VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SEBEMODIFIKUJÍCÍ SE CELULÁRNÍ AUTOMATY

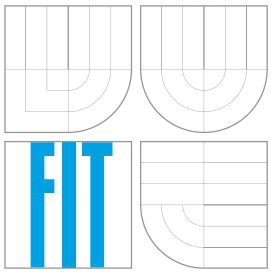
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETER SZABO

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SEBEMODIFIKUJÍCÍ SE CELULÁRNÍ AUTOMATY

SELF-MODIFYING CELLULAR AUTOMATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETER SZABO

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAL BIDLO, Ph.D.

BRNO 2014

Abstrakt

Tato práce se zabývá celulárními automaty s konceptem sebe-modifikace a jich porovnáním s konvenčními celulárními automaty. Pro tento účel jsme vytvořili simulátor, který také umožňuje samostatně definovat logiky umělé inteligence, generátoru čísel a statistického testu, které simulátor využívá. Následně jsou provedeny dva pokusy, které koncept sebe-modifikace demonstrují.

Abstract

This work deals with cellular automata with a concept of self-modification and their comparsion against regular cellular automatons. For this task we constructed a simulator, that lets us define the logic of artificial inteligence, number generator and statistical test, which are used by the automata, on their own. Consequently two experiments are carried out that demonstrate the concept of self-modification.

Klíčová slova

celulární automaty, sebe-modifikace, umělá inteligence, generátor čísel, statistické testování.

Keywords

cellular automata, self-modification, artificial inteligence, number generator, statistical testing.

Citace

Peter Szabo: Sebemodifikující se celulární automaty, bakalářská práce, Brno, FIT VUT v Brně, 2014

Sebemodifikující se celulární automaty

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Michala Bidla, Ph.D.. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Peter Szabo
31. července 2014

Poděkování

Touto cestou bych se chtěl poděkovat svému vedoucímu, Ing. Michalovi Bidlovi, Ph.D. za poskytnuté zdroje a rady, odborné vedení a za ochotu a čas, který mi věnoval. Také bych rád poděkoval rodině a blízkým za podporu, kterou mi při studiu a tvorbě této práce poskytli.

Obsah

1 Úvod **3**

2 Celulárny automat **4**
2.1 Štruktúra . . . 5
2.1.1 Bunka . . . 5
2.1.2 Pole buniek . . . 5
2.1.3 Okolie . . . 6
2.1.4 Pravidlo . . . 6
2.2 Klasifikácia . . . 7
2.3 Aplikácia . . . 7
2.3.1 Štúdium správania . . . 8
2.3.2 Generovanie čísel . . . 8

3 Seba-modifikujúci sa celulárny automat **10**
3.1 Seba-modifikácia . . . 10
3.2 Umelá inteligencia . . . 10
3.3 Heuristika . . . 11

4 Simulátor **12**
4.1 Špecifikácia požiadaviek . . . 13
4.2 Analýza a návrh . . . 14
4.2.1 Diskrétny simulátor . . . 15
4.2.2 Argumenty . . . 16
4.2.3 Log . . . 17
4.2.4 Automat . . . 18
4.2.5 Umelá inteligencia . . . 20
4.2.6 Generátor čísel . . . 21
4.2.7 Štatistický test . . . 22
4.3 Implementácia . . . 23
4.4 Testovanie . . . 23
4.5 Údržba . . . 23

5 Pokusy **24**
5.1 Pokus I . . . 24
5.1.1 Logika umelej inteligencie . . . 25
5.1.2 Logika generátoru čísel . . . 26
5.1.3 Logika štatistického testu . . . 26
5.1.4 Argumenty . . . 26

5.1.5 Výsledok . . . 27
5.1.6 Záver . . . 28
5.2 Pokus II . . . 28
5.2.1 Logika umelej inteligencie . . . 29
5.2.2 Logika generátoru čísel . . . 29
5.2.3 Logika štatistického testu . . . 30
5.2.4 Argumenty . . . 30
5.2.5 Výsledok . . . 31
5.2.6 Záver . . . 32

6 Záver **33**

A Obsah CD **36**

Kapitola 1

Úvod

Celulárne automaty sú známym konceptom z minulosti a sú opísané v nasledujúcej, druhej kapitole. Ich správanie je a bolo predmetom mnohých štúdií a v niektorých prípadoch je ich vývoj vypočítateľný vopred. Takýmto prípadom a prípadom, kedy sa celulárne automaty vyvíjajú nežiadaným spôsobom, môžme zabrániť použitím konceptu seba-modifikácie, ktorý je opísaný v kapitole tri.

Cieľom našej práce je takýto celulárny automat zostrojiť a vhodným spôsobom demonštrovať jeho vlastnosti. Pre tento účel nám poslúži simulátor, ktorý v kapitole štyri navrhneme a zostrojíme špeciálne pre túto úlohu. Ten v sebe bude porovnávať konvenčný celulárny automat a celulárny automat so seba-modifikáciou, pričom seba-modifikácia je proces, ktorý sa riadi pomocou umelej inteligencie. Výstup celulárneho automatu je potom upravený takzvaným generátorom čísel, ktorý odstráni nežiaduce časti skresľujúce výsledky. Končené čísla sú potom štatisticky testované a priemer tejto štatistiky je vypísaný ako výsledok. Tieto tri časti simulátoru – umelá inteligencia, generátor čísel a štatistický test – k svojmu chodu využívajú logiky, ktoré sú definovatelné v samostatných súboroch.

Aby však simulátor generoval výsledky, musíme ho najprv patričným spôsobom nastaviť a to pomocou vstupov, ktoré nám poskytuje. To v sebe zahŕňa klasické nastavenia celulárneho automatu, akými sú veľkosť okolia a použité pravidlo ale aj špecifikáciu logík, podľa ktorých sa bude riadiť umelá inteligencia, generátor čísel a štatistický test. Tieto pokusy ako aj ich výsledky a zhodnotenie sa nachádza v predposlednej kapitole, kapitole päť.

Posledná kapitola zhŕňa našu prácu, diskutuje výhody a nevýhody konceptu seba-modifikácie ako aj vymedzuje niektoré možné rozšírenia.

Kapitola 2

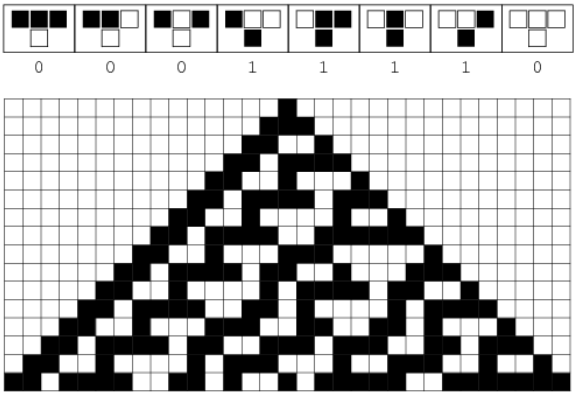
Celulárny automat

Prvé nápady spojené s celulárnymi automatmi sa objavili už v 40. rokoch dvadsiateho storočia, kedy sa John von Neumann snažil navrhnúť samoreplikujúci sa stroj.[10] Spolu so Stanislawom Ulamom, ktorý v tej istej dobe skúmal rast kryštálov a používal podobné metódy, v 50. rokoch vytvorili prvý model celulárneho automatu pre výpočet toku tekutín. Ich model považoval tekutinu za skupinu diskrétnych samostatných jednotiek – buniek, ktorých pohyb sa počítal vždy nezávisle od seba len na základe stavov okolitých buniek. Jednalo sa teda o diskrétny systém v priestore i čase.

V roku 1970 John Conway objavil jednoduchý dvoj-dimenzionálny, dvoj-stavový celulárny automat, ktorý vykazoval komplexné správanie (obrázok 2.1) aké môžeme pozorovať aj pri vývoji spoločenstiev živých organizmov. Neskôr sa tento celulárny automat stal známy ako *Game of Life (Hra života)*.

Obrázek 2.1: Celulárny automat Game of Life počas simulácie

Neskôr v roku 1981 sa celulárnymi automatmi začal zaoberať aj Stephen Wolfram, no narozdiel od svojich predchodcov sa zameral na jedno-dimenzionálne celulárne automaty s dvomi stavmi (tiež nazývané elementárne).[17] O dva roky neskôr publikoval svoje prvé zistenia týkajúce sa *pravidla 30* (obrázok 2.2), čo je takzvaný *Wolframov kód*. Tento celulárny automat preukazuje chaotické správanie a jeho vzory môžme pozorovať aj v prírode. Wolframovi sa neskôr podarilo dokázať prepojenie na štatistickú fyziku a fakt, že celulárne automaty je možné použiť na modelovanie komplikovaných fyzikálnych systémov.



Obrázek 2.2: Graficky znázornené pravidlo 30 a začiatok vývoja celulárneho automatu

Tri kľúčové vlastnosti celulárnych automatov:

- homogenita – pre všetky bunky platia rovnaké vlastnosti
- lokalita – nový stav závisí len na stave danej bunky a stavov buniek jej okolia
- paralelizmus – výpočet nových stavov buniek môže prebiehať súčasne, čiže nezávisle od seba

Ďalej si presnejšie špecifikujeme štruktúru celulárnych automatov, ich klasifikáciu a aplikáciu.

2.1 Štruktúra

Celulárny automat je pole buniek, ktoré sa vyvíjajú za použitia pravidiel fungujúcich na základe istých vzťahov medzi bunkami (ich okolím).[6]

2.1.1 Bunka

Jedná sa o základný element celulárneho automatu. Môže nadobúdať jeden z konečnej množiny stavov (napríklad u binárneho celulárneho automatu je to množina $\{0, 1\}$). Pri grafickom znázornení celulárneho automatu je stavy možné odlíšiť farebne, čo napomáha k rýchlemu rozoznaniu vzorov, ktoré v celulárnom automate skúmame.

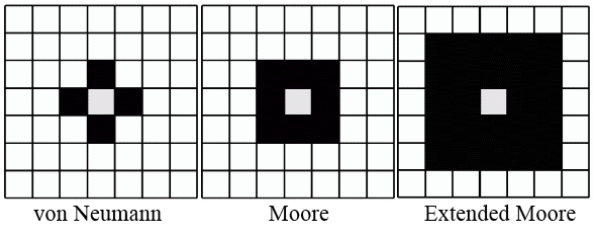
2.1.2 Pole buniek

Jeho úlohou je rovnomerne rozdeliť priestor v jednej a viac dimenziách (najčastejšie sa pracuje s 1, 2 a 3 dimenzionálnymi celulárnymi automatmi). Pri jedno-dimenzionálnych celulárnych automatoch je prehľadne možné zachytiť aj ich priebeh v čase, ako je to viditeľné na obrázku 2.2 (celulárny automat sa v čase vyvíja smerom nadol, resp. každý nový riadok reprezentuje novú generáciu celulárneho automatu).

Stav všetkých buniek v jednom čase označujeme za **konfiguráciu** celulárneho automatu. Pole však môže byť aj nekonečné.

2.1.3 Okolie

Z hľadiska istej bunky v poli nám vymedzuje okolité bunky, ktoré budú zohľadnené pri výpočte nového stavu. Okrem veľkosti sa taktiež rozlišuje jeho typ, ktorý závisí od dimenzie (obrázok 2.3). Pri jedno-dimenzionálnom celulárnom automate však rozlišujeme iba veľkosť okolia. Táto veľkosť nám pri nekonečných poliach buniek určuje o koľko buniek celulárny automat narastie každou novou generáciou.



