Bakalářská práce

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra kybernetiky

Algoritmus generující operace ve vícehodnotové logice

Jan Kozák

Květen 2017
Vedoucí práce: Ing. Milan Petrík, Ph.D.

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Jan K o z á k

Studijní program: Otevřená informatika (bakalářský)

Obor: Informatika a počítačové vědy

Název tématu: Algoritmus generující operace ve vícehodnotové logice

Pokyny pro vypracování:

Implementujte v jazyce C nebo C++ algoritmus popsaný v článku "Rees coextensions of finite, negative tomonoids".
Výstupem algoritmu má být např. textový soubor přehledně popisující všechny fuzzy konjunkce vygenerované až po danou velikost. Vzhledem k obrovské náročnosti algoritmu klaďte důraz na operační a paměťovou optimalizaci. Přidejte i možnost rozdělení běhu programu do více procesů nebo vláken, aby bylo možné využít více jader počítače. Předpokládá se, že program bude testován a spouštěn na operačním systému Linux (nebo podobném).

Seznam odborné literatury:

[1] Navara Mirko, Olšák Petr – Základy fuzzy množin – Praha, 2007
[2] Petrík Milan, Vetterlein Thomas – Rees coextensions of finite, negative tomonoids – Journal of Logic and Computation, 2015

Vedoucí bakalářské práce: Ing. Milan Petrík, Ph.D.

Platnost zadání: do konce zimního semestru 2017/2018

L.S.

prof. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 23. 5. 2016

Poděkování / Prohlášení

Chtěl bych poděkovat své rodině a své přítelkyni Barboře Dratvové za jejich dlouholetou podporu. Také bych chtěl poděkovat vedoucímu práce Ing. Milanovi Petríkovi, Ph.D. za jeho přínosné připomínky k práci.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 26. 5. 2017

....

Abstrakt / Abstract

Monoidy s lineárním uspořádáním (zkráceně tomonoidy) jsou matematickou strukturou, která může mimo jiné popisovat sémantiku konečných fuzzy logik. Cílem této práce je implementovat v jazyce C/C++ algoritmus popsaný Milanem Petríkem a Thomasem Vetterleinem v článcích „*Rees coextensions of finite, negative tomonoids*“ a „*Algorithm for generating finite totally ordered monoids*“, jenž k libovolnému tomonoidu vygeneruje všechna jeho jednoprvková rozšíření, tzv. reesovské koextenze, která jsou opět tomonoidy. Tímto způsobem je v principu možné vygenerovat všechny reprezentace konečných fuzzy logik, nicméně časová náročnost algoritmu je exponenciální v závislosti na počtu prvků tomonoidu. Práce se tedy také zabývá možnostmi paralelizace procesu a vhodným způsobem ukládání vygenerovaných výsledků. Cílovou platformou implementace je operační systém Linux.

Klíčová slova: konečné negativní tomonoidy, vícehodnotová logika, fuzzy logika

Totally ordered monoids (abbreviated as tomonoids) are mathematical structures which can be utilized – among other – to describe semanthics of finite-valued fuzzy logics. Main goal of the thesis is an implementation of the algorithm described by Milan Petrík and Thomas Vetterlein in articles “*Rees coextensions of finite, negative tomonoids*” and ‘*Algorithm for generating finite totally ordered monoids*” in C/C++. The algorithm generates one-element extensions (called Rees coextensions) to a given tomonoid which are, in turn, tomonoids as well. Using this approach it is possible - in principle - to create all the representations of finite-valued fuzzy logics, however the time complexity of the algorithm is exponential in respect to the cardinality of tomonoid. Due to this fact the thesis also considers parallelization of the algorithm and an appropriate method of storing the generated results. The target implementation platform is the Linux operating system.

Keywords: finite negative tomonoids, multivalued logic, fuzzy logic

/ Obsah

1 Úvod . . . 1
1.1 Fuzzy množiny a fuzzy logika . . . 1
1.1.1 Fuzzy konjunkce . . . 2
1.2 Teorie a terminologie . . . 3
1.2.1 Reprezentace tomonoidů . . 4
1.2.2 Reesovské koextenze . . . 4
1.2.3 Jednoprvkové reesovské koextenze . . . 6
2 Algoritmus . . . 8
2.3 Poznámky . . . 10
3 Implementace . . . 11
3.1 Generátor tomonoidů . . . 11
3.1.1 Seznam přepínačů . . . 11
3.1.2 Objektový návrh . . . 13
3.2 Implementace algoritmu . . . 14
3.2.1 Reprezentace tomonoidů v programu . . . 14
3.2.2 Generování reesovských koextenzí . . . 15
3.3 Paralelizace výpočtu . . . 17
3.3.1 Prohledávání stromu tomonoidů . . . 18
3.3.2 Synchronizace . . . 18
3.4 Ukládání výsledků . . . 19
3.4.1 Zobrazování tomonoidů . . 20
3.5 Srovnání . . . 21
4 Závěr . . . 23
Literatura . . . 24
A Seznam zkratek . . . 27
B Obsah přiloženého CD . . . 28

Tabulky

3.1. Porovnání doby běhu I. 21
3.2. Porovnání doby běhu II. 22
3.3. Porovnání doby běhu III. 22
3.4. Porovnání velikostí výstupních souborů 22

Obrázky

1.1. Příklad funkce příslušnosti 2
1.2. Dvourozměrná reprezentace tomonoidu 4
1.3. Asociativita tomonoidu 5
1.4. Řetězec reesovských kvocientů . . 6
1.5. Příklady diskrétních t-norem 7
3.1. Příklad vyhledání výsledku operace 15
3.2. Přiřazování volných tříd 16
3.3. Příklad uložení do souboru 20
3.4. Screenshot zobrazení tomonoidu 21

Kapitola 1
Úvod

Cílem práce je implementovat v jazyce C/C++ algoritmus generující operace v konečných vícehodnotových logikách navržený v článku *Rees coextensions of finite, negative tomonoids*[1]. K jejich popisu je použita struktura tzv. konečných negativních monoidů s lineárním uspořádáním, zkráceně (z angličtiny) *tomonoidů*. Ukazuje se, že počet tomonoidů, a tím pádem i časová náročnost algoritmu, roste exponenciálně v závislosti na počtu prvků tomonoidu. Dalším úkolem práce tedy je prozkoumat možnosti paralelizace výpočtu a navrhnout vhodný způsob ukládání a reprezentace výsledků. Teoretické základy vedoucí k algoritmu jsou vysvětleny v sekci 1.2, následuje rozbor algoritmu v kapitole 2 a jeho implementace je popsána v kapitole 3.

1.1 Fuzzy množiny a fuzzy logika

Tomonoidy jsou obecnou matematickou strukturou, která může sloužit k popisu operací v rozličných odvětvích matematiky či informatiky. V článcích, které slouží jako základní zdroj pro tuto práci[1,2], jsou tomonoidy interpretovány především jako reprezentanti konjunkcí v konečných vícehodnotových logikách, proto je na úvod práce zařazena tato sekce.

Za prvotní příspěvek k teorii fuzzy množin je považován článek *Fuzzy Sets*[3] z roku 1965, jehož autorem je Lofti A. Zadeh. Na ní pak v 70. letech dále formuloval základy fuzzy logiky. Nicméně vícehodnotovými logikami se zabývali matematici už dříve, např. Jan Łukasiewicz nebo Kurt Gödel[4, s. 21]. Jako příklad v praxi často užívané vícehodnotové logiky uveďme tříhodnotovu logiku dotazovacího jazyka SQL.

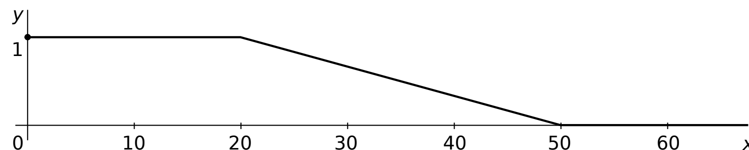
Jednou z motivací k zavedení teorie fuzzy množin a fuzzy logiky je skutečnost, že lidé v běžném životě často operují s vágně definovanými pojmy, které v klasických protějšcích těchto teorií nelze snadno (nebo výstižně) popsat. Zatímco k zavedení množiny neplnoletých občanů České republiky zcela postačuje klasická teorie množin, u množiny *mladých* občanů České republiky už by pravděpodobně vznikl problém kvůli nejasné definici toho, kdo je ještě mladý a kdo už ne. Lze samozřejmě zavést pevnou věkovou hranici, např. 26 let, do které osobu prohlásíme za mladou, nicméně je otázkou, jestli tato pevná hranice dostatečně dobře vystihuje pojem *mladý*. Opravdu je člověk do 26 let úplně mladý a v den svých 26. narozenin už najednou vůbec mladý není?

Příslušnost prvku x z univerzální množiny X k množině A lze v klasické teorii množin popsat *charakteristickou funkcí* $\mu_A : X \to \{0,\ 1\}$:

$$\mu_A = \begin{cases} 1 & \text{pro } x \in A, \\ 0 & \text{pro } x \notin A. \end{cases}$$

V teorii fuzzy množin se používá zobecněná charakteristická funkce (také zvaná *funkce příslušnosti*)

$$\mu_A : X \to \langle 0,\ 1 \rangle.$$



Obrázek 1.1. Funkce příslušnosti μ_A z rovnice (1).

Pokud nyní označíme věk člověka jako x a množina A bude reprezentovat věk, do kterého je člověk mladý, mohli bychom tento pojem definovat mnohem uspokojivěji třeba takto:

$$\mu_A = \begin{cases} 1 & \text{pro } x \in \langle 0,\ 20), \\ \frac{50-x}{30} & \text{pro } x \in \langle 20, 50) \\ 0 & \text{pro } x \in \langle 50, +\infty). \end{cases} \tag{1}$$

Tedy do 20 let bychom člověka označili jako jasně mladého, pak by se do 50 let příslušnost člověka k množině mladých lidí postupně snižovala a nakonec člověka staršího 50 let bychom už do množiny mladých lidí nezařadili. Obdobně bychom mohli zavést množinu B, která by reprezentovala věk, od kterého už je člověk *starý*. Nemuseli bychom se však nutně držet toho, že $\mu_B = 1 - \mu_A$. Mohli bychom třeba definovat, že 35letý člověk přísluší z 0,5 k množině mladých lidí, ale pouze z 0,1 k množině starých lidí.

Dalším ze způsobů popisu fuzzy množiny je popis pomocí *systému řezů*

$$\mathcal{R}_A : \langle 0, 1\rangle \to \mathcal{P}(X).$$

Ten každému $\alpha \in \langle 0, 1\rangle$ přiřadí tzv. α-řez:

$$\mathcal{R}(\alpha) = \mu_A^{-1}(\langle \alpha, 1\rangle) = \{x \in X : \mu_A(x) \geq \alpha\}$$

Poznamenejme, že množina μ_A^{-1} je *ostrá* (tj. klasická, není fuzzy). Reprezentace fuzzy množiny systémem řezů se také nazývá *horizontální reprezentace*, reprezentace funkcí příslušnosti se nazývá *vertikální reprezentace*.

