručný úvod do teorie genetických algoritmů. Není zde popsána teorie schémat, o které se předpokládá, že vysvětluje chování genetických algoritmů. Podrobnější informace včetně popisu teorie schémat poskytují zdroje [4] – hlavní informační zdroj této kapitoly, [5] a [6]. Zdroj [6] se zaměřuje hlavně na problematiku genetických algoritmů s reálně kódovanými chromozómy.

Genetické algoritmy, jak již jejich název sám napovídá, byly inspirovány přírodou a představou biologické evoluce. Termín „genetické algoritmy“ zavedl v roce 1975 John Holland pro množinu algoritmů, které měly několik shodných rysů. Zpočátku byly navrženy a používány pro simulaci biologické evoluce, teprve později byly aplikovány pro řešení optimalizačních úloh, jelikož se na tomto poli ukázaly jako velmi výkonné. Jako jejich velká přednost se ukázala schopnost adaptace na měnící se okolní podmínky.

3.1 Terminologie

Všechny živé organismy se skládají z buněk. V každé buňce je stejná sada **chromozomů** (**Chromosome**), které se skládají z molekul deoxyribonukleové kyseliny DNA. Chromozomy určují vlastnosti celého organismu a skládají se z **genů** (**Gene**), což jsou bloky DNA. Každý gen reprezentuje jistou vlastnost celého organismu, např. barvu očí. Různé hodnoty, jakých geny mohou nabývat, nazýváme **allela** (**Allele**). Různé organismy mají různý počet genů. Množina chromozomů, popisujících veškeré fyzické vlastnosti organismu, se nazývá **genom** (**Genome**). Konkrétní nastavení genů se nazývá **genotyp** (**Genotype**). Spolu s vnějšími vlivy po narození určuje genotyp tzv. **fenotyp** (**Phenotype**), což jsou veškeré fyzické a psychické vlastnosti jedince.

Při reprodukci organismů dochází ke **křížení** (**Crossover**), neboli **rekombinaci** (**Recombination**). V rámci tohoto procesu se vybírají geny rodičů a jejich kombinací se vytváří genotyp nového jedince – potomka (**Offspring**). Může také docházet k náhodným **mutacím** (**Mutation**), tedy ke změnám malé části genotypu. **Úspěšnost** (**Fitness**) hodnotíme jako pravděpodobnost toho, že jedinec přežije do své reprodukce, nebo jako počet jeho potomků. Reprodukci si zajistí pouze některé organismy a s vyšší pravděpodobností to budou takové, které mají vyšší fitness.

V kontextu genetických algoritmů se pod pojmem chromozom rozumí přípustné řešení daného problému, uloženého ve tvaru řetězce předem dané abecedy. Většinou je to abeceda binární, v poslední době se však prosazuje, hlavně v optimalizaci problémů na spojitých prostorech, také kódování reálnými čísly. Řetězec má ve většině případů pevnou délku, ale objevují se i aplikace s proměnnou délkou řetězce. Genem potom muže být jeden znak řetězce nebo jeho úsek. Allela je hodnota, kterou muže gen nabývat. U binárního chromozomu je to tedy buď 0, nebo 1. Křížení je jedním ze základních nástrojů tvorby nových kandidátských chromozomů. Ty jsou tvořeny výměnou genetického materiálu rodičů. Křížení pracuje obvykle na dvou chromozomech, výjimečně se objevují implementace, kdy je křížení definováno na více než dvou chromozomech najednou. Mutace mění hodnoty náhodně zvolených genů na jiné. V případě binární abecedy jednoduše změní 0 na 1 a naopak. Většina aplikací genetických algoritmu má přípustné řešení uloženo v jednom chromozomu. V souvislosti s optimalizací se často mluví o **prohledávaném prostoru**. Pod tímto pojmem se rozumí množina všech možných chromosomu – řetězců dané abecedy, popisujících přípustné řešení. Pokud se prohledávaný prostor rozšíří o jednu dimenzi, do

které se poté vynese hodnota fitness daného řešení, vznikne tzv. **fitness plocha**. Na této ploše lze sledovat útvary různé krajinotvorné útvary, např. kopce, údolí apod. Tyto překážky musí být překonány pro nalezení optimálního řešení, což je logicky globální maximum.

3.2 Pravidla a vlastnosti

Genetické algoritmy nemají samy o sobě žádnou přesnou definici, protože pod tímto pojmem se rozumí množina algoritmů, které se vzájemně mohou v různých aspektech značně lišit. Všechny ale dodržují základní všeobecná pravidla.

3.2.1 Všeobecná pravidla

1. Algoritmus vždy pracuje s množinou chromozomů. Tato množina se nazývá **populace** (**Population**). Pokud se hovoří o posloupnosti populací, používá se termín **generace** (**Generation**). V každém kroku je vždy prostřednictvím genetických operátorů vytvořena nová populace z populace předešlé. Tato nová populace má většinou stejný počet jedinců, jako populace předchozí. Stará populace se potom na dalším vývoji algoritmu nijak nepodílí.

2. **Selekce** je operátor, kterým jsou vybírány chromozomy ze staré populace pro následné křížení. Tyto chromozomy jsou vybírány na základě hodnoty jejich fitness. Existuje mnoho různých druhů selekce, bezesporu nejpoužívanější je metoda tzv. **Ruletového kola**. Jeho činnost spočívá v tom, že jednotlivé chromozomy získají na obvodu ruletového kola takovou část, která odpovídá poměru jejich fitness k průměru fitness populace. Poté se kolem ”točí” a je vybírán ten chromozom, na jehož úseku se kolo zastaví. Stejný chromozom je možno zvolit vícekrát. Je tedy zřejmé, že čím má chromosom vyšší hodnotu fitness, tím větší má šanci stát se vzorem pro tvorbu nové generace a tím větší šanci má šířit svou genetickou informaci dále. Tento operátor simuluje mechanismus přirozeného výběru v přírodě, kde přežívají jen „nejsilnější“ jedinci.

3. **Křížení** je hlavním operátorem pro tvorbu nových chromozomů. Většinou je implementováno tak, že ze dvou jedinců vybraných operátorem selekce vytvoří dva nové jedince, výjimečně se objevují realizace, kdy operátor pracuje nad jiným počtem chromozomů. Při nejjednodušším **jednobodovém křížení** je náhodně zvolen bod v chromozomu a odpovídající si části v chromozomech před (nebo za) tímto bodem se vymění. Pro názornost předpokládejme binární řetězec délky 6. Mějme dva chromozomy (rodiče) 110100, 100011 a křížící bod 2. Potom po křížení jsou vytvořeny dva nové řetězce 100100 a 110011. Přirozeným a používaným rozšířením je **n-bodové křížení**, kdy je na počátku zvoleno *n* křížících bodu a úseky ohraničené těmito body jsou střídavě přesouvány nebo zůstávají v původním chromozomu. Posledním, běžně používaným křížením je tzv**. uniformní křížení**, kdy je každý gen s předem danou pravděpodobností vyměněn. Samozřejmě existují další varianty křížení, ale tyto jsou, měřeno rozšířeností, nejdůležitější.

4. **Mutace** se uplatňuje po operátoru křížení a jejím hlavním úkolem je zajistit, aby populace nekonvergovala do jednoho chromozomu. Mutace s jistou malou pravděpodobností změní hodnotu genu. Při binárním chromozomu jednoduše změní 0 na 1 a naopak. V případě, že je množina allel větší, vybere mutace náhodně allelu různou od původní. V případě, kdy je množina allel rovna množině reálných čísel, se často k hodnotě genu přičte číslo z generátoru náhodných čísel (často se používá generátor s Gaussovou distribucí pravděpodobnosti s nulovým posunutím, nebo s uniformní distribucí na intervalu symetrickém podle bodu nula). Většina implementací mutace pracuje podle uvedeného schématu s tím, že jsou přizpůsobeny použité abecedě.

5. **Elitismus** není běžné uváděn a nepatří mezi operátory, které by charakterizovaly genetické algoritmy a některé implementace ho ani nepoužívají. Při vytváření nové populace se lehce stane, že chromozom s nejvyšší dosud nalezenou hodnotou fitness ztratíme. Tím ale dochází ke ztrátě velmi zajímavého chromozomu. Proto se používá operátor elitismu, který vždy zkopíruje několik nejlepších (z hlediska fitness) chromozomu ze staré populace do nové.

3.2.2 Definice fitness funkce

Kriticky důležitou součástí genetického algoritmu je definice fitness funkce, která by měla co nejlépe postihnout povahu zkoumaného problému. Mnoho optimalizačních problémů má povahu minimalizace jisté cenové funkce. I když bude mít úloha charakter maximalizace, bude třeba definovat určitou transformaci z účelové funkce úlohy na fitness funkci, a to např. pro to, že účelová funkce nemusí garantovat, že bude na celém svém definičním oboru nezáporná (záporná fitness nemá pro genetické algoritmy smysl).

3.2.3 Obecné schéma genetického algoritmu

1. **Inicializace počáteční populace.**
   Chromozomy, které budou tvořit počáteční populaci, se volí náhodně z prohledávaného prostoru. Většinou na ně nejsou kladena žádná dodatečná omezení. Některé implementace používají počáteční populaci větší mohutnosti, než jsou populace vytvářené během algoritmu. Vychází to z předpokladu, že algoritmus si na počátku vybere lepší chromosomy, ze kterých bude tvořit první generaci.

2. **Výpočet fitness.**
   Výpočtem fitness chromozomu v populaci se určí, jak je který vhodný pro to, stát se rodičem a šířit tak dál své geny. V tomto kroku se také napočítávají další důležité veličiny pro operátor selekce (např. průměrná fitness celé populace).

3. **Test ukončujících podmínek**
   Vyhodnotí se zastavovací kriterium a v případě jeho splnění algoritmus skončí.

4. **Vytvoření nové generace.**
   Nová populace vznikne pomocí operátoru křížení a selekce. Operátor selekce vybírá chromozomy, na které se poté aplikuje operátor křížení s předem určenou

pravděpodobností. Některé chromozomy se tedy beze změny zkopírují do nové generace.

5. **Mutace**
   Na všechny chromozomy nové generace se aplikuje s danou pravděpodobností operátor mutace.

6. **Elitismus** (volitelně)
   S pomocí elitismu zkopírujeme několik nejlepších chromozomů ze staré generace do nové generace.

7. **Obměna generací**
   Stará generace se dále již neúčastní výpočtu a je nahrazena generací novou.

8. **Opakování**
   Návrat do kroku 2.

3.3 Ukončující podmínky

Jsou důležitou součástí genetického algoritmu a jejich volba není vždy snadná. V praxi se používají nejčastěji tyto přístupy:

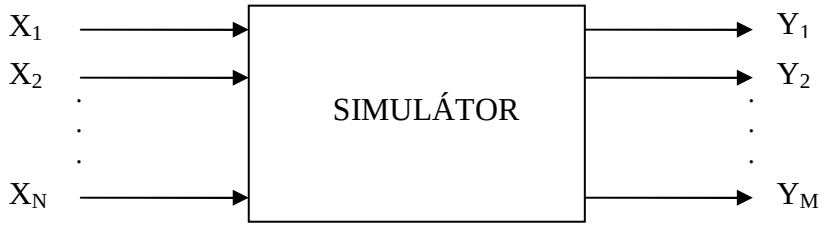
1. Algoritmus se zastaví po pevně zvoleném počtu generací.
2. Algoritmus se zastaví, pokud nedošlo ke zlepšení řešení během pevně stanoveného počtu generací, tj. aktuální hodnota nejlepšího fitness se během tohoto počtu generací nezměnila
3. Je definována mez, při jejímž překročení se bere řešení, které tuto hodnotu svou fitness překročilo, jako dostačující.

4 Optimalizace simulovaných systémů

Kapitola podrobně rozebírá a formalizuje problém optimalizace simulátoru. Pro tyto účely jsou zde zavedeny nezbytné formalismy, definující pojmy jako: simulátor, parametry simulátoru, vektorové kritérium apod. Důraz byl kladen na také přesné vymezení pojmu optimální množiny.

4.1 Simulátor

Pro účely optimalizace lze na simulátor jistého systému pohlížet jako na systém, který se skládá z množiny **vstupů** a množiny **výstupů**. Každý vstup z množiny vstupů daného simulátoru je reprezentován jistou vstupní proměnnou, která má definovanou doménu. Totéž platí analogicky pro množinu výstupů.



Obrázek 4.1: Jednoduché schéma simulátoru

V oboru simulace a modelování se nemluví o množině vstupů, ale množině **parametrů**. Význam však zůstává tentýž. Místo o množině výstupů se potom mluví o **výstupních charakteristikách** systému, jejichž podmnožinu tvoří uživatelem definovaná kritéria.

Množinu parametrů simulátoru lze rozdělit do 2 kategorií podle toho, zda jsou či nejsou interpretovatelné v kontextu simulovaného systému. V systému hromadné obsluhy budou prakticky interpretovatelné parametry např. střední hodnota doby mezi příchody jednotlivých zakázek, střední hodnota doby zpracování zakázky, apod.

Parametry, které nejsou v kontextu simulovaného systému prakticky interpretovatelné, mají smysl pouze v kontextu simulátoru a zavádí do něj nezbytnou stochastičnost. Typickým představitelem této kategorie parametrů jsou hodnoty násad generátorů náhodných čísel. Zkratka PG nechť označuje pojem **generátor náhodných čísel**. PG většinou pracují takovým způsobem, že mezi hodnotou násady PG a posloupností vygenerovaných náhodných čísel, vzniklých voláním tohoto PG s iniciální hodnotou této násady, existuje vzájemně jednoznačné zobrazení. V praxi simulace je pohodlné mít K různých PG ($K \geq 1$), teoreticky však stačí pouze 1. Hodnoty násad PG výpočet simulátoru sice ovlivňují, ale nejsou prakticky interpretovatelné a tvoří tak náhodnou složku výpočtu simulátoru.

Z důvodu přehlednějšího matematického popisu, který bude následovat, nechť je množina parametrů jistým způsobem uspořádaná do **vektoru parametrů**.

Definice

$SIMS$ …množina simulujících systémů (simulátorů) diskrétních simulačních modelů

$sim\_par\_num : SIMS \to \mathbb{N}$ ...zobrazení, které pro daný simulátor vrací počet jeho prakticky interpretovatelných parametrů

$sim\_npg\_num : SIMS \to \mathbb{N}$ ...zobrazení, které pro daný simulátor vrací počet jeho prakticky neinterpretovatelných parametrů – násad PG

$$sim\_par\_dom : \left\{ \begin{array}{l} (sim, par\_id): \\ sim \in SIMS, \\ par\_id \in 1,..,sim\_par\_num(sim) \end{array} \right\} \to POW(\mathbb{R})$$

...zobrazení, které pro daný simulátor a číslo jeho prakticky interpretovatelného parametru vrací doménu hodnot tohoto parametru

sim_cart_dom …zobrazení, které pro daný simulátor vrací kartézský součin domén všech jeho prakticky interpretovatelných parametrů

sim_aprior_dom …zobrazení, které pro daný simulátor vrací všechny vektory hodnot z kartézského součinu domén hodnot všech jeho prakticky interpretovatelných parametrů, které jsou pro výpočet apriorně přípustné

Axiomy

$$\forall (sim, par\_id) \in DOM(sim\_par\_dom);$$
$$\left(\left| sim\_par\_dom(sim, par\_id) \right| > 0 \right)$$

$$DOM(sim\_cart\_dom) = SIMS$$

$$\forall sim \in DOM(sim\_cart\_dom);$$
$$\left(\begin{array}{l} sim\_cart\_dom(sim) = \\ \left(\begin{array}{l} sim\_par\_dom(sim,1) \times \\ sim\_par\_dom(sim,2) \times \\ \vdots \\ sim\_par\_dom(sim, sim\_par\_num(sim)) \end{array} \right) \end{array} \right)$$

$$DOM(sim\_aprior\_dom) = SIMS$$

$$\forall sim \in DOM(sim\_aprior\_dom);$$
$$\left(sim\_aprior\_dom(sim) \subseteq sim\_cart\_dom(sim) \right)$$

4.1.1 Výpočet se simulátorem

Výpočet simulátoru je plně determinován hodnotami všech svých parametrů. Jak již bylo dříve uvedeno, tyto parametry lze rozdělit podle toho, zda jsou či nejsou v simulovaném systému prakticky interpretovatelné. Kterýkoliv vektor hodnot prakticky interpretovatelných parametrů daného simulátoru, který je z hlediska výpočtu apriorně přípustný, se nazývá **variantou simulátoru** – je prvkem hodnot zobrazení sim_aprior_dom daného simulátoru. Slovo „apriorně“ je zde použito pro zdůraznění skutečnosti, že trajektorie výpočtu se může později ukázat jako nepřípustná i v případě tohoto apriorně přípustného vektoru.

Vektor hodnot prakticky neinterpretovatelných parametrů daného simulátoru nechť je nazýván **vektorem násad simulátoru**. Hodnotami jeho položek jsou přirozená čísla.

Výpočet simulátoru s jednou a toutéž variantou může obecně proběhnout více různými trajektoriemi podle hodnot jeho vektoru násad PG, které se váží k jednotlivým výpočtům s toutéž variantou.

Trajektorie výpočtu je z hlediska praktické interpretace neúnosně složitým útvarem. Interpretovány jsou a mohou být pouze některé veličiny a funkce simulárního času, definované na trajektorii výpočtu. Tyto veličiny a závislosti jsou zpravidla jednoduše programátorsky realizovatelná, avšak matematicky přitom často obtížně vyjádřitelná. V praxi simulace hrají tyto prvky rozmanité role, mohou být pozorovány či ignorovány z celé řady prakticky důležitých důvodů. Pouze některé z těchto veličin a časových funkcí jsou složkami vektorového kritéria, zvoleného pro posouzení kvality dané varianty simulátoru.

Bez újmy na obecnosti lze předpokládat, že se optimalizační algoritmus bude snažit všechny složky vektorového kritéria maximalizovat. Pokud se výpočet dostane do nepřípustného stavu, bude tento výpočet s danou variantou okamžitě ukončen a všechny hodnoty vektorového kritéria, příslušné této variantě, budou nastaveny na hodnotu $-\infty$.

Definice

SIM_TIMES …množina všech přípustných hodnot celkových dob běhu simulátorů

$sim\_total\_time : SIMS \rightarrow SIM\_TIMES$ …zobrazení, které pro daný simulátor vrací celkovou dobu simulace

$sim\_crit\_num : SIMS \rightarrow \mathbb{N}$ …zobrazení, které pro daný simulátor vrací počet položek kritéria

sim_crit_values …zobrazení, které pro daný simulátor, variantu, vektor násad a hodnotu simulárního času vrací vektor hodnot kritérií

Axiomy

$SIM\_TIMES \subseteq \mathbb{R}^{+} \cup \{0\}$

$$DOM(sim\_crit\_values) = \left\{ \begin{array}{l} (sim, variant, npg, time): \\ sim \in SIMS, \\ variant \in sim\_aprior\_dom(sim), \\ npg \in \mathbb{N}^{sim\_npg\_num(sim)}, \\ time \in \langle 0, sim\_total\_time(sim) \rangle \end{array} \right\}$$

$$\forall (sim, variant, npg, time) \in DOM(sim\_crit\_values);$$
$$\left(sim\_crit\_values(sim, variant, npg, time) \in \mathbb{R}^{sim\_crit\_num(sim)} \right)$$

4.2 Optimalizace simulátoru

4.2.1 Optimalizační kritérium

Vzniká otázka, jak porovnat kvalitu dvou různých variant vzhledem k jednomu zvolenému kritériu na celém intervalu simulárního času. Vzhledem k tomu, že se optimalizátor bude snažit složky kritérií maximalizovat, má smysl říci, že kvalita dané varianty je vzhledem k danému kritériu na daném časovém intervalu tím větší, čím je větší obsah plochy, shora ohraničené hodnotami tohoto kritéria na tomtéž intervalu. Přesněji řečeno, bude to střední hodnota všech hodnot těchto obsahů, protože jedna a tatáž varianta definuje obecně jinou trajektorii výpočtu podle aktuálního vektoru násad.

Definice

ti_crit_values ... zobrazení, které pro daný simulátor, variantu, vektor násad a hodnotu simulárního času vrací vektor hodnot časových integrálů kritérií

ti_mean_crit_values ... zobrazení, které pro daný simulátor, variantu a hodnotu simulárního času vrací vektor středních hodnot časových integrálů kritérií

Axiomy

$$DOM(ti\_crit\_values) = \left\{ \begin{array}{l} (sim, variant, npg, time): \\ sim \in SIMS, \\ variant \in sim\_aprior\_dom(sim), \\ npg \in \mathbb{N}^{sim\_npg\_num(sim)}, \\ time \in \langle 0, sim\_total\_time(sim) \rangle \end{array} \right\}$$

$$\forall (sim, variant, npg, time) \in DOM(ti\_crit\_values);$$
$$\left(ti\_crit\_values(sim, variant, npg, time) \in \mathbb{R}^{sim\_crit\_num(sim)} \right)$$

$$DOM(ti\_mean\_crit\_values) = \left\{ \begin{array}{l} (sim, variant, time): \\ sim \in SIMS, \\ variant \in sim\_aprior\_dom(sim), \\ time \in \langle 0, sim\_total\_time(sim) \rangle \end{array} \right\}$$

Vrací vektor středních hodnot integrálů kritérií dané varianty a hodnoty simulárního času:

$$\forall (sim, variant, time) \in DOM(ti\_mean\_crit\_values);$$

$$\left(\begin{array}{l} ti\_mean\_crit\_values(sim, variant, time) \\ = \\ \underset{\forall npg \in \mathbb{N}^{sim\_ngp\_num(sim)}}{E} (ti\_crit\_values(sim, variant, npg, time)) \end{array} \right)$$

Vzhledem k tomu, že kritérium kvality dané varianty simulátoru je obecně vícesložkové, je třeba předně definovat, jakým způsobem budou porovnávány 2 vektory středních hodnot časových integrálů kritérií. Problém je v tom, že vektory reálných hodnot lze uspořádat pouze částečně, nikoliv úplně. V oblasti optimalizace multikriteriálních problémů je používána tzv. **Pareto optimalita**. Pro vysvětlení tohoto termínu je třeba zavést ještě několik dalších termínů, z nichž některé byly převzaty z [7]

Multikriteriální optimalizační problém (obecná definice)

Multikriteriální optimalizační problém (MOP) je definován jako problém maximalizace $F(x) = \left(f_1(x), ..., f_k(x) \right)$ vzhledem k podmínkám: $g_i(x) \le 0, i \in 1,..,m$ a $x \in DEC\_SPACE$. Řešení tohoto problému maximalizuje složky vektoru $F(x)$, kde $x = \left(x(1),..,x(n) \right)$ je *n*-složkový reálný vektor, náležející do DEC_SPACE.

Pareto nedominovanost

Reálný konečný vektor $u = (u(1),...,u(k))$ dominuje vektoru $v = (v(1),...,v(k))$ právě tehdy, když platí:

$$\forall i \in 1,..,k, \exists j \in 1,..,k;$$
$$\left(\left(u(i) \ge v(i) \right) \wedge \left(u(j) > v(j) \right) \right)$$

Značení: $u \ge v$

Pareto optimalita

Řešení $x \in DEC\_SPACE$ je **Pareto-optimální** vzhledem k DEC_SPACE právě tehdy, když platí:

$$\neg\exists x' \in DEC\_SPACE;$$
$$\left(v(f_1(x'),..,f_k(x')) \geq u(f_1(x),..,f_k(x))\right)$$

Tedy dané řešení je optimální právě tehdy, když neexistuje prvek z množiny parametrů, jehož vektor hodnot optimalizovaných funkcí dominuje vektoru hodnot optimalizovaných funkcí tohoto řešení.

Pareto optimální množina

Pareto optimální množina (P^*) je vzhledem k $F(x)$ daného MOP, definována jako:

$$P^* = \left\{ \begin{array}{l} x \in DEC\_SPACE: \\ \left(\begin{array}{l} \neg x' \in DEC\_SPACE; \\ F(x') \geq F(x) \end{array} \right) \end{array} \right\}$$

Pareto Front

Pareto Front (PF^*) je vzhledem k $F(x)$ a P^* daného MOP definována jako:

$$PF^* = \left\{ \begin{array}{l} u: \\ u = F(x), \\ x \in P^* \end{array} \right\}$$

Prvky množiny PF^* se nazývají také jako **nedominované vektory**. PF^* je tedy množina vektorů, jejíž žádný prvek není dominován žádným jiným prvkem takovým, který představuje hodnoty optimalizovaných funkcí, vyčíslovaných na množině volitelných parametrů daného MOP.