Obrázek 2.3: Rôzne druhy okolí dvoj-dimenzionálnych celulárnych automatov

2.1.4 Pravidlo

Je to funkcia, ktorá nám z aktuálneho stavu istej bunky a stavov buniek jej okolia definujú nový stav tejto bunky. Matematicky by ho bolo možné definovať ako funkciu f,

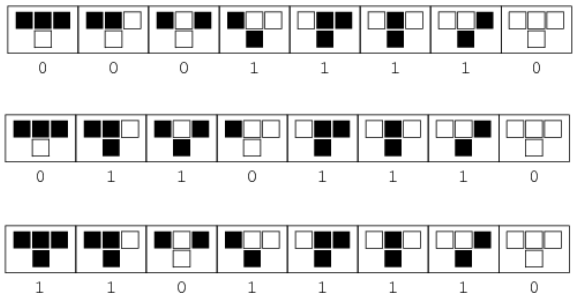
$$s(t+1) = f(s(t), N_s(t))$$

nazývanú tiež *lokálna prechodová funkcia*, kde s je stav menenej bunky, t je čas poslednej zmeny a N_s je okolie menenej bunky.

Počet možných pravidiel závisí na počte možných stavov bunky a veľkosti jej okolia a je ho možné vyjadriť vzťahom,

$$R = S^{S^{(2n+1)^D}} \tag{2.1}$$

kde R je výsledný počet pravidiel, S je počet možných stavov bunky, n je rozmer okolia a D je počet dimenzií.



Obrázek 2.4: Grafické znázornenie pravidiel 30, 110, 222

Ďalej sa bližšie pozrieme na Wolframov kód, čo je systém na pomenúvanie elementárnych celulárnych automatov.

Wolframov kód

Každý elementárny celulárny automat je jedno-dimenzionálny, pričom každá bunka môže nadobudnúť jeden z dvoch stavov. Tieto celulárne automaty majú najčastejšie jednorozmerné okolie, čo z nich robí najjednoduchší prípad celulárnych automatov. Ak tieto hodnoty dosadíme do (2.4) zistíme, že pre elementárne celulárne automaty existuje 256 možných pravidiel, kedže existuje 8 možných konfigurácií okolia, ktoré môžu generovať dva druhy stavov.

$$R = S^{S^{(2n+1)^D}} = 2^{2^{(2*1+1)^1}} = 256$$

Pri definícii pravidiel sa však konfigurácie okolia nemenia – mení sa len generovaný stav. Toto umožnilo Wolframovi špecifikovať pevné poradie konfigurácií a ich generované stavy považovať za binárne číslice, ktoré po úprave tvoria Wolframov kód, ako to znázorňuje obrázok 2.4 (napríklad $(00011110)_2 = (30)_{10}$, $(01101110)_2 = (110)_{10}$, $(11011110)_2 = (222)_{10}$ a podobne). Niektoré z týchto pravidiel sú však izomorfné a považované za totožné.

2.2 Klasifikácia

Už pri prvých pozorovaniach bolo zrejmé, že niektoré celulárne automaty preukazujú zdielané správanie. Najčastejšie sa jednalo o vytváranie rôznych vzorov, ktoré sa neskôr opakovali. Po pokusoch klasifikovať tieto vzory sa Wolfram rozhodol klasifikovať samotné pravidlá. Tak vznikli štyri základné triedy celulárnych automatov, ktoré boli neskôr upresnené.[2]

Podľa zložitosti to sú:

- Trieda 1: Celulárne automaty, u ktorých sa takmer všetky počiatočné vzory rýchlo vyvinú do stabilného, homogénneho stavu. Všetka náhodnosť z pôvodného vzoru zmizne.
- Trieda 2: Celulárne automaty, u ktorých sa takmer všetky počiatočné vzory rýchlo vyvinú do stabilných alebo oscilujúcich štruktúr. Časť náhodnosti z pôvodného vzoru môže byť odfiltrovaná, avšak časť je zachovaná. Lokálne zmeny v pôvodnom vzore zvyknú zostať lokálnymi.
- Trieda 3: Celulárne automaty, u ktorých sa takmer všetky počiatočné vzory vyvinú pseudo-náhodným alebo chaotickým spôsobom. Všetky stabilné štruktúry, ktoré sa objavia, sú rýchlo zničené okolitým šumom. Lokálne zmeny v pôvodnom vzore sa zvyknú šíriť neurčito.
- Trieda 4: Celulárne automaty, u ktorých sa takmer všetky počiatočné vzory vyvinú do štruktúr, ktoré interagujú komplexnými a zaujímavými spôsobmi, s formáciou lokálnych štruktúr, ktoré sú schopné prežiť dlhé obdobie. Výsledkom môžu byť stabilné alebo oscilujúce štruktúry triedy 2, ale počet krokov na dosiahnutie tohto stavu môže byť veľmi vysoký, aj pri relatívne jednoduchých počiatočných vzoroch. Lokálne zmeny v pôvodnom vzore sa môžu šíriť neurčito.

2.3 Aplikácia

Celulárne automaty sú vďaka svojej štruktúre aplikovateľné i výhodné na použitie v mnohých oblastiach. Ako bolo už v úvode kapitoly spomenuté, ich vývoj bol spojený s fyzikálnym

výskumom, avšak ich názov korení v biológii (kvôli ich bunkovej stavbe). Najčastejšie sa pomocou nich počíta dynamika plynov a tekutín, rôzne bunkové siete ako aj ich samotné správanie pri náhodných podmienkach.

Niektoré z celulárnych automatov sú dokonca turingovsky kompletné, čo nám umožňuje vďaka nim vykonávať teoreticky ľubovoľný výpočet vykonateľný na počítači (ako napríklad *pravidlo 110* z obrázku 2.4 alebo *Game of Life* z obrázku 2.1).

Ich relatívne jednoduchá fyzická implementácia nám na druhej strane ponúka vyššiu rýchlosť ako ich softvérová obdoba.

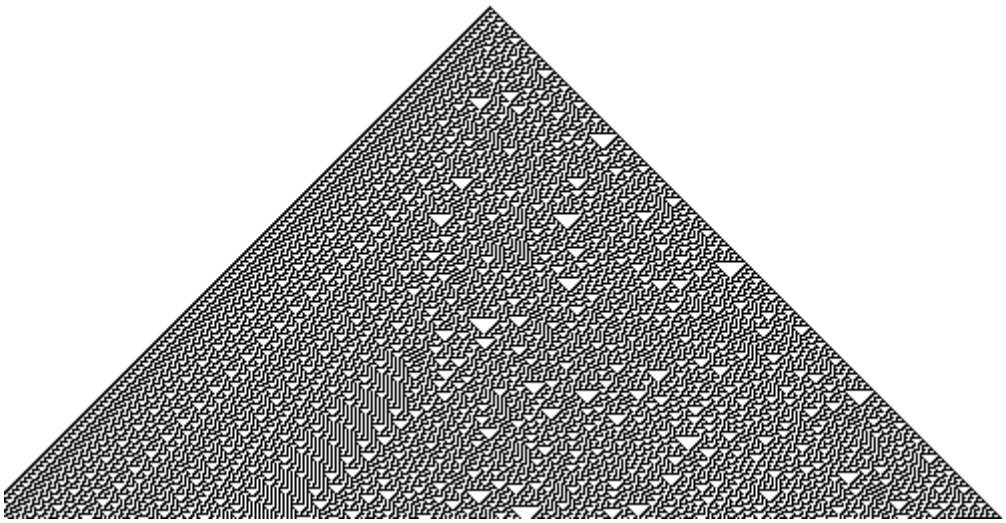
2.3.1 Štúdium správania

Jednotlivé vlastnosti, ako možnosť zobraziť celulárne automaty graficky, sprístupnili túto oblasť širšej verejnosti. K tomu prispelo aj Conwayove zverejnenie celulárneho automatu *Game of Life*, ktorý pripomína správanie sa živých organizmov.

U celulárnych automatov sa rozlišovalo nielen správanie vzorov ale aj správanie celých celulárnych automatov (podkapitola 2.2). Ich veľké množstvo a počet rôznych kombinácií, podľa ktorých sa majú celulárne automaty vyvíjať, robia zo štúdia správania stále úplne neprebádanú a lákavú oblasť (napríklad samo-replikujúci sa vzor do celulárneho automatu *Game of Life (Hra života)* sa podarilo zostrojiť až v roku 2013, čo je 43 rokov po vzniku automatu).[11] Zaujímavé správanie však pri svojej jednoduchosti preukazujú už elementárne celulárne automaty.

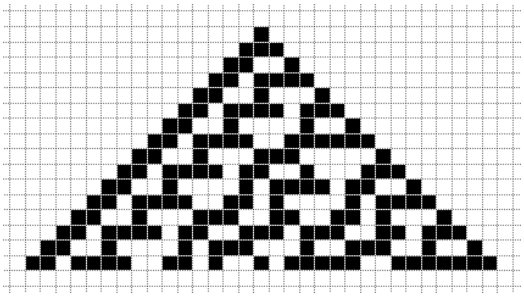
2.3.2 Generovanie čísel

Ďalšou z oblastí, v ktorej sa dodnes využívajú celulárne automaty je generovanie čísel, kedže konfigurácie binárnych celulárnych automatov je možné považovať za binárne čísla. Pri použití celulárnych automatov triedy 3 (podkapitola 2.2) je dokonca možné generovať i pseudo-náhodné čísla. Takýmto celulárnym automatom je aj *pravidlo 30*, ktoré ako prvé publikoval Wolfram po začatí jeho štúdií celulárnych automatov, ako to je spomenuté v úvode kapitoly. Toto pravidlo sa dodnes využíva ako generátor veľkých náhodných čísel.

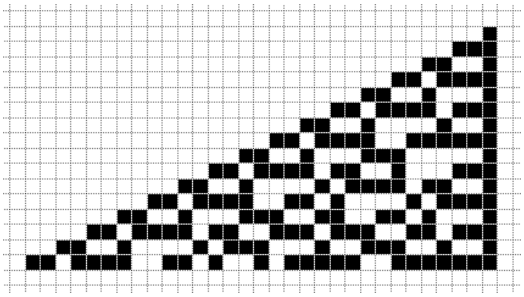


Obrázek 2.5: Vývoj celulárneho automatu pravidlo 30

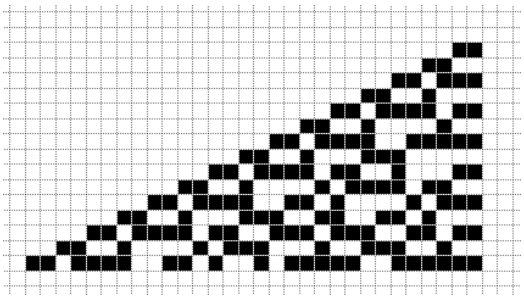
Keď sa však pozrieme na celulárny automat po dlhšej dobe behu (obrázok 2.5), je viditeľné, že na ľavej strane celulárneho automatu sa začali vytvárať vzory, pričom naj-pravejšia bunka celulárneho automatu je opakovane rovnaká. To nás núti konfiguráciu celulárneho automatu upraviť, ak chceme celulárny automat použiť ako zdroj pseudo-náhodných čísel.