1.1.1 Fuzzy konjunkce

K výrokovým a množinovým operacím z klasických teorií zavádí fuzzy teorie jejich fuzzy protějšky. Pro účely práce se omezíme pouze na definici pojmu *fuzzy konjunkce*. Fuzzy konjunkce je binární operace $\wedge_\bullet : \langle 0, 1\rangle^2 \to \langle 0, 1\rangle$ splňující následující axiomy pro všechna $\alpha, \beta, \gamma \in \langle 0, 1\rangle$ [4, s. 30]:

$$\alpha \wedge_\bullet \beta = \beta \wedge_\bullet \alpha \qquad \text{(komutativita)}$$

$$\alpha \wedge_\bullet (\beta \wedge_\bullet \gamma) = (\alpha \wedge_\bullet \beta) \wedge_\bullet \gamma \qquad \text{(asociativita)}$$

$$\beta \leq \gamma \Rightarrow \alpha \wedge_\bullet \beta \leq \alpha \wedge_\bullet \gamma \qquad \text{(monotonie)}$$

$$\alpha \wedge_\bullet 1 = \alpha \qquad \text{(okrajová podmínka)}$$

Tečka u znaménka pro klasickou konjunkci naznačuje, že ve fuzzy logice může tyto podmínky splňovat více operací, které lze rozlišovat např. indexem. Fuzzy konjunkce se také jinak označují jako *trojúhelníkové normy* (*t-normy*). Níže jsou uvedeny tři jejich příklady:

- standardní konjunkce:

$$\alpha \wedge_S \beta = \min(\alpha, \beta),$$

- Łukasiewiczova konjunkce:

$$\alpha \wedge_L \beta = \begin{cases} \alpha + \beta - 1 & \text{pro } \alpha + \beta - 1 > 0, \\ 0 & \text{jinak,} \end{cases}$$

- drastická konjunkce:

$$\alpha \wedge_D \beta = \begin{cases} \alpha & \text{pro } \beta = 1, \\ \beta & \text{pro } \alpha = 1, \\ 0 & \text{jinak.} \end{cases}$$

1.2 Teorie a terminologie

Tato sekce shrnuje teorii a vysvětluje termíny, které budou využity při popisu implementovaného algoritmu. Základním pojmem, se kterým zachází celá práce, je *monoid s lineárním uspořádáním*, dále v práci zkracovaný jako *tomonoid* (z anglického *totally ordered monoid*).

Monoid je množina M, na níž je definována binární operace $\odot : M \times M \to M$. Tuto dvojici zapisujeme jako $(M, \odot)$ a vyznačuje se následujícími vlastnostmi:

- $\odot$ je asociativní, tj. $\forall\ a, b, c \in M : (a \odot b) \odot c = a \odot (b \odot c)$
- existuje v ní neutrální prvek, tj. $\exists n \in M\ \forall x \in M : x \odot n = n \odot x = x$.

Příkladem monoidu je operace sčítání přirozených čísel $(\mathbb{N}, +)$ nebo násobení matic velikosti $n \times n$ $(\mathbb{R}^{n \times n}, \cdot)$. Druhý příklad ukazuje, že s výjimkou operací zahrnujících neutrální prvek (zde jednotková matice příslušných rozměrů) nelze od binární operace definované na monoidu očekávat komutativitu. Dále v práci bude neutrální prvek monoidu značen jako 1, neboť použitá operace $\odot$ bude mít multiplikativní charakter.

Lineární (neostré) uspořádání na množině M je binární relace R, která je:

- reflexivní $(\forall\ a \in M : aRa)$
- antisymetrická $(\forall\ a, b \in M : aRb\ \wedge\ bRa\ \Rightarrow\ a = b)$
- tranzitivní $(\forall\ a, b, c \in M : aRb\ \wedge\ bRc\ \Rightarrow\ aRc)$
- trichotomická $(\forall\ a, b \in M : aRb \vee bRa)$.

Trichotomie zaručuje, že každý prvek množiny M lze v rámci R porovnat s jakýmkoliv jiným, čímž se lineární uspořádání liší od obecnějšího *částečného uspořádání*. Záměnou reflexivity za ireflexivitu $(\forall\ a \in M : \neg\ aRa)$ je definováno *ostré* lineární uspořádání. Typickým příkladem lineárního uspořádání je relace „$\leq$“ (menší nebo rovno) na základních číselných oborech $(\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R})$.

Lineární uspořádání $\leq$ nazveme *kompatibilním* s operací $\odot$, pokud pro všechna $a, b, c \in M$ platí $a \leq b \Rightarrow (a \odot c \leq b \odot c\ \wedge\ c \odot a \leq c \odot b)$. V takovém případě nazveme operaci $\odot$ *monotónní* vůči $\leq$. Příkladem monoidu s kompatibilním lineárním uspořádáním je násobení přirozených čísel $(\mathbb{N}; \leq, \cdot, 1)$.

Tomonoidem budeme zvát monoid $(S, \odot)$ s kompatibilním lineárním uspořádáním $\leq$ a neutrálním prvkem 1 – značeno jako $(S; \leq, \odot, 1)$. Práce se bude dále zabývat především konečnými *negativními* (tj. 1 je největším prvkem S) tomonoidy. Nebude-li uvedeno jinak, bude se při použití pojmu tomonoid předpokládat také splnění těchto dvou vlastností.

Jak bylo předvedeno výše, tomonoidy nemusí být nutně komutativní (pro všechna $a, b \in M$ platí $a \odot b = b \odot a$), ovšem pokud jsou, představují fuzzy konjunkci a korespondují tedy s *diskrétními* trojúhelníkovými normami. Nejmenší (*triviální*) tomonoid obsahuje pouze prvek 1.

0	***x***	***y***	***z***	**1**	
0	x	y	z	1	**1**
0	0	x	x	z	***z***
0	0	0	0	y	***y***
0	0	0	0	x	***x***
0	0	0	0	0	**0**

Obrázek 1.2. Reprezentace pětiprvkového tomonoidu S s prvky 0, x, y, z, 1 pomocí S^2. Jednotlivé buňky tabulky představují výsledek operace $\odot$, přičemž levý operand je v řádku a pravý ve sloupci – výsledek operace $z \odot y = x$ je tedy v 2. řádku a 3. sloupci. Je evidentní, že tomonoid reprezentovaný touto tabulkou není komutativní ($z \odot y \neq y \odot z$).

1.2.1 Reprezentace tomonoidů

Pro zacházení s tomonoidem $(S; \leq, \odot, 1)$ je v našem případě vhodnější místo práce přímo nad samotnou množinou S uvažovat jeho reprezentaci nad kartézským součinem $S \times S \rightarrow S^2$. Konkrétněji pro všechny dvojice prvků (a, b), $(c, d) \in S^2$ je definována relace $\sim$ předpisem

$$(a, b) \sim (c, d) \Leftrightarrow a \odot b = c \odot d.$$

Relaci $\sim$ nazveme vrstevnicovou ekvivalenci. Znakem $\trianglelefteq$ pak označme uspořádání S^2 po komponentách, tedy:

$$\forall a, b, c, d \in S : (a, b) \trianglelefteq (c, d) \Leftrightarrow (a \leq b \wedge c \leq d)$$

(toto je na rozdíl od relace $\leq$ pouze částečné uspořádání). Nechť $\sim$ je ekvivalence na S^2 splňující následující podmínky[2, s. 2–3]:

(i) pro všecha $a, b, c, d, e \in S$: $(a, b) \sim (1, d)$ a $(b, c) \sim (1, e)$ implikuje $(d, c) \sim (a, e)$ (tato podmínka souvisí s asociativitou tomonoidu),
(ii) pro všechna $a, b \in S$ existuje právě jedno $c \in S$ takové, že $(a, b) \sim (1, c) \sim (c, 1)$,
(iii) pro všechna $a, b, c, d, a', b', c', d' \in S$:, $(a, b) \sim (a', b') \trianglelefteq (c, d) \sim (c', d') \trianglelefteq (a, b)$.

V takovém případě nazveme $(S^2; \trianglelefteq, \sim, (1, 1))$ *systémem vrstevnic tomonoidu* (angl. *tomonoid partition*). Mezi tomonoidy a jejich systémem vrstevnic existuje vzájemně jednoznačné zobrazení[2, s. 3]. Pro jednotlivé třídy ekvivalance na systému vrstevnic tomonoidu budeme dále používat místo zápisu $(a, b) \sim (1, c) \sim (c, 1)$ zkráceně $(a, b) \sim c$.

1.2.2 Reesovské koextenze

Na úvod sekce zavedeme k neutrálnímu prvku 1 další tři význačné prvky tomonoidu:

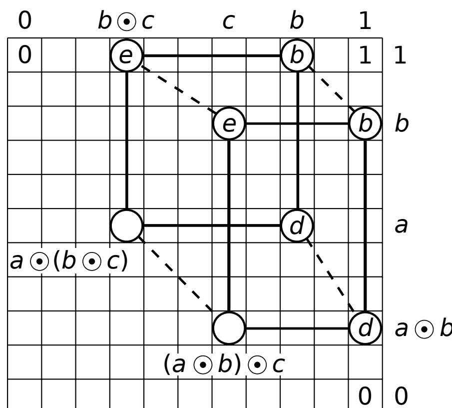
- **nula** (0) – nejmenší prvek tomonoidu,
- **atom** (α) – druhý nejmenší prvek tomonoidu,
- **koatom** (κ) – druhý největší prvek tomonoidu.

Vzhledem k monotónnosti operace $\odot$ platí $a \odot 0 = 0 \odot a = 0$ pro všechna $a \in S$.

Kongruence na monoidu $(M, \odot, 1)$ je relace ekvivalence $\approx$ na M (tj. $\approx \subseteq M \times M$), která pro všechna $a, b, c, d \in M$ splňuje:

$$a \approx c \wedge b \approx d \Rightarrow a \odot b \approx c \odot d.$$

Kongruence tomonoidu S je relace ekvivalence $\approx$ taková, že (i) $\approx$ je kongruence na monoidu S, (ii) každá z tříd ekvivalence $\approx$ je *konvexní*[1, s. 4]. Podmnožina C



Obrázek 1.3. Geometrická interperetace požadavku (i) ze sekce 1.2.1. Vytvořme nad tabulkou výsledků dva obdélníky tak, aby se jeden dotýkal horní hrany a druhý prvé hrany tabulky. Pokud dvojice jejich pravých horních vrcholů, pravých dolních vrcholů a levých horních vrcholů leží ve stejné třídě ekvivalence, potom i dvojice levých dolních vrcholů musí ležet ve stejné třídě ekvivalence. Toto je důsledek asociativity tomonoidů a jeho interpretace souvisí s tzv. *Reidemeisterovou podmínkou* [2, s. 4][5, s. 12].

částečně uspořádané množiny je konvexní, pokud z předpokladu $a, c \in C$ a $a \leq b \leq c$ vyplývá $b \in C$.

Třídu ekvivalence kongruence $\approx$ obsahující prvek $a \in M$ značíme $[a]_\approx$, množinu všech tříd ekvivalence pak $\langle M \rangle_\approx$. Na $\langle M \rangle_\approx$ můžeme zavést binární operaci $\odot_\approx$ předpisem:

$$[a]_\approx \odot_\approx [b]_\approx = [a \odot b]_\approx$$

Struktura $(\langle M \rangle_\approx, \odot_\approx, [1]_\approx)$ je opět monoidem a nazývá se kvocientem monoidu M podle kongruence $\approx$ (značeno $M/\approx$).