Globální maximum multikriteriálního optimalizačního problému

Nechť platí tyto předpoklady:

$$|DEC\_SPACE| > 0$$
$$k \geq 2$$
$$F: DEC\_SPACE \subseteq \mathbb{R}^n \rightarrow \mathbb{R}^k$$

Množina PF^* je **globální maximum**, právě tehdy, když platí:

$$\begin{pmatrix} \forall vec \in PF^{*}, \exists x \in DEC\_SPACE; \\ (F(x) = vec) \end{pmatrix}$$

$$\wedge$$

$$\begin{pmatrix} \forall vec \in PF^{*}, \forall i \in 1,..,k; \\ (vec(i) < \infty) \end{pmatrix}$$

$$\wedge$$

$$\begin{pmatrix} \forall vec \in PF,^{*} \neg \exists x \in DEC\_SPACE; \\ (F(x) \geq vec) \end{pmatrix}$$

4.2.2 Globální maximum simulátoru

Je intuitivně zřejmé, že by bylo vhodné definovat globální optimum daného simulátoru analogicky s tím, jak je definováno globální maximum multikriteriálního optimalizačního problému. K tomu je zapotřebí definovat jakýsi ekvivalent pojmu P^{*}, což by měla být jakási optimální množina řešení daného simulátoru.

Optimální řešení daného simulátoru je takový apriorně přípustný vektor hodnot jeho prakticky interpretovatelných parametrů – tedy varianta, jehož příslušný vektor středních hodnot časových integrálů kritérií není dominován žádným jiným vektorem středních hodnot časových integrálů jiné varianty téhož simulátoru.

Globální maximum daného simulátoru je množina vektorů středních hodnot časových integrálů kritérií, jejíž příslušné varianty tvoří všechna optimální řešení tohoto simulátoru. Z předchozího vyplývá, že globální maximum simulátoru je tvořeno vzájemně nedominovanými vektory.

Definice

sim_opt_sol …zobrazení, které pro daný simulátor vrací množinu všech jeho optimálních řešení

sim_glob_max …zobrazení, které pro daný simulátor vrací jeho globální maximum

Axiomy

Množina optimálních řešení simulátoru

$DOM(sim\_opt\_sol) = SIMS$

$$\forall sim \in DOM(sim\_opt\_sol);$$

$$\left(sim\_opt\_sol(sim) = \left\{ \begin{array}{l} variant \in sim\_aprior\_dom(sim): \\ ti\_values = ti\_mean\_crit\_values(sim, variant, total\_time), \\ total\_time = sim\_total\_time(sim), \\ \left(\begin{array}{l} \neg\exists dom\_variant \in sim\_aprior\_dom(sim); \\ ti\_mean\_crit\_values(sim, dom\_variant, total\_time) \geq ti\_values \end{array} \right), \\ \left(\begin{array}{l} \forall i \in 1,..,sim\_crit\_num(sim); \\ ti\_values(i) < \infty \end{array} \right) \end{array} \right\} \right)$$

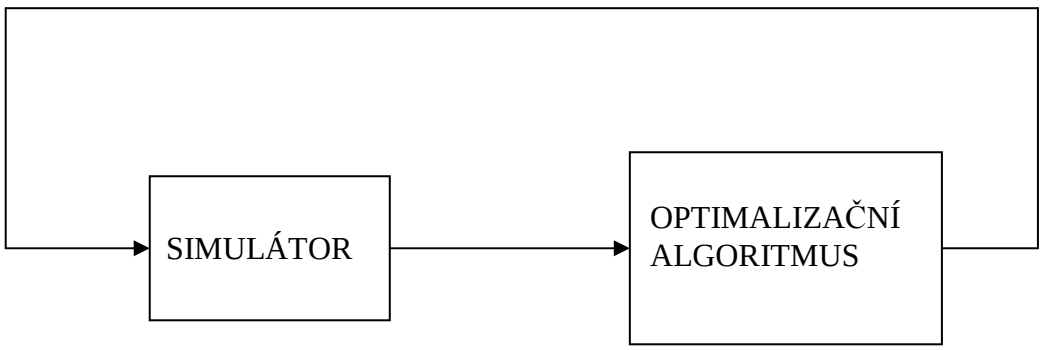
$$DOM(sim\_glob\_max) = SIMS$$

$$\forall sim \in DOM(sim\_glob\_max);$$

$$\left(sim\_glob\_max(sim) = \left\{ \begin{array}{l} ti\_vector \in \mathbb{R}^{sim\_crit\_num(sim)}: \\ total\_time = sim\_total\_time(sim), \\ \left(\begin{array}{l} \exists opt\_variant \in sim\_opt\_sol(sim); \\ ti\_vector = ti\_mean\_crit\_values(sim, opt\_variant, total\_time) \end{array} \right) \end{array} \right\} \right)$$

4.2.3 Problém optimalizace simulátoru

Analogicky s MOP lze problém optimalizace daného simulátoru definovat jako problém hledání globálního maxima tohoto simulátoru.



Obrázek 4.2: Optimalizační algoritmus simulátoru využívá průběžně hodnot výstupních proměnných optimalizovaného simulátoru, podle nichž následně řídí vlastní strategii hledání optimálního řešení

5 Optimalizátor PAROPMULTI

Kapitola dopodrobna popisuje optimalizátor PAROPTMULTI, který byl vyvinut RNDr. Jiřím Weinbergerem, CSc., a jehož výkonné jádro tvoří algoritmus tzv. alternujících heuristik. Popsat tuto metodu optimalizace v její celé úplnosti jak z matematického, tak z programátorského hlediska, dalece přesahuje vymezení této práce a ani k tomu neexistuje dobrý důvod. Proto je její popis redukován na jisté teoretické minimum, nutné pro pochopení jejích základních principů. Úplný popis metody PAROPTMULTI nalezne čtenář v [4].

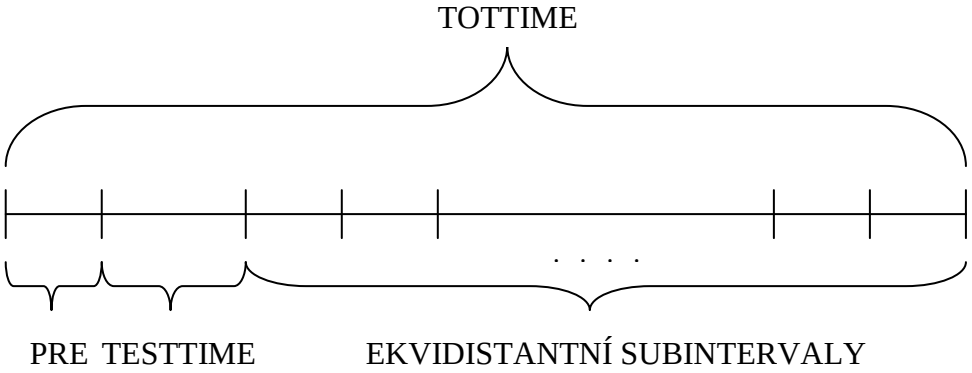
5.1 Dekompozice simulárního času

Dekompozice simulárního času je jeho jisté rozdělení na vzájemně disjunktní podintervaly. Podle potřeby lze samozřejmě provést libovolnou dekompozici, avšak oba optimalizátory shodně definují význam jednotlivých podintervalů a částečně také relace poměru jejich délek.

První podinterval tvoří tzv. **náběhového období**, které je charakteristické tím, že se v něm simulovaný systém chová většinou jinak než za „plného provozu“. Např. v systému hromadné obsluhy je taková situace charakteristická tím, že fronty jsou prázdné, zařízení neobsazena (za předpokladu nulového výchozího stavu simulátoru) atd. Vliv náběhového období může tedy značně ovlivnit reprezentativnost výsledků simulačních experimentů, a proto se jej oba optimalizátory snaží co nejvíce potlačit.

Po náběhovém období následuje **nulté období**. Zbytek je rozdělen na ekvidistantní (stejně velké) časové podintervaly. Nulté období je zpravidla delší než jeden ekvidistantní podinterval.

Simultánně prováděné výpočty jisté množiny variant simulátoru se v hraničních bodech podintervalů simulárního času pozastavují a po provedení konkrétní činnosti opět pokračují ve výpočtu. V okamžicích pozastavení simultánně počítaných variant dochází k rozhodnutí o vyřazení „slabých variant“, místo nichž jsou automatizovaně vytvářeny varianty nové. Rozhodnutí, na základě něhož je jistá varianta shledána vzhledem k ostatním variantám jako „slabá“, logicky vyplývá z porovnání vektorových kritérií. To, jakým způsobem se vytvářejí nové varianty, je plně v kompetenci použitého optimalizačního algoritmu.



Obrázek 5.1: Dekompozice simulárního času

5.2 Výkonné jádro

Jádro optimalizačního algoritmu je tvořeno heuristickým algoritmem, založeném na principu alternujících heuristik. Používá **adaptivní strategii** nasazování jistých elementárních vyhledávacích procedur, která je řízena na základě průběžného vyhodnocování jejich užitečnosti v té které fázi výpočtu.

5.2.1 Transformace hodnot parametrů

Výkonné jádro optimalizátoru PAROPTMULTI interně pracuje na N-rozměrné jednotkové krychli, tedy v $C_N = \langle 0,1 \rangle^N$ (N je počet prakticky interpretovatelných parametrů). Uživatel optimalizátoru si definuje (nejčastěji lineární) zobrazení g_i, $\forall i \in 1,..,N; g_i : \langle 0,1 \rangle \to D_i$ (D_i je doména hodnot i-tého parametru). Hledání extrémů pak probíhá nezávisle na řešeném problému v C_N, přičemž hodnoty, které se vztahují k původnímu problému, jsou získávány pomocí jednotlivých zobrazení g_i.

Extremalizační algoritmus může být tedy vyvíjen bez ohledu na to, na co bude později aplikován.

5.2.2 Schéma algoritmu

Předpoklady

1. $N > 1$
2. $\forall i \in 1,..,N; D_i$ je libovolná neprázdná
3. $D \equiv D_1 \times D_2 ... \times D_N$
4. $D_0 \subseteq D$, D_0 je množina, na které je f definováno
5. $f : D_0 \to \mathbb{R}$
6. Lze rozhodnout, zda pro libovolné $d \in D$ platí, že $d \in D_0$.
7. Hodnoty f lze na D_0 vyčíslit.
8. f není výsledkem nějakého čistě náhodného ohodnocení bodů množiny D_0.
   Má lokální i globální zákonitosti, byť eventuálně s četnými poruchami.
   Poruchou je zde míněna nespojitost, ostrá změna atd.

Průběh

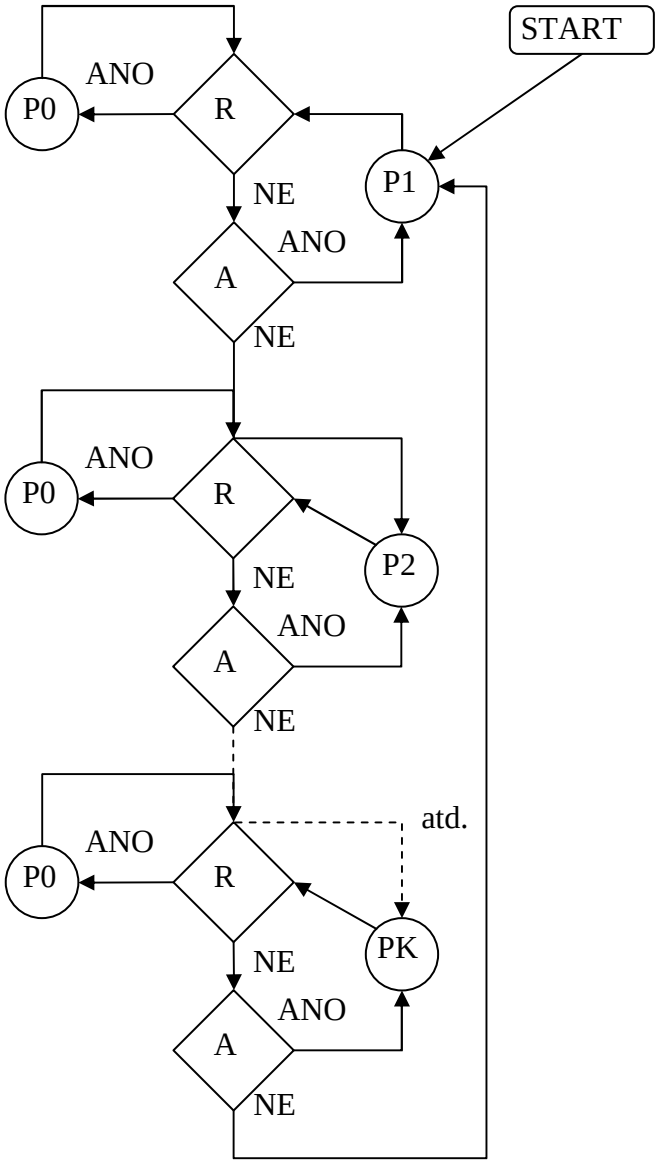
1. Je libovolně stanoveno M bodů množiny D.
2. Pro tyto body jsou stanoveny hodnoty zobrazení f.
3. Na základě veškeré dostupné informace, tj. celkem M dvojic tvaru $\langle x, f(x) \rangle$ (řešení) je vybranou elementární procedurou stanoven další nový bod z množiny D včetně hodnoty zobrazení f v tomto bodě.
4. Z aktuálního souboru řešení je vyřazeno takové, jehož hodnota je v porovnání s ostatními nejhorší (např. minimální, pokud hledáme maximum). Máme tedy opět M řešení.
5. Pokud není nalezeno řešení požadované kvality, skok na 2.
6. Konec.

Pokud je některé vygenerované řešení nepřípustné, je automaticky vygenerováno řešení jiné. Číslo M je voleno intuitivně, řádově 10^1 až 10^2.

5.2.3 Adaptivní strategie výběru elementární procedury

Algoritmus definuje celkem $K, K > 1$ elementárních vyhledávacích procedur, označených jako $P1, P2, ...PK$. Procedura $P0$ nechť je definována jako: ”zrcadlové překlopení” druhého “nejlepšího” sledovaného bodu z D kolem “nejlepšího” sledovaného bodu. Dále jsou definovány 2 booleovské procedury RECORD a AGAIN. Procedura RECORD nabývá hodnoty *true* právě tehdy, když naposled stanovený bod je ze sledovaných “nejlepší”. Procedura AGAIN nabývá hodnoty *true*, jestliže právě vyřazovaný bod nebyl stanoven elementární vyhledávací procedurou, která byla použita v bezprostředně předcházejícím kroku.

Výhoda je v tom, že tyto procedury si může každý definovat sám a není tedy nutné setrvávat u jedné, neměnné sady procedur. Totéž platí o modifikacích samotné adaptivní strategie. Lze téměř s jistotou tvrdit, že rezervy jsou zde skutečně značné.



Obrázek 5.2: Adaptivní strategie výběru elementárních vyhledávacích procedur. V každém bloku <R> je ještě test na ukončení celého výpočtu.

5.3 Celkové schéma optimalizátoru

Optimalizátor udržuje v průběhu výpočtu množinu tzv. **živých variant**, která se obecně mění. Živá varianta je taková varianta, která nebyla vyřazena a jejíž výpočet nebyl ukončen. Varianty, které se ukázaly být z hlediska vektorového kritéria jako méně kvalitní, jsou z výpočtu vyřazovány a nahrazovány variantami kvalitnějšími. Pro vzájemné porovnání kvality dvou různých variant se používají metody, jejichž princip bude nyní podrobněji popsán.

5.3.1 Porovnávání kvality variant

Dominovanost variant

Pokud je varianta A **dominovaná** variantou B téhož simulátoru, znamená to, že vektor kritérií varianty A je dominován – ve smyslu Pareto dominovanosti, vektorem kritérií varianty B.

Pomocné skalární kritérium

Toto kritérium není určeno k výběru nejlepších variant, ale k vyřazování těch variant, které s největší pravděpodobností mezi nejlepší varianty nepatří. Integrální součástí tohoto kritéria jsou váhy složek vektorového kritéria, které může zadat uživatel optimalizátoru. Pokud nejsou hodnoty těchto vah explicitně zadány, je s nimi dále počítáno, jako by byly rovné hodnotě 1.

Hodnota pomocného skalárního kritéria jisté varianty z :

$$POM\_CRIT(z) = \sum_{i=1}^{m} dom(z,i) * w_i \ ,$$

kde:

$dom(z,i)$ …počet živých variant, které varianta z dominuje v *i*-té složce kritéria

w_i …váha *i*-tého kritéria

$\forall i \in 1,..,m; w_i \in \mathbb{R}^+$

5.3.2 Experimentální studie

Vstupní parametry

1. **V –** násada generátoru pseudonáhodných čísel (NGPČ), jež slouží heuristickému vyhledávacímu algoritmu
2. **NPA –** počet volitelných parametrů pro daný problém
3. **PRE –** doba náběhu simulátoru
4. **TESTTIME –** nulté, zpravidla nejdelší období, které je součástí disjunktního rozkladu celého simulovaného období na subintervaly
5. **TOTTIME** – celkové simulované období (včetně doby náběhu)
6. **NTT** – počet ekvidistantních subintervalů, na které je rozděleno období $\langle PRE + TESTTIME, TOTTIME \rangle$
7. **LISTLIMIT –** počet simultánně počítaných variant simulátoru
8. **N_CRIT –** počet složek vektorového kritéria definovaného na simulátoru
9. **INTERACTIVE –** true nebo *false* – při interaktivním nebo plně automatické práci

Průběh výpočtu

1. Je zadána výchozí hodnota NGPČ, kterou budou mít všechny varianty stejnou; nechť je uložena v proměnné *SEED _ START* .
2. Uživatel si může vytvořit expertní varianty a zadat je do optimalizátoru. Počet celkově zadaných expertních variant je shora omezen hodnotou *LISTLIMIT*.
3. Počet expertních variant je doplněn na hodnotu *LISTLIMIT* – je použit mechanismus, který vytváří varianty automatizovaně pseudonáhodným výběrem.
4. Kontrola, zda každá z variant splňuje apriorní podmínky. U varianty, kde tomu tak, není, se tato nahradí novou variantou. Pokud splňuje všech *LISTLIMIT* variant apriorní podmínky řešení, potom výpočet dospěl do stavu, kterému se říká: „byla vytvořena vedoucí skupina variant“
5. Ve všech variantách vedoucí skupiny se rozběhnou simulární děje a zastaví se v simulárních časech: *TT(0) = PRE + TESTTIME*

Poznámka platná pro celý výpočet: Pokud v nějaké ze simultánně počítaných variant dojde k nepřípustnému stavu, je tato varianta z výpočtu okamžitě vyřazena a nahrazena novou, automatizovaně vytvořenou variantou.

6. Pro všechny varianty vedoucí skupiny variant jsou stanoveny hodnoty vektorového kritéria.
7. Pokud je některá živá varianta dominována jinou živou variantou, je tato varianta z výpočtu odstraněna. Místo ní je vygenerována varianta nová, která je počítána až do času *TT(0)*. Pokud je nová varianta z výpočtu odstraněna též, pokus o doplnění živých variant pokračuje tak dlouho, dokud se nezdaří. Případně je výpočet po jistém počtu nezdařených pokusů pozastaven a experimentátor požádán o změnu řídících parametrů experimentu. Pro všechny živé varianty, které se podaří zdárně dopočítat do času *TT(0)* jsou podle bodu 6 stanoveny hodnoty vektorů kritérií.
8. Pro vektory kritérií všech živých variant je vypočtena hodnota pomocného skalárního kritéria a všechny živé varianty jsou podle tohoto kritéria seřazeny do nerostoucí posloupnosti. Na základě této posloupnosti je vygenerována nová varianta a rozběhne se její výpočet, který je pozastaven také v simulárním čase *TT(0)*. Všechny dominované varianty jsou postupně z výpočtu odstraněny – viz bod 7.
9. Pro vektory kritérií všech živých variant je opět vypočtena hodnota pomocného skalárního kritéria, podle něhož jsou varianty seřazeny do nerostoucí posloupnosti. Poslední varianta je poté z výpočtu odstraněna.
10. Opakuje se výpočetní cyklus popsaný v bodech 5 – 9 s tím, že roli okamžiku *TT(0)* hrají postupně okamžiky *TT(1), TT(2),..TT(NTT)*.

Po skončení výše uvedené experimentální studie má uživatel k dispozici seznam *LISTLIMIT* vzájemně nedominovaných variant simulátoru, jejichž výpočty dospěly až do času *TOTTIME*. Ke každé z těchto variant je zároveň k dispozici i vektor hodnot kritérií.

Uživatel má možnost provést i replikovanou experimentální studii. Ta se bude lišit od předchozí studie hodnotou *SEED _ START* , může se však také lišit volbou počátečních řešení, což bývají v praxi často koncové varianty předchozí studie. PAROPTMULTI má

pro replikované experimentální studie implementovánu přímou podporu, takže lze pohodlně celý předchozí postup iterovat.

6 Návrh optimalizační metody

Kapitola popisuje návrh optimalizační metody. Součástí je popis, analýza a zdůvodnění výběru různých alternativ řešení, které se vztahují k různým aspektům fungování optimalizační metody. Popsán bude hlavně podkladový genetický algoritmus a jeho spojení s dekompozicí simulárního času. Konečným výstupem bude potom kompletní popis navržené optimalizační metody.

6.1 Požadavky

Nejprve bude vhodné připomenout, které požadavky bude muset optimalizační metoda respektovat. Tyto požadavky lze rozdělit na 2 skupiny podle toho, zda byly či nebyly vyjádřeny explicitně v zásadách pro vypracování.

Explicitní požadavky:

1. Optimalizační metoda má být založena na principu genetických algoritmů.
2. Genetický algoritmus se má uplatňovat již za běhu optimalizovaných simulátorů, nikoliv až po dokončení jejich běhů.

Implicitní požadavky:

1. Domény všech volitelných parametrů optimalizovaných systémů jsou podmnožiny reálných čísel.
2. Optimalizační kritérium je obecně vícesložkové (vektor) a domény všech složek jsou podmnožiny reálných čísel.
3. Kvalita výsledků navrhované metody by měla být srovnatelná s kvalitou výsledků PAROPTMULTI.

6.2 Analýza 1. explicitního požadavku a návrh řešení

Z explicitního požadavku č. 1 vyplývá, že bude třeba jistým způsobem propojit varianty simulátoru s jedinci populace genetického algoritmu. Přímočaré a intuitivní řešení je prostě spojit každého jedince populace s jednou variantou ve vztahu vzájemně jednoznačného přiřazení. Pokud tedy vznikne požadavek na výpočet hodnot kritérií daného jedince, dá tento jedinec jakýsi „signál“ variantě, která je mu přiřazena, a tato varianta bude simulována po jistou dobu. V době simulace této varianty se pak budou průběžně aktualizovat hodnoty vektoru kritérií, které budou po přerušení/ukončení výpočtu s variantou k dispozici příslušnému – „volajícímu“ jedinci.

6.3 Analýza 2. explicitního požadavku a návrh řešení

Druhý explicitní požadavek implikuje nutnost provést dekompozici simulárního času. Výpočty jednotlivých variant simulátoru budou iterativně spouštěny a pozastavovány v bodech dekompozice simulárního času.

Vyvstává zde otázka:

> Jakým způsobem reprezentovat iterace genetického algoritmu v rámci dekompozice simulárního času?

Přirozeným řešením tohoto problému se jeví vyjít z PAROPTMULTI. Aktuální generace jedinců bude reprezentována také jakousi vedoucí skupinou variant, jejichž výpočty budou spouštěny a pozastavovány v jednotlivých bodech dekompozice. Každá iterace algoritmu proběhne tím způsobem, že se kvaziparalelně spustí výpočty všech variant vedoucí skupiny. Výpočty posléze dosáhnou následujícího bodu dekompozice, kde budou pozastaveny. Na základě hodnot kriteriálních vektorů jedinců, jednoznačně přiřazeným variantám vedoucí skupiny, vytvoří podkladový genetický algoritmus generaci potomků, která jistým způsobem nahradí předchozí generaci. Vyvstává další otázka, jak toto nahrazení vlastně provést. Rozumných alternativ řešení tohoto problému je více.