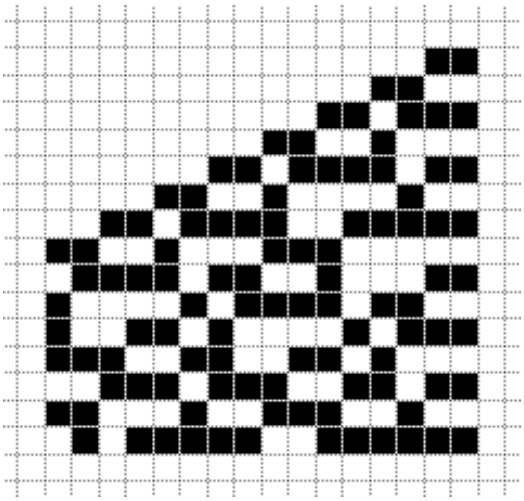
(a) Klasické pravidlo 30



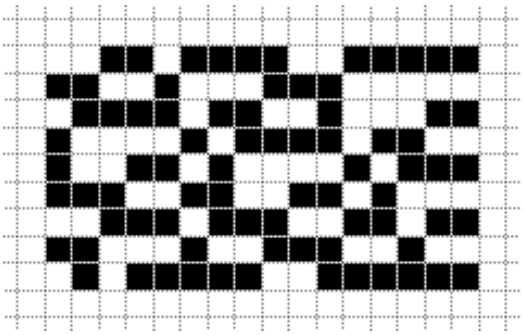
(b) Konfigurácie zarovnané podľa rádu buniek



(c) Konfigurácie bez naj-pravšej bunky



(d) Konfigurácie zaberajúce 16 buniek



(e) Výsledné pseudo-náhodné čísla

Obrázek 2.6: Generovanie čísel z konfigurácií celulárneho automatu

Najjednoduchší spôsob ako konfiguráciu v tomto prípade upraviť, je odstrániť naj-pravejšiu bunku, ktorá je opakovane rovnaká a vzory z ľavej strany nechať prirodzene pretiecť, kedže číselné dátové štruktúry majú limitovanú a pevnú veľkosť (jednotlivé kroky sú znázornené na obrázku 2.6). Iné celulárne automaty však môžu vyžadovať iné úpravy.

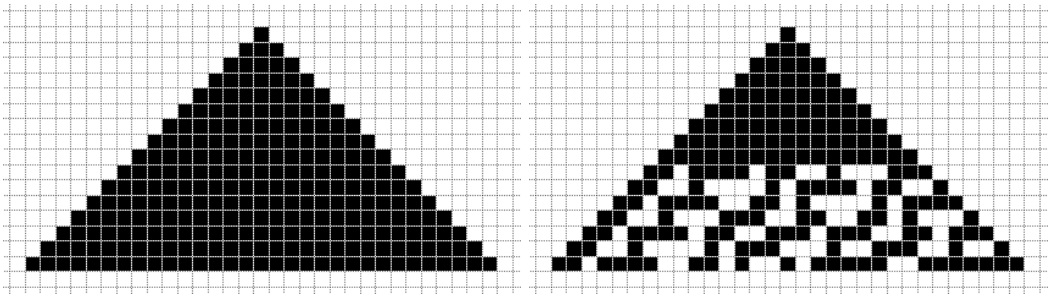
Kapitola 3

Seba-modifikujúci sa celulárny automat

Jedná sa o rozšírenie klasických celulárnych automatov a umožňujú nám zmenu stavu vybraných buniek mimo pravidlo (lokálnu prechodovú funkciu). To je však komplikovaná činnosť, ktorá vyžaduje podrobnejšie skúmanie.

3.1 Seba-modifikácia

Aby sme boli schopní seba-modifikáciu vytvoriť, musíme ju najprv presne popísať. Ako bolo už v úvode kapitoly spomenuté, musíme byť schopní, na základe istého podnetu, zmeniť stav vopred neznámeho počtu buniek. Tento podnet, ako aj samotný výber a zmena stavov buniek však musí nastať v programe, teda bez akéhokoľvek vstupu užívateľa. K tomu bude nutné vytvoriť **umelú inteligenciu**, ktorá bude tento proces riadiť.



(a) Klasické pravidlo (b) Pravidlo po seba-modifikácii

Obrázek 3.1: Znázornenie seba-modifikácie

3.2 Umelá inteligencia

Umelou inteligenciou označujeme inteligenciu, ktorú vykazujú stroje alebo programy.[4][8] Môže sa však jednať o špecifickú inteligenciu, preukázanú pri riešení konkrétneho problému. Táto oblasť je však stále predmetom výskumu a nie je úplne presne definovaná. Typicky sa však za jej vzor považuje ľudský rozum a jeho schopnosť riešiť problémy.

Problémy spojené so seba-modifikáciou

1. rozhodnúť sa, či má modifikácia nastať
2. vybrať bunky, u ktorých má modifikácia nastať
3. spôsob, akým sa majú vybrané bunky modifikovať

Takéto problémy sa však v umelej inteligencii riešia celkom bežne, napríklad pri informovanom prehľadávaní stavového priestoru. [13][14] Umelá inteligencia pri tom využíva **heuristiky**.

3.3 Heuristika

Fakt, že celulárne automaty generujú komplexné vzory alebo vykazujú chaotické správanie nám nie vždy umožňuje špecifikovať seba-modifikáciu presne, tak aby nám bola čo najviac prospešná. Aj keď je to možné, často správne riešenie iba priblížime, či už s časových alebo výpočtových nárokov. Presne na to nám slúžia **heuristiky**. Dokážu nám za istých podmienok s istou presnosťou dodať výsledok, ktorý je využitý v ďalšom výpočte k dosiahnutiu výsledku, alebo je sám osebe považovaný za výsledok. Práve takéto heuristiky použijeme k riešeniu problémov popísaných v predošlej podkapitole.

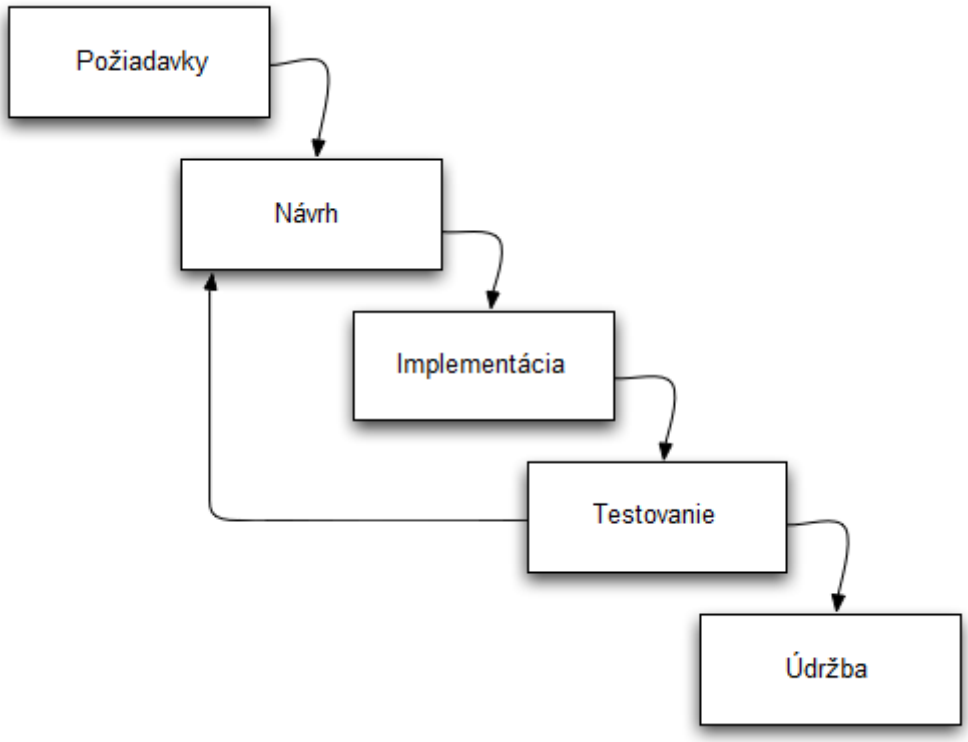
Nakoľko sa však jedná iba o približné riešenia, ich účinnosť a spoľahlivosť nie je dokázateľná, môže byť však štatisticky skúmaná.

Kapitola 4

Simulátor

K potvrdeniu predošle uvedených teoretických znalostí sme nútení zostrojiť simulátor. Jeho cieľom bude pomocou užívateľom špecifikovaného štatistického testu porovnať vlastnosti konfigurácií (respektíve z nich vygenerovaných čísel), bez aj za použitia seba-modifikácie, ktorá musí byť taktiež špecifikovaná užívateľom. Pre jednoduchosť budeme pracovať len s elementárnymi celulárnymi automatmi s nekonečným polom buniek.

K vývoju simulátoru použijeme upravený vodopádový model (obrázok 4.1), čo je sekvenčný vývojový proces s piatimi fázami.[3][16]



Obrázek 4.1: Vodopádový model

Začneme so špecifikáciou požiadaviek, ktoré na simulátor máme.

4.1 Špecifikácia požiadaviek

Okrem zrejmých požiadaviek na simulátor, akými sú napríklad možnosti interakcie a špecifikácie spôsobov akými budú prebiehať, nám simulátor musí poskytovať možnosť:

- porovnať celulárny automat bez seba-modifikácie a so seba-modifikáciou
- nastaviť celulárny automat uvedením Wolframovho kódu
- nastaviť veľkosť okolia
- nastaviť počiatočnú konfiguráciu
- nastaviť počet čísel použitých v teste (podmienka simulácie)
- pristupovať k jednotlivým bunkám celulárneho automatu
- modifikovať jednotlivé bunky celulárneho automatu
- špecifikovať umelú inteligenciu
- špecifikovať generátor čísel
- špecifikovať testy
- priebežne zaznamenávať medzi-výsledky simulácie

Vrámci umelej inteligencie musíme byť schopní:

- špecifikovať heuristiku, ktorá nám určí či nastane seba-modifikácia
- špecifikovať heuristiku, ktorá nám určí bunky u ktorých nastane seba-modifikácia
- aktualizovať dáta, na základe ktorých heuristika pracuje

Vrámci generátoru čísel musíme byť schopní:

- špecifikovať spôsob, akým sa má upraviť konfigurácia celulárneho automatu
- špecifikovať koľko prvých čísel má byť ignorovaných

Vrámci testov musíme byť schopní:

- špecifikovať štatistický test
- priebežne vkladať testované dáta
- vypisovať medzi-výsledky

4.2 Analýza a návrh

Táto podkapitola sa zaoberá analýzou predošle uvedených požiadaviek a návrhom simulátoru a jeho častí. Táto časť je jednou z naj-dôležitejších pri vývoji aplikácie, nakoľko v nej presne špecifikujeme jej funkcionalitu i napriek nepresným nárokom na rozsah aplikácie, prípadne neúplne špecifikovaným požiadavkám. Dobrý návrh aplikácie nám tiež ušetrí čas pri implementácii a taktiež nám ponúka možnosti ako aplikáciu jednoducho rozšíriť.

V prvom rade si musíme určiť o aký simulátor sa jedná. Nakoľko sú celulárne automaty diskrétne v čase, použijeme na diskrétny simulátor. To nám určí základnú štruktúru aplikácie ako aj jej priebeh.

Ďalej si musíme určiť akým spôsobom budeme so simulátorom komunikovať. Nakoľko sa snažíme skúmať štatistické vlastnosti konfigurácií celulárnych automatov, postačí nám na komunikáciu rozhranie príkazového riadku. Vďaka nemu sme schopní simulátor spustiť a argumentmi mu predať potrebné nastavenia, ktoré sú počas simulácie nemenné. Narozdiel od vstupných operácií, ktoré sme schopní zadať všetky naraz, musia byť tie výstupné oddelené. Preto použijeme logy (súbory na zápis), do ktorých sa budú iba zapisovať medzi-výsledky.

V neposlednom rade je nutné navrhnúť celulárny automat schopný seba-modifikácie (ako bol popísaný v kapitole 3). Taktiež musíme byť schopní unifikovaným spôsobom špecifikovať logiku či už umelej inteligencie, generátoru čísel alebo štatistického testu a túto logiku patrične použiť.

Už počas nastavovania aplikácie však môžu nastať výnimky, ktoré je taktiež nutné definovať a spracovávať.