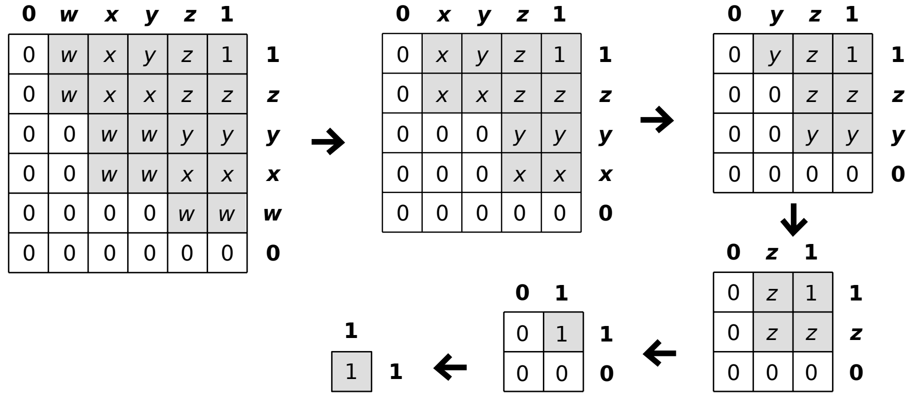
Operaci indukovanou $\odot_\approx$ na $\langle S \rangle_\approx$ označíme opět jako $\odot$. Dále definujeme lineární uspořádání $\leq$ na $\langle S \rangle_\approx$ tak, že pro $a, b \in S$ platí $[a]_\approx \leq [b]_\approx$ pokud $a \approx b$ nebo $a < b$. Množina $(\langle S \rangle_\approx; \leq, \odot_\approx, [1]_\approx)$ je opět tomonoidem a nazývá se tomonoidovým kvocientem vůči $\approx$. Jedním z typů kongruence tomonoidu, který bude dále využit, je tento:

- Nechť $(S; \leq, \odot, 1)$ je tomonoid a $q \in S$. Pro $a, b \in S$ definujeme $a = b$, pokud $a \approx b$ nebo $a, b \leq q$. Potom $\approx_q$ je kongruence tomonoidu[1, s. 4].

Jinak řečeno, všechny prvky tomonoidu S menší než q jsou sloučeny do jedné třídy ekvivalence, zatímco ostatní prvky zůstanou jedinými prvky svých vlastních tříd. Jak je ukázáno níže, taková kongruence souvisí s pojmem *ideálu pologrupy*.

$(M, \odot)$ je pologrupa, pokud $\odot$ je asociativní binární operace. Na rozdíl od monoidu tedy v pologrupě není vyžadována existence neutrálního prvku (ale ani zakázána, tudíž každý monoid je i pologrupou). Množina $I \subset M$ je ideálem pologrupy M, pokud pro všecha $a \in I$, $b \in M$ platí $(a \odot b) \in I$ a $(b \odot a) \in I$ (tedy pokud je jeden z operandů prvkem ideálu, pak je i výsledek operace prvkem ideálu). Aplikací těchto pojmů na tomonoid S s prvkem $q \in S$ nahlédneme, že množina $\{a \in S \mid a \leq q\}$, odpovídající „sloučené“ třídě ekvivalence z výše definované relace $\approx_q$, je díky negativitě tomonoidu ideálem pologrupy S.

Relace $\approx_q$ na tomonoidu S je tzv. *reesovskou kongruencí* podle q. Příslušný kvocient označíme S/q a nazveme ho *reesovský kvocient* S podle q. Reesovský kvocient tomonoidu je opět tomonoid, který vzniká sloučením všech prvků menších nebo rovných q do 0[6, 1]. Z libovolného tomonoidu lze tedy jednoduchým způsobem získat tomonoid s *menším* počtem prvků.



Obrázek 1.4. Řetězec reesovských kvocientů, ve kterém jsou z šestiprvkového tomonoidu postupně vytvářeny menší tomonoidy, dokud nevznikne triviální tomonoid. Každý menší tomonoid vzniká sloučením nuly a atomu do jednoho prvku („nové nuly“).

Tomonoid S nazveme *reesovskou koextenzí* tomonoidu S/q a speciálně koextenzi podle kvocientu α pak nazveme *jednoprvkovou reesovskou koextenzí* (tomonoidu S/α). Jak bylo předvedeno, získat z daného tomonoidu s menším počtem prvků jako jeho reesovský kvocient je snadné. V práci bude vysvětleno, jakým způsobem naopak získat z tomonoidu nový tomonoid s větším počtem prvků, a to tak, že k němu budou vytvořeny všechny jeho jednoprvkové reesovské koextenze.

1.2.3 Jednoprvkové reesovské koextenze

V této sekci bude vysvětlen postup, jakým k libovolnému tomonoidu sestrojit všechny jeho jednoprvkové reesovské koextenze. Důležitou roli při tom budou mít *idempotentní prvky*. Prvek a tomonoidu S je idempotentní, pokud $a \odot a = a$. V každém netriviálním tomonoidu existují alespoň dva idempotentní prvky – 1 a 0 (v triviálním pouze 1). Pokud v tomonoidu kromě těchto dvou prvků není žádný další idempotentní prvek, jedná se o tzv. *archimédovský* tomonoid (a opačně, pokud je v tomonoidu kromě 1 a 0 ještě jiný idempotentní prvek, jedná se o tzv. *nearchimédovský* tomonoid). Vlastnosti tomonoidů zaručují, že v archimédovském tomonoidu pro všechna $a, 0 < a < 1$ platí $a \odot a < a$.

Nyní zavedeme několik dalších množin, které mají vztah k tomonoidu $(S; \leq, \odot, 1)$ a jeho systému vrstevnic $(S^2; \trianglelefteq, \odot, (1,1))$. Při zacházení s S označíme jako $S^\star$ množinu všech prvků S kromě nejmenšího, tedy $S^\star = S \setminus \{0\}$, a jako $\bar{S}$ lineárně uspořádanou množinu $\bar{S} = S^\star \cup \{0, \alpha\}$, ve které pro všechna $a \in S^\star$ platí $0 < \alpha < a$. Množinu $\bar{S}$ nazveme *nulu zdvojující extenzí* tomonoidu S.

Zavedem dvě podmnožiny množiny $\bar{S}^2$:

$$\mathcal{P} = \{(a,b) \in \bar{S}^2 \mid a, b \in S^\star \text{ a existuje } c \in S^\star \text{ takové, že } (a,b) \sim c\}, \tag{2}$$

$$\mathcal{Q} = \bar{S}^2 \setminus \mathcal{P}. \tag{3}$$

Tedy $\mathcal{P}$ je množina všech dvojic, které se v tomonoidu vyhodnotí jinak, než 0, a naopak $\mathcal{Q}$ obsahuje všechny ostatní dvojice a navíc i dvojice s nově přidaným prvkem α.

Zvolme v tomonoidu dvojici nenulových idempotentních prvků $(\varepsilon_l, \varepsilon_r)$ a označme symbolem $\dot{\sim}$ nejmenší relaci ekvivalence nad $\bar{S}^2$, pro kterou platí následující pravidla[2, s. 5–6]:

(E1) $(a,b) \mathrel{\dot{\sim}} (c,d)$ pro všechna $(a,b), (c,d) \in \mathcal{P}$ taková, že $(a,b) \sim (c,d)$.

(E2) $(d,c) \mathrel{\dot{\sim}} (a,e)$ pro všechna $(a,b), (b,c) \in \mathcal{P}$ a $(d,c), (a,e) \in \mathcal{Q}$ taková, že $(a,b) \sim d$ a $(b,c) \sim e$.

0	w	x	y	z	1	
0	w	x	y	z	1	**1**
0	0	0	0	0	z	**z**
0	0	0	0	0	y	**y**
0	0	0	0	0	x	**x**
0	0	0	0	0	w	**w**
0	0	0	0	0	0	**0**

0	w	x	y	z	1	
0	w	x	y	z	1	**1**
0	w	x	y	z	z	**z**
0	w	x	y	y	y	**y**
0	w	x	x	x	x	**x**
0	w	w	w	w	w	**w**
0	0	0	0	0	0	**0**

0	w	x	y	z	1	
0	w	x	y	z	1	**1**
0	0	w	x	y	z	**z**
0	0	0	w	x	y	**y**
0	0	0	0	w	x	**x**
0	0	0	0	0	w	**w**
0	0	0	0	0	0	**0**

Obrázek 1.5. Příklady komutativních tomonoidů reprezentujících diskrétní trojúhelníkové normy pro šestiprvkovou množinu. **Vlevo:** drastická konjunkce. **Uprostřed:** standardní konjunkce. **Vpravo:** Łukasiewiczowa konjunkce – jediná archimédovská konjunkce, ve které pro všechna $x, y \in S, x \leq y$ existuje z takové, že $y \odot z = z \odot y = x$ (geometricky: v každém řádku/sloupci s indexem y jsou všechny prvky od 0 do y) [7].

(E3'a) $(a, e) \dot{\sim} 0$ pro všechna $a, b, c, e \in S^\star$ taková, že $(a, b) \in \mathcal{Q}$, $(b, c) \sim e$ a $c < \varepsilon_r$. Zároveň $(d, c) \dot{\sim} 0$ pro všechna $a, b, c, d \in S^\star$ taková, že $(b, c) \in \mathcal{Q}$,$(a, b) \sim d$ a $a < \varepsilon_l$.

(E3'b) $(a, e) \dot{\sim} (a, b)$ pro všechna $a, b, c, e \in S^\star$ taková, že $(a, b) \in \mathcal{Q}$, $(b, c) \sim e$ a $c \geq \varepsilon_r$. Zároveň $(d, c) \dot{\sim} (b, c)$ pro všechna $a, b, c, d \in S$ taková, že $(b, c) \in \mathcal{Q}$,$(a, b) \sim d$ a $a \geq \varepsilon_l$.

(E3'c) $(a, b) \dot{\sim} 0$ pro všechna $a, b, c > 0$ taková, že $(a, b), (b, c) \in \mathcal{Q}$,$a < \varepsilon_l$ a $c \geq \varepsilon_r$. Zároveň $(b, c) \dot{\sim} 0$ pro všechna $a, b, c > 0$ taková, že $(a, b), (b, c) \in \mathcal{Q}$,$a \geq \varepsilon_l$ a $c < \varepsilon_r$.

(E4'a) $(1, 0) \dot{\sim} (0, 1) \dot{\sim} (a, \alpha) \dot{\sim} (\alpha, b)$ pro všechna $a < \varepsilon_l$ a $b < \varepsilon_r$. Zároveň $(a, b) \dot{\sim} 0$ pro všechna $(a, b), (c, d) \in \mathcal{Q}$ taková, že $(a, b) \trianglelefteq (c, d) \dot{\sim} 0$.

(E4'b) $(1, \alpha) \dot{\sim} (\alpha, 1) \dot{\sim} (\varepsilon_l, \alpha) \dot{\sim} (\alpha, \varepsilon_r)$. Zároveň $(a, b) \dot{\sim} \alpha$ pro všechna $(a, b), (c, d) \in \mathcal{Q}$ taková, že $(a, b) \trianglerighteq (c, d) \dot{\sim} \alpha$

Strukturu $(\bar{S}^2; \trianglelefteq, \dot{\sim}, (1, 1))$ nazveme $(\varepsilon_l, \varepsilon_r)$-*ramifikací* $(S^2; \trianglelefteq, \sim, (1, 1))$.