6.3.1 Alternativa 1

Nejjednodušší je vyjít opět z přímo z PAROPTMULTI, kde byla v typickém případě v každém bodě časové dekompozice vytvořena jedna nová varianta, která byla poté dopočítána do času vedoucí skupiny variant. V případě genetického algoritmu by úprava tohoto přístupu spočívala v tom, že by do času vedoucí skupiny variant byly dopočítány všechny varianty, příslušné generaci potomků. Poté by vzniklo sjednocení obou skupin variant. Z tohoto sjednocení variant by poté byla vybrána nová vedoucí skupina variant. Výběr nové vedoucí skupiny variant by závisel již pouze na genetickém algoritmu.

6.3.2 Alternativa 2

Výchozí princip je totožný jako u alternativy č. 1 s tím rozdílem, že z dopočítávání do času vedoucí skupiny variant budou vyjmuty varianty takových potomků, kteří mají z hlediska vektorů integrálů středních hodnot kritérií nejhorší vlastnosti. Určit takové potomky lze pomocí nedominovaného třídění (viz odstavec 6.1.4) těchto vektorů. Varianty potomků by byly potom kvaziparalelně dopočítávány do simulárního času vedoucí skupiny variant. V každém bodu dekompozice by byl výpočet variant potomků pozastaven, potomci nedominovaně setříděni do front, a varianty, příslušné potomkům fronty s nejvyšším pořadovým číslem, definitivně vyřazeny z dalších výpočtů. Následně by byly výpočty zbylých variant opět spuštěny. V případě, kdy by vznikla pouze 1 fronta – vektory všech potomků by byly vzájemně nedominované, by se prostě dál pokračovalo ve výpočtu bez vyřazování jakékoliv varianty.

6.3.3 Diskuze alternativ řešení

Alternativa č. 1 je velmi přímočará, její implementace by byla snazší než u alternativy č. 2. Možnou nevýhodou je to, že do výpočtu jsou zahrnuty vždy všichni potomci včetně těch, jejichž vektory integrálů středních hodnoty kriterií jsou v porovnání s ostatními nejhorší. Tento problém se snaží řešit alternativa č. 2, která aktuálně nejhorší potomky z výpočtu pravidelně eliminuje. Alternativa č. 2 tedy nevylučuje možnost, aby byla do vedoucí skupiny variant dopočítána pouze 1 varianta, což je pravidlem v PAROPTMULTI.

Obecně lze tedy říci, že vzhledem ke spotřebě simulárního času, nebude alternativa č. 2 nikdy horší než alternativa č. 1, v některých případech bude naopak dosahovat mnohem lepších výsledků.

Podstatná nevýhoda alternativy č. 2 je to, že eliminace je prováděna s omezeným množstvím informace – zejména v počátečních bodech dekompozice. Zařazení týchž potomků do front se s každým bodem obecně mění, takže může docházet k eliminaci takových potomků, kteří by se v dalších bodech dekompozice, případně v rámci vedoucí skupiny variant, mohli ukázat jako nadprůměrní. Tento přístup by se tedy dobře uplatnil v situacích, kde by se zařazování potomků do jednotlivých nedominovaných front v bodech dekompozice měnilo pouze málo. GA_PAROPTMULTI má být ale obecně použitelný optimalizátor, bez explicitní deklarace aplikovatelnosti na konkrétní typy simulovaných systémů. Z tohoto důvodu a z důvodu snazší a přímočaré implementace byla zvolena alternativa č. 1.

6.4 Návrh podkladového genetického algoritmu

Oblast optimalizace multikriteriálních problémů se v současné době vyvíjí dosti rychle, a proto existuje již poměrně dost genetických algoritmů, které splňují první 2 implicitní požadavky. S ohledem na jednoduchost, výkonnost a kvalitu výsledných řešení byl vybrán NSGA-II. Tento algoritmus je podrobně rozepsán v [8], zde budou uvedeny pouze některé jeho aspekty.

NSGA-II neklade na zakódování, křížení a mutaci explicitně žádné požadavky. Zdroj [8] popisuje srovnávací studii, ve které byl problém kódován jak binárně, tak reálně. Naopak, NSGA-II specificky definuje výpočet fitness, operátor selekce a dosti se opírá o elitismus.

6.4.1 Výpočet hodnot vektorů kritérií

Vzhledem k tomu, jak je definováno maximum simulátoru, jeví se přirozené zvolit za kritérium, které bude genetický algoritmus optimalizovat, vektor hodnot časových integrálů kritérií optimalizovaného simulátoru, viz odstavec 4.2.1. Algoritmus bude tedy udržovat vlastně 2 různé vektory kritérií. Uživatelský, za jehož aktualizaci odpovídá sám uživatel optimalizační metody (což se děje v rámci výpočtu simulátoru), a vektor kritérií daného optimalizátoru, který odpovídá časovým integrálům uživatelských kritérií.

6.4.2 Kódování řešení do chromozomů

Výkonné jádro PAROPTMULTI pracuje v C_N. Výhody tohoto přístupu byly již zmíněny a není důvod neimplementovat jej také do podkladového genetického algoritmu. První implicitní požadavek není s tímto rozhodnutím v rozporu. Vzhledem k těmto skutečnostem se jeví jako přirozené zvolit **kódování reálnými hodnotami**. Toto kódování je obzvláště vhodné v těch případech optimalizace, kdy jsou rozhodovací proměnné (volitelné parametry) spojitého charakteru, což je v simulační optimalizaci běžný případ.

Každý gen reprezentuje jednu rozhodovací proměnnou zadaného problému. Vzhledem k tomu, že podkladový genetický algoritmus bude pracovat v C_N, je doména hodnot každého genu omezena na interval $\langle 0,1 \rangle$. Každý chromozom tedy představuje vektor reálných čísel, které tvoří souřadnice příslušného řešení v C_N. Pokud by byl stejný chromozom kódován binárně, musel by být pro každou složku vektoru řešení (ve stavovém prostoru) kvůli přesnosti vyhrazen dostatečný počet bitů, což by vedlo ke značnému nárůstu délky chromozomů, menší přehlednosti, větší pracnosti, tedy ke ztížené implementaci. Další výhodou reálného kódování je přesnější a rychlejší vyhledávání kvalitnějšího řešení na lokální úrovni. V případě binárního kódování je takovéto „odlaďování“ obtížné z důvodu Hamming cliff efektu.

6.4.3 Velikost populace

Velikost populace je závislá na povaze řešeného problému. Některé zdroje považují obecně za minimální přiměřenou velikost populace 20 až 30 jedinců, jiné 50 a více. V případě optimalizace simulátorů, kde k získání hodnoty účelové funkce je zapotřebí netriviální časové kvantum (minimálně v řádu vteřin), by použití populace o vysokém počtu jedinců nebylo únosné. Proto byla velikost populace v implementovaném algoritmu pevně nastavena na hodnotu 30. V dalším textu bude tato hodnota reprezentována identifikátorem *POP_SIZE*.

6.4.4 Křížení a mutace

Volba operátorů křížení a mutace přímo závisí na tom, jak je problém zakódován. Pro problémy kódované reálnými hodnotami existuje již řada operátorů křížení a mutace. V práci [5], v níž byl také proveden výzkum vlivu jednotlivých operátorů na kvalitu výsledného řešení a jeho vyhodnocení, se lze dočíst, že všeobecně nejlepší volbou byl operátor s názvem: BLX-α. Informace ohledně těchto testů jsou příliš rozsáhlé na to, aby zde byly všechny uvedeny, důležité však je toto:

1. V testu byl zařazen i algoritmus s binárním kódováním s dostatečnou přesností s dvoubodovým křížením. Tento algoritmus vykazoval v porovnání s ostatními obecně horší výsledky.
2. Algoritmy s BLX-α a neuniformní mutací dosáhly nejlepších výsledků.

Výše zmíněný výzkum přispěl nemalou měrou k tomu, že křížící operátor byl implementován jako BLX-α (α = 0.5) a mutační operátor byl implementován jako neuniformní.

Rozdělení intervalů hodnot genů

Předpoklady:

1. $m \in \mathbb{N}$ …délka chromosomu
2. $i \in 1,..,m$
3. $a_i, b_i \in \mathbb{R}$
4. $c_i^1, c_i^2 \in \langle a_i, b_i \rangle$ …hodnoty 2 genů, které se budou rekombinovat
5. $\alpha_i = \min\{c_i^1, c_i^2\},\ \beta_i = \max\{c_i^1, c_i^2\}$
6. $\alpha_i' \in \langle a_i, b_i \rangle \wedge \alpha_i' \leq \alpha_i$
7. $\beta_i' \in \langle a_i, b_i \rangle \wedge \beta_i' \geq \beta_i$

Potom tzv. **action interval** $\langle a_i, b_i \rangle$ lze rozdělit na 3 disjunktní podintervaly $\langle a_i, \alpha_i \rangle$, $\langle \alpha_i, \beta_i \rangle$, $\langle \beta_i, b_i \rangle$. V těchto podintervalech se bude nacházet výsledek křížení genů c_i^1, c_i^2.

Podinterval uprostřed - $\langle \alpha_i, \beta_i \rangle$, se nazývá **exploitation zone**. Dva uvedené postranní intervaly nazývají se **exploration zones**. Interval $\langle \alpha_i', \beta_i' \rangle$ se nazývá **relaxed exploitation zone**.

Každý křížící operátor má jistý stupeň **exploitation** a/nebo **exploration** podle toho, v jakých podintervalech budou vznikat nové geny. Výkon genetického algoritmu je silně závislý na stupni exploration a exploitation použitého operátoru křížení. Operátor **BLX-α**, který bude dále popsán, vytváří nové geny uniformně v relaxed exploitation zone.

Křížící operátor BLX-α

Nechť $C_1 = (c_1^1 ... c_n^1)$, $C_2 = (c_1^2 ... c_n^2)$ jsou 2 chromozomy, které byly vybrány operátorem selekce ke vzájemnému křížení. Potom potomek je:

$$H = (h_1, ..., h_i, ..., h_n),$$

kde:

h_i je náhodně vybrané číslo z rovnoměrného rozdělení pravděpodobnosti na intervalu:

$\langle c_{\min} - I * \alpha, c_{\max} + I * \alpha \rangle$

$c_{\max} = \max(c_i^1, c_i^2)$

$c_{\min} = \min(c_i^1, c_i^2)$

$I = c_{\max} - c_{\min}$

Efekt operátorů křížení reálně kódovaných genetických algoritmů

Efekt operátorů křížení lze studovat ze dvou úhlů pohledu, jak se lze dočíst v [5]: z pohledu genu a z pohledu chromozomu.

Z pohledu genu lze říci, že mnoho operátorů křížení používá exploitation zone, což se jeví jako přirozené, nicméně studie prováděné na operátoru BLX-α a jiných potvrdily, že použití relaxed exploitation zone je také vhodné. V případě absence selekčního tlaku, hodnoty $\alpha \leq 0.0$ budou způsobovat to, že daná populace bude mít tendenci konvergovat do středu svých intervalů hodnot, což může vyústit v malou různorodost této populace. Takto se může stát, že algoritmus bude předčasně konvergovat k neoptimálnímu řešení. V případě, že $\alpha = 0.5$, bude zajištěna rovnováha mezi konvergencí (exploitation) a divergencí (exploration), jelikož pravděpodobnost toho, že potomek bude ležet mimo interval hodnot svých rodičů, je stejná jako pravděpodobnost toho, že bude ležet v tomto intervalu.

Z pohledu chromozomu je efekt křížících operátorů zřejmý z geometrického hlediska. Předpokládejme, že X, Y jsou chromozomy ke vzájemnému křížení. Potom nechť H_{XY} označuje hyperkrychli definovanou X, Y. Operátor BLX-α potom bude moci generovat každý bod v hyperkrychli o něco větší než H_{XY}, ve které je H_{XY} obsažena.

Mutační neuniformní operátor

Nechť $C = (c_1, ..., c_i, ..., c_n)$ je chromozom a $c_i \in \langle a_i, b_i \rangle$ je gen, který má být mutován. Dále nechť c_i' je již zmutovaný gen, který vznikl aplikací operátoru mutace na gen c_i.

Jestliže je operátor aplikován v generaci t a $g_{\max}$ představuje maximální počet generací, potom:

$$c_i' = c_i + \Delta(t, b_i - c_i), \text{ pokud } \tau = 0$$
$$c_i' = c_i - \Delta(t, c_i - a_i), \text{ pokud } \tau = 1$$

kde:

τ je náhodně vybrané číslo z množiny $\{0,1\}$,

$$\Delta(t, y) = y\left(1 - r^{\left(1 - \frac{t}{g_{\max}}\right)^b}\right),$$

r je náhodně vybrané číslo z intervalu $\langle 0,1 \rangle$

b je parametr, jehož hodnota je zvolena uživatelem a který vyjadřuje míru závislosti na počtu iterací

Funkce $\Delta(t, y)$ dává hodnotu v intervalu $\langle 0, y \rangle$, takže pravděpodobnost toho, že vrátí číslo blíže k 0, se s postupem algoritmu zvyšuje. Velikost intervalu, ve kterém bude hodnota zmutovaného genu, se tedy bude postupem algoritmu zmenšovat. To ve svém důsledku způsobí, že na počátku algoritmu se budou zmutované chromozomy nacházet se stejnou pravděpodobností v celém prostoru volitelných parametrů problému, postupem času se však budou nacházet stále blíže ke svým původním nezmutovaným chromozomům.

6.4.5 Nondominated Sorting (Nedominované třídění)

Tento algoritmus je cele popsán v [8] a nemá smysl jej zde cele opisovat. V této stati bude naznačen pouze účel tohoto algoritmu. Algoritmus dostane jako svůj vstup množinu konečných vektorů reálných hodnot. Výstupem bude skupina tzv. **nedominovaných front**, která je setříděnou reprezentací vstupu podle relace Pareto Dominance.

Jednoduše řečeno, algoritmus ze vstupní množiny "oddělí" podmnožinu navzájem se nedominujících vektorů a vytvoří novou prázdnou nedominovanou frontu, do níž tuto podmnožinu umístí. Tento postup iteruje do té doby, než je vstupní množina prázdná. Každá fronta tedy obsahuje jistou podmnožinu vstupní množiny vektorů, přičemž platí, že žádné 2 vektory v téže frontě se navzájem nedominují. Každý vektor z fronty s vyšším pořadím je přitom dominován nějakým vektorem z fronty s pořadím nižším. Lze tedy říci, že pořadí fronty, do níž vektor patří, je jakýmsi indikátorem kvality (fitness) daného vektoru v porovnání s ostatními vektory.

Značení a předpoklady

$INPUT\_VECTORS$ …neprázdná vstupní množina konečných vektorů reálných čísel

K …počet vygenerovaných front

$FRONT_i$ …v pořadí i-tá vygenerovaná fronta

Platí

$K \in \mathbb{N}$

$$\bigcup_{i=1}^{K} FRONT_i = INPUT\_VECTORS$$

$$\forall i \in 1,..,K, \forall vec \in FRONT_i;$$
$$\left(\neg \exists vec\_dom \in FRONT_i; vec\_dom \geq vec\right)$$

$$\forall i, j \in 1,..,K, i < j, \forall vec \in FRONT_j;$$
$$\left(\exists vec\_dom \in FRONT_i; vec\_dom \geq vec\right)$$

6.4.6 Hustota v prostoru hodnot účelových funkcí

Tímto výpočtem je získán jakýsi odhad míry toho, jak je vektor hodnot účelových funkcí jistého jedince „natěsnán" k jiným takovým vektorům vstupní množiny jedinců v prostoru vektorů hodnot účelových funkcí. Míře takového „natěsnání" se v anglicky psaných zdrojích říká **crowding distance**. Je smysluplné, aby vstupní množina jedinců měla vzájemně nedominované vektory hodnot kritérií.

Značení a předpoklady

Z …vstupní množina jedinců, jejichž vektory kritérií jsou vzájemně nedominované

$z, z\_left, z\_right \in Z$

M …počet účelových funkcí

$f_i(z)$ …hodnota i-té účelové funkce jedince z

Pokud platí, že:

$$\left(\neg\exists a \in Z;\ f_i\left((z) - f_i(a)\right) < \left(f_i(z) - f_i(z\_left)\right)\right)$$

$$\wedge$$

$$\left(\neg\exists b \in Z;\ f_i\left((b) - f_i(z)\right) < \left(f_i(z\_right) - f_i(z)\right)\right)$$

potom:

$$adjDist(z,i) = f_i(z\_right) - f_i(z\_left)$$

jinak:

$$adjDist(z,i) = \infty$$

Hodnota crowding distance jedince z v kontextu množiny Z:

$$crowDist(z) = \sum_{i=1}^{M} adjDist(z,i)$$

6.4.7 Relace $\prec_n$

Ideou vytvoření tohoto operátoru byla snaha o co nejlepší rozprostření vektorů hodnot účelových funkcí výsledné množiny řešení na Pareto Front. Tento operátor je používán ve výběru nejlepších jedinců do následující generace. Ze dvou jedinců preferuje takového, který pochází z nedominované fronty s nižším pořadím. Pokud oba jedinci patří do téže fronty, pak preferuje toho, jehož vektor hodnot účelových funkcí se nachází v "řidší" lokalitě prostoru hodnot vektorů účelových funkcí.

Značení a předpoklady

i, j …jedinci populace

Předpokládejme, že všichni jedinci populace mají definovány 2 atributy:

1. *rank* …vyjadřuje pořadí fronty, ke které jedinec náleží
2. *dist* …hodnota crowding distance

$i \prec_n j$, jestliže platí: $\left(rank(i) < rank(j)\right) \vee \left(\left(rank(i) = rank(j)\right) \wedge \left(dist(i) > dist(j)\right)\right)$

6.4.8 Selekce

V procesu selekce jedinců, kteří vytvoří populaci potomků, hraje obvyklou roli hodnoty fitness jistého jedince hodnota *rank* spolu s hodnotou *crowding distance*. Algoritmus selekce je zde implementován jako turnaj 2 vybraných jedinců. Nejprve jsou ze stávající populace vybráni 2 jedinci. Každý jedinec je vybírán s rovnoměrným rozdělením pravděpodobnosti na celé populaci. Vítěz je vybrán na základě relace $\prec_n$. V případě, že pro oba jedince neplatí tato relace, je vítěz vybrán náhodně.

6.4.9 Hlavní cyklus výpočtu

Nejprve je náhodně vygenerována prvotní populace velikosti POP_SIZE. Ta je setříděna pomocí algoritmu nedominovaného třídění. Každému jedinci je přiřazena jeho hodnota $rank$, která je rovna pořadí fronty, do které tento jedinec náleží. Následnou aplikací operátorů selekce, křížení a mutace se vytvoří populace potomků velikosti POP_SIZE. Poté je populace rodičů sjednocena s populací potomků do tzv. kombinované populace, velikosti $2POP\_SIZE$. Tato kombinovaná populace je poté setříděna pomocí algoritmu nedominovaného třídění. V pořadí první nedominovaná fronta obsahuje nejlepší jedince v rámci kombinované populace, které je logické uchovat do následující iterace. Pokud velikost této fronty nepřesáhne počet „volných míst“ nové populace (která je v tomto případě rovna POP_SIZE), přejdou všichni její jedinci do nové populace a počet „volných míst“ nové populace se sníží o hodnotu velkosti přidané fronty. Stejný princip se postupně aplikuje na další fronty podle jejich pořadí. Pokud nastane situace, kdy je velikost fronty větší než počet „volných míst“, je tato fronta sestupně setříděna na základě relace $\prec_n$. Zbývající „volná místa“ jsou poté zaplněna prvními jedinci z této setříděné fronty. Nová populace je tak kompletní.

Značení a předpoklady

P_t …populace rodičů generace t

Q_t …populace potomků, vytvořená z P_t

$FRONT_i$ …v pořadí i-tá vygenerovaná nedominovaná fronta

Algoritmus

1. Polož: $R_t = P_t \cup Q_t$
2. Setřiď R_t do nedominovaných front pomocí Pareto Sorting
3. Polož: $P_{t+1} = \emptyset$, $i = 1$
4. Dokud platí: $|P_{t+1}| + |FRONT_i| \leq POP\_SIZE$, potom:
   a. Polož: $P_{t+1} = P_{t+1} \cup FRONT_i$
   b. Polož: $i = i + 1$
5. Setřiď $FRONT_i$ podle relace $\prec_n$ v sestupném pořadí
6. Vyber prvních $POP\_SIZE - |P_{t+1}|$ prvků z $FRONT_i$ a přidej je do P_{t+1}
7. Z P_{t+1} vytvoř novou populaci a přiřaď ji do Q_{t+1}
8. Polož: $t = t + 1$

6.4.10 Ukončující podmínky

Genetický algoritmus skončí, jakmile dosáhne zadaného počtu generací. Počet generací je roven celkovému počtu podintervalů dekompozice simulárního času. Tato dekompozice bude definována uživatelem pro daný problém. Neexistuje tedy žádný pevně daný počet generací.

6.4.11 Parametry genetického algoritmu

Hodnoty zbývajících volitelných parametrů podkladového genetického algoritmu budou určeny na základě takových hodnot, které jsou všeobecně uznávány jako přiměřené. V tomto případě se jedná konkrétně o pravděpodobnost křížení, pravděpodobnost mutace a parametr b. V této práci není prostor pro vypracování podrobné studie, která by se zabývala volbou hodnot konkrétních parametrů na přesně specifikované typy problémů, jejíž výsledky by posloužily jako jakýsi orientační bod.

Kódování ..... **reálné**
Velikost populace **30**
Pravděpodobnost křížení..... **1.0**
pravděpodobnost mutace **0.005**
Operátor selekce ..... **turnaj**
Operátor křížení ..... **BLX-α**
α **0.5**
Operátor mutace..... **neuniformní**
b **2**
Počet generací ..... **nepřímo dán uživatelem**

6.5 Celkové schéma optimalizátoru

Celkové schéma navržené optimalizační metody je podobné PAROPMULTI. V průběhu výpočtu bude udržována vedoucí skupina variant, ze které je v bodech dekompozice simulárního času pomocí genetického algoritmu vygenerována jiná skupina variant – skupina potomků. Poté se kvaziparalelně rozběhnou výpočty potomků a zastaví se v čase vedoucí skupiny variant. Ze sjednocení obou těchto skupin je potom podle hodnot časových integrálů kritérií vybrán jistý počet nejkvalitnějších variant, se kterými se bude ve výpočtu pracovat dále. Ostatní varianty ze sjednocení budou z dalších výpočtů definitivně vyřazeny.

6.5.1 Experimentální studie

Vstupní parametry

Vstupní parametry budou též podobné těm z PAROPTMULTI. Optimalizační metoda musí mít informace o počtu volitelných parametrů a počtu položek vektorového kritéria. Musí mít také informace o dekompozici simulárního času. Jelikož genetický algoritmus pracuje také stochasticky, musí být pro tyto účely zadána též násada PG. V tomto ohledu zde není s PAROPTMULTI žádný rozdíl, až na názvy samotných parametrů.

Není zde parametr s podobnou sémantikou jako LISTLIMIT. Je to logické, neboť uživatel algoritmu by měl být od principu fungování podkladového genetického algoritmu zcela odstíněn. Navíc, nevhodná volba velikosti populace by silně ovlivnila chování algoritmu.

Není zde ani parametr INTERACTIVE, neboť mechanismus interaktivní spolupráce uživatele v průběhu výpočtu nebyl navržen, a tedy zatím nebude ani implementován. Uživatel pouze zadá parametry, případně navrhne množinu počátečních řešení a potom už pouze čeká na výsledek.