Po prvotnej analýze nám teda vzniklo osem logických celkov:

1. diskrétny simulátor
2. argumenty
3. logy
4. celulárny automat
5. umelá inteligencia
6. generátor čísel
7. štatistický test
8. výnimky

Pre návrh a implementáciu simulátoru som sa rozhodol použiť objektovo orientované programovacie paradigma.[5][15] V ďalších podkapitolách sú podrobnejšie analyzované jednotlivé logické celky okrem výnimiek a je uvedený aj návrh jednotlivých objektov v jazyku UML (pomocou triednych diagramov).[1]

4.2.1 Diskrétny simulátor

Jadro našej aplikácie je tvorené simulačným cyklom. Nakoľko sú celulárne automaty diskrétne v čase, bude tento simulátor diskrétny. To nám určuje základnú štruktúru a priebeh našej aplikácie nasledovným spôsobom:

Algoritmus diskrétneho simulátoru [6][12]

1. **Začiatok simulácie**
2. Inicializácia prvkov simulátoru
3. Inicializácia podmienky cyklu
4. Naplánovanie prvého behu simulačného cyklu
5. **Začiatok simulačného cyklu** – typicky nekonečný cyklus
6. Použijeme prvky simulátoru
7. Aktualizujeme štatistický test
8. Zaznamenáme medzi-výsledky
9. Vyvinieme prvky simulátoru v čase
10. **Koniec simulačného cyklu** – ak nevyhovie podmienka cyklu
11. Zaznamenáme výsledok štatistického testu
12. **Koniec simulácie**

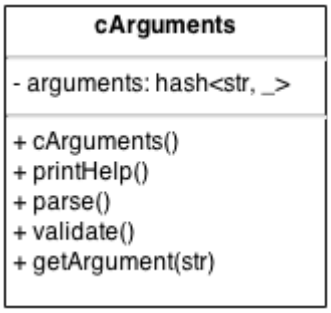
Prvkami simulátoru v tomto prípade myslíme jednotlivé objekty, ktoré sa zúčastňujú simulácie. V našom prípade sa bude jednať o celulárny automat, umelú inteligenciu a generátor čísel.

Simulačný cyklus je typicky nekonečný cyklus s ukončovacou podmienkou a obsahuje operácie vykonávané jednotlivými prvkami simulátoru ako extrahovanie čísel z konfigurácií celulárneho automatu a práca s umelou inteligenciou. Podmienkou cyklu (čo je náš diskrétny čas) bude počet čísel, ktoré sme extrahovali z vygenerovaných konfigurácií celulárneho automatu. Taktiež sú v ňom zaznamenávané jednotlivé medzi-výsledky, ktoré nám vygenerujú jednotlivé prvky simulátoru. V poslednom rade sa vykoná vývoj týchto prvkov v čase, konkrétne vývoj celulárnych automatov.

Po ukončení simulačného cyklu je zaznamenaný výsledok štatistického testu, čo je priemer hodnôt výsledkov logiky štatistického testu pre jednotlivé vygenerované čísla.

4.2.2 Argumenty

Nakoľko jediné vstupy, ktoré v simulátore realizujeme sú nastavenia, ktoré sa počas behu aplikácie nemenia, zadáme tieto vstupy ako argumenty programu. To nás núti vytvoriť objekt, ktorý sa bude starať o spracovanie ako aj o validáciu týchto argumentov.



Obrázek 4.2: Triedny diagram – argumenty aplikácie

Tento objekt teda musí obsahovať štruktúru, do ktorej sa spracované argumenty uložia. Nato nám poslúži hash, ktorého kľúčom bude reťazec, ktorý daný argument výstižne pomenúva a ktorého hodnota bude spracovaný argument žiadaného typu.

Nakoľko môžu byť tieto argumenty zadané nesprávne, musíme argumenty aplikácie z príkazového riadku najprv spracovať a potom previesť ich validáciu, kde skontrolujeme ich jednotlivé kombinácie a neskôr skontrolujeme či sa ich typ rovná požadovanému. V prípade chyby dôjde k vyvolaniu výnimky, ktorá je ďalej v aplikácii spracovaná.

Ďalej nám objekt musí poskytovať prístup k týmto argumentom, čo nás núti definovať metódu na prístup k nim.

Kedže spôsob zadania argumentov aplikácie je nám vopred známy, vieme tieto argumenty popísať už pri tvorbe tohto objektu a pred-spracovať ich. To sa realizuje v konštruktore a umožňuje nám to v prípade potreby vypísať pomocnú správu popisujúcu jednotlivé vstupy aplikácie a spôsob ich zadania.

Možné vstupy aplikácie zadané ako argumenty:

- žiadosť o výpis pomocnej správy
- wolframov kódu
- veľkost okolia
- počiatočná konfigurácia
- počet generovaných čísel
- počet ignorovaných čísel
- súbor s logikou umelej inteligencie, generátoru čísel a štatistického testu
- prepínač umožňujúci iba generáciu konfigurácií celulárneho automatu
- prepínač umožňujúci iba generáciu čísel
- súbor na zápis výsledku
- súbory na zápis medzi-výsledkov

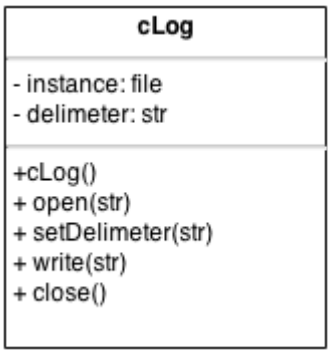
4.2.3 Log

V predošlej kapitole sme si špecifikovali vstup aplikácie, ktorý sa narozdiel od výstupu aplikácie zadáva jednotným spôsobom a naraz. Kedže je však predmetom našej práce skúmanie konceptu seba-modifikácie, je dobré počas priebehu zaznamenávať aj medzi-výsledky. To nás núti vytvoriť objekt, ktorý tieto medzi-výsledky dokáže zaznamenať do samostatných súborov, podľa toho o aký medzi-výsledok sa jedná. Špecifikujeme si teda niekoľko medzi-výsledkov, ktoré budeme chcieť z prvkov simulátoru zaznamenávať.

Možné medzi-výsledky simulácie:

- konfigurácie celulárneho automatu pred seba-modifikáciou
- konfigurácie celulárneho automatu po seba-modifikácii
- extrahované čísla z konfigurácie bez použitia seba-modifikácie
- extrahované čísla z konfigurácie s použitím seba-modifikácie
- výsledok testu logiky pre konkrétne číslo bez použitia seba-modifikácie
- výsledok testu logiky pre konkrétne číslo s použitím seba-modifikácie
- výsledok štatistického testu bez použitia seba-modifikácie
- výsledok štatistického testu s použitím seba-modifikácie

Pre jednotnosť výstupu chceme taktiež možnosť zaznamenať výsledok štatistického testu do súboru, dokopy teda môžme zaznamenávať až sedem druhou údajov.



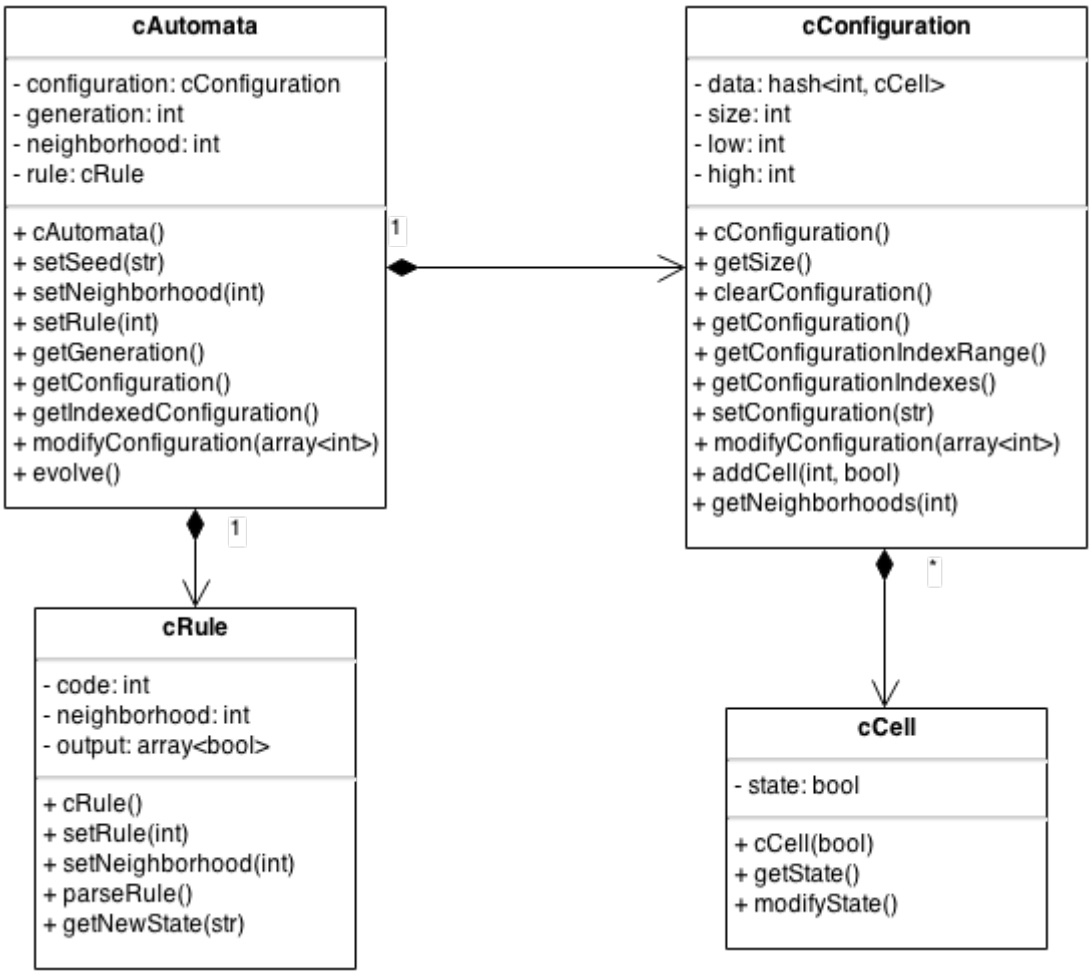
Obrázek 4.3: Triedny diagram – súbor na zaznamenávanie

Objekt musí obsahovať premennú, kam uložíme inštanciu alebo deskriptor otvoreného súboru a premennú, ktorá bude obsahovať oddeľovač medzi-výsledkov. Tento oddeľovač je nastaviteľný pomocou metódy. Ďalej musí taktiež obsahovať metódu na otvorenie aj zatvorenie požadovaného súboru, ktorého cestu zadáme ako reťazec pomocou argumentu aplikácie. Poslednou metódou tohto objektu je metóda na zápis dát, ktorá zapisuje jednotlivé dáta oddelené oddeľovačom.

4.2.4 Automat

Tento celok popisuje nosnú časť našej aplikácie a ňou je práve seba-modifikujúci sa celulárny automat (kapitola 3).

Každý celulárny automat musí obsahovať veľkosť okolia a konfiguráciu, ktorá je zaznamenaná v samostatnom objekte. Okrem toho ešte obsahuje počítadlo generácií a pravidlo, ktoré je taktiež samostatný objekt. Celulárny automat taktiež definuje metódy, ktoré umožňujú manipuláciu s týmito prvkami, ako napríklad nastavenie veľkosti okolia, nastavenie počiatočnej konfigurácie, alebo nastavenie pravidla zadaním Wolframovho kódu. Konfiguráciu je možné vypísať ako binárny reťazec, prípadne ako hash aj s príslušnými indexmi, čo využíva umelá inteligencia. Tá ďalej vyžaduje aj metódu na výpis aktuálnej generácie a metódu, ktorá na základe zoznamu indexov modifikuje stav vybraných buniek. Medzi najhlavnejšiu metódu však patrí metóda na vývoj celulárneho automatu, ktorá za použitia súčasnej konfigurácie a pravidla vygeneruje novú konfiguráciu.