Nyní ještě krátce okomentujeme jednotlivá pravidla. V bodě (E1) přeneseme do ramifikace nenulové výsledky z S^2 a v bodě (E2) si vynucujeme příslušnost asociovaných prvků v $\mathcal{Q}$ ke stejné třídě ekvivalence. Body (E3'a) a (E3'c) vyhledávají prvky z $\mathcal{Q}$, které budou kvůli monotonii nebo asociativitě vždy v třídě ekvivalence s 0, a to podle pozice jejich komponent vůči idempotentním prvkům. Tu zkoumá i bod (E3'b), který však jenom vyhledává, které prvky jsou spolu v relaci. Konečně, v bodech (E4'a) a (E4'b) se prvky (ε_l, α) a (α, ε_r) (a všechny větší z $\mathcal{Q}$) přiřadí do třídy ekvivalence s α, všechny menší pak do třídy ekvivalence s 0.

Nyní ještě zavedeme pojem *seskupení* (angl. *coarsening*) relace na S^2. Nechť $\sim_1$, $\sim_2$ jsou relace ekvivalence na S^2. Relace $\sim_2$ je seskupením $\sim_1$, pokud $\sim_1 \subseteq \sim_2$, tedy každá třída ekvivalence $\sim_2$ je sjednocením tříd ekvivalence $\sim_1$ [2, s. 5].

Pokud u $(\varepsilon_l, \varepsilon_r)$-ramifikace tomonoidu

(i) nedojde k tomu, že $(1, 0) \dot{\sim} (1, \alpha)$,
(ii) a provedeme dostatečné seskupení relace $\dot{\sim}$ do $\bar{\sim}$ tak, aby v každé třídě ekvivalence $\bar{\sim}$ byl právě jeden prvek typu $(1, c)$,

získáme systém vrstevnic tomonoidu $(\bar{S}^2; \trianglelefteq, \bar{\sim}, (1, 1))$, který je jednoprvkovou reesovskou koextenzí S^2 vůči idempotentním prvkům ε_l, ε_r. Navíc všechny jednoprvkové koextenze S^2 vůči ε_l, ε_r, pokud existují, vzniknou tímto způsobem[1, s. 17].

Nyní tedy dokážeme obrátit postup zmíněný v sekci 1.2.2 a k tomonoidu sestrojit nový tomonoid, který obsahuje o jeden prvek *více*.

Kapitola 2
Algoritmus

Ze závěrů zmíněných v sekci 1.2.3 a pravidel (E1)–(E4'b) byl formulován následující algoritmus[2, s. 7–9], který k tomonoidu S vygeneruje všechny jeho jednoprvkové reesovské koextenze. Po provedení prvního kroku (zkopírování nenulových výsledků rozšiřovaného tomonoidu) mohou být zbylé dvojice v jednom ze tří stavů – ve třídě ekvivalence s 0, ve třídě ekvivalence s α, nebo jsou dosud nepřiřazené.

Algoritmus pracuje se dvěma operacemi: $(a,b) \dot{\sim} z$ přiřadí dvojici do třídy ekvivalence se z ($z \in \{0, \alpha\}$) a $(a,b) \dot{\sim} (c,d)$ přířadí dvě dvojice do stejné třídy ekvivalence. Velmi důležité je zachování tranzitivy a monotonie operace $\odot$. Operace $(a,b) \dot{\sim} \alpha$ tedy např. vyžaduje, aby také následně proběhlo $(u,v) \dot{\sim} \alpha$ pro všechny $(u,v) \in \mathcal{Q} : (a,b) \trianglelefteq (u,v)$, a také pro všechny další prvky, které jsou s nimi ve stejné třídě ekvivalence. Obdobně $(a,b) \dot{\sim} (c,d)$ slučuje do jedné třídy ekvivalence i všechny další dvojice, které už jsou ve třídě ekvivalence s (a,b) nebo (c,d). Při běhu algoritmu se může stát, že nějaká dvojice bude přiřazena do třídy ekvivalence s 0 i s α. V takovém případě k dané dvojici idempotentních prvků reesovská koextenze neexistuje.

Vstup:

– $(S^2; \trianglelefteq, \sim, (1,1))$ ... systém vrstevnic tomonoidu $(S; \leq, \odot, 1)$
– $(\varepsilon_l, \varepsilon_r)$ ... dvojice nenulových idempotentních prvků tomonoidu

Výstup:

– $(\bar{S}^2; \trianglelefteq, \bar{\sim}, (1,1))$ ... jednoprvková Reesova koextenze $(S^2; \trianglelefteq, \sim, (1,1))$ vůči $(\varepsilon_l, \varepsilon_r)$

Algoritmus:

Inicializace

1. Nechť $\bar{S}$ je nulu zdvojující extenze S.
2. Nechť 0, α, a κ jsou nulou, atomem a koatomemem $\bar{S}$ a množiny $\mathcal{P}$ a $\mathcal{Q}$ jsou definovány podle (2) a (3).
3. Nechť $\dot{\sim}$ je relace ekvivalence na $\bar{S}^2$ (následující kroky ji budou definovat).

Část (E1)

4. Pro všechna $(a,b), (c,d) \in \mathcal{P}$:

   - přiřadíme $(a,b) \dot{\sim} (c,d)$, pokud $(a,b) \sim (c,d) \sim e$ pro $e \in \bar{S} \setminus \{0, \ \alpha\}$

Část (E2)

5. Pro všechna $(a,b), (b,c) \in \mathcal{P}$:

   - nechť pro $d \in \bar{S}$ platí $(a,b) \sim d$,
   - nechť pro $e \in \bar{S}$ platí $(b,c) \sim e$,
   - potom $(a,e) \dot{\sim} (d,c)$.

Části (E4'a) a (E4'b)

6. Přiřadíme $(1,0) \dot{\sim} (0,1) \dot{\sim} 0$.

7. Přiřadíme $(a, \alpha) \mathrel{\dot\sim} (\alpha, b) \mathrel{\dot\sim} 0$ pro $a < \varepsilon_l$ a $b < \varepsilon_r$.
8. Přiřadíme $(\varepsilon_l, \alpha) \mathrel{\dot\sim} (\alpha, \varepsilon_r) \mathrel{\dot\sim} \alpha$.

Část (E3'a)

9. Pro všechna $a \in \bar{S}$ taková, že $\alpha < a < \varepsilon_l$:

   - nechť $b \in \bar{S}$ je největším prvkem takovým, že $(a, b) \in \mathcal{Q}$,
   - nechť $c \in \bar{S}$ je největším prvkem takovým, že $c < \varepsilon_r$,
   - nechť pro $e \in \bar{S}$ platí $(b, c) \sim e$,
   - pokud $e > \alpha$, potom přiřadíme $(a, e) \mathrel{\dot\sim} 0$.

10. Pro všechna $c \in \bar{S}$ taková, že $\alpha < c < \varepsilon_r$:

   - nechť $b \in \bar{S}$ je největším prvkem takovým, že $(b, c) \in \mathcal{Q}$,
   - nechť $a \in \bar{S}$ je největším prvkem takovým, že $a < \varepsilon_l$,
   - nechť pro $d \in \bar{S}$ platí $(a, b) \sim d$,
   - pokud $d > \alpha$, potom přiřadíme $(d, c) \mathrel{\dot\sim} 0$

Část (E3'c)

11. Pro všechna $a \in \bar{S}$ taková, že $\varepsilon_l < a < 1$:

   - nechť $b \in \bar{S}$ je největším prvkem takovým, že $(a, b) \in \mathcal{Q}$,
   - nechť $c \in \bar{S}$ je největším prvkem takovým, že $(b, c) \in \mathcal{Q}$ a $c < \varepsilon_r$,
   - potom přiřadíme $(b, c) \mathrel{\dot\sim} 0$.

12. Pro všechna $c \in \bar{S}$ taková, že $\varepsilon_r < c < 1$:

   - nechť $b \in \bar{S}$ je největším prvkem takovým, že $(b, c) \in \mathcal{Q}$,
   - nechť $a \in \bar{S}$ je největším prvkem takovým, že $(a, b) \in \mathcal{Q}$ a $a < \varepsilon_l$,
   - potom přiřadíme $(a, b) \mathrel{\dot\sim} 0$.

Část (E3'b)

13. Pro všechna $b \in \bar{S}$ taková, že $\alpha < b < 1$:

   - nechť pro $e \in \bar{S}$ platí $(b, \varepsilon_r) \sim e$,
   - pokud $e < b$, tak:

     • pro všechna $a \in \bar{S}$ taková, že $\alpha < a < \varepsilon_l$ a $(a, b) \in \mathcal{Q}$, přiřadíme $(a, e) \mathrel{\dot\sim} (a, b)$

14. Pro všechna $b \in \bar{S}$ taková, že $\alpha < b < 1$:

   - nechť pro $d \in \bar{S}$ platí $(\varepsilon_l, b) \sim d$,
   - pokud $d < b$, tak:

     • pro všechna $c \in \bar{S}$ taková, že $\alpha < c < \varepsilon_r$ a $(b, c) \in \mathcal{Q}$, přiřadíme $(d, c) \mathrel{\dot\sim} (b, c)$

Seskupení

15. Nechť $\mathrel{\bar\sim} := \mathrel{\dot\sim}$.
16. Všem dvojicím $(a, b) \in \bar{S}^2$, které zůstaly nepřiřazené k 0 nebo k α, libovolně přiřadíme $(a, b) \mathrel{\bar\sim} 0$ nebo $(a, b) \mathrel{\bar\sim} \alpha$.

K získání všech jednoprvkových koextenzí tomonoidu opakujeme algoritmus pro všechny možné dvojice jeho nenulových idempotentních prvků. Navíc můžeme

vytvořit ještě jednu (nearchimédovskou) koextenzi tak, že provedeme pouze první dva kroky a následně přiřadíme:

- $(1,0) \mathrel{\bar{\sim}} (0,1) \mathrel{\bar{\sim}} 0$,
- $(a,b) \mathrel{\bar{\sim}} (c,d)$ pro všechna $(a,b),(c,d) \in \mathcal{P}$ taková, že $(a,b) \sim (c,d) \sim e$, $e \in \bar{S} \setminus \{0,\ \alpha\}$,
- $(\alpha,\alpha) \mathrel{\bar{\sim}} \alpha$.

Pokud chceme generovat pouze komutativní tomonoidy, je třeba ještě za krok 5 přidat podmínku $(a,b) \mathrel{\dot{\sim}} (b,a)$ pro všechna $a,b \in \bar{S}$.

2.3 Poznámky

Ukazuje se, že k některým dvojicím idempotentních prvků nelze sestrojit jednoprvkové reesovské koextenze. Je otevřenou otázkou, zdali o sestrojitelnosti koextenze lze rozhodnout ještě před vykonáním algoritmu.

V části (E2) lze, označíme-li nejmenší idempotentní prvek tomonoidu jako φ, vynechat všechny dvojice $(a,b),(b,c) \in \mathcal{P}$ takové, že $(a,b),(b,c) \trianglerighteq (\varphi,\varphi)$, protože potom nutně i dvojice $(a,e),(d,c) \trianglerighteq (\varphi,\varphi)$, a tudíž musí být i ony prvky $\mathcal{P}$.

Označíme-li jako n počet prvků rozšiřovaného tomonoidu S, potom je časová náročnost algoritmu od inicializace až do části (E3'a) polynomiální vůči n. Za předpokladu, že výsledek operace $a \odot b$ jsme schopni zjistit v konstantním čase, je nejnáročnější část (E2), ve které se při určování asociovaných dvojic pro $O(n^2)$ dvojic z $\mathcal{P}$ provede až $O(n)$ porovnání, tudíž její časová náročnost je $O(n^3)$.