1. **DECISIONS_NUMBER – počet** volitelných parametrů pro daný problém
2. **OBJECTIVES_NUMBER – počet** složek vektorového kritéria definovaného na simulátoru
3. **PRE** – doba náběhu simulátoru
4. **TESTTIME** – nulté, zpravidla nejdelší období, které je součástí disjunktního rozkladu celého simulovaného období na subintervaly
5. **TOTTIME – celkové** simulované období (včetně doby náběhu)
6. **NTT** – počet ekvidistantních subintervalů, na které je rozděleno období $\langle PRE + TESTTIME, TOTTIME \rangle$
7. **MAIN_RANDOM_SEED** – násada generátoru náhodných čísel (NGPČ), jež slouží genetickému algoritmu optimalizační metody
8. **MODEL_RANDOM_SEED** – náhodná složka všech variant experimentální studie

Průběh výpočtu

1. Jsou načteny parametry studie. Pokud v této fázi nastane chyba, je optimalizátor ukončen a vypíše na Standardní výstup hlášení o této chybě.
2. Uživatel si může vytvořit expertní varianty a zadat je do optimalizátoru. Počet celkově zadaných expertních variant je shora omezen velikostí populace genetického algoritmu – tj. hodnotou *POP_SIZE*.
3. Počet všech variant je doplněn na hodnotu *POP_SIZE* – je použit mechanismus, který vytváří varianty automatizovaně pseudonáhodným výběrem.
4. Kontrola, zda každá z variant splňuje apriorní podmínky. Varianta, kde tomu tak není, se nahradí variantou novou. Pokud splňuje všech *POP_SIZE* variant apriorní podmínky řešení, potom výpočet dospěl do stavu, kterému se říká: „byla vytvořena vedoucí skupina variant“.
5. Ve všech variantách vedoucí skupiny se rozběhnou simulární děje a zastaví se v simulárních časech: *TT(0) = PRE + TESTTIME*.

Poznámka platná pro celý výpočet: Pokud v nějaké ze simultánně počítaných variant dojde k nepřípustnému stavu, je tato varianta z výpočtu okamžitě vyřazena a nahrazena novou, automaticky vytvořenou variantou.

6. Pro všechny varianty vedoucí skupiny variant jsou stanoveny hodnoty vektorového kritéria.
7. Ze skupiny vedoucích variant je vygenerována nová skupina variant velikosti *POP_SIZE* – tzv. **skupina potomků**. Ve všech variantách skupiny potomků se rozběhnou simulární děje a zastaví se v simulárních časech *TT(0)*.
8. Vedoucí skupina variant je sjednocena se skupinou potomků. Z tohoto sjednocení je genetickým algoritmem vybrána skupina prvních *POP_SIZE* variant, která bude tvořit vedoucí skupinu variant v následující iteraci algoritmu.
9. Opakuje se výpočetní cyklus popsaný v bodech 5 – 9 s tím, že roli okamžiku *TT(0)* hrají postupně okamžiky *TT(1), TT(2),..TT(NTT)*.

Po skončení výše uvedené experimentální studie má uživatel k dispozici seznam *POP_SIZE* variant simulátoru, jejichž výpočty dospěly až do času *TOTTIME*. Na rozdíl od PAROPTMULTI je však třeba podotknout, že tyto varianty nemusí být z hlediska vektorů kritérií vzájemně nedominované.

7 Implementace optimalizační metody

Tato kapitola detailně popisuje implementaci optimalizační metody, navržené v předchozí kapitole, do optimalizátoru GA_PAROPTMULTI. Implementace je provedena v jazyce Simula 67. Soubory zdrojového kódu GA_PAROPTMULTI budou dodány na přiloženém CD včetně uživatelské dokumentace.

Nejprve je diskutována dekompozice optimalizátoru do jednotlivých modulů. Následuje podrobný popis implementace těchto modulů včetně popisu jejich vzájemné spolupráce.

7.1 Rozdělení na hlavní moduly

Již z návrhu optimalizační metody intuitivně vyplývá, že podkladový genetický algoritmus by bylo vhodné vyčlenit do samostatného modulu. Tento modul nechť je implementován ve formě hlavní třídy (i s případnými interními třídami), jejíž název je *GA_Algorithm*. Zbytek optimalizátoru, který bude využívat služeb *GA_Algorithm*, nechť je implementován modulem, jehož hlavní třída nechť má název *GA_Paroptmulti*.

Otázkou je, jaký typ vyčlenění *GA_Algorithm* zvolit. V zásadě se nabízí 3 alternativy: dědičnost, kompozice a kombinace obou předchozích.

7.1.1 Alternativa 1 – Dědičnost

GA_Algorithm bude nadtřídou *GA_Paroptmulti*. Zde existuje několik podalternativ, které se vzájemně liší tím, v jaké části *GA_Algorithm* by byl umístěn hlavní cyklus genetického algoritmu.

Pokud by byl umístěn přímo v životních pravidlech, potom by nezbývalo nic jiného, než implementovat do životních pravidel *GA_Algorithm* a jeho podtřídy synchronizační mechanismus, který by zajišťoval správnou posloupnost prolínání příkazů životních pravidel obou tříd. Vyžadovalo by to použití návěští (i virtuálních).

V případě, že by byl hlavní cyklus umístěn do procedury, potom by se musel pro *GA_Algorithm* zajišťovat konzistentní stav mezi každými dvěma voláními této procedury. Nebylo by třeba volat procedury *detach*, *call*, *resume* – implementace by se přiblížila způsobu řešení téhož problému v jazycích bez podpory kvaziparalelnosti.

7.1.2 Alternativa 2 – Kompozice

Objekt třídy *GA_Algorithm*, nebo nějaké její podtřídy, by byl položkou třídy *GA_Paroptmulti*.

V případě, že by byl hlavní cyklus implementován v životních pravidlech třídy *GA_Algorithm*, pak by každá iterace algoritmu proběhla na základě zavolání objektu této třídy prostřednictvím procedur *call* nebo *resume*. Po ukončení iterace by se objekt algoritmu opět odpojil pomocí procedury *detach*.

Pokud by byl hlavní cyklus implementován v proceduře, potom by byla každá iterace provedena na základě volání této procedury od objektu typu *GA_Algorithm*. Dále by samozřejmě platilo to, co bylo popsáno pro 1. alternativu – a příslušnou podalternativu.

7.1.3 Alternativa 3 – Dědičnost a kompozice

Tento způsob je zvolen v PAROPTMULTI, kde je třída, implementující algoritmus jádra (*Source*), rodičovskou třídou zbytku optimalizátoru. Zbytek optimalizátoru tvoří pouze 1 hlavní třída (*Parmulti*), která má ještě několik interních tříd. Třída *Source* implementuje algoritmus jádra prostřednictvím své interní třídy *y_source*, která obsahuje hlavní výpočetní cyklus ve svých životních pravidlech. Objekt typu *y_source* je jednou z položek třídy *Source*. Třída *Parmulti* pak volá ve svých interních procedurách tento objekt prostřednictvím procedury *resume*.

Stejný princip lze samozřejmě aplikovat na GA_PAROPTMULTI. Jemná variace může spočívat v tom, že by *GA_Paroptmulti* volala objekt typu *GA_Algorithm* ve svých životních pravidlech. Další podalternativa by byla, jako v předchozích 2 případech, implementovat hlavní cyklus prostřednictvím procedury.

7.1.4 Diskuze alternativ

Kompozice vyjadřuje přesněji vztahy mezi podkladovým genetickým algoritmem a zbytkem optimalizátoru. Samotný optimalizátor GA_PAROPTMULTI NENÍ totiž genetickým algoritmem, nýbrž pouze POUŽÍVÁ genetický algoritmus – MÁ jej k dispozici. Zjevnou nevýhodou alternativy č. 1 je tedy fakt, že mezi oběma třídami není typ vztahu JE, a podtřída je takto donucena být něčím, čím ve skutečnosti není. To se mimo jiné projeví také tím, že značná část položek *GA_Algorithm* nebude mít v *GA_Paroptmulti* v principu žádný smysl (v horším případě jiný význam) – budou tam naprosto zbytečně. Navíc, implementovat zmiňovaný synchronizační mechanismus by bylo dosti komplikované. Tyto důvody dostatečně diskvalifikují alternativu č. 1 z možností implementace.

Vzhledem k nesprávnému typu vztahu obou tříd sdílí alternativa č. 3 stejné nevýhody jako alternativa č. 1.

Alternativa č. 2 implementuje skutečný vztah obou mezi *GA_Algorithm* a *GA_Paroptmulti*. Zbývá tedy zvolit způsob implementace hlavního cyklu. Žádná z obou podalternativ nevykazuje ve srovnání s tou druhou žádnou podstatnou výhodu nebo nevýhodu. Výběr je tedy z velké míry záležitostí subjektivní preference. Vzhledem k tomu, že kvaziparalelní zpracování je v Simule 67 na vysoké úrovni propracovanosti a vzhledem k tomu, umístění hlavního cyklu do životních pravidel je přece jenom o něco intuitivnější, bylo rozhodnuto zvolit tuto podalternativu.

7.2 Popis třídy *GA_Algorithm*

Třída *GA_Algorithm* implementuje podkladový genetický algoritmus optimalizátoru. Sama je podtřídou standardní simulovské třídy *Simset*, která implementuje práci se seznamy. Implementace samotné *GA_Algorithm* je již velice intuitivní a přímočará. Musí obsahovat entity, které mají přímý vztah ke genetickému algoritmu. Dále by měla obsahovat entity, které podporují běh a usnadňují implementaci samotnou, jako je např. systém výjimek, logování určitých informací apod. Nutnou součástí je také mechanismus propojení s *GA_Paroptmulti*.

7.2.1 Parametry třídy

Genetický algoritmus musí mít k dispozici určitě tyto informace: počet rozhodovacích proměnných, počet kritérií, maximální počet generací. Dalším parametrem bude násada PG, ze které se na počátku vygenerují další násady, které budou později využity hlavně genetickými operátory.

7.2.2 Entity genetického algoritmu

Konstantní hodnoty parametrů genetického algoritmu: velikost populace, pravděpodobnost křížení, pravděpodobnost mutace, hodnoty parametrů genetických operátorů.

Interní třída *Chromosome* implementuje, jak již sám název napovídá, chromozom. Je tvořen polem reálných čísel. Životní pravidla náhodně inicializují každou složku s uniformním rozdělením pravděpodobnosti na intervalu $\langle 0,1 \rangle$.

Interní třída *Individual* zapouzdřuje informace o určitém jedinci populace. Jedinec se vztahuje k jistému chromozomu, jehož reference je položkou třídy *Individual*. Každý jedinec má svůj unikátní celočíselný identifikátor (ID). Mezi další datové položky patří: pole kriteriálních hodnot, pořadí nedominované fronty, hodnota *crowding distance*, indexy rodičovských jedinců. Obsahuje pouze 1 procedurální položku, která implementuje výpočet kriteriálních hodnot. Tato procedura interně volá virtuální proceduru *object_function*, která je virtuální na úrovni hlavní třídy a bude podrobněji popsána později – jejímž skutečným parametrem je právě hodnota ID tohoto jedince. Životní pravidla tvoří 2 příkazy. První z nich vygeneruje novou, unikátní hodnotu ID a přiřadí ji do ID tohoto jedince. Druhý vytvoří nový objekt typu *Chromosome* a přiřadí jej do příslušné reference tohoto jedince.

Interní třída *Population* implementuje populaci jedinců. Její hlavní datovou položkou je pole odkazů na objekty typu *Individual*. Procedura *count_objective_functions* implementuje výpočet kriterií všech jedinců v populaci. Interně volá proceduru *count_indiv_obj*, která zapouzdřuje výpočet kriteriálních hodnot specifikovaného jedince a v případě neúspěchu generuje chybu. Kompletní popis problematiky výpočtu kriteriálních hodnot jedinců bude uveden později.

Obsahuje také množství procedur, které jsou odpovědné za vlastní údržbu populace, jako např. vkládání a odebírání jedinců, výpočet statistik kritérií.

Skupina procedur, které implementují genetické operátory, inicializují prvotní populaci a vytvářejí populaci potomků. Pro každý genetický operátor jsou většinou kromě verze, uvedené v návrhu, deklarovány ještě další verze, a to pro účely testování.

Interní třída *Nondominated_Front*, která implementuje nedominovanou frontu. Udržuje seznam jedinců, jejichž kriteriální vektory jsou vzájemně nedominované. Další datovou položkou je hodnota pořadí této fronty. Procedurální položky jsou určeny pro aktualizace seznamu jedinců a aktualizace pořadí fronty.

Procedury a konstanty, které implementují algoritmus nedominovaného třídění. Do této skupiny patří např. funkční procedura *dominate_relation*, která implementuje relaci dominance 2 vektorů, které jsou uvedeny jako parametry.

7.2.3 Výpočet kriteriálních hodnot jedinců

Při výpočtu kriteriálních hodnot jedinců je zapotřebí zpracovat situaci, kdy se simulační výpočet příslušné varianty jedince dostane do nepřípustného stavu. Informaci o tom, že se tak stalo, dává algoritmu přímo uživatel. Učiní tak nastavením příznaku *variant_sim_error* pomocí procedury *set_variant_sim_error*. Reakce *GA_Algorithm* na vznik této situace je závislá na tom, ve které fázi výpočtu tato situace nastala.

V inicializační fázi prvotní populace (procedura *init_population*) je výpočet kritérií nového jedince zapouzdřen procedurou *count_indiv_obj*. Ta spustí vlastní výpočet kritérií tohoto jedince pomocí jeho procedury *count_objective_functions*. Po ukončení běhu této procedury kontroluje příznak *variant_sim_error*. V případě, že není nastaven, procedura *count_indiv_obj* končí. V opačném případě je nastaven příznak *gen_new_indiv* a příznak *variant_sim_error* je vynulován. Je vygenerován nový jedinec, jehož výpočet kritérií je ihned spuštěn. Celý předchozí postup je iterován maximálně 10 krát. Pokud byl výpočet úspěšný, původní jedinec je v dané populaci nahrazen tím, jehož výpočet byl úspěšný a procedura *count_indiv_obj* končí. V opačném případě je vygenerována chyba *CountObjectives_Error*.

Ve fázi průběhu – procedura *Population.count_objective_functions*, je výpočet kritérií každého jedince aktuální populace také zapouzdřen procedurou *count_indiv_obj*. Pokud byli v průběhu výpočtů vygenerováni noví jedinci, je nastaven příznak *gen_new_indiv*. Noví jedinci nemají nastavené hodnoty proměnných *rank* a *crowding_distance*, přesněji řečeno, hodnoty těchto proměnných jsou rovny 0. Pro začlenění nových jedinců je proto zapotřebí provést opětovné setřídění do nedominovaných front a vypočíst hodnoty jejich *crowding_distance*. Celý tento proces implementuje procedura *resort_population*, která je vyvolána tehdy, když je nastaven příznak *gen_new_indiv*.

V fázi tvorby nového potomka, je nový potomek zrušen (a jeho případný sourozenec také). Následně jsou vybráni noví rodiče, z nichž se vytvoří nová dvojice potomků a začne další výpočet jejich kritérií. Celkem je učiněno maximálně 10 pokusů o výpočet. Jestliže jsou všechny pokusy neúspěšné, je vygenerována chyba *CountObjectives_Error*.

7.2.4 Logování

Logování je Standardní prostředek pro trvalý zápis informací, které jsou z nějakého důvodu pro uživatele důležité. Pro tyto účely byla v samostatném souboru deklarována třída *Log_File*. Třída má jeden textový parametr, který definuje název logového souboru. Postup práce s touto třídou je jednoduchý:

1. Nastavení délky řádky logového souboru – *set_line_length*
2. Otevření logového souboru pro zápis – start_log
3. Vlastní zápis informací do logu – procedury *log_<Základní datový typ>*
4. Ukončení zápisu a uzavření logu – *end_log*

Log_File definuje zápis pro všechny základní datové typy. Pro typ *real* jsou definovány procedury pro zápis čísel v pevné a v plovoucí desetinné čárce.

Interní třída *Algorithm_Log*, která je podtřídou *Log_File*, umožňuje zapisovat v průběhu algoritmu informace o aktuální populaci jedinců. Do přehledné tabulky jsou vypsány všechny hodnoty datových položek jedinců. Pod tabulkou jsou navíc zapsány statistické informace týkajících se hodnot kritérií. Možnost logování lze využít hlavně ve fázi testování a vylaďování genetického algoritmu.

7.2.5 Chyby a výjimky

Simula 67 nemá implementován žádný mechanismus zpracování chyb a výjimečných stavů. Pokud taková situace nastane, potom program prostě skončí běhovou chybou. Ve snaze implementovat alespoň jisté primitivní zpracování chyb, byla do samostatného souboru deklarována třída *Error*. Má jeden textový parametr, který popisuje vzniklou chybu. Dále má jednu interní datovou položku textového typu, jejíž hodnotou je typ chyby. Uživatel může ve své třídě popsat chyby pomocí jednotlivých podtříd třídy *Error*, přičemž definuje typ a popis chyby svými vlastními řetězci.

Do *GA_Algorithm* bylo zapracováno ošetření zatím tří různých chyb. Tyto chyby vznikají v jistých procedurách, kde je jejich přítomnost kontrolována běžnými programovými prostředky, které Simula 67 nabízí. V případě, že je přítomnost chyby potvrzena, je vygenerován objekt této chyby a ten je následně předán k tomu určené **catch-proceduře**, která provede vlastní zpracování chyby. Zpracování veškerých chyb, které budou dále popsány, povede k definitivnímu ukončení práce s objektem algoritmu. Sama přítomnost výskytu takového druhy chyby je indikována příznakem *error_occurred* a uživatel může jeho hodnotu testovat pomocí procedury *get_error_occurred*.

AddExpertVariant_Error vznikne v proceduře *add_expert_variant* tehdy, když uživatel přidá svou expertní variantu „příliš pozdě“ – viz životní pravidla – nebo když počet jím přidaných variant překročí hodnotu velikosti populace, tj. 30. Procedura *add_expert_variant* kontroluje tyto podmínky pomocí obyčejných *if* příkazů. V případě vzniku chyby je vytvořen nový objekt *AddExpertVariant_Error*, kterému je předán popis chyby podle dané situace. Následně je řízení předáno na speciální návěští procedury *add_expert_variant*, kde je objekt chyby předán proceduře *catch_AddExpertVariant_Error*. Tato procedura zjistí, zda je aktivní logování algoritmu. V kladném případě zapíše zprávu o chybě do logu, v opačném případě na Standardní výstup. Poté je nastaven chybový příznak *error_occurred*.

StartLog_Error vznikne tehdy, pokud uživatel zadá přání logovat průběh algoritmu „příliš pozdě“ – viz životní pravidla. Způsob zpracování chyby je analogické jako u předchozí chyby. Rozdíl je v tom, že nyní je objekt chyby předán proceduře *catch_StartLog_Error*, která vypíše zprávu o chybě na Standardní výstup.

CountObjectives_Error vznikne tehdy, když je překročen maximální počet pokusů o výpočet kriteriálních hodnot jedince. Tato chyba vzniká v procedurách *count_indiv_obj* a *generate_child_population*. Objekt této chyby je předán proceduře *catch_CountObjectives_Error*. Ta v případě, že je nastaven příznak logování, zapíše zprávu o chybě do logu. V opačném případě je zpráva vypsána na Standardní výstup. Poté je nastaven chybový příznak a výpočet je přesměrován do životních pravidel na návěstí, kde se eventuálně provede ukončení logování. Následně je výpočet životních pravidel definitivně ukončen.

7.2.6 Životní pravidla

Nejprve jsou vygenerovány násady PG, které budou používány procedurami genetických operátorů. Dále se vytvoří nový objekt populace a inicializují hodnoty příznakových proměnných, např. že počet expertních variant zadaných uživatelem je 0, že nevznikla chyba atd. Poté se provede *detach*, čímž se objekt odpojí. Pouze v tomto a žádném jiném mezidobí může uživatel zadat úmysl logovat a přidávat své expertní varianty.

Při následném připojení objektu k operačnímu řetězci se nejprve zkontroluje, zda není nastaven příznak chyby. Pokud je nastaven, potom se provádění pravidel přesune na svůj konec, kde je eventuálně ještě ukončeno logování – pokud byl nastaven příznak logování. V případě, že nenastala žádná chyba, inicializuje se prvotní populace a spočtou se její statistiky. Pokud je nastaven příznak logování, bude prvotní populace též zalogována. Poté se objekt pomocí *detach* opět odpojí.

Při opětovném připojení objektu se provede 1 iterace hlavního cyklu algoritmu. Na počátku každé iterace se přitom kontroluje, zda nedošlo k překročení počtu generací a zda není nastaven příznak chyby. Tělo cyklu začíná podmínkou, zda je aktuální populace prvotní generací. Pokud ano, výpočet ihned přejde na vygenerování populace potomků. V opačném případě se nejprve vypočtou kriteriální hodnoty všech jedinců v populaci a až poté se přejde ke generování potomků. Při výpočtu kriteriálních hodnot jedinců aktuální populace se skupina variant, příslušná této populaci, posune do následujícího bodu dekompozice simulárního času, kde je pozastavena. V rámci generování populace potomků potom simulační výpočty variant, příslušných těmto potomkům, probíhají až do dosažení hodnoty simulárního času variant, příslušných aktuální populaci jedinců. Zbytek iterace proběhne podle popisu v odstavci 6.4.9. V případě, že je nastaven příznak logování, se nově vytvořená populace zaloguje. Na konci každé iterace se objekt algoritmu pomocí *detach* odpojí.

Po ukončení provádění cyklu se v případě nastaveného příznaku ukončí logování a provádění životních pravidel je definitivně ukončeno.

7.2.7 Propojení s *GA_Paroptmulti*

Hlavní propojovací entitou je virtuální procedura *object_function*, která má tyto parametry:

1. Hodnoty rozhodovacích proměnných volajícího jedince, což jsou hodnoty allel jeho referencovaného chromozomu – pole reálných hodnot
2. Hodnoty kritérií volajícího jedince – pole reálných hodnot
3. Hodnota ID volajícího jedince – celé číslo

Jak již bylo výše napsáno, tuto proceduru volá jedinec nepřímo prostřednictvím své procedury *count_objective_functions*. Je zodpovědností uživatele, aby provedl výpočet s variantou, příslušnou danému ID, a aktualizoval hodnoty kritérií. Pokud se výpočet příslušné varianty ukáže jako nepřípustný, potom uživatel volá proceduru *set_variant_sim_error*, která nastaví příznak nepřípustného výpočtu.

Uživatel by měl též průběžně testovat chybový příznak *error_occurred*. K tomu slouží procedura *get_error_occurred*, která vrací hodnotu tohoto příznaku.

7.2.8 Veřejné rozhraní

get_population_size

- vrací velikost populace jedinců

add_expert_variant(variant)

- přidá expertem definovanou variantu do populace
- parametrem je přidávaná varianta, což je pole reálných hodnot
- generuje chybu typu *AddExpertVariant_Error*

start_log(file_name)

- uživatel dává voláním této procedury najevo úmysl logovat průběh algoritmu
- parametrem je název logového souboru
- generuje chybu typu *StartLog_Error*

get_error_occurred

- vrací *true*, pokud došlo za běhu algoritmu k chybě, která znemožňuje jeho další běh, jinak vrací *false*

is_variant_alive(variant_ID)

- vrací *true*, pokud aktuální populace obsahuje jedince, jehož ID je rovné hodnotě parametru. V opačném případě vrací *false*.
- parametr je ID jedince

set_variant_sim_error

- voláním této procedury nastaví uživatel příznak, který říká, že výpočet varianty, která je příslušná jedinci, od něhož byla naposled vyvolána procedura *count_objective_functions*, skončil chybou

7.3 Popis třídy GA_Paroptmulti

Třída *GA_Paroptmulti* je realizuje zapouzdření celého optimalizačního procesu. Sama je podtřídou standardní simulovské třídy *Simset*, která implementuje práci se seznamy. Tuto třídu, nebo nějakou její podtřídu, bude používat přímo uživatel optimalizátoru GA_PAROPTMULTI.

GA_Paroptmulti musí obsahovat prostředky pro práci s objektem typu *GA_Algorithm*. Nutná je podpora dekompozice simulárního času. Další nezbytnou součástí je podpora vedoucí skupiny variant, která vyžaduje mapování jedinců genetického algoritmu na varianty simulátoru.

Pro interakce s optimalizovanými simulátory bude nutné zavést procedury, jejichž přesný význam bude deklarován uživatelem. Jde především o transformaci hodnot rozhodovacích proměnných, se kterými pracuje podkladový genetický algoritmus, na hodnoty prakticky interpretovatelných (volitelných) parametrů simulátoru. Uživatel musí též deklarovat apriorní omezení na hodnoty volitelných parametrů simulátoru.