Obrázek 4.4: Triedny diagram – celulárny automat

Pravidlo je objekt, ktorý nám z danej konfigurácie okolia dokáže určiť nový stav bunky. Tento objekt je však nutné nastaviť a to zadaním veľkosti okolia a Wolframovho kódu. Po zadaní týchto údajov je nutné pravidlo spracovať, čím sa pred-počítajú výstupy jednotlivých konfigurácií okolia s danou veľkosťou, nakoľko Wolframov kód je v podstate binárne číslo, ktoré nám definuje výstupy pre konfigurácie okolí buniek konfigurácie celulárneho automatu (podkapitola 2.1 resp. 2.1.4).

Konfigurácia celulárneho automatu obsahuje štruktúru – hash, ktorého kľúčom je index bunky a ktorého hodnotou je samotná bunka. Objekt taktiež uchováva informáciu o veľkosti konfigurácie, ako aj rozsah indexov jej buniek. Tieto informácie sú nám dostupné za pomoci niekoľkých metód, akými sú napríklad výpis veľkosti konfigurácie, výpis konfigurácie ako binárneho reťazca, výpis indexov buniek konfigurácie, alebo ich rozsahu. Konfiguráciu je taktiež možné nastaviť (či už v celku binárnym reťazcom alebo jednotlivo po bunkách), nakoľko je nutné zadať počiatočnú konfiguráciu, alebo vygenerovať novú. Pri generovaní novej konfigurácie je však nutné starú vymazať. Objekt ďalej obsahuje metódu, ktorá nám podľa veľkosti okolia vráti z danej konfigurácie celulárneho automatu zoznam konfigurácií jednotlivých okolí buniek, ktorý je využitý pri generovaní ďalšej konfigurácie celulárneho automatu. Objekt taktiež obsahuje už spomenutú metódu, ktorá nám modifikuje bunky vymedzené zoznamom indexov.

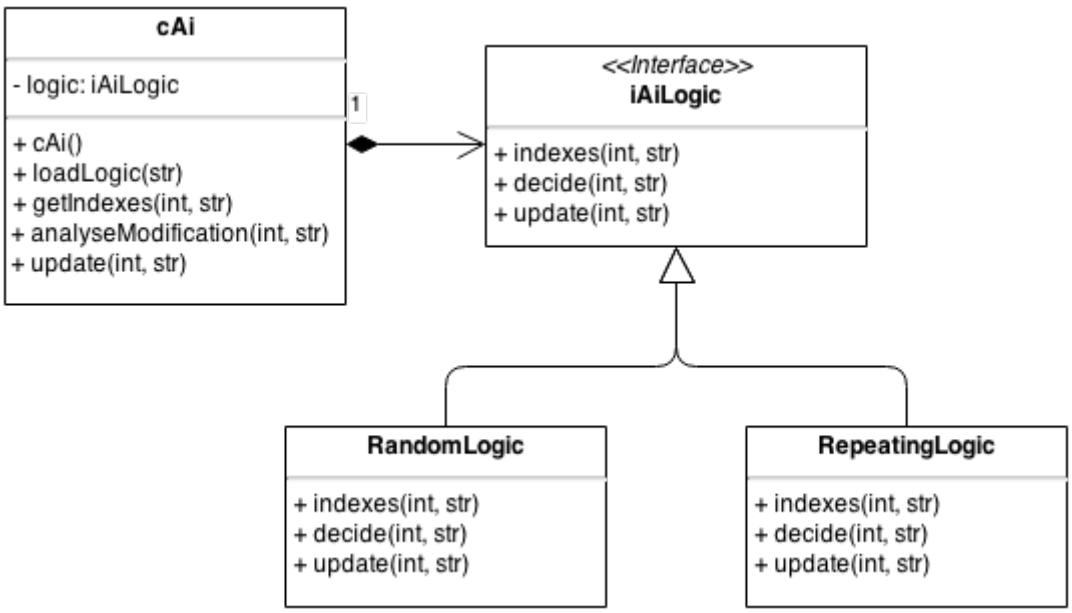
Posledným objektom v tomto celku je bunka, ktorá obsahuje premennú uchovávajúcu jej stav, metódu na výpis tohto stavu, ako aj na jeho modifikáciu. Modifikáciou bunky binárneho celulárneho automatu rozumieme negáciu jej aktuálneho stavu, to znamená, že ak je bunka aktívna, stane sa neaktívnou a naopak. Konkrétny stav bunky sa určuje iba pri jej vytvorení konfiguráciou a preto nastavenie realizuje konštruktor tohto objektu.

4.2.5 Umelá inteligencia

Ďalšou dôležitou časťou simulátoru je umelá inteligencia, ktorá nám za použitia heuristík riadi seba-modifikáciu.

Nakoľko má byť užívateľ schopný definovať umelú inteligenciu, rozdelíme ju na dve časti:

1. umelá inteligencia
2. logika umelej inteligencie – má rozhranie



Obrázek 4.5: Triedny diagram – umelá inteligencia

Umelá inteligencia bude objekt, ktorý je prvkom simulácie a logika umelej inteligencie bude objekt, ktorý umelá inteligencia použije pre riadenie. Logikou budú v tomto prípade heuristiky, ktoré nám za použitia počítadla generácií celulárneho automatu a jeho konfigurácie vyriešia problémy popísane v podkapitole 3.2. Týmito problémami sú:

1. rozhodnutie, či má nastať modifikácia
2. zoznam indexov buniek, u ktorých ma modifikácia nastať

Prvý z problémov bude riešiť metóda, ktorej výstupom bude hodnota typu bool určujúca či modifikácia nastane alebo nie. Bude pritom využívať počítadlo generácií celulárneho automatu a binárny reťazec reprezentujúci jeho konfiguráciu.

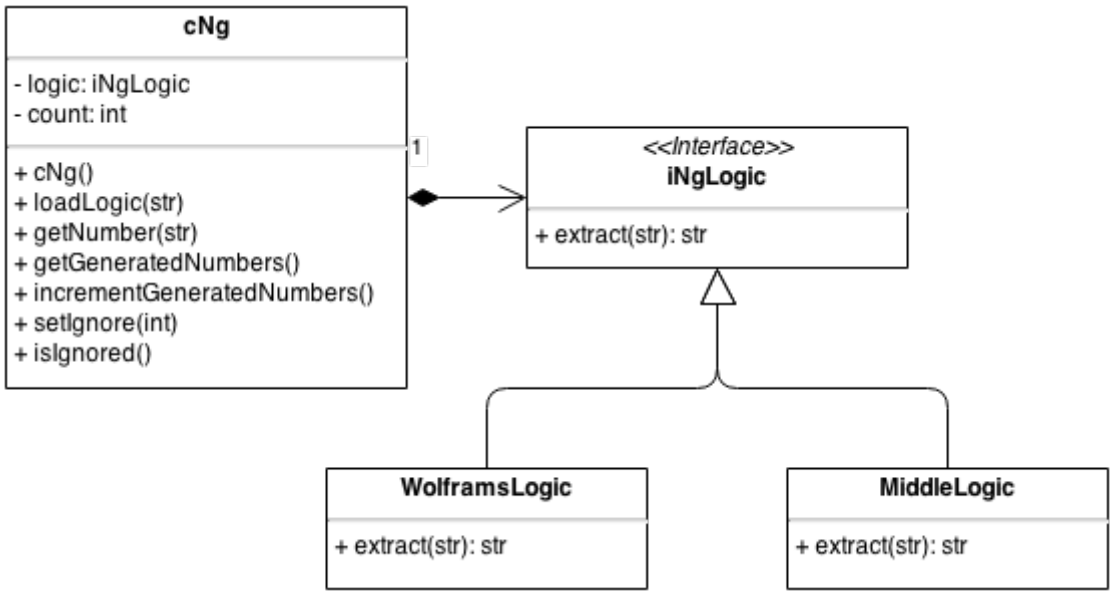
Druhý z problémov bude riešiť metóda s rovnakými argumentmi, ktorej výstupom však bude zoznam indexov, ktoré nám vymedzia bunky konfigurácie celulárneho automatu u ktorých nastane modifikácia.

Poslednou z metód bude aktualizácia heuristiky, ktorá zaznamená aktuálnu generáciu celulárneho automatu a binárny reťazec reprezentujúci konfiguráciu po modifikácii.

4.2.6 Generátor čísel

Generátor čísel je jednoduchý objekt, ktorého základnou úlohou je úprava konfigurácie celulárneho automatu tak, aby sme ju mohli považovať za číslo. Taktiež ako umelá inteligencia sa bude skladať z dvoch častí:

1. generátor čísel
2. logika generátoru čísel – má rozhranie



Obrázek 4.6: Triedny diagram – generátor čísel

Tento objekt bude okrem logiky, ktorá nám bude realizovať extrakciu čísel, obsahovať taktiež počítadlo vygenerovaných čísel. Hodnotu tohto počítadla nám sprístupňuje metóda a ďalšia ho v prípade, kedy generujeme konfigurácie celulárneho automatu a nepoužívame generátor čísel, navyšuje. To je potrebné kedže počítadlo vygenerovaných čísel používame ako podmienku cyklu. Objekt ďalej obsahuje metódu na extrakciu čísel a metódu, ktorá nám určí či má by číslo ignorované.

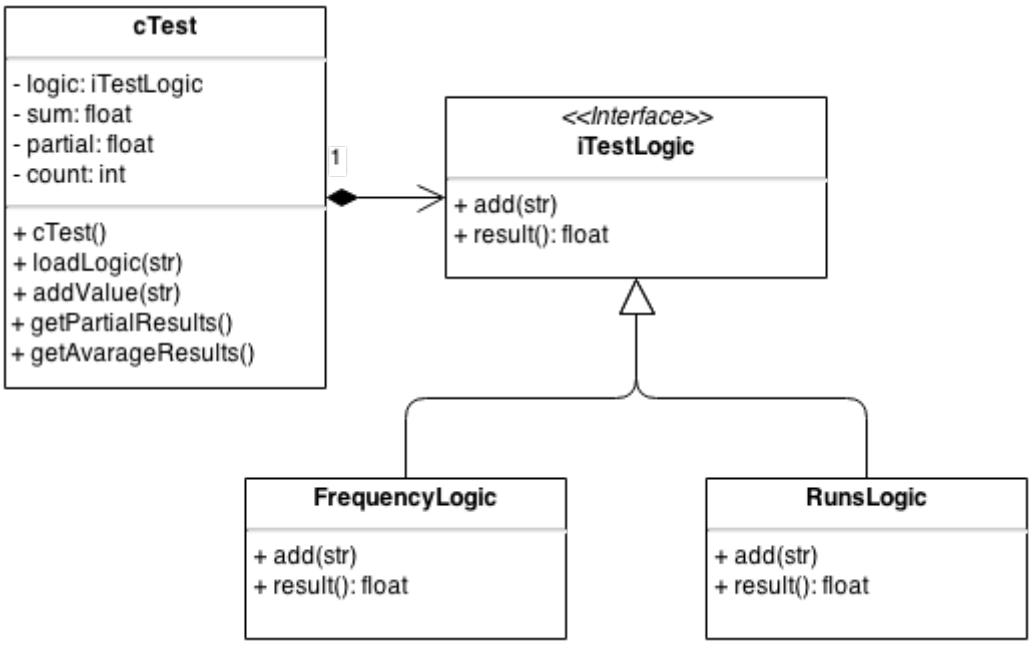
Objekt bude taktiež obsahovať metódu ktorá načíta logiku generátoru čísel zo súboru zadaného argumentom aplikácie.