Časová náročnost poslední fáze algoritmu, seskupení, je už ovšem exponenciálně závislá na n. Obecně, pokud po předchozích krocích algoritmu zůstane k nepřiřazených tříd ekvivalence, můžeme provést až 2^k jejich různých přiřazení do tříd ekvivalence 0 nebo α (ne všechny ovšem musí být validní, protože může dojít k nesplnění požadavku na monotonii). Nejhorší případ nastane při rozšiřování tomonoidu reprezentujícího drastickou t-normu (*drastický tomonoid* – viz Obrázek 1.5 Vlevo na str. 7), kdy vznikne $(n-2)^2$ nepřiřazených tříd ekvivalence (každá dvojice $(a,b) \in \bar{S}^2, 1 > a > \alpha, 1 > b > \alpha$). Lze ukázat [8], že počet p jednoprvkových archimédovských[1]) koextenzí drastického tomonoidu je potom

$$p = \begin{cases} \binom{2n}{n} & \text{(obecně)}, \\ 2^n & \text{(vyžadujeme-li komutativitu)}. \end{cases}$$

[1]) Ještě lze navíc vytvořit jednu nearchimédovskou koextenzi.

Kapitola 3
Implementace

Zadání práce vyžadovalo, aby byl algoritmus naprogramován v jazyce C nebo C++. Pro implementaci byl zvolen jazyk C++ ve standardu C++11, který – kromě mnoha dalších věcí – zavedl do standardní knihovny jazyka tzv. chytré ukazatele (angl. *smart pointers*) a hashovací tabulky.

Chytré ukazatele slouží ke snadnější správě dynamicky alokované paměti, v implementaci je používán `std::shared_ptr`, který umožňuje v programu používat vícenásobně ukazatel na objekt, přičemž ke smazání referencovaného objektu dojde automaticky (z pohledu programátora) ve chvíli, kdy zanikne poslední aktivně používaný ukazatel na něj.

Hashovací tabulky umožnily ve standardu C++11 vytvoření kontejnerů `std::unordered_map` a `std::unordered_set`, u kterých standardní operace (vyhledávání prvku, vložení nového prvku a smazání prvku) má *v průměru* konstantní časovou složitost. Původní implementace abstraktních datových typů množina (`std::set`) a asociativní pole (`std::map`) využívá binárních vyhledávacích stromů, kvůli kterým standardní operace na nich mají v průměru logaritmickou časovou složitost[9].

3.1 Generátor tomonoidů

Hlavním výstupem práce je generátor tomonoidů, který je spustitelný z příkazové řádky. Jeho výstupem je soubor se všemi vygenerovanými tomonoidy do předem zadané velikosti (syntaxe souboru je blíže popsána v sekci 3.4). Program pak nabízí i možnost načíst tomonoid z uloženého souboru a použít ho jako kořen pro generování (implicitním kořenem je triviální tomonoid). Před spuštěním lze také specifikovat, jestli vygenerované soubory mají být archimédovské a/nebo komutativní.

3.1.1 Seznam přepínačů

V této sekci jsou uvedeny přepínače programu. Obdobný seznam je zobrazen, pokud je při spuštění programu použit přepínač `-h` nebo `--help`.

```
-max <LEVELS>
```

Určuje hloubku prohledávání. Parametr `LEVELS` musí tedy být větší než 0. Implicitní hodnota je 5.

```
-a
```

Generovat pouze archimédovské tomonoidy.

```
-c
```

Generovat pouze komutativní tomonoidy.

```
-i <FILENAME> -id <ID>
```

Načíst tomonoid ze souboru `FILENAME` s identifikátorem `ID`. Ten je následně použit jako kořen pro generování.

```
-o <FILENAME>
```

Uložit vygenerované tomonoidy do souboru `FILENAME`. Pokud není použit, výstup se ukládá do souboru se jménem vygenerovaným podle času spuštění ve složce *output*.

```
-odir <DIRNAME>
```

Uložit výstup s generickým jménem do složky `DIRNAME`.

```
-multi [<THREADS>]
```

Povolí běh programu ve více vláknech. Nepovinným parametrem `THREADS` lze nastavit počet spuštěných vláken, musí tedy být nezáporným celým číslem (1 je validní, nicméně evidentně neužitečná hodnota). Pokud je parametr `THREADS` vynechán, nastaví se počet vláken na hodnotu vrácenou funkcí `std::thread::hardware_concurrency`.

```
-optsave
```

Povolí optimalizaci při ukládání. Pokud není použit, uloží se všechny nové výsledky, v opačném případě program vynechá ty, které jsou díky monotonii operace $\odot$ redundantní. Toto prohledávání je jen mírně pomalejší, proto lze v zásadě doporučit použít tento přepínač vždy.

Všechny přepínače jsou nepovinné a při spouštění programu nezáleží na jejich pořadí. Mohou být kombinovány zcela libovolně s výjimkou přepínačů `-odir` a `-o`, ze kterých může být vždy použit nejvýše jeden.

```
./tomonoid [-max <LEVELS>] [-a] [-c] [-i <FILENAME> -id <ID>]
           [(-o <FILENAME> | -odir <DIRNAME>)] [-multi [<THREADS>]]
           [-optsave]
```

Nyní bude uvedeno několik příkladů spuštění programu z příkazové řádky.

```
./tomonoid
```

Takto spuštěný program vygeneruje všechny tomonoidy do velikosti 7 (nejmenší tomonoid používaný v programu není triviální, ale dvouprvkový, více v sekci 3.2.1) a uloží je ve složce *output* do souboru s generickým jménem podle času spuštění. Program poběží pouze v jednom vlákně.

```
./tomonoid -max 8 -a -odir out -optsave -multi 3
```

Program vygeneruje pouze archimédovské tomonoidy do velikosti 10, které uloží v optimalizované formě ve složce *out* do souboru s generickým jménem. Program poběží ve třech vláknech.

```
./tomonoid -max 6 -a -c -o output/example.txt -optsave -i input/in.txt
 -id 7 -multi
```

Program vygeneruje archimédovské komutativní koextenze tomonoidu načteného ze souboru *input/in.txt* s ID 7, které mají nejvýše o 6 prvků více než zvolený kořen. Výstup bude optimalizován a uložen do souboru *output/example.txt*. Je povolena paralelizace, ovšem program sám určí, kolik vláken bude použito.

3.1.2 Objektový návrh

Tato sekce podává krátký popis tříd, které byly vytvořeny při implementaci algoritmu.

```
enum ElementType{BOTTOM, ORDINARY, TOP};
```

Výčet tří význačných typů prvků v tomonoidu – `BOTTOM` reprezentuje nejmenší prvek 0, `TOP` neutrální a největší prvek 1 a `ORDINARY` všechny ostatní.

```
class Element {
    unsigned int order;
    ElementType type;
public:
    static const Element bottom_element, top_element;
    ...
};
```

Třída reprezentující prvky tomonoidu. Kromě statických instancí *bottom_element* a *top_element* jsou všechny ostatní typu `ORDINARY`. Atribut `order` udává sestupné pořadí prvku od druhého největšího prvku (tj. největšího `ORDINARY` prvku). To je vhodnější pro indexaci, protože zůstává zachováno napříč tomonoidy různých velikostí.

```
class TableElement {
    std::shared_ptr<const Element> left, right;
    ...
};
```

Dvojice prvků z kartézského součinu $S \times S$ tomonoidu S. K referencování jednotlivých instancí třídy Element je použit výše zmíněný chytrý ukazatel.

```
class ElementCreator {
    std::vector<std::shared_ptr<const Element>>* elements_array;
};
```

Singleton, který před začátkem generování vytvoří potřebný počet instancí třídy Element (ten snadno určíme z velikosti kořenového tomonoidu a maximální hloubky prohledávání).

```
class Tomonoid {
public:
    std::vector<Tomonoid*>* calculateExtensions();
    typedef std::unordered_map<TableElement,
                                   std::shared_ptr<const Element> >
            results_map;
    ...
private:
    Tomonoid* previous;
    unsigned int size;
    results_map importantResults;
    bool atomNotIdempotent;
    ...
};
```

Nejdůležitější třída celého programu reprezentující tomonoid. Většina běhu programu probíhá na instancích této třídy v těle metody `calculateExtensions`, která generuje všechny jednoprvkové reesovské koextenze daného tomonoidu. Nově vygenerované výsledky operace $\odot$ jsou ukládány v atributu `importantResults`. Každý tomonoid si také

udržuje ukazatel na tomonoid, ze kterého vznikl (tedy na svůj reesovský kvocient S/α, viz sekci 1.2.2). Implementace algoritmu a způsob ukládání výsledků v rámci běhu programu jsou podrobněji vysvětleny v následující sekci.

3.2 Implementace algoritmu

Tato sekce popisuje, jakým způsobem byl formulovaný algoritmus převeden do kódu v jazyce C++. Především je objasněno, jaké kroky skrývá výše zmíněná metoda `calculateExtensions` třídy Tomonoid. Nejdříve bude ovšem vysvětleno, jakým způsobem udržuje program v paměti vygenerované výsledky operace v jednotlivých tomonoidech.

3.2.1 Reprezentace tomonoidů v programu

Na úvod se vraťme k obrázku 1.4 na straně 6, který zobrazuje řetězec reesovských kvocientů vedoucích od tomonoidu se šesti prvky až k triviálnímu tomonoidu. Poznamenejme, že jiným způsobem z původního tomonoidu k triviálnímu dojít nelze, nejvýše může být do jeho ideálu zahrnuto více prvků, čímž by se některé kroky přeskočily. Tuto možnost nicméně nebudeme brát v úvahu, protože algoritmus vychází z předpokladu, že vždy vytváříme jednoprvkové koextenze. Každý netriviální tomonoid je tedy jednoprvkovou koextenzí jediného menšího tomonoidu, který můžeme označit jako jeho rodiče. Tomonoidy tím pádem mají stromovou strukturu.

Program této struktury vygenerovaných tomonoidů využívá ke snížení paměťové náročnosti ukládání výsledků a používá k tomu trojici atributů `previous`, `importantResults` a `atomNotIdempotent`. Pokud platí $\alpha \odot \alpha \sim \alpha$, potom je `atomNotIdempotent` nastaven na *false* a to k reprezentaci nových výsledků zcela postačuje. Pokud α není idempotentní, pak jsou do `importantResults` uloženy všechny dvojice $(a, b) \sim \alpha$ (tedy instance třídy TableElement). Výsledky pro $(c, d) \sim e$, kde $e > \alpha$, se zjistí rekurzivně z předchozího tomonoidu referencovaného ukazatelem `previous`. Algoritmus pro zjištění výsledku pro dvojici (a, b) na tomonoidu S je následující:

1. Pokud je $(a, b) \sim c$ v `importantResults`, vrať c, jinak
2. přiřaď do hodnoty d největší známý idempotentní prvek φ takový, že $(\varphi, \varphi) \trianglelefteq (a, b)$,
3. pokud je $a = \alpha$ nebo $b = \alpha$, vrať d,
4. jinak vrať maximum z d a výsledku (a, b) pro tomonoid `previous`.

Krok 2 obvykle znamená pouze kontrolu hodnoty `atomNotIdempotent`, pokud je *true*, provede se $d := 0$, jinak $d := \alpha$. Pouze v případě použití přepínače `-i` se u zvoleného kořenového tomonoidu ukládají i hodnoty ostatních idempotentních prvků a v tom případě může být do d přiřazena hodnota vyšší než α.