Stejně jako v případě *GA_Algorithm*, měla by *GA_Paroptmulti* též obsahovat další podpůrné prostředky, jako systém výjimek a logování.

7.3.1 Varianty simulátoru, třídy *Expert* a seznamy expertů

Interní třída *Expert* implementuje experta, který simuluje jistou variantu. Má jeden parametr typu pole reálných hodnot, které představuje hodnoty volitelných parametrů simulátoru. Hodnoty volitelných parametrů nastavuje sám optimalizátor a nebudou se po celou dobu existence experta měnit – uživatel je může pouze číst. Protože *GA_Paroptmulti* pracuje se seznamy expertů, je tato třída podtřídou třídy *link*. *GA_Paroptmulti* implementuje simultánní výpočty variant prostřednictvím kvaziparalelního výpočtu objektů typu *Expert*. Uživatel *GA_Paroptmulti* vytvoří

podtřídu třídy *Expert*, v níž si nadeklaruje kompletní výpočet optimalizovaného simulátoru. Musí však postupovat s ohledem na dekompozici simulárního času – přesný postup bude uveden později.

Mezi nejdůležitější datové položky třídy *Expert* patří tyto:

1. hodnoty volitelných parametrů simulátoru
2. hodnoty uživatelských kritérií
3. příznak, zda nastal v průběhu výpočtu varianty ilegální stav
4. jednoznačný identifikátor experta
5. časové integrály uživatelských kritérií

Privátní procedurální položky mají na starost aktualizaci hodnot časových integrálů uživatelských kritérií.

Na počátku vykonávání životních pravidel je objekt vložen do seznamu aktuální populace expertů. Poté je vygenerován jednoznačný ID, který je tomuto expertovi přiřazen. Posledním příkazem je *detach*.

Veřejné rozhraní třídy Expert

objectives_profits

- hodnoty uživatelských kritérií

illegal_state

- příznak, zda nastal v průběhu výpočtu varianty ilegální stav

get_ID

- vrací ID experta

get_parameters(parameters)

- vrací v parametru hodnoty volitelných parametrů varianty
- parametrem je pole nových hodnot, kam se zapíší hodnoty volitelných parametrů

get_time_integral(obj_index)

- vrací hodnotu časového integrálu kritéria
- parametr je index kritéria

print_parameters

- vypisuje hodnoty volitelných parametrů na Standardní výstup

Interní třída *Experts_List*, která je podtřídou třídy *head*, implementuje seznamy objektů třídy *Expert* včetně výpočtu statistik kritérií a jejich časových integrálů. Tyto seznamy implementují skupiny variant simulátoru.

7.3.2 Propojení s GA_Algorithm

Jak již bylo uvedeno, každý jedinec genetického algoritmu vyvolá v rámci výpočtu svých hodnot kritérií výpočet s jemu jednoznačně přiřazenou variantou simulátoru. Mapování jedince na experta, který příslušnou variantu simuluje, je implementováno pomocí jednoduchých hašovacích tabulek.

Interní třída *HT_Int* implementuje hašovací tabulku. Má jeden celočíselný parametr, který vyjadřuje počet klíčů tabulky. Tato tabulka na jednotlivých klíčích udržuje seznamy celočíselných párů typu <klíč,hodnota>. Množina hodnot, ze které pochází klíče párů, je mapována na klíče tabulky. Je-li třeba přidat nový pár, je z 1. hodnoty v páru vypočtena hodnota klíče tabulky. Nový pár je potom přidán do seznamu příslušnému tomuto klíči. Při získávání 2. hodnoty páru je nejprve vstupní klíč páru namapován na klíč tabulky, pomocí kterého je získán příslušný seznam. Z tohoto seznamu je vrácena taková hodnota, jejíž klíč je roven vstupnímu klíči. Při odebírání jsou z příslušného seznamu odebrány všechny páry, jejichž hodnota je totožná s hodnotou vstupního páru.

V *GA_Paroptmulti* existují 2 hašovací tabulky. Tabulka *variant_to_expert_map* mapuje ID jedinců na ID expertů a tabulka *expert_to_variant_map* mapuje naopak ID expertů na ID jedinců.

Interní třída *Expert_Algorithm* je podtřídou třídy *GA_Algorithm* a deklaruje proceduru *object_function*, která je v *GA_Algorithm* virtuální. Po tom, co je vyvolána tato procedura, jsou hodnoty rozhodovacích proměnných – jejich hodnoty jsou v poli jednoho z parametrů této procedury – transformovány prostřednictvím procedury *transform* na hodnoty volitelných parametrů simulátoru. Transformované hodnoty jsou zkontrolovány na apriorní přípustnost procedurou *constraints*. Pokud daná varianta není apriorně přípustná, potom je zavolána procedura *set_variant_sim_error* a výpočet *object_function* je ukončen. Procedury *transform* a *constraints* jsou virtuální na úrovni třídy *GA_Paroptmulti* a je na uživateli, aby deklaroval jejich přesnou funkčnost.

V případě, že je varianta apriorně přípustná, je učiněn pokus o získání experta, který je příslušný jedinci, který *object_function* vyvolal. To je implementováno pomocí hašovací tabulky, která mapuje ID jedinců na ID expertů. Klíčem je zde jednoduše ID volajícího jedince. Pokud je vrácena korektní hodnota ID experta, je podle tohoto ID nalezen odpovídající expert a spuštěn výpočet jeho varianty po jistou dobu. Poté je výpočet varianty pozastaven a hodnoty kriterií volajícího jedince jsou aktualizovány hodnotami časových integrálů uživatelských kritérií experta, který danou variantu simuloval. Tímto je výpočet *object_function* ukončen.

Jestliže hašovací tabulka vrátí hodnotu oznamující, že příslušný expert neexistuje, je pomocí procedury *get_new_expert* dynamicky vytvořen nový expert. Tato procedura je na úrovni *GA_Paroptmulti* virtuální a její přesnou funkčnost deklaruje uživatel. Hodnoty volitelných parametrů varianty nového experta jsou nastaveny podle transformovaných hodnot rozhodovacích proměnných volajícího jedince. Následně je proveden výpočet varianty až do aktuální hodnoty simulárního času a aktualizovány hodnoty kritérií volajícího jedince. Poté jsou ještě aktualizovány obě hašovací tabulky.

Pokud se stane, že se výpočet varianty dostane do nepřípustného stavu, je na uživateli, aby nastavil příznak *illegal_state* příslušného experta. Tento příznak je v obou případech kontrolován bezprostředně po přerušení výpočtu varianty. Pokud je příznak nastaven, je opět vyvolána procedura *set_variant_sim_error* a výpočet *object_function* je ukončen.

7.3.3 Logování

Interní třída *Paroptmulti_Log*, která je podtřídou *Log_File*, umožňuje zapisovat v průběhu optimalizace informace o aktuální skupině expertů.

Pro každý interval dekompozice simulárního času je zapsána skupina expertů. Pro lepší přehlednost jsou vytvořeny vždy 3 tabulky – volitelné parametry, uživatelská

kritéria a časové integrály uživatelských kritérií. Pod těmito tabulkami jsou zapsány statistické informace hodnot časových integrálů uživatelských kritérií.
Hlavním důvodem, proč použít logování je fakt, že je to jediný prostředek, jak získat informace o množině výsledných řešení, která je logicky uvedena na konci logu – uživatel tedy nemusí procházet celý soubor, který obsahuje velmi podrobné informace o průběhu celé studie. Stejně jako v případě *GA_Algorithm*, logování lze také využít ve fázi testování a vylaďování optimalizátoru.

7.3.4 Konfigurační soubor a jeho zpracování

Optimalizátor definuje řadu parametrů, které musí uživatel nejprve nastavit. Za účelem experimentování s optimalizátorem vzniká potřeba persistentního ukládání a opětovného načítání konfiguračních dat. Na rozdíl od PAROPTMULTI, kde uživatel definuje parametry optimalizátoru prostřednictvím parametrů třídy *Parmulti*, jsou parametry optimalizátoru GA_PAROPTMULTI načítány z konfiguračního souboru, jehož název je parametrem třídy *GA_Paroptmulti*. Pro podporu práce s konfiguračními soubory byla do samostatného souboru deklarována třída *Config_File*.

Třída *Config_File* umožňuje načítání hodnot uživatelem definovaných parametrů z textového konfiguračního souboru, který je textovým parametrem třídy. Konfigurační soubor musí mít jistý formát, který definuje *Config_File*. Podrobný popis všech pravidel, která je nutno dodržet při tvorbě konfiguračního souboru, je uveden v uživatelské dokumentaci.
Základní pracovní proces je následující:

1. Přidání definic parametrů, které se budou ze souboru načítat – *add_parameter(param_name, param_type)*
2. Čtení konfiguračního souboru – read_config_file
3. Vracení načtených hodnot definovaných parametrů – get_<TYPE>_param_value*(param_name)*

Typ parametru může být kterýkoliv základní datový typ, kromě *char*. Uživatel může do konfiguračního souboru přidávat i komentáře – přesná syntaxe je uvedena v uživatelské dokumentaci.

Pokud dojde v průběhu prvních 2 bodů k chybě, jejíž přítomnost je indikována chybovým příznakem, nelze již s objektem dále pracovat – resp. uživatel by neměl s objektem dále pracovat, neboť jeho chování je nedefinované.

Vzhledem k tomu, že *Config_File* používá Standardní simulovské procedury pro konverzi textových řetězců na cílový datový typ, není možné chyby, které vzniknou v průběhu konverze, zpracovat uvnitř třídy a nastavit chybový příznak. V takovém případě program prostě spadne.

Pro načítání parametrů experimentální studie deklaruje třída *GA_Paroptmulti* proceduru *read_config_parameters*, která je vyvolána na počátku životních pravidel. Pakliže dojde v průběhu této procedury k chybě, je prostřednictvím zpracování chyby nastaven chybový příznak a výpočet životních pravidel *GA_Paroptmulti* je ukončen.

Pokud proběhlo načítání bez chyb, jsou načtené hodnoty parametrů zkontrolovány, zda neobsahují prakticky nesmyslné hodnoty – např. zápornou hodnotu celkového simulárního času apod. V případě, že se u některého z parametrů objeví takováto nesmyslná hodnota, je opět nastaven chybový příznak a výpočet životních pravidel *GA_Paroptmulti* je ukončen.

7.3.5 Chyby a výjimky

Mechanismus zpracování chyb a výjimečných stavů je totožný s tím, který byl uveden v popisu *GA_Algorithm*. Třída *GA_Paroptmulti* deklaruje též svůj chybový příznak *error_occurred*, který by měl uživatel optimalizátoru pomocí procedury *get_error_occurred* průběžně testovat. Jsou zde zpracovány celkem 3 typy chyb.

AddExpertVariant_Error má téměř totožnou sémantiku jako stejnojmenná chyba v *GA_Algorithm*. Rozdíl je v tom, že objekt podkladového genetického algoritmu *gen_alg*, jehož typ je *Expert_Algorithm*, není uživateli optimalizátoru přístupný. Pokud uživatel přidá expertní variantu „příliš pozdě“ – viz životní pravidla, je vygenerována tato chyba. Objekt této chyby je předán proceduře *catch_AddExpertVariant_Error*. Tato procedura zjistí, zda je nastaven příznak logování optimalizační studie. V kladném případě zapíše zprávu o chybě do logu, v opačném případě na Standardní výstup. Poté nastaví chybový příznak. V případě, že uživatel přidá svou variantu v časovém mezidobí pro to určeném, je na objektu *gen_alg* vyvolána procedura *add_expert_variant*. V případě, že dojde k chybě uvnitř tohoto objektu, je výsledkem zpracování této chyby nastavení chybového příznaku. Chybový příznak objektu algoritmu je v proceduře *add_expert_variant* také testován a pokud je skutečně nastaven, potom je nastaven též chybový příznak v *GA_Paroptmulti*.

StartLog_Error má také téměř totožnou sémantiku jako stejnojmenná chyba v *GA_Algorithm*. Vznikne tehdy, pokud uživatel zadá přání logovat průběh optimalizační studie „příliš pozdě“ – viz životní pravidla. Výsledkem je opět nastavení chybového příznaku.

ReadConfigParameters_Error vznikne tehdy, když dojde v procesu načítání konfiguračního souboru k chybě. Objekt chyby je předán proceduře *catch_ReadConfigParameters_Error*, která vypíše zprávu o chybě na Standardní výstup, nastaví chybový příznak a provede odskok na konec životních pravidel *GA_Paroptmulti*.

7.3.6 Životní pravidla

Životní pravidla začínají načtením parametrů experimentální studie a kontrolou jejich načtených hodnot – viz odstavec konfigurační soubor a jeho zpracování. Pokud dojde v této fázi chybě, je nastaven chybový příznak a výpočet životních pravidel je definitivně ukončen.

V případě, že načtení a kontrola parametrů proběhla bezchybně, pokračuje výpočet nastavením hodnoty délky ekvidistantního podintervalu dekompozice simulárního času – **časového kroku**, která je uložena do proměnné *time_step_length*. Poté je vytvořen objekt podkladového genetického algoritmu, jehož reference je uložena do proměnné *gen_alg*. Následuje kontrola chybového příznaku *gen_alg* a pokud je nastaven, je též nastaven chybový příznak *GA_Paroptmulti* a výpočet životních pravidel je ukončen.

Pokud v předchozí fázi nedošlo k chybě, pokračuje výpočet vytvořením seznamu aktuálních expertů – zatím prázdného a vytvořením hašovacích tabulek *variant_to_expert_map* a *expert_to_variant_map*. Poté je provádění životních pravidel přerušeno pomocí příkazu *inner*, který provede odskok na výpočet životních pravidel podtřídy třídy *GA_Paroptmulti*. Tuto podtřídu nebo blok prefixovaný třídou *GA_Paroptmulti*, deklaruje uživatel optimalizátoru, který může v rámci životních pravidel této podtřídy nastavit logování průběhu experimentální studie – pomocí *start_log*, nebo přidávat své expertní varianty – pomocí *add_expert_variant*. Po dokončení výpočtu životních pravidel uživatelovy podtřídy je zkontrolován chybový

příznak. Pokud je nastaven, je po eventuálním ukončení logování výpočet životních pravidel ukončen.

Výpočet pokračuje inicializací prvotní populace jedinců podkladového genetického algoritmu. To je provedeno voláním procedury *run_GA*, která nejprve pomocí Standardní procedury *call* volá objekt *gen_alg* a poté zkontroluje chybový příznak téhož objektu. Pokud je nastaven, je nastaven též chybový příznak *GA_Paroptmulti* a výpočet životních pravidel je po eventuálním ukončení logování ukončen. V případě úspěšné inicializace prvotní populace jedinců je volána procedura *update_current_experts*, která z aktuální populace expertů odstraní takové, jejichž mapovaní jedinci se již nenachází v aktuální populaci podkladového genetického algoritmu – viz procedura *is_variant_alive*. Následuje vygenerování potomků z aktuální populace jedinců – opět pomocí procedur *run_GA* a *update_current_experts*. Pokud je nastaven příznak logování, vypočtou se statistiky hodnot časových integrálů kritérií a aktuální populace expertů je zalogována.

Čítač časových kroků je nastaven na hodnotu 1 a výpočet životních pravidel vstoupí do *while* cyklu. Na počátku každé iterace se přitom kontroluje, zda nedošlo k překročení maximální hodnoty počtu časových kroků a zda není nastaven příznak chyby. Tělo cyklu nejprve provede volání procedur *run_GA* a *update_current_experts*. V případě, že je nastaven příznak logování, je aktuální populace expertů zalogována. Posledním příkazem těla cyklu je inkrementace čítače časových kroků.

Po ukončení cyklu je eventuálně ukončeno logování a výpočet životních pravidel *GA_Paroptmulti* je definitivně ukončen.

7.3.7 Veřejné rozhraní

get_config_file_name

- vrací název konfiguračního souboru včetně úplné cesty

get_decisions_num

- vrací počet volitelných parametrů simulátoru

get_objectives_num

- vrací počet uživatelských kritérií

get_pre

- vrací délku náběhového období

get_testtime

- vrací délku prvního intervalu, zpravidla nejdelšího

get_tottime

- vrací hodnotu celkového simulárního času

get_ntt

- vrací počet ekvidistantních podintervalů časové dekompozice, na které je rozděleno období $\langle PRE + TESTTIME, TOTTIME \rangle$

get_ model_random_seed

- vrací násadu PG, která bude použita v optimalizovaném simulátoru

get_time_step_length

- vrací délku jednoho ekvidistantního časového intervalu

add_expert_variant(variant)

- umožňuje uživateli přidat do populace jedinců svou expertní variantu
- parametrem je pole reálných hodnot rozhodovacích proměnných podkladového genetického algoritmu
- generuje chybu typu *AddExpertVariant_Error*

start_log(file_name)

- umožňuje uživateli nastavit logování průběhu experimentální studie
- parametrem je název logového souboru
- generuje chybu typu *StartLog_Error*

get_error_occurred

- vrací hodnotu chybového příznaku

8 Testování a porovnávání optimalizátorů na uměle vytvořených instancích

Náplní této kapitoly je testování a porovnání obou optimalizátorů na uměle vytvořených instancích. Kapitolu lze rozdělit do dvou částí.

První část začíná zdůrazněním výhod uměle vytvořených instancí ve smyslu možnosti uživatelské volby jejich klíčových, uživatelem definovaných vlastností. Dále pokračuje přesným vymezením a formalizací pojmů, na jejichž základě je na konci vystavěn celý systém testovacích tříd.

Druhá část se zabývá vlastním testováním a porovnáváním kvality výsledných řešení obou optimalizátorů. Podrobně popisuje vlastní testovací proces včetně definic metrik kvalit. Konec je tvořen tabulkami výsledků a interpretací jejich hodnot.

8.1 Umělé testovací instance a jejich vlastnosti

V předchozích kapitolách byly představeny 2 různé optimalizátory simulovaných systémů. Logicky následuje jednoduchá otázka: Který optimalizátor je lepší? Odpověď však jednoduchá není, a to nejenom proto, že je nutné zavést jisté formalizace.

Optimalizátory se dají porovnávat na různých simulátorech různých simulovaných systémů. Pokud není explicitně určena jakási třída simulovaných systémů, na jaké je daný optimalizátor speciálně konstruován, lze předpokládat použití takového optimalizátoru obecně na jakýkoliv smysluplný problém, což je také případ obou popisovaných optimalizátorů. Jak tedy otestovat optimalizátory na všeobecnou použitelnost? Existuje jakási sada problémů, která se k tomuto účelu používá a která se v oblasti simulační optimalizace považuje za standart? Takovou sadu se mi najít nepodařilo, avšak ohlasy po její potřebě již zaznamenat lze. Rozhodl jsem se proto vytvořit si vlastní sadu testovacích instancí, které budou mít jisté, experimentátorem požadované vlastnosti.

8.1.1 Zobrazení volitelných parametrů na vektory hodnot kritérií

Základním prvkem struktury testovacích instancí je zobrazení – dále uvedených vlastností, prostoru hodnot volitelných parametrů na prostor hodnot vektorů kritérií. Takovéto zobrazení – prvek množiny $MAPS$, bude dále v textu označováno slovem: „mapování“. Doména hodnot každého volitelného parametru je interval $\langle 0,1 \rangle$, prostor hodnot všech volitelných parametrů je tedy n-rozměrná jednotková krychle, kde n je počet volitelných parametrů.

Doména každého volitelného parametru je rozdělena na disjunktní ekvidistantní podintervaly. Celý prostor parametrů je rozdělen na navzájem disjunktní krychle stejné dimenze (podprostory), jejichž délka hrany je rovna právě délce výše zmiňovaného disjunktního ekvidistantního podintervalu. Pojmem podprostor bude dále v textu implicitně myšlen podprostor volitelných parametrů, dokud nebude řečeno jinak. Pro všechny body patřící do jednoho podprostoru platí, že jejich vektory kritérií mají totožné hodnoty. Vektory hodnot kritérií patří do množiny konečných vektorů reálných hodnot – u stejného zobrazení mají stejný počet složek. Dvě mapování jsou stejného

typu právě tehdy, když mají totožnou množinu podprostorů a stejný počet položek vektorového kritéria. Dvě mapování mají totožnou množinu podprostorů právě tehdy, když mají stejný počet volitelných parametrů a doména hodnot volitelných parametrů je rozdělena na stejný počet podintervalů.

Každý podprostor má přidělen jednoznačný identifikátor, což je přirozené číslo z intervalu: 1..*počet podprostorů*. Na tento identifikátor bude v následujících textu odkazováno zkratkou: ID. Pro všechna mapování stejného typu platí, že zobrazení, která přidělují jejich podprostorům jednoznačné identifikátory, jsou totožná.

Definice

$MAPS$ …množina všech mapování použitých v testovacích instancích

$SUBSPACES$ ...množina všech podprostorů množiny $MAPS$

$REAL\_VECTORS$ …množina konečných vektorů reálných hodnot

$CN\_VECTORS$ ...množina konečných reálných vektorů, jejichž hodnoty jsou omezeny na interval $\langle 0,1 \rangle$

$map\_dec\_num : MAPS \rightarrow \mathbb{N}$ ...zobrazení, které pro dané mapování vrací počet jeho volitelných parametrů

$map\_obj\_num : MAPS \rightarrow \mathbb{N}$ ...zobrazení, které pro dané mapování vrací počet jeho kritérií

$map\_subint\_num : MAPS \rightarrow \mathbb{N}$ ...zobrazení, které pro dané mapování vrací počet podintervalů hodnot každého z jeho volitelných parametrů

$map\_sub\_num : MAPS \rightarrow \mathbb{N}$ ...zobrazení, které pro dané mapování vrací počet jeho podprostorů

$map\_get\_obj\_values : MAPS \times CN\_VECTORS \rightarrow REAL\_VECTORS$
...zobrazení, které pro dané mapování a vektor volitelných parametrů vrací vektor jeho kriteriálních hodnot

$map\_get\_sub\_values :$

$$MAPS \times \left\{ \begin{array}{l} sub\_id : \\ sub\_id \in 1..map\_sub\_num(map) \end{array} \right\} \rightarrow REAL\_VECTORS$$

…zobrazení, které pro dané mapování a ID podprostoru vrací vektor jeho kriteriálních hodnot

$map\_equal\_type : MAPS \times MAPS \rightarrow \{true, false\}$ …zobrazení, které pro 2 daná mapování vrací logickou hodnotu podle toho, zda jsou obě stejného typu

$map\_subs : MAPS \rightarrow POW(SUBSPACES)$ …zobrazení, které pro dané mapování vrací množinu jeho všech podprostorů

$$map\_get\_sub\_ID : \left\{ \begin{array}{l} (map, sub): \\ map \in MAPS, \\ sub \in map\_subs(map) \end{array} \right\} \to \mathbb{N}$$

…zobrazení, které pro dané mapování a jeho podprostor vrací ID tohoto podprostoru

Axiomy

$$\forall map \in MAPS;$$
$$\left(map\_subint\_num(map) = e^{\ln(map\_subs\_num(map))/map\_dec\_num(map)} \right)$$

$$\forall map \in MAPS;$$
$$\left(map\_sub\_num(map) = \left| map\_subs(map) \right| \right)$$

$$\forall (map1, map2) \in DOM(map\_equal\_type);$$
$$\left(\begin{array}{l} \left(\begin{array}{l} map\_subs(map1) = map\_subs(map2) \wedge \\ map\_obj\_num(map1) = map\_obj\_num(map2) \end{array} \right) \\ \Leftrightarrow \\ \left(map\_equal\_type(map1, map2) = true \right) \\ \\ \end{array} \right)$$

$$\forall map1, map2 \in MAPS;$$
$$\left(\begin{array}{l} \left(map\_subs(map1) = map\_subs(map2) \right) \Leftrightarrow \\ \left(\begin{array}{l} map\_dec\_num(map1) = map\_dec\_num(map2) \wedge \\ map\_subint\_num(map1) = map\_subint\_num(map2) \end{array} \right) \end{array} \right)$$

$$\forall map \in MAPS;$$
$$\left(\begin{array}{l} \forall sub \in map\_subs(map); \\ map\_get\_sub\_ID(map, sub) \in 1, .., \left| map\_subs(map) \right| \end{array} \right)$$

$$\forall map \in MAPS, \forall sub1, sub2 \in map\_subs(map), sub1 \neq sub2;$$
$$\left(map\_get\_sub\_ID(map, sub1) \neq map\_get\_sub\_ID(map, sub2) \right)$$

$$\forall map1, map2 \in MAPS, map\_equal\_type(map1, map2) = true;$$
$$\left(\begin{array}{l} \forall sub \in map\_subs(map1); \\ map\_get\_sub\_ID(map1, sub) = map\_get\_sub\_ID(map2, sub) \end{array} \right)$$

8.1.2 Dělení prostoru volitelných parametrů

Každé mapování definuje počet ekvidistantních podintervalů intervalu $\langle 0,1 \rangle$. Tímto jsou rovněž definovány hraniční hodnoty těchto podintervalů, včetně hodnot 0 a 1. Výchozí body jednotlivých podprostorů, které jsou n-rozměrnými krychlemi, jsou vektory, jejichž počet položek je rovný počtu volitelných parametrů – nechť je tato hodnota rovna *n*. Složky těchto vektorů tvoří všechny kombinace hraničních hodnot podintervalů – kromě hodnoty 1. Každý podprostor je pak kartézským součinem právě *n* podintervalů tvaru $\langle low\_bound(i), low\_bound(i) + subint\_length \rangle$, kde dolní mez je i-tou položkou vektoru výchozího bodu krychle tohoto podprostoru a *subint_length* je délkou jednoho ekvidistantního podintervalu volitelných parametrů.