Jednu takúto logiku sme si už opísali v podkapitole 2.3.2 a znázornili na obrázku 2.6.

4.2.7 Štatistický test

V poslednom rade si opíšeme štatistický test, čo je objekt, ktorý nám generuje výsledky simulácie a ktorého cieľom je vyhodnotiť vlastnosti čísel extrahovaných z konfigurácií celulárneho automatu. Taktiež bude mať dve časti dvoch častí:

1. štatistický test
2. logika štatistického testu – má rozhranie



Obrázek 4.7: Triedny diagram – štatistický test

Tento objekt bude obsahovať sumu, kam sa postupne pričítajú výsledky logiky testu, čo sú výsledky istého testu pre jedno číslo. Posledný pripočítaný výsledok je taktiež uchovaný, nakoľko ho môžme chcieť zaznamenať. Objekt pre túto činnosť definuje metódu.

V objekte sa ďalej nachádza aj počítadlo čísel, ktoré je použité v metóde na výpis výsledku testu (teda priemeru hodnôt). Tento výsledok môže byť taktiež zaznamenávaný priebežne.

4.3 Implementácia

Táto kapitola vychádza z návrhu a obsahuje vývoj aplikácie, jej programovanie v konkrétnom programovacom jazyku, čo zahŕňa aj konkrétnu implementáciu algoritmov a dátových štruktúr. Výsledkom sú prepojené prvky tvoriace jeden funkčný celok.

Pre vývoj tohto simulátoru využijeme programovací jazyk **Python 3**, nakoľko nám umožňuje použiť objektovo orientované programovacie paradigma. Je to skriptovací jazyk najmä využívaný vedeckou komunitou pre jeho rýchlosť, jednoduchú syntax a čitateľnosť kódu a jeho schopnosť spracovávať veľké čísla.[7] Pre implementáciu sme použili jeho najnovšiu verziu, ktorá však nie je kompatibilná so staršou (Python 2).

Pri implementácii využívame aj knižnice, ktoré sú však dostupné v každej distribúcii programovacieho jazyka. Jedná sa o:

- argparse – umožňuje spracovávať argumenty zadané rôznymi zaužívanými spôsobmi
- sys – sprístupňuje nám štandardný vstup a výstup ako aj moduly v iných adresároch ako v koreňovom
- os – umožňuje nám pracovať so zložkami bez závislosti na platforme
- inspect – umožňuje nám previesť kontrolu zadaných logík

4.4 Testovanie

Testovanie je fáza vývoju softvéru, v ktorej sa snažíme preukázať funkcionalitu aplikácie. Nakoľko náš simulátor poskytuje veľké množstvo medzi-výsledkov, sme schopní aplikáciu otestovať ako koncový užívateľ, takzvaný *black-box testing*.[9]

Nakoľko simulátor vyžaduje k svojmu chodu užívateľom špecifikované logiky, jeho hĺbkové testovanie nie je možné. Pre jeho chod sme teda nútený definovať aspoň jednoduché logiky, ktoré aspoň preukážu jeho schopnosť načítať a použiť logiku.

Na základe zistení v tejto kapitole boli doplnené špecifickejšie chybové hlášky a odstránené niektoré menšie chyby.

Výsledný systém bol testovaný na platforme Linux aj Windows.

4.5 Údržba

Poslednou časťou vodopádového modelu je údržba, čo je činnosť, ktorú realizujeme po nasadení systému do prevádzky. V našom prípade sa môže jednať o zápis logík špecifickým spôsobom vyžadovaným aplikáciou, nakoľko užívateľ nemusí programovací jazyk ovládať.

Náš simulátor však pre logiky používa rozhrania, čo užívateľovi presne špecifikuje štruktúru, ktorá musí byť dodržaná. Programovací jazyk Python je navyše veľmi dobre čitateľný a jednoduchý, čo umožňuje túto činnosť vykonávať aj užívateľovi.

Kapitola 5

Pokusy

Záverom našej práce budú pokusy, ktoré budeme realizovať na našom simulátory. To zahŕňa špecifikáciu argumentov, špecifikácie logík umelej inteligencie, generátoru čísel a štatistického testu, požitie týchto logík v simulácii ako aj interpretáciu výsledkov. Tieto pokusy budú zreteľne demonštrovať isté vlastnosti seba-modifikácie. Tieto vlastnosti nemusia byť z celkového hľadiska výhodné, avšak naše pokusy budú založené na istých predpokladoch, ktoré sa budeme snažiť preukázať a dokázať štatistickým testom.

Ako bolo uvedené v podkapitole 4.2.5 a na obrázku 4.5, logika umelej inteligencie má rozhranie, ktoré núti nás implementovať nasledujúce tri metódy:

- metóda, ktorá rozhodne či nastane modifikácia
- metóda, ktorá vráti zoznam indexov
- metóda, ktorá aktualizuje heuristiku

Ako bolo uvedené v podkapitole 4.2.6 a na obrázku 4.6, logika generátoru čísel má rozhranie, ktoré núti nás implementovať nasledujúcu metódu:

- metóda, ktorá extrahuje čísla z konfigurácií

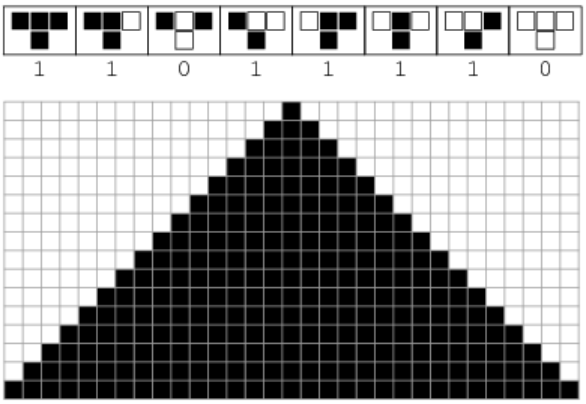
Ako bolo uvedené v podkapitole 4.2.7 a na obrázku 4.7, logika štatistického testu má rozhranie, ktoré núti nás implementovať nasledujúce dve metódy:

- metóda, ktorá pridáva čísla do testu
- metóda, ktorá vráti výsledok

5.1 Pokus I

Prvým z týchto pokusov založíme na predpoklade, že i celulárne automaty ktoré sa vyvíjajú stabilným spôsobom môžu vďaka seba-modifikácii začať preukazovať chaotické správanie. Využijeme pri tom celulárny automat, ktorý generuje iba bunky s jedným stavom. Takýmto celulárnym automatom je pravidlo 222 (obrázok 5.1). Logiku umelej inteligencie vytvoríme

takým spôsobom, aby sa zhruba štvrtina buniek modifikovala. Konfigurácie celulárneho automatu budú následne upravené podobným spôsobom ako bolo spomenuté v podkapitola 2.3.2. Aby sme tento predpoklad dokázali, budeme musieť navrhnúť aj logiku štatistického testu, ktorý bude počítať pomer počtu buniek jedného stavu k počtu buniek druhého stavu. Budeme zaznamenávať ako priebeh výsledok logiky testu, tak aj priebeh výsledkov štatistického testu (priemer).



Obrázek 5.1: Graficky znázornené pravidlo 222 a začiatok vývoja celulárneho automatu

Teraz si presne špecifikujeme jednotlivé logiky a argumenty aplikácie, s ktorými prevedieme simuláciu, znázorníme si priebeh výsledkov a vyvodíme záver.

5.1.1 Logika umelej inteligencie

Aby sa celulárny automat dokázal seba-modifikovať, musíme najprv vytvoriť logiku umelej inteligencie, ktorá túto modifikáciu riadi.

Táto logika umelej inteligencie bude náhodným spôsobom vyberať bunky, ktoré sa modifikujú a preto ju nazveme *RandomLogic*. Tieto bunky však môžu byť po modifikácii istú dobu zamrazené – nebude teda možné ich modifikovať.

Metóda, ktorá rozhodne či nastane modifikácia, prejde každú zadanú bunku v konfigurácii a ak o nej existuje záznam v heuristike a bunka nie je zmrazená, nastane modifikácia. V opačnom prípade alebo v prípade, kedy je prikrátka, modifikácia nenastane.

Metóda, ktorá vráti zoznam indexov prejde konfiguráciu obdobným spôsobom, vyberie bunky so záznamom, ktoré nie sú zmrazené a s pravdepodobnosťou 25% ich pridá do zoznamu.

Metóda, ktorá aktualizuje heuristiku prejde každú bunku v konfigurácii a ak o bunke existuje záznam s časom zmrazenia, tak tento čas zníži. Ak tento záznam neexistuje, vytvorí ho a nastaví jeho hodnotu tak, aby bolo možné bunku modifikovať.

Logika teda nebude čisto náhodná ale bude obsahovať aj prvok zmrazenia bunky, ktorý zabraňuje modifikácii bunky.

5.1.2 Logika generátoru čísel

Aby sme z konfigurácie odstránili bunky, ktoré by nám mohli skresľovať výsledky, vytvoríme logiku generátoru čísel.

Táto logika bude fungovať na princípe, aký používa aj Wolfram pri generovaní pseudonáhodných čísel a aký bol opísaný v podkapitole 2.3.2 a na obrázku 2.6. Preto ju nazveme *WolframsLogic*.

Metóda, ktorá extrahuje číslo z konfigurácie teda najprv odstráni najpravšiu bunku a následne vráti posledných 32 buniek. Ak je veľkosť konfigurácie menšia, doplní sa číslo zľava o pasívne bity.

Táto logika teda generuje 32 bitové čísla.

5.1.3 Logika štatistického testu

Aby sme boli schopní preukázať či logika umelej inteligencie napĺňa alebo nenapĺňa stanovené predpoklady, musíme zostrojiť logiku štatistického testu, ktorá patričným spôsobom preskúma vlastnosti vyplývajúce z nášho predpokladu.

Táto logika štatistického testu bude skúmať, aký je pomer počtu aktívnych bitov k celkovému počtu bitov. Túto logiku nazveme *FrequencyLogic*, nakoľko skúma frekvenciu aktívnych bitov v čísle.

Metóda, ktorá pridáva čísla bude teda prechádzať každé jedno zadané číslo bit po bite a počítať koľko z nich je aktívnych. Metóda taktiež zaznamená dĺžku týchto čísel.

Metóda, ktorá vráti výsledok následne počet aktívnych bitov vydelí celkovým počtom bitov čím nám vznikne pomer aktívnych bitov v konfigurácii.

Výsledkom logiky testu bude teda číslo od 0 do 1, pričom optimálna hodnota bude 0.5 (rovnaký počet aktívnych aj pasívnych bitov v čísle). Logika sa pri každom pridanom čísle inicializuje do pôvodného stavu.

5.1.4 Argumenty

Aby simulácia správne fungovala, musíme si špecifikovať argumenty. Tieto argumenty nastavia simulátor do stavu v ktorom nám bude dávať najlepšie výsledky.