Díky tomuto rekurzivnímu přístupu lze v algoritmu přeskočit část (E1), ve které se kopírují výsledky z rozšiřovaného tomonoidu. Časová náročnost zjištění výsledku operace je pak lineární v závislosti na počtu prvků tomonoidu.

Vzhledem k tomu, že pro všechna $a \in S$ jsou už z definice známy výsledky operací $a \odot 0$, $0 \odot a$, $a \odot 1$ a $1 \odot a$, není nutné tyto případy ukládat v paměti. K popisu tomonoidu velikosti n tedy stačí znát jen nenulové výsledky nejvýše pro jeho $(n-2)^2$ dvojic `ORDINARY` prvků. Tímto způsobem také reprezentace tomonoidů v programu funguje, důsledkem toho je, že v rámci programu je nejmenším tomonoidem jediný existující dvouprvkový tomonoid $S = (0, 1)$.

0	w	x	y	z	1	
0	w	x	y	z	1	**1**
0	w	?	?	?	z	**z**
0	0	w	w	?	y	**y**
0	0	w	w	?	x	**x**
0	0	0	0	w	w	**w**
0	0	0	0	0	0	**0**

→

0	x	y	z	1	
0	x	y	z	1	**1**
0	x	x	?	z	**z**
0	0	?	?	y	**y**
0	0	0	x	x	**x**
0	0	0	0	0	**0**

→

0	y	z	1	
0	y	z	1	**1**
0	0	?	z	**z**
0	0	y	y	**y**
0	0	0	0	**0**

Obrázek 3.1. Příklad vyhledání výsledku $y \odot z$ v šestiprvkovém tomonoidu. Otazníkem jsou značeny výsledky, které jsou v daném tomonoidu neznámé.

3.2.2 Generování reesovských koextenzí

Tato sekce popisuje, jakým způsobem byly implementovány jednotlivé kroky algoritmu ukryté v metodě `calculateExtensions` třídy Tomonoid. Jak bylo zmíněno v předešlé sekci, díky zvolené reprezentaci tomonoidů začíná implementace až od části (E2). V té se určují asociované dvojice prvků z $\mathcal{Q}$, což je v programu reprezentováno tak, že k instanci třídy TableElement je přiřazena množina dalších instancí TableElement:

```
typedef std::map< TableElement, std::set<TableElement> >
        associated_mapset;
```

V této fázi se přiřazení $(a,e) \mathrel{\dot\sim} (d,c)$ z kroku 5 algoritmu provede tak, že do instance typu `associated_mapset` jsou vloženy jako klíče (a,e) i (d,c) a druhá dvojice je pak vždy přidána do objektu `std::set<TableElement>`.

Kroky 6–13 algoritmu (části (E4), (E3'a) a (E3'c)), ve kterých se určuje, které prvky musí nutně spadat do tříd ekvivalence s 0 nebo s α, jsou implementovány pomocí dvou polí sloupcových „zarážek“.

```
int atomCol[this->size - 1];
int zeroCol[this->size - 1];
```

Do těchto polí se ukládá `order` prvku, který v daném sloupci $\bar{S}^2$ tvoří hranici mezi nepřiřazenou a přiřazenou třídou ekvivalence.

Část (E3'b) je implementována obdobně jako krok 5. Po jejím provedení se iterací přes `associated_mapset` vytvoří sjednocením jednotlivé třídy ekvivalence $\dot\sim$, což je v programu reprezentováno jako množina množin.

```
typedef std::set<TableElement> equivalenceClass;
std::set<equivalenceClass> classes_set;
```

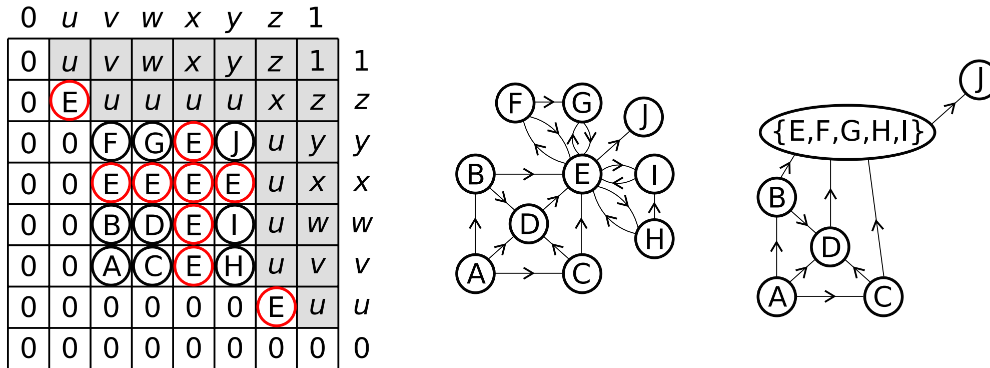
Nechť T_1 a T_2 jsou třídy ekvivalence $\dot\sim$. Definujme relaci $T_1 \lhd_G T_2$:

$$\text{Nechť } (a,b) \in T_1, (c,d) \in T_2 \text{ a } T_1 \neq T_2. \text{ Pak } T_1 \lhd_G T_2, \text{ pokud } (a,b) \trianglelefteq (c,d).$$

Z pohledu reprezentace tomonoidu pomocí $\bar{S}^2$, $T_1 \lhd_G T_2$, pokud některý z prvků T_1 leží vlevo dole od některého z prvků T_2. Pomocí této relace nyní můžeme formulovat pravidla nutná pro zachování monotonie ve chvíli, kdy některou třídu ekvivalence přiřadíme k 0 nebo α:

- Pokud $T_1 \lhd_G T_2$ a $T_1 \mathrel{\dot\sim} \alpha$, pak i $T_2 \mathrel{\dot\sim} \alpha$.
- Pokud $T_2 \lhd_G T_1$ a $T_1 \mathrel{\dot\sim} 0$, pak i $T_2 \mathrel{\dot\sim} 0$.

Z polí `atomCol` a `zeroCol` určíme, které třídy ekvivalence je potřeba sloučit do třídy ekvivalence s 0, resp. α. Z množiny `classes_set` pak tyto třídy ekvivalence vyřadíme,



Obrázek 3.2. **Vlevo:** Tomonoid s vyznačenými nepřiřazenými třídami ekvivalence před krokem 16 algoritmu. **Uprostřed:** Orientovaný graf G reprezentující relaci $\lhd_G$ na předchozím tomonoidu. **Vpravo:** Kondenzace grafu G do silně souvislých komponent. Ta je následně využita pro určení všech validních přiřazení jednotlivých tříd ekvivalence k 0 nebo k α.
Pozn.: Pro větší přehlednost byly v grafu vynechány hrany spojující vrcholy, mezi kterými vede orientovaná cesta přes jiný vrchol. Podle definice relace $\lhd_G$ by např. z vrcholu A správně měla vést hrana do všech ostatních vrcholů, nicméně i v přiřazovacím algoritmu postačuje, pokud z A vedou hrany pouze do B a C. Obecně ovšem relace $\lhd_G$ *není* tranzitivní.

čímž v ní zůstanou pouze nepřiřazené třídy ekvivalence, které je podle kroku 16 možno libovolně (za dodržení předchozích pravidel) přiřadit buď k 0, nebo k α.

Následující algoritmus předvádí, jakým způsobem program generuje všechna validní přiřazení zbylých tříd. Vychází z reprezentace vztahů mezi nepřiřazenými třídami ekvivalence pomocí orientovaného grafu G, jehož vrcholy jsou třídy ekvivalence a hrany vedou z T_i do T_j, pokud $T_i \lhd_G T_j$. Obrázek 3.2 ukazuje, že tento graf může být cyklický (a obecně ani nemusí být souvislý). Algoritmus však vyžaduje, aby vstupní graf byl acyklický, proto je ještě nutné graf G kondenzovat. V programu je k tomu použit Tarjanův algoritmus pro hledání silně souvislých komponent[10]. Množina T značí nepřiřazené třídy ekvivalence tomonoidu $\bar{S}^2$.

Vstup:
- G ... graf reprezentující vztahy nepřiřazených tříd ekvivalence podle relace $\lhd_G$.
- $f : T \to \{0, \alpha\}$ ... funkce přiřazující výsledek operace $\odot$ k volným třídám ekvivalence
- K ... množina reesovských koextenzí tomonoidu S

Výstup:
- $\bar{K}$ ... množina známých reesovských koextenzí doplněná o ty, které vznikly z grafu G

Algoritmus

1. Vytvoříme množinu V vrcholů grafu G, do kterých nevede žádná hrana.
2. Pokud je V prázdná, sestrojíme podle funkce f reesovskou koextenzi k, přiřadíme $\bar{K} = K \cup \{k\}$ a ukončíme algoritmus.
3. Vybereme libovolný vrchol $v \in V$ a vytvoříme množinu vrcholů U:

$$U = \{u \in G \mid u = v \text{ nebo z } v \text{ vede orientovaná cesta do } u\}$$

4. Pro všechna $u \in U : f(u) = \alpha$.
5. Sestrojíme graf odebráním všech vrcholů v U: $G_1 = G \setminus U$.
6. Provedeme algoritmus pro vstup (G_1, f, K) a jeho výstup označíme K_1.
7. Sestrojíme graf odebráním vrcholu v: $G_2 = G \setminus v$.
8. $f(v) = 0$.

9. Provedeme algoritmus pro vstup (G_2, f, K) a jeho výstup označíme K_2.
10. $\bar{K} = K \cup K_1 \cup K_2$.

3.3 Paralelizace výpočtu

Kvůli časové náročnosti algoritmu bylo dalším úkolem této práce navrhnout způsob, jakým generování tomonoidů paralelizovat. K tomuto účelu byla využita další novinka v standardu C++11, třídy `std::thread` a `std::mutex`[11]. Tyto třídy byly do standardu zahrnuty pro lepší přenositelnost aplikací mezi různými platformami a poskytují programátorovi vyšší míru abstrakce, než např. v rámci Unixových systémů obvyklé používání funkcí knihovny `pthread` vycházející ze standardu *POSIX Threads*[12][1]).

Program paralelizuje generování tomonoidů tak, že každému vláknu je vždy přiřazen jeden tomonoid, na kterém je následně zavolána metoda `calculateExtensions`. Všechny vygenerované koextenze daného tomonoidu jsou pak uloženy do prioritní fronty, ze které si vlákno po skončení metody `calculateExtensions` vyžádá další tomonoid k rozšíření. Pro kontrolu paralelizace byly vytvořeny následující struktury.

```
struct TomoCount {
  Tomonoid* tomo;
  TomoCount* previous = NULL;
  unsigned int id;

  inline bool operator<(TomoCount& other)
  {
     return this->tomo->getSize() < other.tomo->getSize() ?
            true :
            this->id > other.id;
  }
};
```

V této struktuře se uloží ukazatel na Tomonoid, který bude dále rozšiřován, a na strukturu reprezentující jeho rodiče. Také je v ní uloženo unikátní ID tomonoidu, které je použito později při jeho ukládání (ID rodiče se zjistí v atributu `previous`). Také je definován operátor $<$, který je využit při zařazování do prioritní fronty.

```
struct NextCall
{
  int level;
  TomoCount* currentTomo;
  ...
  inline bool operator<(NextCall& other)
  {
    return *(this->currentTomo) < *(other.currentTomo);
  }
};

std::priority_queue<NextCall*> tomonoid_queue;
```

Této struktuře se přiřadí ukazatel na TomoCount a také aktuální hloubka prohledávání. Je na ní opět definován operátor $<$, který jen odkazuje na obdobnou metodu struktury

[1]) Na POSIXových platformách jsou nicméně funkce knihovny `pthread` volány třídou `std::thread` interně.