Definice:

$DIV\_VECTORS$ …vektory dělících hodnot intervalu $\langle 0,1 \rangle$ - konečné rostoucí posloupnosti reálných hodnot z $\langle 0,1 \rangle$, každé 2 sousední hodnoty mají stejnou vzdálenost

$map\_get\_div : MAPS \rightarrow DIV\_VECTORS$ …zobrazení, které pro dané mapování vrací vektor dělících bodů intervalu hodnot jeho volitelných parametrů

$map\_cubes\_points : MAPS \rightarrow POW(CN\_VECTORS)$ …zobrazení, které pro dané mapování vrací výchozí body krychlí podprostorů jeho prostoru volitelných parametrů

$map\_get\_subspace$:

$$MAP \times \left\{ \begin{array}{l} cn\_vector: \\ cn\_vector \in CN\_VECTORS, \\ |cn\_vector| = map\_dec\_num(map), \\ \left(\begin{array}{l} \forall i \in 1,..,|cn\_vector|, \\ subint\_length = 1 / map\_subint\_num(map); \\ (1 - cn\_vector(i)) \geq subint\_length \end{array} \right) \end{array} \right\} \rightarrow SUBSPACES$$

…zobrazení, které pro dané mapování a výchozí bod uvnitř prostoru volitelných parametrů vrací podprostor

Axiomy:

$\forall map \in DOM(map\_get\_div), div\_vector = map\_get\_div(map);$
$\left(|div\_vector| = map\_subint\_num(map) + 1 \right)$

$\forall map \in DOM(map\_get\_div), div\_vector = map\_get\_div(map);$
$\left(div\_vector(1) = 0 \wedge div\_vector(|div\_vector|) = 1 \right)$

Každé dvě hraniční hodnoty jednotlivých podintervalů, které tvoří hranice jednoho podintervalu, mají stejnou vzájemnou vzdálenost.

$$\forall map \in DOM(map\_get\_div), div\_vector = map\_get\_div(map);$$

$$\begin{pmatrix} \forall i \in 1,..,|div\_vector|-1, subint\_lentgh = 1/map\_subint\_num(map); \\ (div\_vector(i+1) - div\_vector(i)) = subint\_length \end{pmatrix}$$

$$\forall map \in DOM(map\_cubes\_points);$$

$$map\_cubes\_points(map) = \left\{ \begin{matrix} cn\_vector: \\ cn\_vector \in CN\_VECTORS, \\ |cn\_vector| = map\_dec\_num(map), \\ \begin{pmatrix} \forall i \in 1,..,|cn\_vector|, \\ div\_vector = map\_div\_vector(map), \\ \exists j \in 1,..,|div\_vector|-1; \\ cn\_vector(i) = div\_vector(j) \end{pmatrix} \end{matrix} \right\}$$

$$\forall (map, cn\_vector) \in DOM(map\_get\_subspace),$$

$$subint\_length = 1/map\_subint\_num(map);$$

$$\begin{pmatrix} map\_get\_subspace(map, cn\_vector) = \\ \langle cn\_vector(1), cn\_vector(1) + subint\_length \rangle \times \\ \langle cn\_vector(2), cn\_vector(2) + subint\_length \rangle \times \\ \vdots \\ \times \langle cn\_vector(|cn\_vector|), cn\_vector(|cn\_vector|) + subint\_length \rangle \end{pmatrix}$$

$$\forall map \in MAPS, dec\_num = map\_dec\_num(map);$$

$$map\_subs(map) = \left\{ \begin{matrix} map\_get\_subspace(map, point): \\ point \in map\_cubes\_points(map) \end{matrix} \right\}$$

8.1.3 Testovací instance

Testovací instance je konečný vektor dvojic typu: <simulární čas, mapování>. Všechna mapování jsou stejného typu. Pro definovanou hodnotu simulárního času, pokud se nachází v nějaké dvojici, a pro definovaný vektor hodnot volitelných parametrů vrací vektor hodnot kritérií. Zobrazení **časových integrálů kritérií** je definováno pro jistý podprostor a simulární čas, jehož hodnota nechť je rovna *time*. Prostorem hodnot tohoto zobrazení jsou konečné reálné vektory hodnot časových integrálů kritérií. Integrál každého kritéria je definován na intervalu $\langle 0, time \rangle$.

Definice

$INSTANCES$ ...množina testovacích instancí

$TIME\_VALUES$ ...množina všech hodnot simulárního času

$TIME\_VECTORS$ …množina všech vektorů, jejichž složky tvoří možné hodnoty simulárního času. Hodnoty vektorů jsou konečné rostoucí posloupnosti nezáporných reálných čísel.

$inst\_dec\_num : INSTANCES \rightarrow \mathbb{N}$ ...zobrazení, které pro danou instanci vrací počet volitelných parametrů všech jejích mapování

$inst\_obj\_num : INSTANCES \rightarrow \mathbb{N}$ ...zobrazení, které pro danou instanci vrací počet kritérií všech jejích mapování

$inst\_sub\_num : INSTANCES \rightarrow \mathbb{N}$ ...zobrazení, které pro danou instanci vrací počet podprostorů všech jejích mapování

$inst\_times : INSTANCES \rightarrow TIME\_VECTORS$ …zobrazení, které pro danou instanci vrací vektor hodnot všech jejích simulárních časů (časový vektor)

$inst\_time\_exists : INSTANCES \times TIME\_VALUES \rightarrow \{true, false\}$ …zobrazení, které pro danou instanci a hodnotu simulárního času indikuje, zda je tato hodnota přítomna v časovém vektoru této instance

$$inst\_get\_map : \left\{ \begin{array}{l} (inst, time): \\ inst \in INSTANCES, \\ (inst\_time\_exists(inst, time) = true) \end{array} \right\} \rightarrow MAPS$$

…zobrazení, které pro danou instanci a hodnotu simulárního času vrací mapování

$$inst\_get\_obj\_values : \left\{ \begin{array}{l} (inst, time, sub\_id): \\ inst \in INSTANCES, \\ tinst\_time\_exists(inst, time) = true, \\ sub\_id \in 1,..,inst\_sub\_num(inst) \end{array} \right\} \rightarrow REAL\_VECTORS$$

…zobrazení, které pro danou instanci, hodnotu simulárního času a ID podprostoru vrací vektor kriteriálních hodnot

$$inst\_get\_ti\_values : \left\{ \begin{array}{l} (inst, sub\_id, time): \\ inst \in INSTANCES, \\ sub\_id \in 1,..,inst\_sub\_num(inst), \\ inst\_time\_exists(inst, time) = true \end{array} \right\} \rightarrow REAL\_VECTORS$$

…zobrazení, které pro danou instanci, ID podprostoru a hodnotu simulárního času vrací vektor časových integrálů kritérií

Axiomy

$$\forall time \in TIME\_VALUES; time \in \mathbb{R}^{+} \cup \{0\}$$

$$\forall inst \in INSTANCES;$$
$$\left(\begin{array}{l} \forall time\_id \in 1,..,|inst\_times(inst)|, \\ time = inst\_times(inst)(time\_id), \\ map = inst\_get\_map(time); \\ \left(\begin{array}{l} inst\_dec\_num(inst) = map\_dec\_num(map) \wedge \\ inst\_obj\_num(inst) = map\_obj\_num(map) \wedge \\ inst\_sub\_num(inst) = map\_sub\_num(map) \end{array}\right) \end{array}\right)$$

$$\forall (inst, time) \in DOM(inst\_time\_exists);$$
$$\left(\begin{array}{l} \\ (inst\_time\_exists(inst, time) = true) \\ \Leftrightarrow \\ \left(\begin{array}{l} times = inst\_times(inst), \exists i \in 1,..,|times|; \\ times(i) = time \end{array}\right) \end{array}\right)$$

$$\forall (inst, time, sub\_id) \in DOM(inst\_get\_obj\_values);$$
$$\left(\begin{array}{l} inst\_get\_obj\_values(inst, time, sub\_id) = \\ map\_get\_sub\_values(inst\_get\_map(inst, time), sub\_id) \end{array}\right)$$

$$\forall (inst, sub\_id, time) \in DOM(inst\_get\_ti\_values);$$
$$(|inst\_get\_ti\_values(inst, sub\_id, time)| = inst\_obj\_num(inst))$$

$$\forall (inst, sub\_id, time) \in DOM(inst\_get\_ti\_values),$$
$$ti\_values = inst\_get\_ti\_values(inst, sub\_id, time),$$
$$\forall obj\_id \in 1,..,|ti\_values|,$$
$$times = inst\_times(inst), time = times(time\_id);$$
$$\left(\begin{array}{l} ti\_values(obj\_id) = \\ \sum_{i=2}^{time\_id} \left\{\begin{array}{l} ti\_part: \\ ti\_part = (times(i) - times(i-1))\, obj\_values(obj\_id), \\ obj\_values = inst\_get\_obj\_values(inst, times(i-1), sub\_id) \end{array}\right\} \end{array}\right)$$

8.1.4 Relace v kontextu jedné testovací instance

Vzhledem k tomu, že všechny vektory volitelných parametrů, které náleží do stejného podprostoru, mají definován tentýž vektor kritérií, je logické dívat se na nedominované třídění vektorů kritérií všech podprostorů přeneseně jako přímo na „třídění podprostorů (podle nějakého kritéria)“. Vůbec není nutné omezovat se pouze na vektor kritérií, předchozí postup lze rozšířit obecně na jakékoliv zobrazení, jehož definičním oborem je množina podprostorů mapování a oborem hodnot množina konečných reálných vektorů. Vzhledem k povaze testovaných systémů, bylo jako vhodný kandidát vybráno zobrazení časových integrálů kritérií.

Termín **pořadí podprostoru** (v kontextu jisté instance, mapování a simulárního času samozřejmě) nechť je definován jako: pořadí nedominované fronty, do které náleží vektor časových integrálů kritérií tohoto podprostoru (vzhledem k dané instanci, mapování a simulárnímu času).

Diference podprostorů je definována jako absolutní hodnota rozdílu jejich pořadí v kontextu dvojice hodnot simulárních časů jisté instance. Tato hodnota má navíc smysl pouze u podprostorů, jejichž ID jsou totožná – totožných podprostorů. **Celková diference podprostorů** je definována na dvojici hodnot simulárních časů jisté instance. Její hodnota je rovna součtu jednotlivých diferencí všech podprostorů a je indikátorem jakési podobnosti obou mapování s ohledem na rozdělení vektorů časových integrálů kritérií do nedominovaných front.

Definice

$$inst\_front\_num : \left\{ \begin{array}{l} (inst, time): \\ inst \in INSTANCES, \\ inst\_time\_exists(inst, time) = true \end{array} \right\} \rightarrow \mathbb{N}$$

…zobrazení, které pro danou instanci a simulární čas vrací počet nedominovaných front podprostorů

$$inst\_get\_rank : \left\{ \begin{array}{l} (inst, time, sub\_id): \\ inst \in INSTANCES, \\ inst\_time\_exists(inst, time) = true, \\ sub\_id \in 1,..,inst\_sub\_num(inst) \end{array} \right\} \rightarrow \mathbb{N}$$

…zobrazení, které pro danou instanci, simulární čas a ID podprostoru vrací pořadí tohoto podprostoru

$$inst\_get\_rank\_diff : \left\{ \begin{array}{l} (inst, time1, time2, sub\_id): \\ inst \in INSTANCES, \\ inst\_time\_exists(inst, time1) = true, \\ inst\_time\_exists(inst, time2) = true, \\ sub\_id \in 1,..,inst\_sub\_num(inst) \end{array} \right\} \rightarrow \mathbb{N}$$

…zobrazení, které pro danou instanci, 2 hodnoty simulárního času a ID podprostoru, vrací diferenci podprostorů

$$inst\_get\_total\_diff : \left\{ \begin{array}{l} (inst, time1, time2): \\ inst \in INSTANCES, \\ inst\_time\_exists(inst, time1) = true, \\ inst\_time\_exists(inst, time2) = true \end{array} \right\} \rightarrow \mathbb{N}$$

…zobrazení, které pro danou instanci a 2 hodnoty simulárního času vrací celkovou diferenci podprostorů

Axiomy

$$\forall (inst, time1, time2, sub\_id) \in DOM(inst\_get\_rank\_diff);$$
$$\left(\begin{array}{l} inst\_get\_rank\_diff(inst, time1, time2, sub\_id) = \\ abs(inst\_get\_rank(inst, time1, sub\_id) - inst\_get\_rank(inst, time2, sub\_id)) \end{array} \right)$$

$$\forall (inst, time1, time2) \in DOM(inst\_get\_total\_diff);$$
$$\left(\begin{array}{l} inst\_get\_total\_diff(inst, time1, time2) = \\ \sum_{sub\_id=1}^{inst\_sub\_num(inst)} inst\_get\_rank\_diff(inst, time1, time2, sub\_id) \end{array} \right)$$

8.1.5 Třídy testovacích instancí

Každá třída testovacích instancí sdružuje instance, které mají předem definované společné vlastnosti. Instance, které patří do stejné třídy, musí mít totožné časové vektory a totožný typ mapování. Typ mapování a vektor hodnot simulárního času každé instance definuje třída. Přesněji řečeno: třída definuje vektor hodnot dekomponovaného simulárního času. Simulární čas je dekomponován na ekvidistantní podintervaly, přičemž délka těchto podintervalů je určena parametry té které třídy – hodnotou celkového simulárního času (*total_time*) a počtem těchto podintervalů. Výše uvedené podmínky lze považovat pouze za jakýsi základ k definici dalších podmínek – je zřejmé, že množina instancí, která tyto podmínky splňuje, může být z hlediska průběhu celkové diference podprostorů značně rozmanitá.

Třída zavádí **časově-diferenční prostor**, což je prostor dvou dimenzí: simulárního času a celkové diference podprostorů. Interval hodnot dimenze simulárního času je roven $\langle 0, total\_time \rangle$. Interval hodnot celkové diference podprostorů se vztahuje k dvěma sousedním hodnotám simulárního času a nabývá hodnot $\langle min\_diff, max\_diff \rangle$, kde levý, resp. pravý okraj tohoto intervalu představuje minimální, resp. maximální možnou hodnotu celkové diference podprostorů. Z předchozího tedy vyplývá, že interval diferenční složky je variabilní s hodnotou simulárního času. Každá třída podle svých parametrů rozděluje každý z obou intervalů na definované počty ekvidistantních podintervalů. Každému diferenčnímu podintervalu je navíc přidělen jednoznačný identifikátor v podobě přirozeného čísla.

Každá třída jednoznačně přiděluje každému časovému podintervalu jistý podinterval diferenční. Toto zobrazení lze uchovat v tzv. **typovém vektoru třídy**, jehož počet položek je rovný počtu časových podintervalů. Hodnota i-té položky tohoto vektoru je

ID diferenčního podintervalu, který je přidělen i-tému časovému podintervalu. Hodnota celkové diference podprostorů dvou sousedních hodnot simulárního času, které leží obě ve stejném časovém podintervalu, musí být v takovém intervalu hodnot, jehož ID je definován v položce typového vektoru, příslušné tomuto časovému podintervalu. V případě, že obě hodnoty leží každá v jiném časovém podintervalu, se omezení hodnoty diference vztahuje na podinterval hodnot diferencí příslušný časovému podintervalu, v němž leží vyšší hodnota simulárního času.

Každá kombinace hodnot typového vektoru jednoznačně identifikuje danou třídu. Všechny třídy, které mají totožné hodnoty svých parametrů, kromě hodnot typových vektorů, tvoří **systém tříd**. Velikost systému tříd, definovaného na jistých hodnotách parametrů, je tedy roven počtu všech validních kombinací hodnot typového vektoru.

Definice

$CLASSES$ …množina všech tříd testovacích instancí

$TYPE\_VECTORS$ …množina všech typových vektorů tříd, složkami jsou přirozená čísla

$cl\_dec\_num : CLASSES \rightarrow \mathbb{N}$ ...zobrazení, které pro danou třídu vrací počet volitelných parametrů všech mapování každé její instance

$cl\_obj\_num : CLASSES \rightarrow \mathbb{N}$ ...zobrazení, které pro danou třídu vrací počet kritérií všech mapování každé její instance

$cl\_sub\_num : CLASSES \rightarrow \mathbb{N}$ ...zobrazení, které pro danou třídu vrací počet podprostorů všech mapování každé její instance

$cl\_ti\_num : CLASSES \rightarrow \mathbb{N}$ …zobrazení, které pro danou třídu vrací počet ekvidistantních časových podintervalů

$cl\_di\_num : CLASSES \rightarrow \mathbb{N}$ …zobrazení, které pro danou třídu vrací počet ekvidistantních diferenčních podintervalů

$cl\_total\_time : CLASSES \rightarrow \mathbb{R}$ …zobrazení, které pro danou třídu vrací konečnou hodnotu simulárního času každé její instance

$cl\_expti\_num : CLASSES \rightarrow \mathbb{N}$ …zobrazení, které pro danou třídu vrací počet ekvidistantních časových podintervalů každé její instance

$cl\_get\_inst : CLASSES \rightarrow POW(INSTANCES)$ …zobrazení, které pro danou třídu vrací množinu jejích instancí – v rámci daného systému tříd

$cl\_times : CLASSES \rightarrow TIME\_VECTORS$ …zobrazení, které pro danou třídu vrací vektor hodnot simulárních časů všech jejích instancí

$cl\_type : CLASSES \rightarrow TYPE\_VECTORS$ …zobrazení, které pro danou třídu vrací její typový vektor

$$cl\_get\_time\_int: \left\{ \begin{array}{l} (class, time): \\ class \in CLASSES, \\ time \in \langle 0, cl\_total\_time(class) \rangle \end{array} \right\} \to \mathbb{N}$$

…zobrazení, které pro danou třídu a hodnotu simulárního času vrací časový podinterval, do něhož tato hodnota patří

$$inst\_get\_min\_diff: \left\{ \begin{array}{l} (inst, time, front\_num): \\ inst \in INSTANCES, \\ inst\_time\_exists(inst, time) = true \\ front\_num \in 1,..,inst\_sub\_num(inst) \end{array} \right\} \to \mathbb{R}$$

…zobrazení, které pro danou instanci, hodnotu simulárního času a počet front následující hodnoty simulárního času vrací hodnotu minima celkové diference podprostorů

$$inst\_get\_max\_diff: \left\{ \begin{array}{l} (inst, time, front\_num): \\ inst \in INSTANCES, \\ tinst\_time\_exists(inst, time) = true, \\ front\_num \in 1,..,inst\_sub\_num(inst) \end{array} \right\} \to \mathbb{R}$$

…zobrazení, které pro danou instanci, hodnotu simulárního času a počet front následující hodnoty simulárního času vrací hodnotu maxima celkové diference podprostorů

Axiomy

Základní podmínky, které musí splňovat všechny instance jisté třídy.

$\forall class \in CLASSES;$

$$\left(\begin{array}{l} \forall inst \in cl\_get\_inst(class); \\ cl\_dec\_num(class) = inst\_dec\_num(inst) \wedge \\ cl\_obj\_num(class) = inst\_obj\_num(inst) \wedge \\ cl\_sub\_num(class) = inst\_sub\_num(inst) \wedge \\ cl\_times(class) = inst\_times(inst) \end{array} \right)$$

Složky vektoru hodnot simulárního času představují dekompozici simulárního času na ekvidistantní podintervaly.

$\forall class \in CLASSES;$

$$\left(\left| cl\_times(class) \right| = cl\_expti\_num(class) \right)$$

$$\forall class \in CLASSES, last = \left| cl\_times(class) \right|;$$

$$\left(cl\_times(class)(last) = cl\_total\_time(class) \right)$$

$$\forall class \in CLASSES;$$
$$\left(\begin{array}{l}\forall idx \in 1,..,cl\_expti\_num(class), ti\_length = cl\_total\_time(class) / \left|cl\_times(class)\right|; \\ cl\_times(class)(idx) = idx * ti\_length\end{array}\right)$$

Typový vektor dané třídy definuje pro každý časový podinterval jistý podinterval hodnot celkové diference podprostorů, který je vyjádřen svým ID.

$$\forall class \in CLASSES;$$
$$\left(\left|cl\_type(class)\right| = cl\_ti\_num(class)\right)$$

$$\forall class \in CLASSES, \forall i \in 1,..,\left|cl\_type(class)\right|;$$
$$\left(cl\_type(class)(i) \in 1,..,cl\_di\_num(class)\right)$$

$$\forall (class, time) \in DOM(cl\_get\_time\_int),$$
$$tm\_length = cl\_total\_time(class) / cl\_ti\_num(class);$$
$$\left(cl\_get\_time\_int(class, time) = \left\lfloor time / tm\_length \right\rfloor + 1\right)$$

$$\forall class \in CLASSES, \forall inst \in cl\_get\_inst(class), \forall i \in 1,..,\left|cl\_times(class)\right| - 1,$$
$$time1 = cl\_times(class)(i),$$
$$time2 = cl\_times(class)(i+1),$$
$$min\_diff = inst\_get\_min\_diff(inst, time1, time2),$$
$$max\_diff = inst\_get\_max\_diff(inst, time1, time2),$$
$$diff\_int\_length = \left(max\_diff - min\_diff\right) / cl\_di\_num(class),$$
$$ti\_id = cl\_get\_time\_int(class, time2),$$
$$min\_bound = \left(cl\_type(class)(ti\_id) - 1\right) * diff\_int\_length,$$
$$max\_bound = min\_bound + diff\_int\_length$$
$$\left(inst\_get\_total\_diff\left(inst, time1, time2\right) \in \left\langle min\_bound, max\_bound\right\rangle\right)$$

8.2 Testování

Cílem testování bylo porovnat kvalitu výsledků obou optimalizátorů na daném systému tříd testovacích instancí. Během testování se však ukázalo, že již pro systémy tříd velikostí řádově v desítkách instancí, by bylo časově velmi náročné vůbec vygenerovat všechny testovací instance. Proto bylo testování uskutečněno pouze na jistých, vybraných třídách tohoto systému. Z každé třídy bylo vygenerováno celkem 10 testovacích instancí, na kterých byly oba optimalizátory vyzkoušeny.

8.2.1 Parametry systému testovacích tříd

Parametry systémů testovacích tříd byly voleny především s ohledem na omezené časové možnosti. Počet různých tříd roste exponenciálně s počtem ekvidistantních časových podintervalů. Dále je zapotřebí vzít v úvahu skutečnost, že algoritmus generátoru testovacích instancí má zhruba kubickou časovou složitost vzhledem počtu podprostorů. I pro malé systémy tříd může být tedy problém vůbec vygenerovat zadaný počet instancí z každé třídy.