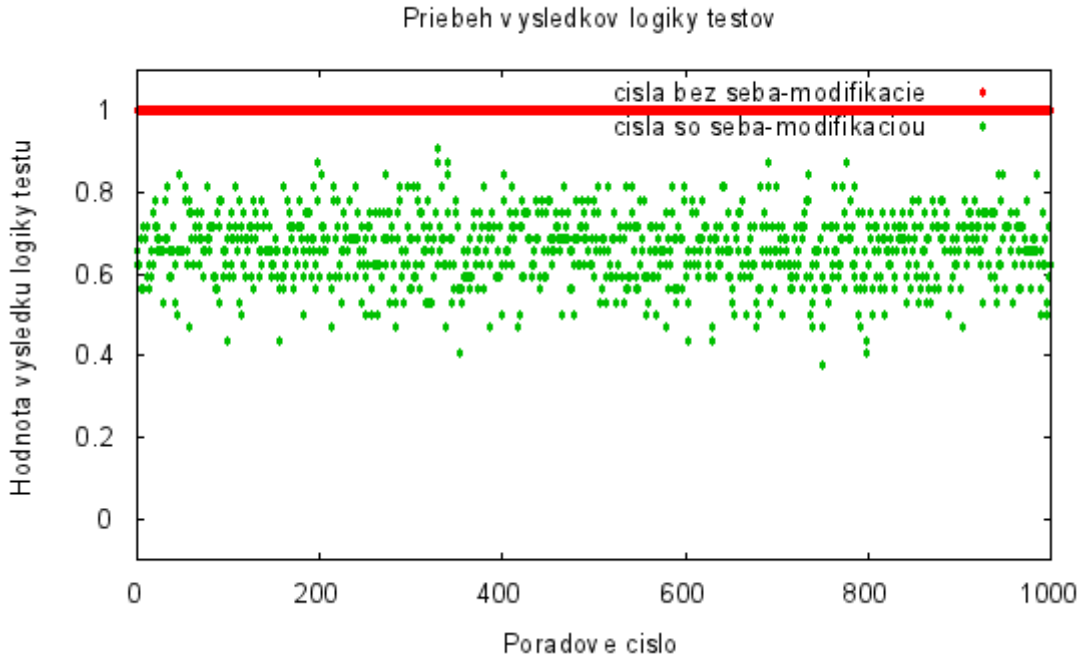
V prvom rade budeme chcieť špecifikovať veľkosť okolia, ktoré bude jedno rozmerné a pravidlo použité na generovanie nových stavov. Následne špecifikujeme počet čísel, ktoré chceme generovať a štatisticky testovať. Východzie nastavenie je 100 čísel, my však použijeme 1000 čísel pre väčšiu presnosť. Následne nastavíme počet ignorovaných čísel, čo je v našom prípade 16, nakoľko generujeme 32 bitové číslo, ktoré je pri začiatku vývoja celulárneho automatu dopĺňané zľava o nuly (konfigurácia celulárneho automatu narastie každou generáciou o dve bunky). Ďalej určíme cesty k jednotlivým logikám a súborom kam sa uložia medzi-výsledky a výsledok.

Argumenty príkazovej riadky budú vyzerať nasledovne:

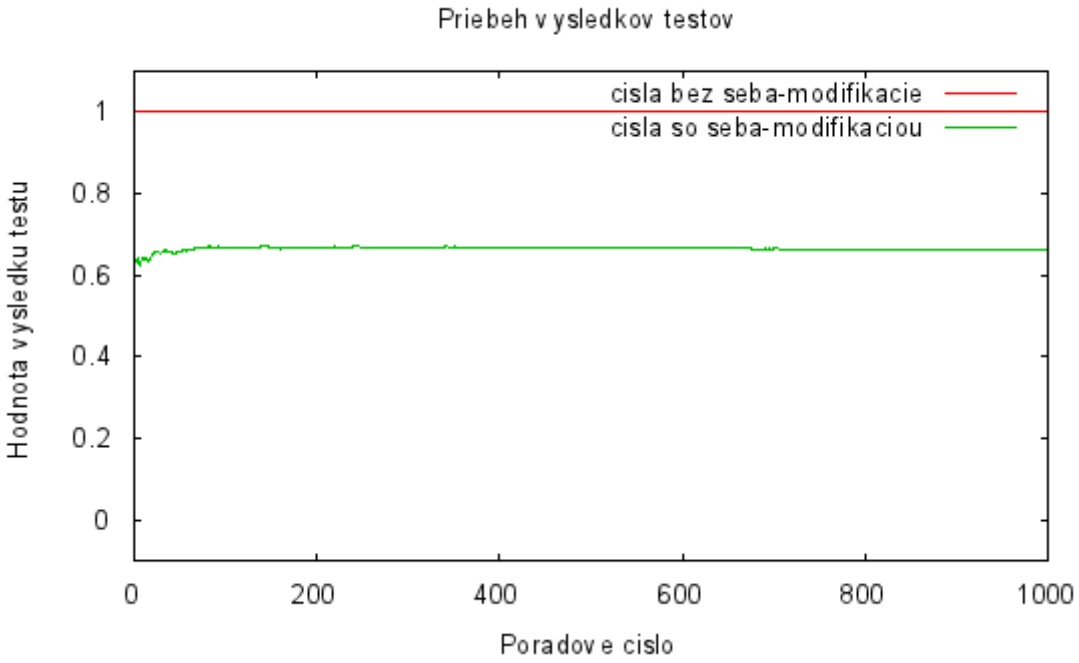
```
--neighborhood 1 --rule 222 --count 1000 --ignore 16
--ai ai_logic\randompick.py
--ng ng_logic\wolframs.py
--test test_logic\frequency.py
--output results\output.txt
--configurations1 results\configurations1.txt
--configurations2 results\configurations2.txt
--numbers1 results\numbers1.txt
--numbers2 results\numbers2.txt
--partial1 results\partial1.txt
--partial2 results\partial2.txt
--avarage1 results\avarage1.txt
--avarage2 results\avarage2.txt
```

5.1.5 Výsledok

Výsledok simulácie je zachytený na dvoch obrázkoch. Obrázok 5.2 znázorňuje priebeh výsledkov logiky testu, čiže výsledok testu logiky pre každé jedno číslo. Obrázok 5.3 znázorňuje priebeh výsledkov štatistického testu, čiže priemer výsledkov zatiaľ testovaných čísel.



Obrázek 5.2: Postupne zaznamenané výsledky logiky testu



Obrázek 5.3: Postupne zaznamenané výsledky testu

5.1.6 Záver

Hlavným výstupom simulácie je nasledovný text:

```
Avarage of the test without self-modification: 1.0
Avarage of the test with self-modification: 0.6623125
```

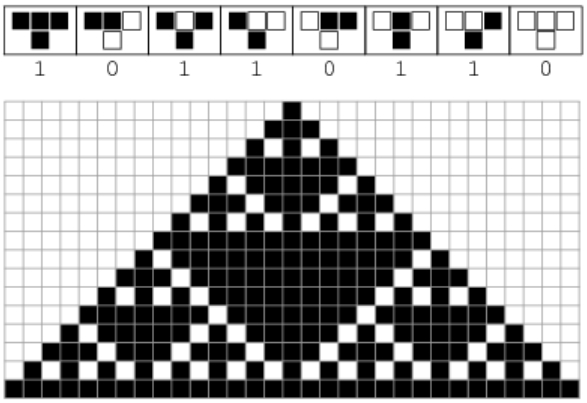
Ako už aj z obrázkov (5.2 a 5.3) vidno, že celulárny automat bez seba-modifikácie mal všetky bunky aktívne, čiže aj extrahované čísla mali všetky bity aktívne. To znamená že zaberali 100% čísla. Ak sa však pozrieme na čísla extrahované z konfigurácie celulárneho automatu so seba-modifikáciou zistíme, že aktívne bity zaberali okolo 60% až 80%.

Výsledná hodnota 66% teda potvrdzuje náš predpoklad, že i stabilne sa správajúce celulárne automaty môžu pomocou seba-modifikácie vykazovať na chaotické (alebo aspoň viac náhodné) správanie.

5.2 Pokus II

Druhý z týchto pokusov založíme na predpoklade, že niektoré celulárne automaty generujú vzory. Tieto vzory sú tvorené bunkami jedného stavu, pričom bunky druhého stavu tvoria ich okraje. Tieto vzory teda obsahujú niekoľko za sebou idúcich buniek s rovnakým stavom, ktoré navyše svoj stav uchovávajú aj v priebehu času. Takýmto celulárnym automatom je aj pravidlo 182 (obrázok 5.4). Logika umelej inteligencie bude navrhnutá takým spôsobom, aby sa bunka s rovnakým stavom v priebehu istého času modifikovala. Na preukázanie tohto predpokladu však budeme musieť zostrojiť iný test, ktorý bude testovať pomer najdlhšieho počtu za sebou idúcich buniek s jedným stavom k najdlhšiemu počtu za sebou idúcich

buniek s druhým stavom.



Obrázek 5.4: Graficky znázornené pravidlo 182 a začiatok vývoja celulárneho automatu

Teraz si presne špecifikujeme jednotlivé logiky a argumenty aplikácie s ktorými prevedieme simuláciu, znázorníme si priebeh výsledkov a vyvodíme záver.

5.2.1 Logika umelej inteligencie

Aby sa celulárny automat dokázal seba-modifikovať, musíme najprv vytvoriť logiku umelej inteligencie, ktorá túto modifikáciu riadi.

Táto logika umelej inteligencie bude počítať aktívne bunky v priebehu času, ktoré sú aspoň tri generácie rovnaké a preto ju nazveme *RepeatingLogic*.

Metóda, ktorá rozhodne či nastane modifikácia prejde každú zadanú bunku v konfigurácii a ak o nej existuje záznam v heuristike a bunka je aktívna po dobu viac ako troch generácií, nastane modifikácia. V opačnom prípade modifikácia nenastane.

Metóda, ktorá vráti zoznam indexov prejde konfiguráciu obdobným spôsobom a pridá indexy buniek o ktorých existuje záznam, že sú viac ako tri generácie rovnaké do zoznamu.

Metóda, ktorá aktualizuje heuristiku prejde každú bunku v konfigurácii a ak je bunka aktívna, zvýši záznam o generácii. Ak je bunka pasívna tak tento záznam vynuluje.

Logika bude založená na jednoduchom počítaní, ktoré však znemožňuje tvorbu vzorov.

5.2.2 Logika generátoru čísel

Aby sme z konfigurácie odstránili bunky, ktoré by nám mohli skresľovať výsledky, vytvoríme logiku generátoru čísel.

Táto logika bude vracať stred konfigurácie a preto bude mať názov *MiddleLogic*. V strede konfigurácie sa totižto nachádza väčšie množstvo väčších vzorov.

Metóda, ktorá extrahuje číslo z konfigurácie teda najprv zistí veľkosť štvrtiny konfigurácie

a následne odstráni prvú a poslednú štvrtinu. V prípade že je konfigurácia menšia ako 4, vráti sa táto konfigurácia ako konečné číslo.

Táto logika teda vracia zhruba polovicu konfigurácie.

5.2.3 Logika štatistického testu

Aby sme boli schopní preukázať či logika umelej inteligencie napĺňa alebo nenapĺňa stanovené predpoklady, musíme zostrojiť logiku štatistického testu, ktorá patričným spôsobom preskúma vlastnosti vyplývajúce z nášho predpokladu.

Táto logika štatistického testu bude skúmať, aký je pomer veľkosti najdlhšieho počtu za sebou idúcich aktívnych bitov k najdlhšiemu počtu za sebou idúcich pasívnych bitov. Túto logiku nazveme *RunsLogic*, nakoľko skúma behy (počty za sebou idúcich) bitov v čísle.

Metóda, ktorá pridáva čísla bude teda prechádzať každé jedno zadané číslo bit po bite a počítať, aký je najdlhší počet za sebou idúcich aktívnych a pasívnych bitov.

Metóda na vrátenie výsledkov následne počet najdlhšie za sebou idúcich aktívnych bitov vydelí počtom najdlhšie za sebou idúcich pasívnych bitov. V prípade že je jedna z hodnôt nulová, výsledok bude tiež nulový.

Výsledkom logiky testu bude teda číslo, určujúce koľko násobne väčší je najdlhší beh aktívnych bitov voči tým pasívnym, pričom optimálna hodnota bude 1 (rovnaká veľkosť najdlhších behov aktívnych aj pasívnych bitov). Logika sa pri každom pridanom čísle inicializuje do pôvodného stavu.