`TomoCount`. Podle tohoto operátoru jsou instance `NextCall` zařazovány do prioritní fronty `tomonoid_queue` – ta pak ve chvíli, kdy si od ní některé z vláken vyžádá další tomonoid, předá ke zpracování ten aktuálně největší. Podle definice operátoru v těchto stukturách je TomoCount větší, pokud jím reprezentovaný tomonoid má více prvků. V případě shodné velikosti je větší ten, který má menší ID (byl vytvořen dříve). Díky tomuto řazení se prioritní fronta chová podobně, jako zásobník používaný pro *prohledávání do hloubky* (angl. *depth-first search* – DFS).

3.3.1 Prohledávání stromu tomonoidů

Jak již bylo zmíněno, tomonoidy mají stromovou strukturu, neboť ke každému netriviálnímu tomonoidu lze najít právě jeden tomonoid, který je jeho reesovským kvocientem podle atomu. Tento strom můžeme procházet buď do šířky (angl. *breadth-first search* – *BFS*), což by v případě tomonoidů znamenalo (pokud má kořenový tomonoid n prvků), že nejdříve vytvoříme všechny tomonoidy velikosti $n+1$, poté všechny tomonoidy velikosti $n+2$ atd.; nebo do hloubky, tj. v každé úrovni se vybere jeden tomonid, který je dále rozšiřován, a jeho sourozenci přijdou na řadu, až když jsou vytvořeny koextenze ke všem jeho potomkům (a potomkům potomků atd.).

Vzhledem k tomu, že cílem programu je vygenerovat všechny tomonoidy do dané velikosti, je z pohledu výsledku jedno, jestli použijeme BFS nebo DFS. Vzhledem k exponenciálnímu nárůstu počtu tomonoidů mezi jednotlivými úrovněmi je však lepší vybrat méně paměťově náročnou variantu, kterou je v tomto případě DFS. Jednotlivé tomonoidy se totiž v implementaci odkazují na své rodiče, takže je nutné udržovat v paměti ke každému tomonoidu velikosti n všech $(n-1)$ jeho předchůdců. V případě BFS bychom tedy museli držet v paměti *všechny* dosud vygenerované tomonoidy. Alternativně bychom mohli počet tomonoidů v paměti snížit tím, že bychom rodiče mazali a jejich výsledky uložili do všech jejich potomků, nicméně tento přístup je ve skutečnosti paměťově ještě náročnější. Výsledek, který byl předtím uložen pouze v rodičovském tomonoidu, by nyní byl uložen ve *všech* jeho jednoprvkových koextenzích.

V případě ideálního DFS by k tomonoidu velikosti n stačilo mít v paměti právě pouze $(n-1)$ jeho předchůdců. Tohoto stavu nicméně v programu nemůže být docíleno ani při jeho spuštění v jediném vlákně, protože metoda `calculateExtensions` vrací všechny koextenze k danému tomonoidu. Kvůli tomu jsou v paměti také sourozenci všech předchozích tomonoidů.

Ve chvíli, kdy při prohledávání stromu dojde program do maximální povolené hloubky, postačí nový tomonoid pouze uložit do souboru a následně ho smazat. Obdobně můžeme z paměti vymazat tomonoid ve chvíli, kdy smažeme všechny jeho potomky. Tímto způsobem se tedy v paměti udržuje jen aktuálně rozpracovaná větev stromu, a to i v případě spuštění ve více vláknech. Ty sice v každé úrovni nadbytečně otevřou další větve, nicméně díky použití prioritní fronty dojde na jejich zpracování nejdříve ve chvíli, kdy je prozkoumán celý podstrom jednoho z jejich sourozenců.

3.3.2 Synchronizace

Důležitým aspektem vícevláknového programování je synchronizace přístupu ke sdíleným prostředkům a komunikace mezi vlákny. Synchronizace přístupu je nutná v případě, že vlákna mění obsah sdíleného prostředku. Tehdy je nutné sekce kódu, ve kterých se k prostředku přistupuje (tzv. *kritické sekce*), blokovat, aby v jednu chvíli mohlo prostředek měnit pouze jedno vlákno (a aby žádné vlákno prostředek nepoužívalo, když ho jiné mění).

K blokování kritických částí kódu byla v programu využita na počátku sekce zmíněná třída `std::mutex`. Ta vynucuje synchronizaci tak, že nad kritickou sekcí vytvoří tzv.

zámek. Ve chvíli, kdy nějaké vlákno chce přistoupit ke kritické sekci, musí zámek odemknout. To se mu povede, pokud žádné jiné vlákno ve stejnou chvíli nevykonává kód dané sekce. V opačném případě je vlákno blokováno (musí čekat) do doby, než druhé vlákno vykoná kód kritické sekce. Je tedy vhodné, aby mutexů bylo co nejméně a aby kritické sekce trvaly co nejkratší dobu, protože v opačném mohou být ostatní vlákna zbytečně blokována a výhody paralelizace se tím zmenší.

V rámci programu bylo potřeba synchronizovat vlákna v těchto situacích:

- při přístupu k prioritní frontě,
- při ukládání tomonoidů do souboru,
  - zapisování na disk je náročnou operací, proto se řetězce reprezentující tomonoid uloží nejdříve do tzv. bufferu
  - k zápisu do souboru na disku dojde až ve chvíli, kdy je buffer dostatečně zaplněný
  - obě tyto operace je však třeba synchronizovat s ostatními vlákny
- při přiřazování ID (k tomu je použit globální čítač).

Naopak není nutné synchronizovat přístupy k předchozím tomonoidům ve chvíli, kdy hledáme výsledek operace. Ten sice na jednom tomonoidu může vyhledávat více vláken najednou, nicméně jedná se pouze o čtení a tomonoid není nijak měněn. To je pro program zcela zásadní, protože nutnost synchronizace nad touto operací by ho mohla výrazně zpomalit – při použití DFS totiž všechna vlákna prohledávají stejnou větev tomonoidů.

3.4 Ukládání výsledků

Tomonoidy jsou ukládány do textových souborů podle pravidel podobných JavaScript Object Notation (JSON)[13]. Každý vygenerovaný tomonoid je objektem s několika atributy - počtem prvků (včetně 0 a 1), příznakem komutativity (booleanovsky 0/1), unikátním ID v rámci souboru, ID rodiče, výčtem idempotentních prvků a nakonec výčtem nových výsledků v uspořádaných trojicích [*levý operand*, *pravý operand*, *výsledek*] nebo dvojicích [*levý operand*, *pravý operand*], v tomto případě se za výsledek považuje atom příslušného tomonoidu. Z výčtu atributů je patrné, že při ukládání je využito faktu, že každý tomonoid (kromě triviálního) je jednoprvkovou reesovskou koextenzí rodičovského tomonoidu, na který je odkázáno. Pokud danému tomonoidu jeho předchůdce chybí (buď se jedná o triviální tomonoid, nebo byl kořenem při generování souboru), je ID předchozího tomonoidu 0 (tím pádem žádný tomonoid v sobouru nesmí mít ID 0).

Pro úsporu paměti jsou při ukládání vynechány názvy atributů, tudíž vygenerované soubory striktně vzato neodpovídají specifikaci JSON, nicméně převedení do formy odpovídající JSON gramatice je triviální. Na rozdíl od JSON je však kvůli tomu nutné dodržovat přesně jejich pořadí. Obdobně jako při generování jsou také vynechány výsledky v případech, kdy jedním z operandů je 0 nebo 1. Ostatním prvkům v tomonoidu velikosti n jsou přiřazena sestupně čísla od 1 do $(n-2)$, neboť oproti přirozenějšímu vzestupnému pořadí toto danému prvku přiřazuje stejné číslo v tomonoidech libovolné velikosti, a je tedy vhodnější pro jejich rekurzivní definice.

Dalším krokem pro úsporu paměti je ukládání pouze nenulových výsledků, ostatně prvku 0 není ani přiřazeno číslo pro jeho reprezentaci. Jako důsledek postačuje v každém nově vygenerovaném tomonoidu ukládat pouze dvojice $(a, b) \sim \alpha$, případně pouze $[\alpha]$, do pole idempotentních prvků, pokud $(\alpha, \alpha) \sim \alpha$. Ukládání výsledků ve formátu $[a,b,\alpha]$ je také validní, ale s výjimkou tomonoidu bez předchůdce, u kterého může být nutné

definovat i dvojice $(c, d) \sim e$ pro $e > \alpha$, je informace o výsledku operace zbytečná. Využít lze i monotonie operace $\odot$, díky které v případě, kdy pro dvě dvojice prvků platí $(a, b) \sim (c, d) \sim \alpha$ a $(a, b) \trianglelefteq (c, d)$, postačuje uložit pouze dvojici (a, b). Tím pádem je v každém sloupci (nebo řádku) uložena nejvýše jedna dvojice prvků, tudíž paměťová náročnost uložení *nových* výsledků v tomonoidu velikosti n je $O(n)$. [1])

0	w	x	y	z	1	
0	w	x	y	z	1	**1**
0	0	w	w	w	z	**z**
0	0	0	0	w	y	**y**
0	0	0	0	w	x	**x**
0	0	0	0	0	w	**w**
0	0	0	0	0	0	**0**

Obrázek 3.3. Multiplikační tabulka tomonoidu velikosti 6 a její možné reprezentace:
1. `{6,1,1,0,[],[[3,1]]}`,
2. `{6,1,1,0,[],[[1,3,4]]}`,
3. `{6,0,1,0,[],[[3,1],[1,3]]}`,
4. `{6,0,1,0,[],[[3,1,4],[2,1,4],[1,3,4],[1,2,4],[1,1,4]]}`.

Příklad 1 využívá všech možností, jak zmenšit velikost souboru, příklad 2 redundantně definuje výsledek operace, příklad 3 nevyužívá komutativity tomonoidu a příklad 4 opět redundantně definuje výsledky operace a nevyužívá ani její monotonie. Všechny reprezentace jsou nicméně validní.

Pozn.: V předchozím odstavci je diskutován způsob, jak co nejlépe ukládat vygenerované tomonoidy, nicméně i méně optimální zápisy jsou stále validní. Jako příklad lze uvést uložení nearchimedovského tomonoidu do pole s výsledky jako $[\alpha,\alpha]$ nebo dokonce $[\alpha,\alpha,\alpha]$, případně nevyužití monotonie, při kterém už však paměťová náročnost ukládání nových výsledků asymptoticky vzroste na $O(n^2)$.

Specifikace JSON vyžaduje ukládání znaků odpovídající normě Unicode. Pro ukládání tomonoidů jsou použity číslice a pět ze šesti strukturovacích znaků JSON – `{`, `}`, `[`, `]` a `,` (vynechána je `:`, která odděluje jména atributů od jejich hodnot, jak je však zmíněno výše, jména jsou v souboru vynechána). Při použití kodování UTF-8 je k uložení každého z těchto znaků zapotřebí 1 byte. Celkově je ovšem použito pouze 15 znaků, tudíž k uložení jednoho znaku postačují 4 bity. To ukazuje, že při použití modifikovaného kódování je možno velikost ukládaných souborů snížit o polovinu, navíc pro použití uloženého souboru ve formátu JSON je i bez toho nutný překlad, během kterého by se případně provedla i změna kódování ze modifikovaného na UTF-8. Modifikované kódování nicméně při ukládání tomonoidu není využito především kvůli tomu, aby i základní formát – obdobně jako formát JSON – zůstal čitelný pro člověka.