Počet volitelných parametrů mapování.....**2**
Počet kritérií mapování.....**2**
Počet podprostorů mapování **64**
Počet ekvidistantních časových podintervalů.....**3**
Počet ekvidistantních diferenčních podintervalů.....**3**
Celkový simulární čas.....**100**
Počet ekvid. časových podintervalů instancí **100**

8.2.2 Testované třídy

Vzhledem k tomu, že časové požadavky na testování celého systému tříd je obtížné v rozumné době splnit, byly pro testování vybrány pouze některé třídy, které jsou jistým způsobem zajímavé. Mezi vybranými třídami se nachází dvě takové, které jsou si navzájem protikladné vzhledem k průběhu celkových diferencí podprostorů. Výběr pak doplňuje třída střední celkové diference podprostorů celého intervalu simulárního času. Typové vektory vybraných tříd jsou uvedeny níže:

1. (1, 1, 1)
2. (2, 2, 2)
3. (3, 3, 3)

8.2.3 Parametry optimalizátorů

Parametry testovaných optimalizátorů, které se vztahují k časové dekompozici, vychází přímo z parametrů testovaného systému tříd. Hodnoty parametrů dekompozice simulárního času, jejichž názvy jsou pro oba optimalizátory totožné, jsou uvedeny níže. Problém náběhového období je vzhledem ke vlastnostem testovacích instancí irelevantní, proto jsou PRE a TESTTIME nastaveny na hodnotu 0.

PRE **0**
TESTIME.....**0**
TOTTIME.....**100**
NTT.....**100**

Parametr **MODEL_RANDOM_SEED** optimalizátoru GA_PAROPTMULTI nemá pro uměle vytvořené instance smysl, stejně tak jako hodnota předávaná do procedury *r_start* optimalizátoru PAROPTMULTI.

Parametr **LISTLIMIT** optimalizátoru PAROPTMULTI byl pro všechny testy nastaven na hodnotu 30 – množiny výsledných řešení obou optimalizátorů budou stejně velké.

Parametry, které se vztahují k počtu volitelných parametrů a počtu kritérií jsou shodné s parametry ekvivalentního významu testovaného systému tříd.

8.2.4 Optimalizační kritérium

Volba optimalizačního kritéria vyplývá přímo z definice problému optimalizace simulátoru. Podstatný rozdíl je v tom, že ekvivalent vektoru násad nemá v kontextu testovacích instancí smysl, neboť hodnoty kritérií jsou pro každý vektor volitelných parametrů vzhledem k hodnotám simulárního času pevně definovány. Stačí tedy určit vektor časových integrálů kritérií pro každý podprostor, a to na celém intervalu simulárního času. Množinu takovýchto vektorů jisté instance lze použitím Pareto Sorting roztřídit do jednotlivých nedominovaných front, přičemž v pořadí 1. nedominovaná fronta tvoří ekvivalent maxima simulátoru, nechť je nazvána **optimem instance**.

Z výstupů obou optimalizátorů budou extrahovány vzájemně nedominované vektory časových integrálů kritérií výsledné množiny řešení (vektory volitelných parametrů) – taková množina vektorů nechť je označena termínem **množina výsledných hodnot optimalizátoru**. Výsledná množina optimalizátoru PAROPTMULTI tento nárok garantuje, výsledná množina GA_PAROPTMULTI nikoliv. Proto budou vektory časových integrálů výsledné množiny optimalizátoru GA_PAROPTMULTI postoupeny algoritmu Pareto Sorting – v pořadí 1. fronta již nárok, uvedený výše, splňuje.

8.2.5 Metriky kvality řešení

Pro multikriteriální optimalizaci jsou na výslednou množinu nedominovaných vektorů kladeny dva základní požadavky:

1. Konvergence k Pareto Front
2. Rovnoměrné rozprostření na Pareto Front

Je jasné, že tyto požadavky lze stěží adekvátně realizovat prostřednictvím pouze jedné metriky. V literatuře lze najít příklady těchto metrik. Tato práce čerpala v tomto smyslu hlavně z [9], odkud byly převzaty metriky: Error Ratio, Generational Distance a Maximum Pareto Front Error. Tyto metriky jsou zaměřeny především na vyčíslení míry konvergence, i když jimi lze získat také jistou informaci o rozprostření výsledné množiny na Pareto Front.

V[9], který byl hlavním informačním zdrojem této stati, jsou definovány termíny PF_{TRUE} a PF_{KNOWN}. Termín PF_{TRUE} lze v jistém smyslu považovat za ekvivalent termínu optimum instance. Rovněž termín PF_{KNOWN} je možné v jistém smyslu považovat za ekvivalent termínu: množina výsledných hodnot optimalizátoru. Metriky, které byly vybrány z [9], jsou definovány pro PF_{TRUE} a PF_{KNOWN}. Při popisu těchto metrik v této práci však budou místo těchto termínů používány termíny: optimum instance a množina výsledných hodnot optimalizátoru.

Pro rozprostření byla v kontextu této práce definována metrika Percentage of Pareto Front Coverage, která přímo zohledňuje vlastnosti testovacích instancí.

Definice:

$test\_inst$ …testovací instance

$inst\_opt$ …optimum testovací instance $test\_inst$

opt_result …množina výsledných hodnot optimalizátoru

$res\_get\_id : opt\_result \leftrightarrow 1,..,|opt\_result|$ …zobrazení, které přiděluje každému prvku množiny výsledných hodnot optimalizátoru jeho jednoznačný identifikátor

Axiomy:

$test\_inst \in INSTANCES$

$|opt\_result| \geq 1$

$\forall ti\_vector \in inst\_opt, \exists sub\_id \in 1,..,inst\_sub\_num(test\_inst);$
$(ti\_vector = inst\_get\_ti\_values(test\_inst, sub\_id, inst\_total\_time(test\_inst)))$

$\forall ti\_vector \in opt\_result, \exists sub\_id \in 1,..,inst\_sub\_num(test\_inst);$
$(ti\_vector = inst\_get\_ti\_values(test\_inst, sub\_id, inst\_total\_time(test\_inst)))$

$\forall opt\_vec \in inst\_opt, \neg\exists res\_vec \in opt\_result;$
$(res\_vec \geq opt\_vec)$

Error Ratio

Hodnotou této metriky je podíl počtu prvků výsledných hodnot optimalizátoru, které nepatří do optima dané instance, a velikosti optima instance. Pokud jsou výsledné hodnoty optimalizátoru podmnožinou optima, nabývá metrika hodnoty 0. Naopak, pokud je průnik obou množin prázdný, je výsledek roven 1.

$$\forall i \in 1,..,|opt\_result|, res \in opt\_result, res\_get\_id(res) = i;$$
$$((res \in inst\_opt) \rightarrow e_i = 0)$$

$$\forall i \in 1,..,|opt\_result|, res \in opt\_result, res\_get\_id(res) = i;$$
$$((res \notin inst\_opt) \rightarrow e_i = 1)$$

$$E = \frac{\sum_{i=1}^{|opt\_result|} e_i}{|opt\_result|}$$

Generational Distance

Tato metrika reprezentuje, volně řečeno, míru vzdálenosti množiny výsledných hodnot od optima testované instance. Nulová hodnota této metriky odpovídá případu, kdy je množina výsledných hodnot podmnožinou optima testované instance.

d_i …Euklidovská vzdálenost v prostoru hodnot vektorů úplných časových integrálů mezi i-tým prvkem množiny výsledných hodnot testovaného optimalizátoru a prvkem z optima testované instance, který je tomuto i-tému prvku nejblíže

p …pro tento testovací proces byla zvolena hodnota 2

$$G = \frac{\left(\sum_{i=1}^{|opt\_result|} d_i^{\,p} \right)^{1/p}}{|opt\_result|}$$

Maximum Pareto Front Error

Hodnotou této metriky je maximální vzdálenost každého prvku z množiny výsledných hodnot a jemu nejbližšího prvku z optima. Nulová hodnota indikuje, že množina výsledných hodnot je podmnožinou optima, všechny ostatní hodnoty naopak poukazují na to, že alespoň jeden prvek z množiny výsledných hodnot do optima nepatří.

d_i …definována stejně jako u Generational Distance

$$ME = \max \left\{ \begin{matrix} d_i : \\ i \in 1,..,|opt\_result| \end{matrix} \right\}$$

Percentage of Pareto Front Coverage

Hodnotou této metriky je podíl velikosti množiny průniku optima instance s množinou výsledných hodnot optimalizátoru a velikosti optima instance. Rozmezí hodnot tvoří interval $\langle 0,1 \rangle$. Nulová hodnota značí skutečnost, že žádná z výsledných hodnot optimalizátoru nepatří do optima testované instance. Hodnota 1 naopak vyjadřuje, že optimum instance je podmnožinou výsledných hodnot optimalizátoru.

$$PF\_Coverage = \frac{|opt\_result \cap inst\_opt|}{|inst\_opt|}$$

8.2.6 Popis testovacího procesu

Pro každou testovanou třídu bylo vygenerováno celkem 10 různých instancí. Každý optimalizátor byl na každé instanci spuštěn 10 krát, jinými slovy: bylo uskutečněno 10

různých běhů každého optimalizátoru na každé instanci. Hodnoty testovacích metrik se pro každý běh zapisovaly zvlášť. Z těchto výsledků byly poté sestaveny dva různé výstupy, které budou dále popsány.

8.2.7 Výstup základních výběrových statistik metrik kvalit

Množina výsledných hodnot optimalizátoru, která je získána v rámci jistého běhu na testovací instanci, je v rámci jisté metriky kvality porovnána s optimem této testovací instance. Pro každý z 10-ti běhů takto vznikne soubor hodnot, na kterém je spočtena výběrová střední hodnota a výběrový rozptyl.

Postup uvedený výše byl aplikován na každý optimalizátor, třídu, instanci třídy a metriku kvality. Hodnoty těchto výběrových charakteristik jsou rozděleny podle třídy a zapsány do tabulek 5.1, 5.3 a 5.5.

8.2.8 Výstup Wilcoxon Rank Sum Test

Pro přímé porovnání výsledků obou optimalizátorů na testovacích instancích byl použit **Wilcoxon Rank-Sum test**, což je neparametrický test pro rozhodnutí o tom, zda 2 nezávislé výběry pocházejí ze stejného rozdělení. Kompletní popis tohoto testu lze najít v [10], zde bude nastíněna pouze jeho podstata a aplikace na problematiku porovnávání výsledků testů.

Podmínky pro použití tohoto testu:

1. Oba testované výběry jsou navzájem nezávislé.
2. Každé pozorování v každém výběru je nezávislé na ostatních.
3. Pozorování jsou vzájemně porovnatelná – lze určit, zda dvě pozorování jsou totožná, nebo které je větší

Splnění podmínek testu:

1. Každá dvojice testovaných výběrů byla pořízena z výsledků běhů dvou různých, na sobě zcela nezávislých optimalizátorů. Běhy obou optimalizátoru byly provedeny nezávisle na sobě na totožných instancích. V žádném běhu nedefinuje žádný optimalizátor nějaké expertní varianty (které by byly výsledkem jiného běhu).
2. Každý běh daného optimalizátoru na dané instanci je nezávislý na jakémkoliv jiném běhu téhož optimalizátoru na též instanci.
3. Všechna pozorování patří do domény reálných čísel.

Výběry, které byly předmětem testu, jsou množiny 10-ti hodnot jisté metriky kvality. Každá z těchto hodnot byla vypočtena z daného běhu daného optimalizátoru na dané instanci v rámci dané třídy. Dvojice testovaných výběru se vztahují k dané třídě, dané instanci a dané metrice kvality.

Použita byla testová statistika W_A, jejíž hodnoty jsou součty pořadí hodnot jednoho z testovaných výběrů – zde se sčítaly hodnoty výběru příslušného GA_PAROPTMULTI.

Byla testována nulová hypotéza H_0: “Oba výběry pocházejí ze stejné populace” proti alternativní hypotéze H_1: “Oba výběry pocházejí z populací s rozdílným rozdělením pravděpodobnosti”. Hladina významnosti byla pro všechny testy zvolena $\alpha = 0.05$ V případě platnosti H_0 lze říci, že kvalita výsledků obou optimalizátorů na dané instanci dané třídy je vzhledem k dané metrice kvality srovnatelná.

V opačném případě je jasné, že rozdíl kvalit výsledků obou optimalizátorů je zřetelný. O “vítězi” lze potom rozhodnout přímo z umístění hodnoty W_A vzhledem ke střední hodnotě rozdělení W_A. Pro metriky: Error Ratio, Generational Distance a Max Pareto Front Error platí, že pokud je hodnota W_A situována nalevo od střední hodnoty, jsou výsledky GA_PAROPTMULTI kvalitnější a naopak, pravé umístění svědčí ve prospěch PAROPTMULTI. Pro metriku Percentage of Pareto Front Coverage naopak platí, že pokud je hodnota W_A umístěna napravo od střední hodnoty, jsou výsledky GA_PAROPTMULTI kvalitnější.

8.2.9 Výsledky testů a jejich shrnutí

Výsledky testování pro všechny třídy shrnují tabulky 5.1 až 5.6. Tabulky Wilcoxon Rank-Sum testů obsahují pro každou metriku kvality 2 sloupce. Sloupec *„TestVal“* obsahuje hodnoty W_A. Sloupec *„W”* obsahuje zkratku “vítězného” optimalizátoru, nebo písmeno “N” v případě, že kvality řešení obou optimalizátorů jsou srovnatelné a tedy statisticky nevýznamné.

Třída (1,1,1)

Výsledky testování třídy (1,1,1) shrnují tabulky 5.1 a 5.2. Obě tabulky vykazují zřetelnou dominaci kvalit řešení optimalizátoru GA_PAROPTMULTI, a to u všech metrik kromě Max Pareto Front Error, kde jsou kvality obou optimalizátorů vyrovnány. Rozdíl v kvalitách je patrný hlavně u metriky Percentage of Pareto Front Coverage, kde kromě jednoho případu jasně dominoval GA_PAROPTMULTI. Ani v jednom případě se nestalo, že by vítězem byl PAROPTMULTI.

Třída (2,2,2)

Výsledky testování třídy (2,2,2) shrnují tabulky 5.3 a 5.4. GA_PAROPTMULTI vykazuje kromě metriky Max Pareto Front Error absolutní převahu – ve všech případech byly vzhledem k ostatním metrikám kvality jeho řešení statisticky významně lepší.

Třída (3,3,3)

Výsledky testování třídy (3,3,3) shrnují tabulky 5.5 a 5.6. Vzhledem k metrice Max Pareto Front Error se oproti předchozím třídám nezměnilo nic – výsledky jsou vyrovnané. Vzhledem k ostatním metrikám však GA_PAROPTMULTI vykazuje oproti výsledkům na předchozích třídách zřetelné zhoršení. Vzhledem k metrice Error Ratio zvítězil pouze v těsné většině případů, vzhledem k metrice Generational Distance

zvítězil GA_PAROPTMULTI již pouze ve 3 případech. Naopak, zřetelnou převahu udržel GA_PAROPTMULTI vzhledem k Percentage of Pareto Front Coverage.

Tabulka 5.1: Základní výběrové statistiky metrik kvalit výsledných řešení testovaných optimalizátorů na třídě (1,1,1).

ID	OPT	Error Ratio		Gen. Distance		Max PF Error		% PF Coverage	
		Mean	Var	Mean	Var	Mean	Var	Mean	Var
1	GA	2.917e-2	3.491e-3	3.082e3	8.236e7	2.455e4	5.275e9	3.867e-1	3.381e-3
	PA	5.167e-1	1.358e-1	1.092e5	9.127e9	1.700e5	1.224e10	8.665e-2	4.488e-3
2	GA	7.000e-1	2.100e-1	3.012	6.484	3.012	6.484	3.000e-1	2.100e-1
	PA	9.000e-1	9.000e-2	2.487	1.015e1	2.487	1.015e1	1.000e-1	9.000e-2
3	GA	1.708e-1	2.307e-2	6.962e-1	4.304e-1	2.686	7.736	4.125e-1	3.453e-2
	PA	6.500e-1	1.525e-1	1.616	1.078	2.181	3.372	6.250e-2	3.906e-3
4	GA	2.083e-1	3.646e-2	1.030	1.124	3.190	1.182e1	3.375e-1	2.203e-2
	PA	8.500e-1	5.250e-2	2.352	1.311	3.053	5.409	3.750e-2	3.281e-3
5	GA	2.233e-1	8.173e-2	8.046e-1	1.120	2.555	1.458e1	4.286e-1	3.673e-2
	PA	7.833e-1	7.250e-2	1.948	1.062	2.749	4.723	7.144e-2	9.185e-3
6	GA	3.790e-2	3.402e-3	4.578e-2	4.963e-3	3.624e-1	3.064e-1	4.235e-1	1.578e-2
	PA	5.750e-1	8.396e-2	1.492	6.665	2.005	5.724	7.058e-2	3.321e-3
7	GA	1.362e-1	5.131e-3	5.162e-1	2.664e-1	3.073	9.853	3.133e-1	6.267e-3
	PA	2.867e-1	4.404e-2	1.631	4.205	4.328	3.763e1	1.400e-1	5.736e-3
8	GA	8.366e-2	2.056e-3	2.257e-1	2.156e-2	2.294	3.095	3.385e-1	3.492e-3
	PA	3.533e-1	2.360e-2	1.438	8.052e-1	3.527	3.172	7.307e-2	1.316e-3
9	GA	1.468e-1	9.320e-3	9.211e-1	7.150e-1	4.665	1.189e1	3.643e-1	1.372e-2
	PA	4.083e-1	1.256e-1	2.357	1.368e1	4.512	5.483e1	9.287e-2	4.134e-3
10	GA	1.517e-1	1.747e-2	7.096e-1	5.053e-1	3.208	1.292e1	3.083e-1	9.790e-3
	PA	7.833e-1	7.250e-2	3.422	2.105	4.720	2.798	4.167e-2	3.126e-3

Tabulka 5.2: Hodnoty testové statistiky Wilcoxon Rank-Sum testu včetně zkratek příslušných „vítězných“ optimalizátorů na třídě (1,1,1).

ID	Error Ratio		Gen. Distance		Max PF Error		% PF Coverage	
	Test Val	W	Test Val	W	Test Val	W	Test Val	W
1	6.700e1	GA	6.800e1	GA	7.500e1	GA	1.550e2	GA
2	9.500e1	N	1.115e2	N	1.115e2	N	1.150e2	N
3	7.100e1	GA	7.900e1	N	1.070e2	N	1.550e2	GA
4	5.800e1	GA	7.600e1	GA	9.900e1	N	1.520e2	GA
5	6.250e1	GA	7.300e1	GA	9.200e1	N	1.470e2	GA
6	5.500e1	GA	5.500e1	GA	6.850e1	GA	1.550e2	GA
7	8.350e1	N	8.600e1	N	1.000e2	N	1.475e2	GA
8	6.400e1	GA	6.500e1	GA	8.700e1	N	1.550e2	GA
9	8.400e1	N	9.600e1	N	1.185e2	N	1.520e2	GA
10	5.600e1	GA	5.800e1	GA	8.700e1	N	1.540e2	GA

Tabulka 5.3: Základní výběrové statistiky metrik kvalit výsledných řešení testovaných optimalizátorů na třídě (2,2,2).

ID	OPT	Error Ratio		Gen. Distance		Max PF Error		% PF Coverage	
		Mean	Var	Mean	Var	Mean	Var	Mean	Var
1	GA	9.050e-2	5.536e-3	3.986e-1	1.695e-1	2.643	7.447	4.429e-1	1.000e-2
	PA	6.833e-1	1.192e-1	1.712	7.483e-1	2.255	5.652e-1	4.286e-2	2.246e-3
2	GA	1.600e-1	2.390e-2	8.241e-1	6.832e-1	3.163	9.000	4.222e-1	1.926e-2
	PA	7.000e-1	6.000e-2	2.465	2.792	3.391	1.900	6.666e-2	2.962e-3
3	GA	1.376e-1	1.471e-2	6.214e-1	3.271e-1	2.912	7.612	3.636e-1	1.323e-2
	PA	7.500e-1	1.125e-1	1.367	3.738e-1	1.641	2.991e-1	3.636e-2	1.984e-3
4	GA	6.393e-2	2.944e-3	3.837e-1	1.371e-1	3.535	1.011e1	3.727e-1	4.464e-3
	PA	5.833e-1	9.584e-2	1.576	6.995e-1	2.432	6.569e-1	4.546e-2	1.654e-3
5	GA	1.383e-1	1.422e-2	6.175e-1	5.461e-1	2.694	1.064e1	4.500e-1	1.938e-2
	PA	8.500e-1	5.250e-2	1.114	1.022e-2	1.311	4.93e-32	3.750e-2	3.281e-3
6	GA	2.576e-2	1.552e-3	6.027e-1	8.794e-1	7.000	1.175e2	3.889e-1	1.989e-3
	PA	4.000e-1	6.669e-3	8.764	3.533	2.206e1	1.227e1	5.926e-2	3.291e-4
7	GA	1.176e-1	1.026e-2	3.539e-1	2.037e-1	1.717	3.690	3.857e-1	2.164e-2
	PA	6.000e-1	9.556e-2	2.037e2	3.687e5	5.018e2	2.247e6	5.715e-2	1.837e-3
8	GA	5.500e-1	1.058e-1	2.234	2.277	4.044	8.379	2.750e-1	4.312e-2
	PA	9.000e-1	4.000e-2	4.294	3.314	4.788	2.650	5.000e-2	1.000e-2
9	GA	5.023e-2	1.702e-3	2.424e-1	5.594e-2	2.867	7.720	3.454e-1	3.342e-3
	PA	3.750e-1	7.293e-3	5.416e-1	1.521e-2	1.444	4.93e-32	5.455e-2	3.307e-4
10	GA	7.080e-2	3.540e-3	4.020e-1	1.533e-1	3.485	1.248e1	3.818e-1	5.456e-3
	PA	7.500e-1	6.250e-2	2.592	1.945	3.781	8.548	2.272e-2	5.164e-4

Tabulka 5.4: Hodnoty testové statistiky Wilcoxon Rank-Sum testu včetně zkratek příslušných „vítězných“ optimalizátorů na třídě (2,2,2).

ID	Error Ratio		Gen. Distance		Max PF Error		% PF Coverage	
	Test Val	W	Test Val	W	Test Val	W	Test Val	W
1	6.300e1	GA	6.500e1	GA	1.035e2	N	1.550e2	GA
2	5.800e1	GA	7.550e1	GA	1.000e2	N	1.520e2	GA
3	6.300e1	GA	7.500e1	GA	1.125e2	N	1.550e2	GA
4	6.300e1	GA	6.400e1	GA	1.125e2	N	1.550e2	GA
5	5.500e1	GA	7.800e1	GA	1.050e2	N	1.550e2	GA
6	5.500e1	GA	5.500e1	GA	7.350e1	GA	1.550e2	GA
7	6.300e1	GA	6.800e1	GA	9.150e1	N	1.550e2	GA
8	7.600e1	GA	7.800e1	GA	9.100e1	N	1.340e2	GA
9	5.500e1	GA	7.100e1	GA	1.100e2	N	1.550e2	GA
10	5.500e1	GA	5.500e1	GA	1.000e2	N	1.550e2	GA

Tabulka 5.5: Základní výběrové statistiky metrik kvalit výsledných řešení testovaných optimalizátorů na třídě (3,3,3).

ID	OPT	Error Ratio		Gen. Distance		Max PF Error		% PF Coverage	
		Mean	Var	Mean	Var	Mean	Var	Mean	Var
1	GA	2.500e-1	3.333e-2	6.086e-1	6.243e-1	2.016	1.004e1	4.000e-1	2.400e-2
	PA	8.000e-1	1.100e-1	2.043	1.195	2.483	2.364	6.000e-2	8.400e-3
2	GA	6.000e-1	2.400e-1	3.217	1.025e1	3.217	1.025e1	4.000e-1	2.400e-1
	PA	1.000	0.000	2.862	1.516	2.862	1.516	0.000	0.000
3	GA	7.663e-2	1.680e-3	4.411e-1	1.091e-1	4.562	1.111e1	3.808e-1	4.275e-3
	PA	5.750e-1	8.396e-2	2.471	1.795	5.065	1.544e1	4.615e-2	1.420e-3
4	GA	1.330e-1	2.917e-2	7.889e1	2.484e4	1.375e3	7.545e6	3.704e-1	5.485e-3
	PA	3.700e-1	5.877e-2	6.543e-1	1.838e-1	1.592	2.814e-1	7.036e-2	1.220e-3
5	GA	6.000e-1	2.400e-1	3.499	1.255e1	3.499	1.255e1	4.000e-1	2.400e-1
	PA	1.000	0.000	4.451	8.521e-1	4.451	8.521e-1	0.000	0.000
6	GA	7.090e-2	1.344e-2	1.657e2	6.532e4	2.166e3	1.094e7	3.436e-1	1.735e-3
	PA	3.083e-1	7.785e-2	2.908e3	2.130e7	8.045e3	1.460e8	5.385e-2	1.243e-3
7	GA	6.000e-1	2.400e-1	3.125	8.880	3.125	8.880	4.000e-1	2.400e-1
	PA	1.000	0.000	2.123	0.000	2.123	0.000	0.000	0.000
8	GA	4.500e-1	1.725e-1	3.049	1.045e1	3.970	1.470e1	3.333e-1	8.889e-2
	PA	1.000	0.000	5.415	9.383	5.415	9.383	0.000	0.000
9	GA	2.345e-2	1.313e-3	8.694e-2	2.595e-2	1.120	4.339	3.667e-1	1.975e-3
	PA	4.783e-1	8.006e-2	1.219	3.498e-1	2.871	2.126	5.000e-2	1.358e-3
10	GA	1.682e-1	1.952e-2	2.185	5.867	1.452e1	2.355e2	3.160e-1	3.344e-3
	PA	7.167e-1	8.362e-2	1.868	5.683e-1	2.607	1.97e-31	2.800e-2	9.760e-4

Tabulka 5.6: Hodnoty testové statistiky Wilcoxon Rank-Sum testu včetně zkratek příslušných „vítězných“ optimalizátorů na třídě (3,3,3).