5.2.4 Argumenty

Aby simulácia správne fungovala, musíme si špecifikovať argumenty. Tieto argumenty nastavia simulátor do stavu, v ktorom nám bude dávať najlepšie výsledky.

V prvom rade budeme chcieť špecifikovať veľkosť okolia, ktoré bude jedno rozmerné a pravidlo použité na generovanie nových stavov. Následne špecifikujeme počet čísel, ktoré chceme generovať a štatisticky testovať. Východzie nastavenie je 100 čísel, my však použijeme 1000 čísel pre väčšiu presnosť. Následne nastavíme počet ignorovaných čísel, čo je v našom prípade 3, nakoľko sa v prvých troch generáciách neprejavuje seba-modifikácia. Ďalej určíme cesty k jednotlivým logikám a súborom kam sa uložia medzi-výsledky a výsledok.

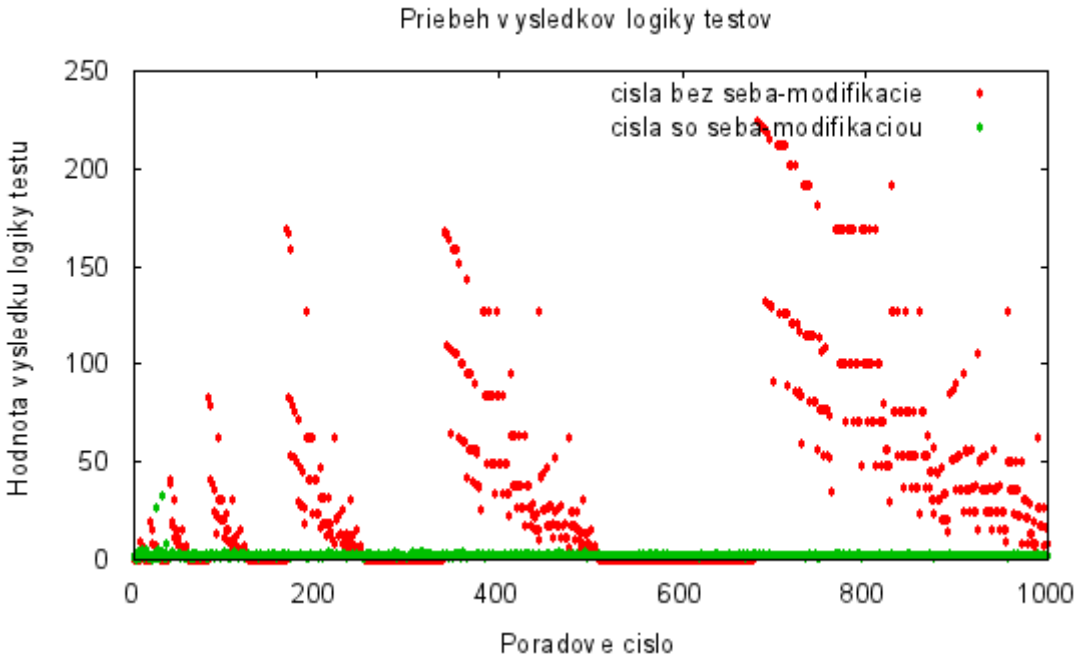
Argumenty príkazovej riadky budú vyzerať nasledovne:

```
--neighborhood 1 --rule 182 --count 1000 --ignore 3
--ai ai_logic\repeating.py
--ng ng_logic\middle.py
--test test_logic\runs.py
--output results\output.txt
--configurations1 results\configurations1.txt
--configurations2 results\configurations2.txt
--numbers1 results\numbers1.txt
--numbers2 results\numbers2.txt
```

```
--partial1 results\partial1.txt
--partial2 results\partial2.txt
--avarage1 results\avarage1.txt
--avarage2 results\avarage2.txt
```

5.2.5 Výsledok

Výsledok simulácie je zachytený na dvoch obrázkoch. Obrázok 5.5 znázorňuje priebeh výsledkov logiky testu, čiže výsledok testu logiky pre každé jedno číslo. Obrázok 5.6 znázorňuje priebeh výsledkov štatistického testu, čiže priemer výsledkov zatiaľ testovaných čísel.



Obrázek 5.5: Postupne zaznamenané výsledky logiky testu

Obrázek 5.6: Postupne zaznamenané výsledky testu

5.2.6 Záver

Hlavným výstupom simulácie je nasledovný text:

```
Avarage of the test without self-modification: 44.943430303030354
Avarage of the test with self-modification: 2.2107315190984864
```

Na prvom obrázku (5.5) je viditeľné, že tento celulárny automat bez seba-modifikácie periodicky generuje vzory. To je ešte zreteľnejšie vidno na druhom obrázku (5.6), ktorý zobrazuje priemer zatiaľ získaných hodnôt. Z tohto obrázku ďalej vyplýva, že pomer veľkosti najväčšieho za sebou idúceho počtu aktívnych bitov k veľkosti najdlhšieho za sebou idúceho počtu pasívnych bitov sa postupom času zväčšuje. Ak sa však pozrieme na čísla extrahované z konfigurácie celulárneho automatu so seba-modifikáciou zistíme, že pomer veľkosti najväčšieho počtu aktívnych bitov k veľkosti najväčšieho počtu pasívnych bitov sa drží pri medzi hodnotou 2 až 3.

Výsledná hodnota so seba-modifikáciou je zhruba dvadsať násobne nižšia od tej bez seba-modifikácie, čo potvrdzuje náš predpoklad, že i s jednoduchou logikou je možné zabrániť generovaniu vzorov alebo tieto vzory zmenšiť.

Kapitola 6

Záver

Táto práca nadväzuje na semestrálny projekt, v ktorom som sa oboznámil s celulárnymi automatmi a ich využitím. Kedže celulárne automaty môžu byť veľmi komplexné, rozhodol som sa zamerať na elementárne celulárne automaty. Následne som preskúmal možnosti, ako tieto celulárne automaty štatisticky testovať.

V rámci tejto práce bol navrhnutý a vytvorený celulárny automat s konceptom seba-modifikácie a simulátor, ktorý takýto celulárny automat porovnáva s konvenčným. Na tento simulátor sme mali niekoľko požiadaviek, ktoré sme analyzovali a založili na nich návrh nášho systému. Medzi ne patrí možnosť špecifikovať logiku umelej inteligencie, generátoru čísel a štatistického testu v samostatnom súbore, bez ďalšieho zásahu do simulátoru. Simulátor pre tento účel definuje rozhrania.

Následne bol simulátor implementovaný a boli na ňom vykonané dva ukážkové pokusy s dvomi odlišnými logikami umelej inteligencie, generátoru čísel aj štatistického testu. Výsledky simulácie, ako aj ich priebeh sú znázornené na obrázkoch a interpretované vhodným spôsobom.

Výsledky našich pokusov, ktoré boli založené na jednoduchých predpokladoch nám preukázali, že vhodne zvolené logiky a nastavenia simulátoru môžu byť prospešné vzhľadom k nášmu štatistickému testu. Ak by sme však čísla podrobnejšie skúmali, mohli by sme odhaliť ďalšie nežiadúce vlastnosti. Podrobné skúmanie čísel by však bolo nad rámec tejto práce, avšak je možné definovať aj zložité logiky.

Okrem toho je možné koncept seba-modifikácie rozšíriť aj na viac-dimenzionálne celulárne automaty. Taktiež by bolo možné vytvoriť grafické užívateľské rozhranie, ktoré by dokázalo vizuálne vyhodnotiť výsledky a špecifikovať požadované logiky. Objektovo orientovaný prístup nám navyše umožňuje ľahko upraviť už vytvorené objekty, alebo tieto objekty použiť v zmysle dedičnosti.

Aj keď sa jedná o relatívne nepreskúmanú oblasť, naše pokusy preukázali, že koncept seba-modifikácie môže byť prínosom, avšak závisí na tom, k čomu celulárne automaty využívame a ako dobre špecifikujeme vstupy simulácie.

Literatura

[1] Group, O. M.: Unified Modeling Language (UML). 2014, [Online; accessed 22-July-2014].
URL http://www.uml.org/

[2] Ilachinski, A.: *A Discrete Universe*. World Scientific, 2002, ISBN 9789812381835.

[3] Ing. Bohuslav Křena, P.; Ing. Radek Kočí, P.: Úvod do softwarového inženýrství. 2010, [Online; accessed 22-July-2014].
URL https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IUS-IT/texts/IUS_opora.pdf

[4] doc. Ing. František Zbořil, C.; Ing. František Zbořil, P.: Základy umělé inteligence. 2012, [Online; accessed 22-July-2014].
URL
https://www.fit.vutbr.cz/study/courses/IZU/private/oporaizu-esf-5a.pdf

[5] Ing. Zbyněk Křivka, P.; Kolář, D. D. I. D.: Principy programovacích jazyku a objektově orientovaného programování. 2008, [Online; accessed 22-July-2014].
URL https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IPP-IT/texts/IPP-II-ESF-1_1_printable.pdf

[6] Peringer, D. I. P.: Modelování a simulace. 2013, [Online; accessed 22-July-2014].
URL https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IMS-IT/lectures/IMS.pdf

[7] Python: Discrete event simulation. 2014, [Online; accessed 22-July-2014].
URL http://www.python.org/

[8] Wikipedia: Artificial intelligence. 2014, [Online; accessed 22-July-2014].
URL http://en.wikipedia.org/w/index.php?title=Artificial_intelligence&oldid=619129480

[9] Wikipedia: Black-box testing. 2014, [Online; accessed 22-July-2014].
URL http://en.wikipedia.org/w/index.php?title=Black-box_testing&oldid=618955521

[10] Wikipedia: Cellular automaton. 2014, [Online; accessed 22-July-2014].
URL http://en.wikipedia.org/w/index.php?title=Cellular_automaton&oldid=612600578

[11] Wikipedia: Conway's Game of Life. 2014, [Online; accessed 22-July-2014].
URL http://en.wikipedia.org/w/index.php?title=Conway%27s_Game_of_Life&oldid=618924641

[12] Wikipedia: Discrete event simulation. 2014, [Online; accessed 22-July-2014].
URL http://en.wikipedia.org/w/index.php?title=Discrete_event_simulation&oldid=601619413

[13] Wikipedia: Heuristic (computer science). 2014, [Online; accessed 22-July-2014].
URL http://en.wikipedia.org/w/index.php?title=Heuristic_(computer_science)&oldid=607561402

[14] Wikipedia: Heuristic function. 2014, [Online; accessed 22-July-2014].
URL http://en.wikipedia.org/w/index.php?title=Heuristic_function&oldid=594928302

[15] Wikipedia: Object-oriented programming. 2014, [Online; accessed 22-July-2014].
URL http://en.wikipedia.org/w/index.php?title=Object-oriented_programming&oldid=619120891

[16] Wikipedia: Waterfall model. 2014, [Online; accessed 22-July-2014].
URL http://en.wikipedia.org/w/index.php?title=Waterfall_model&oldid=617062535

[17] Wolfram, S.: *A New Kind of Science*. Wolfram Media, 2002, ISBN 1579550088.

Příloha A

Obsah CD

Obsah CD:

- bachelor_thesis.pdf - bakalarska praca vo formate PDF
- bachelor_thesis_source_codes.zip - zdrojove kody bakalarskej prace
- experiments.zip - pokusy (pouzite argumenty, pouzite logiky a vsetky medzi-vysledky)
- install.txt - subor s navodom na instalaciu aplikacie
- readme.txt - subor s popisom obsahu CD
- simulator_logic_samples.zip - zlozky so subormi obsahujucimi ukazkove logiky simulatoru
- simulator_source_codes.zip - zdrojove kody simulatoru