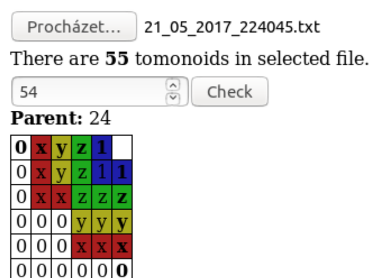
3.4.1 Zobrazování tomonoidů

Ačkoli je formát výstupu čitelný pro člověka, představa o tom, jak daný tomonoid vypadá, by se z výstupu tak jak je získávala jen stěží. Je tedy nutné ještě nějakým způsobem vygenerovaný soubor prezentovat.

[1]) Abychom však získali všechny výsledky v daném tomonoidu, musíme započítat i předchozí články řetězce, tudíž celková paměťová náročnost nutná k uložení všech výsledků daného tomonoidu je $O(n^2)$

Původním záměrem (nad rámec zadání práce) bylo vytvořit také grafické uživatelské rozhraní např. v knihovně Qt[1]), které by kromě zobrazování tomonoidů uložených v souboru dovolovalo také jednodušší spouštění generátoru. Bohužel, tento záměr v době odevzdání práce zůstal nenaplněn. Program je tedy spustitelný pouze z příkazové řádky.

K zobrazování tomonoidů lze nicméně použít internetový prohlížeč, protože prvotním (a původně pouze provizorním) řešením pro jejich prezentaci bylo vytvoření HTML stránky (`viewer.html`) se skriptem v jazyce JavaScript. V ní lze vybrat vygenerovaný soubor, který je převeden do validního formátu JSON (přidají se názvy atributů). Z něj se pomocí JavaScriptové funkce `JSON.parse` vytvoří objekty reprezentující jednotlivé tomonoidy. Po zadání ID do příslušného pole se pak v prohlížeči vykreslí tabulka výsledků operace v daném tomonoidu.



Obrázek 3.4. Screenshot zobrazení tomonoidu v internetovém prohlížeči.

3.5 Srovnání

Tato sekce porovnává novou vícevláknovou implementaci v jazyce C++ s referenční implementací v Pythonu[14]. Porovnávala se doba běhu při generování tomonoidů za těchto podmínek:

1. Generování archimédovských tomonoidů do velikosti 9
2. Generování archimédovských komutativních tomonoidů do velikosti 10
3. Generování všech tomonoidů do velikosti 8

Test probíhal na běžném notebooku s čtyřjádrovým procesorem Intel Core i3-2328M CPU o taktovací frekvenci 2,2 GHz, 4 GB operační paměti na OS Ubuntu 16.04. Výsledky jsou uvedeny níže. Každý z testovacích případů byl spuštěn desetkrát na implementaci v C++ (pětkrát v jediném vlákně a pětkrát ve čtyřech vláknech) a třikrát na implementaci v Pythonu. V tabulkách jsou uvedeny průměrné doby běhu programů.

tomonoidy		doba běhu [s]		
velikost	počet	C++, 1 vl.	C++, 4 vl.	Python
1–8	3933	0,4	0,2	4,3
1–9	58887	6,4	2,7	83,9

Tabulka 3.1. Doba běhu nové a referenční implementace při generování archimédovských tomonoidů do velikosti 9.

[1]) `https://www.qt.io/`

tomonoidy		doba běhu [s]		
velikost	počet	C++, 1 vl.	C++, 4 vl.	Python
1–9	3269	1,0	0,5	5,6
1–10	20656	9,2	3,6	48,8

Tabulka 3.2. Doba běhu nové a referenční implementace při generování komutativních archimédovských tomonoidů do velikost 10.

tomonoidy		doba běhu [s]		
velikost	počet	C++, 1 vl.	C++, 4 vl.	Python
1–7	3005	0,5	0,3	3,9
1–8	30125	8,5	3,6	58,4

Tabulka 3.3. Doba běhu nové a referenční implementace při generování obecných tomonoidů do velikosti 8.

Je evidentní, že ve všech případech je implementace v C++ výrazně rychlejší, a to i při nevyužití paralelizace. Nejmarkantnější rozdíl je při generování archimédovských tomonoidů, kdy je paralelizovaná verze zhruba 20krát rychlejší. V ostatních případech je rozdíl menší, nová implementace je v nich více než 10krát rychlejší.

Kromě doby běhu programu byla ještě srovnána velikost výstupních souborů. Ta je opět v případě nové implementace výrazně menší, nicméně je třeba podotknout, že implementace v Pythonu výstupní soubory nijak nekomprimuje – do souboru je vytisknuta tabulková reprezentace celého tomonoidu a navíc jsou k ní přidány další doplňující informace.

	velikost souboru [MB]	
případ	C++	Python
archim. do velikosti 9	1,9	80,7
archim. kom. do velikosti 10	0,7	32,1
obecný do velikosti 8	0,9	35,3

Tabulka 3.4. Porovnání velikostí výstupních souborů na testovacích příkladech.

Kapitola 4
Závěr

Cíle práce – naimplementovat v jazyce C/C++ algoritmus pro generování tomonoidů a navrhnout i jeho paralelizaci – byly splněny. Vytvořený program umožňuje generovat tomonoidy ve více vláknech, navíc je možné mu jako jeden z argumentů předat již vygenerovaný tomonoid, který je následně použit jako kořen při spuštění algoritmu.

V sekci Srovnání bylo předvedeno, že oproti referenční implementaci v jazyce Python je nová implementace v jazyce C++ na testovaných příkladech výrazně rychlejší, a to i v případě, kdy je k jejímu vykonání použito pouze jedno vlákno. Zároveň i výstupy programu uložené do souboru jsou u nové implementace výrazně menší, nicméně je třeba říct, že formát výstupu referenční implementace evidentně nebyl vytvořen za účelem výsledky nějakým způsobem komprimovat.

Práce podala také negativní odpověď na doposud otevřenou otázku, jestli lze uspořádat třídy ekvivalence, které zůstanou v algoritmu nepřiřazené i po kroku 15. Protipříkladem je obrázek 3.2 na straně 16. Ten nicméně také ukazuje, že není složité zbylé třídy ekvivalence sloučit tak, aby se na nich dalo vytvořit částečné uspořádání.

Zdrojové kódy implementace jsou dostupné i v repozitáři na serveru GitHub.[1])

[1]) https://github.com/kozakja4/thesis

Literatura

[1] PETRÍK, Milan a Thomas VETTERLEIN. Rees coextensions of finite, negative tomonoids. *Journal of Logic and Computation* . 2015, **27**(1), 337-356. DOI: 10.1093/logcom/exv047. ISSN 0955-792x.

[2] PETRÍK, Milan a Thomas VETTERLEIN. Algorithm for Generating Finite Totally Ordered Monoids. In: *Information Processing and Management of Uncertainty in Knowledge-Based Systems*. 2. Eindhoven: Springer, 2016, 532-543. DOI: 10.1007/978-3-319-40581-0_43. ISBN 978-3-319-40581-0. ISSN 1865-0929. Dostupné také z:
`http://link.springer.com/10.1007/978-3-319-40581-0_43`

[3] ZADEH, L.A. Fuzzy sets. *Information and Control*. 1965, **8**(3), 338-353. DOI: 10.1016/S0019-9958(65)90241-X. ISSN 00199958. Dostupné také z:
`http://linkinghub.elsevier.com/retrieve/pii/S001999586590241X`

[4] NAVARA, Mirko a Petr OLŠÁK. *Základy fuzzy množin*. Vyd. 2., přeprac. Praha: Nakladatelství ČVUT, 2007. ISBN 978-80-01-03668-6.

[5] ACZÉL, J. Quasigroups, nets, and nomograms. *Advances in Mathematics*. 1965, **1**(3), 383-450. DOI: 10.1016/0001-8708(65)90042-3. ISSN 00018708. Dostupné také z:
`http://linkinghub.elsevier.com/retrieve/pii/0001870865900423`

[6] REES, D. a P. HALL. On semi-groups. *Mathematical Proceedings of the Cambridge Philosophical Society*. 1940, **36**(04), 387-400. DOI: 10.1017/S0305004100017436. ISSN 0305-0041. Dostupné také z:
`http://www.journals.cambridge.org/abstract_S0305004100017436`

[7] MAYOR, Gaspar a Joan TORRENS. Triangular norms on discrete settings. In: *Logical, Algebraic, Analytic and Probabilistic Aspects of Triangular Norms*. Elsevier, 2005, s. 189. DOI: 10.1016/B978-044451814-9/50007-0. ISBN 9780444518149. Dostupné také z:
`http://linkinghub.elsevier.com/retrieve/pii/B9780444518149500070`

[8] PETRÍK, Milan a Thomas VETTERLEIN. Algorithm to generate the Archimedean, finite, negative tomonoids. *2014 Joint 7th International Conference on Soft Computing and Intelligent Systems (SCIS) and 15th International Symposium on Advanced Intelligent Systems (ISIS)*. IEEE, 2014, 42-47. DOI: 10.1109/SCIS-ISIS.2014.7044822. ISBN 978-1-4799-5955-6. Dostupné také z:
`http://ieeexplore.ieee.org/document/7044822/`

[9] Std::map. *Cppreference.com* [online]. [cit. 2017-05-25]. Dostupné z:
`http://en.cppreference.com/w/cpp/container/map`

[10] TARJAN, Robert. Depth-first search and linear graph algorithms. *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. IEEE, 1971, **1**(2), 114-121. DOI: 10.1109/SWAT.1971.10. ISSN 0097-5397. Dostupné také z:
`http://ieeexplore.ieee.org/document/4569669/`

[11] Thread support library. *Cppreference.com* [online]. [cit. 2017-05-25]. Dostupné z: `http://en.cppreference.com/w/cpp/thread`

[12] <pthread.h>. *The Open Group* [online]. [cit. 2017-05-25]. Dostupné z: `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`

[13] *ECMA-404: The JSON Data Interchange Format*. 1. Geneva: Ecma International, 2013. Dostupné také z: `http://www.ecma-international.org/publications/standards/Ecma-404.htm`

[14] Milan Petrík – Program to find elementary extensions of f.n. tomonoids. *AV ČR* [online]. Praha [cit. 2017-05-25]. Dostupné z: `http://uivtp.cs.cas.cz/~petrik/extensions.php`

Příloha A
Seznam zkratek

angl.	anglicky
BFS	prohledávání do šířky (angl. *Breadth-first search*)
DFS	prohledávání do hloubky (angl. *Depth-first search*)
ID	identifikátor
JSON	JavaScript Object Notation
SQL	Structured Query Language
tomonoid	monoid s lineárním uspořádáním (angl. *totally ordered monoid*)

Příloha B
Obsah přiloženého CD

Přiložené CD obsahuje následující soubory:

```
root
|_ Code ................................ složka se zdrojovými kódy
   |_ install.sh ....................... instalační skript
   |_ Tomonoid ......................... složka se zdrojovými kódy C++ implementace
|_ examples ............................ složka s uloženými příklady tomonoidů
|_ img ................................. složka s obrázky použitými v práci
|_ text.pdf ............................ elektronická verze textu
|_ toomo.js ............................ skript pro načítání tomonoidů ze souboru
|_ viewer.html ......................... prohlížeč tomonoidů
```