ID	Error Ratio		Gen. Distance		Max PF Error		% PF Coverage	
	Test Val	W	Test Val	W	Test Val	W	Test Val	W
1	6.550e1	GA	6.750e1	GA	8.550e1	N	1.505e2	GA
2	8.500e1	N	1.025e2	N	1.025e2	N	1.250e2	N
3	5.500e1	GA	6.000e1	GA	1.045e2	N	1.550e2	GA
4	7.750e1	GA	8.100e1	N	1.035e2	N	1.550e2	GA
5	8.500e1	N	9.300e1	N	9.300e1	N	1.250e2	N
6	8.100e1	N	9.000e1	N	9.000e1	N	1.550e2	GA
7	8.500e1	N	1.100e2	N	1.100e2	N	1.250e2	N
8	7.000e1	GA	8.000e1	N	9.650e1	N	1.400e2	GA
9	5.500e1	GA	5.600e1	GA	7.250e1	GA	1.550e2	GA
10	5.850e1	GA	9.500e1	N	1.200e2	N	1.550e2	GA

9 Testování a porovnávání simulátorů na reálných simulačních modelech

Tato kapitola pojednává o testování popsaných optimalizátorů na reálných simulačních modelech, přesněji a v souladu s terminologií, zavedené ve 2. kapitole, řečeno: o testování na simulátorech reálných simulovaných systémů. Množina různých typů simulovaných systémů je bezpochyby příliš rozsáhlá na to, aby se dala cele otestovat. Mezi nejběžněji se vyskytujícími simulovanými systémy patří systémy hromadné obsluhy, které se proto stanou předmětem testování obou optimalizátorů.

9.1 Stručný úvod do teorie hromadné obsluhy

Teorie hromadné obsluhy (THO) se zabývá zkoumáním systémů hromadné obsluhy (SHO). Do určitého systému hromadné obsluhy přicházejí obecně v náhodných časových okamžicích požadavky (jednotky), které požadují obsluhu. SHO se skládá z čekacího prostoru, který je tvořen frontami příchozích požadavků, a obslužného systému, který tvoří linky (kanály, facility) obsluhy. Doba trvání obsluhy požadavků je obecně náhodná.

9.1.1 Klasifikace systémů hromadné obsluhy

Z výše uvedeného popisu vyplývá, že SHO mají určité charakteristiky, podle nichž je možné je blíže klasifikovat. V teorii hromadné obsluhy se používá **Kendallova klasifikace**. Její rozšířené značení má takovýto tvar:

A / B / C / K / N /D

Význam jednotlivých písmen je následující:
A – rozdělení časových intervalů mezi příchody požadavků do systému
B – rozdělení doby obsluhy
C – počet linek obsluhy
K – maximální počet požadavků v systému (počet obslužných linek plus počet míst pro čekající), nebo pouze počet míst pro čekající
N – počet zdrojů požadavků, neuvádí se, pokud je nekonečný
D – režim fronty

Rozdělení časových intervalů mezi příchody

M…Poissonův proces příchodů
D…doba mezi příchody je pevně stanovená
E_k…Erlangovo rozdělení s parametrem *k*
G…obecné, tj. jakékoliv rozdělení

Rozdělení doby obsluhy

M…exponenciální rozdělení
D…doba obsluhy je pevně stanovená
E_k…Erlangovo rozdělení s parametrem *k*
G…obecné, tj. jakékoliv rozdělení

Režim fronty

FIFO/FCFS…požadavky jsou obsluhovány v pořadí svých příchodů do systému
LIFO/LCFS…požadavky jsou obsluhovány v opačném pořadí svých příchodů do systému
SIRO…požadavky jsou obsluhovány v náhodném pořadí – bez ohledu na pořadí svých příchodů do systému
PNPN…požadavky jsou obsluhovány na základě svých priorit

9.2 Testování

Cílem testování bylo porovnat kvalitu výsledků obou optimalizátorů na simulátorech systémů hromadné obsluhy vybraných charakteristik.

9.2.1 Testované systémy

Testovány byly dva systémy hromadné obsluhy, přesněji řečeno simulátory těchto systémů. Oba obsahovaly pouze jednu facilitu, velikost fronty požadavků nebyla omezena a samotný počet požadavků rovněž nebyl omezen. Facilita obsluhovala požadavky v pořadí jejich příchodů do systému.

Facilita si eviduje celkovou dobu *tt_wait*, kdy byla nečinná – neobsluhuje žádný požadavek a fronta je prázdná. Když k takové situaci dojde, facilita si eviduje čas *tt_beg* té události, a když se objeví příchozí požadavek, přičte se k *tt_wait* hodnota *time-tt_beg* této nečinnosti.

Každý požadavek je schopný si evidovat celkovou dobu *t_wait* strávenou ve frontě. Pokud je tedy po svém příchodu do systému zařazen do fronty, eviduje si v *t_beg* začátek svého čekání. V okamžiku počátku obsluhy tohoto požadavku facilitou, je k hodnotě *t_wait* přičtena hodnota *time-t_beg* tohoto čekání ve frontě.

Systém A

Intervaly mezi příchody jednotlivých požadavků do tohoto systému pochází z normálního rozdělení se střední hodnotou *t_enter* a směrodatnou odchylkou *sigma_enter*. Obsluha facilitou je také náhodná o normálním rozdělení se střední hodnotou *t_service* a směrodatnou odchylkou *sigma_service*. Délka simulačního pokusu je od nuly do *t_end*, ale sběr dat se provádí až od *t_reset*, aby se eliminovaly případné singulární jevy na počátku. Čekací doby jednotlivých požadavků ve frontě se načítají na globální proměnnou *t_glob*. V okamžiku *t_reset* se *t_glob* i *tt_wait* prostě vynulují. Na konci simulačního pokusu dávají tyto dvě hodnoty celkový obraz o čekání.

Klasifikace systému: G / G / 1 / ∞ / ∞ /FIFO

Prakticky interpretovatelné parametry simulátoru systému A:

1. *t_enter*
2. *sigma_enter*
3. *t_service*
4. *sigma_service*

Systém B

Je podobný systému A kromě intervalů příchodů požadavků a časů obsluhy, které mají exponenciální rozdělení. Hodnoty *sigma_enter* a *sigma_sevice* tedy *nejsou* použity.

Kvalifikace systému: G / M / 1 / ∞ / ∞ /FIFO
Prakticky interpretovatelné parametry simulátoru systému B:

1. *t_enter*
2. *t_service*

9.2.2 Definice testů

Všechny testy definovaly stejnou časovou dekompozici simulátorů. Významově totožné parametry časové dekompozice obou testovaných optimalizátorů dostanou tudíž stejné hodnoty:

PRE 0
TESTIME 100
TOTTIME 10000
NTT 100

Parametr **MODEL_RANDOM_SEED** optimalizátoru GA_PAROPTMULTI má pro všechny testy konstantní hodnotu 3, která je totožná s hodnotou předávanou do procedury *r_start* optimalizátoru PAROPTMULTI.

Parametr **LISTLIMIT** optimalizátoru PAROPTMULTI byl pro všechny testy nastaven na hodnotu 30 – množiny výsledných řešení obou optimalizátorů budou stejně velké.

Počet volitelných parametrů je pro simulátor systému A roven 4, pro simulátor systému B roven 2 – viz definice obou systémů. Počet složek vektorového kritéria plyne přímo z definice každého testů – viz dále.

Byly provedeny celkem 4 druhy testů. Druhy 1 – 3 splňují následující předpoklady:

Systém A

$t\_enter \in \langle 0.1, 10 \rangle$

$sigma\_enter \in \langle 0, 10 \rangle$

$t\_service \in \langle 0, 10 \rangle$

$sigma\_service \in \langle 0, 10 \rangle$

Systém B

$t\_enter \in \langle 0.1, 10 \rangle$

$t\_service \in \langle 0, 10 \rangle$

t_enter je zdola omezen již hodnotou 0.1, neboť v případě hodnoty blízké nule by mohlo dojít k situaci, že veškerý strojový čas by byl spotřebován na generování neúnosného množství požadavků, které by později zcela vyčerpaly přidělenou paměť, což by později vedlo k pádu simulátoru.

Test 1

Minimalizace hodnot: $\int_{t=0}^{t\_end} t\_glob$, $\int_{t=0}^{t\_end} tt\_wait$

Test 2

Minimalizace hodnot: $\int_{t=0}^{t\_end} t\_glob$, $\int_{t=0}^{t\_end} tt\_wait$, $\int_{t=0}^{t\_end} abs(t\_enter - V)$.

$V = 5$

Test 3

Minimalizace hodnot:

$$\int_{t=0}^{t\_end} t\_glob, \int_{t=0}^{t\_end} tt\_wait, \int_{t=0}^{t\_end} abs(t\_enter - E), \int_{t=0}^{t\_end} abs(t\_service - S).$$

$E = 2$
$S = 7$

Test 4_1

Minimalizace hodnot: $\int_{t=0}^{t\_end} t\_glob$, $\int_{t=0}^{t\_end} tt\_wait$

Systém A

$t\_enter \in \langle 2, 5 \rangle$

$sigma\_enter \in \langle 0, 10 \rangle$

$t\_service \in \langle 6, 9 \rangle$

$sigma\_service \in \langle 0, 10 \rangle$

Systém B

$t\_enter \in \langle 2,5 \rangle$

$t\_service \in \langle 6,9 \rangle$

Test 4_2

Minimalizace hodnot: $\int_{t=0}^{t\_end} t\_glob$, $\int_{t=0}^{t\_end} tt\_wait$.

Systém A

$t\_enter \in \langle 1,5.5 \rangle$

$sigma\_enter \in \langle 0,10 \rangle$

$t\_service \in \langle 5.6,10 \rangle$

$sigma\_service \in \langle 0,10 \rangle$

Systém B

$t\_enter \in \langle 1,5.5 \rangle$

$t\_service \in \langle 5.6,10 \rangle$

Test 4_3

Minimalizace hodnot: $\int_{t=0}^{t\_end} t\_glob$, $\int_{t=0}^{t\_end} tt\_wait$.

Systém A

$t\_enter \in \langle 5.6,10 \rangle$

$sigma\_enter \in \langle 0,10 \rangle$

$t\_service \in \langle 1,5.5 \rangle$

$sigma\_service \in \langle 0,10 \rangle$

Systém B

$t\_enter \in \langle 5.6,10 \rangle$

$t\_service \in \langle 1,5.5 \rangle$

Test 4_4

Minimalizace hodnot: $\int_{t=0}^{t\_end} t\_glob$, $\int_{t=0}^{t\_end} tt\_wait$.

Systém A

$t\_enter \in \langle 1,7 \rangle$

$sigma\_enter \in \langle 0,10 \rangle$

$t\_service \in \langle 4,10 \rangle$

$sigma\_service \in \langle 0,10 \rangle$

Systém B

$t\_enter \in \langle 1,7 \rangle$

$t\_service \in \langle 4,10 \rangle$

9.2.3 Metriky kvality řešení

V případě, že je kardinalita množiny apriorních řešení optimalizovaného simulátoru rovna ∞ a vyčíslovaná kritéria mají neznámý předpis (což je drtivá většina případů), je nemožné determinovat globální maximum tohoto optimalizovaného simulátoru.

Vzhledem k definici prostoru prakticky interpretovatelných parametrů simulátorů systémů A a B tedy nelze množiny řešení, vrácené testovanými optimalizátory, porovnat s množinami optimálních řešení těchto simulátorů. Další možností je určitým způsobem aproximovat optimální množinu řešení. Ta by mohla být založena na rozdělení prostoru apriorně přípustných řešení na dostatečně velký počet disjunktních podprostorů, přičemž na celém podprostoru by byla střední hodnota vektorového kritéria konstantní. Nevýhoda spočívá v tom, že se změnou přesnosti – počtu podprostorů, se mění také optimální množina. Dosažení vyšší přesnosti vyžaduje samozřejmě vyšší nároky na strojový čas a kapacitu úložiště.

Řešení tohoto problému, které bylo použito v této práci, spočívá v přímém porovnání vektorů kritérií výsledných množin obou optimalizátorů. Vektory kritérií obou výsledných množin se sjednotí a setřídí do nedominovaných front. Každý vektor dostane přiděleno přirozené číslo, jehož hodnota bude rovna pořadí fronty, do níž náleží. Takto vzniknou dva výběry hodnot přirozených čísel, na které se aplikuje Wilcoxon Rank-Sum Test. Podle výsledků testového kritéria lze pak rozhodnout o vzájemné kvalitě výsledných řešení obou optimalizátorů. Vzhledem k tomu, že nižší hodnoty pořadí nedominovaných front korelují s kvalitnějšími řešeními, bude mít výběr, spojený s kvalitnějšími řešeními, průměrně nižší hodnoty vzhledem ke druhému výběru.

Testová statistika *Z*

Vzhledem k tomu, že oba testované výběry hodnot mají rozsah v řádu stovek prvků, nelze již použít jako testovou statistiku W_A. Hodnoty této statistiky jsou totiž tabelovány do rozsahu 12-ti prvků.

Jak uvádí [10], pokud jsou rozsahy obou testovaných výběrů rovné nebo větší než 10, lze brát $W_A \sim N(\mu_A, \sigma_A)$, kde:

$$\mu_A = \frac{n_A(n_A + n_B + 1)}{2}$$

$$\sigma_A = \sqrt{\frac{n_A n_B \left(n_A + n_B + 1\right)}{12}}$$

n_A …rozsah testovaného výběru A

n_B …rozsah testovaného výběru B

Potom platí, že: $P\left(W_A \geq w_A\right) \approx P\left(Z \geq z\right)$, kde:

$$z = \frac{w_A - \mu_A}{\sigma_A}$$

$$Z \sim N(0,1)$$

9.2.4 Výsledky a jejich shrnutí

Výsledky testování pro všechny simulátory shrnují tabulky 6.1 a 6.2. Sloupec *Z-value* obsahuje hodnoty testové statistiky *Z* pro výběr GA_PAROPTMULTI. Sloupec *Winner* obsahuje zkratku optimalizátoru, jehož výsledná řešení byla celkově kvalitnější.

Systém A

Tabulka 6.1 vyjadřuje zcela evidentně, že kvalitnějších řešení dosáhl ve všech testech simulátoru systému A optimalizátor GA_PAROPTMULTI.

Systém B

Výsledky testování na simulátoru systému B, které vyjadřuje tabulka 6.2, hovoří opět ve prospěch optimalizátoru GA_PAROPTMULTI v podstatné většině testů. Pouze ve 2 případech dosáhl PAROPTMULTI kvalitnějších výsledků.

Závěrem lze tedy říci, že GA_PAROPTMULTI dosáhl v podstatné většině testů, na simulátorech obou definovaných systémů hromadné obsluhy, z hlediska kvality výsledných řešení, lepších výsledků. Je však otázkou, zda výsledná řešení GA_PAROPTMULTI jsou z hlediska kvality alespoň uspokojivá v porovnání s optimální množinou řešení, která nebyla k dispozici.

Tabulka 6.1: Hodnoty statistiky Z včetně zkratek „vítězných“ optimalizátorů na testovaném systému A.

TEST ID	Z-value	Winner
1	-1.45e1	GA
2	-2.04e1	GA
3	-1.05e1	GA
4_1	-1.41e1	GA
4_2	-1.31e1	GA
4_3	-1.35e1	GA
4_4	-2.08e1	GA

Tabulka 6.2: Hodnoty statistiky Z včetně zkratek „vítězných“ optimalizátorů na testovaném systému B.

TEST ID	Z-value	Winner
1	1.54e1	PA
2	-4.71	GA
3	7.95	PA
4_1	-9.26	GA
4_2	-1.21e1	GA
4_3	-4.19	GA
4_4	-1.05e1	GA

10 Závěr

10.1 Shrnutí

Smyslem této práce bylo vyvinout a implementovat optimalizační metodu pro simulované systémy, založenou na principu genetických algoritmů, která by se uplatňovala již za běhu simulátorů optimalizovaných systémů. Přímo navazuje na disertační práci RNDr. Jiřího Weinbergera, CSc., který vytvořil podobný typ optimalizátoru s tím rozdílem, že místo metod genetických algoritmů použil jiných technik, které se v praxi ukázaly být velmi úspěšné.

Optimalizační metoda, uvedených vlastností, byla v kontextu této práce vyvinuta a implementována do optimalizátoru GA_PAROPTMULTI. Kvalita výsledných řešení obou optimalizátorů byla porovnána jak na příkladech ryze teoretických, tak na příkladech praktických.

Pro porovnávání kvalit na ryze teoretických příkladech byl vytvořen a formalizován systém tříd testovacích instancí, který umožňuje přesně specifikovat vlastnosti testovacích instancí, patřících do té které třídy. Systém, na jehož třídách se testování provedlo, obsahoval celkem 27 tříd. Testování kvality bylo uskutečněno, pro vysoké časové nároky, pouze na 3 „zajímavějších“ třídách. Na instancích těchto tříd dosáhl GA_PAROPTMULTI v porovnání s PAROPTMULTI lepších kvalit výsledných řešení.

Kvality výsledných řešení optimalizátorů byly porovnány také na simulátorech systémů hromadné obsluhy. Byly specifikovány 2 jednoduché systémy hromadné obsluhy, na jejichž simulátorech se testování provedlo. Také zde GA_PAROPTMULTI celkově předčil PAROTPMULTI ve kvalitě výsledných řešení. Jelikož však nebyla známa optimální množina řešení, nelze říci, jaká by byla kvalita výsledných řešení vzhledem k této optimální množině.

10.2 Budoucnost

Výkonné jádro optimalizátoru GA_PAROPTMULTI netvoří algoritmus, který by byl speciálně určen pro simulační optimalizaci. V nedávné době byly uveřejněny práce – viz [11], které se zabývají vývojem genetických algoritmů speciálně pro účely simulační optimalizace. Bohužel, vzhledem k časové tísni již nemohl být algoritmus jádra znovu reimplementován podle popisu v těchto pracích. Rozhodně by však bylo minimálně zajímavé tyto algoritmy v rámci jiné práce důkladně otestovat a v případě pozitivních výsledků jádro GA_PAROPTMULTI reimplementovat.

Bohužel, příliš velké časové nároky nakonec nedovolily provést testování na celém systému tříd (tříd je celkem 27). I tak bylo ovšem rozdělení časově diferenčního prostoru poměrně hrubé – pouze 3 časové a 3 diferenční intervaly. Bylo by logické třídy, kde není jednoznačný vítěz, rozdělit na jednotlivé podtřídy a provést testování na nich. Aplikací tohoto postupu by bylo možno získat velmi podrobné informace o rozložení kvality řešení testovaných algoritmů na daném systému tříd. Potíž je však v tom, že celkový počet tříd roste exponenciálně vzhledem k počtu časových intervalů testovaného systému tříd.

Též by bylo zajímavé udělat analýzu toho, do jaké třídy, vzhledem k danému systému tříd testovacích instancí, by simulátory testovaných systémů hromadné obsluhy náležely. Vzhledem k definici systému tříd však nemusí náležet do žádné, neboť mohou v určitém časovém intervalu třídy porušovat interval povolených diferencí, definovaný pro tento časový interval. Další problém je v tom, že pro různé hodnoty vektoru násad simulátoru bude průběh vektorového kritéria – a tím pádem i vektoru časových integrálů, obecně různý. Takže není vyloučeno, že obecně různé vektory násad budou implikovat obecně různé třídy. Triviální řešení by bylo, určit třídu pro vektor středních hodnot časových integrálů. Podstatná nevýhoda tohoto řešení však spočívá v tom, že v případě velkého rozptylu hodnot, bude zařazení velmi hrubé až nepřesné, čímž může znehodnotit výsledky eventuálního testování na daném systému tříd.

O rozdělení optimalizovaných analytických funkcí do tříd se lze dočíst např. v [9], o podobném rozdělení simulovaných systémů ve smyslu jejich optimalizace jsem žádné informace v dostupných zdrojích nenašel. Možná by proto mělo smysl teorii, která byla nastíněna v 5. kapitole, více v tomto smyslu rozvinout.

11 Použitá literatura a zdroje

[1] Křivý I., Kindler E. (2001): Simulace a modelování. Učební texty ostravské univerzity, Přírodovědecká fakulta. World Wide Web, *http://prf.osu.cz/kip/dokumenty/Msm.pdf*

[2] Šplíchal J., Mojka A., Weiberger J. (1985): Simulační experimenty a jejich vyhodnocování. INORGA, Praha.

[3] Weinberger J. (1992): Optimalizace a identifikace simulačních modelů složitých systémů. Disertační práce, Univerzita Karlova, Matematicko-fyzikální fakulta.

[4] Pevný T. (2003): Neuronové sítě a genetické algoritmy – vybrané aplikace při řešení některých technických problémů. Diplomová práce, ČVUT, Fakulta Jaderná a Fyzikálně Inženýrská, katedra matematiky. World Wide Web, *http://www.scienceworld.cz/sw.nsf/ed161f5b1ecd72c3c1256e7f00342c53/68a67e1716193394c1256e8b003477e4/$FILE/final.pdf*

[5] Goldberg David E. (1989): Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Pusblishing Company, Inc.

[6] Herrera F., Lozano M., Verdegay J.L. (1996): Tackling Real-Coded Genetic Algorithms: Operators and Tools for Behavioural Analysis. Department of Computer Science and Artificial Intelligence, University of Granada, Spain. File Transfer Protocol, *ftp://decsai.ugr.es/pub/arai/tech_rep/ga-fl/AIRE96.ps.Z*

[7] Herrera F., Lozano M., Sánchez A.M. (2004): Hybrid Crossover Operators for Real-Coded Genetic Algorithms: An Experimental Study. World Wide Web, *http://sci2s.ugr.es/publications/ficheros/HERRERA-SCJ.PDF*

[8] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, T. Meyarivan (2000): Fast and Elitist Multi-Objective Genetic Algorithm: NSGA-II.
Kanpur Genetic Algorithms Laboratory (KanGAL). Indian Institute of Technology Kanpur. World Wide Web, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.7210

[9] David A. Van Veldhuizen (1999): Multiobjective Evolutionary Algorithms: Classification, Analyses, and New Innovations. Disertační práce. World Wide Web, *http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.1823*

[10] The Wilcoxon Rank-Sum Test. World Wide Web, *http://www.stat.auckland.ac.nz/~wild/ChanceEnc/Ch10.wilcoxon.pdf*

[11] Hamidreza Eskandari, Luis Rabelo, Mansooreh Mollaghasemi: Multiobjective Simulation Optimization using an Enhanced Genetic Algorithm. Department of Industrial Engineering and Management Systems, 4000 Central Florida Blvd. University of Central Florida, Orlando, FL 32816, U.S.A. World Wide Web,

http://www.lania.mx/~ccoello/eskandari/05.pdf.gz

[12] Marek Malík (1984): Programování v jazyku Simula-67, 1. svazek. Univerzita Karlova, Praha.

[13] Marek Malík (1984): Programování v jazyku Simula-67, 2. svazek. Univerzita Karlova, Praha.