MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Věra Škopková

Logická hra založená na celulárním automatu

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Otakar Trunda

Studijní program: Informatika

Studijní obor: Programování a softwarové systémy

Praha 2017

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V …..... dne..... Podpis autora

Název práce: Logická hra založená na celulárním automatu

Autor: Věra Škopková

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Otakar Trunda, Katedra teoretické informatiky a matematické logiky

Abstrakt: Tato práce se zabývá využitím celulárního automatu pro vytvoření logické hry. Popisuje klady a zápory jednotlivých typů celulárních automatů a jejich potenciál pro použití v logické hře. Dále se věnuje podrobnému návrhu pravidel hry a tvorbě uživatelského rozhraní. Vytvořená hra podporuje jednorozměrné celulární automaty o dvou až deseti stavech a umí pracovat jak s klasickými, tak s totalistickými přechodovými funkcemi. Cílem hry je odhalit hodnoty buněk, které vygeneroval celulární automat, na základě pravidel přechodové funkce, která jsou hráči známa. Hra je velmi variabilní, nabízí uživateli herní i vizuální parametry, které lze měnit. Navíc hra poskytuje dva nástroje pro vytváření nových celulárních automatů. Prvním z nich je vizuální editor, ve kterém lze pohodlně vytvářet nové či upravovat stávající celulární automaty. Druhým je generátor, který je schopný nalézt pravidla a ohodnocení výchozí generace buněk, z nichž lze vygenerovat konkrétní obrazce.

Klíčová slova: celulární automat, logická hra, uživatelská rozhraní

Title: Logic Game Based on Cellular Automaton

Author: Věra Škopková

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Otakar Trunda, Department of Theoretical Computer Science and Mathematical Logic

Abstract: This thesis is concerned with the usage of cellular automata in development of a logic game. It describes benefits and drawbacks of individual types of cellular automata and its potential for being used in a logic game. It contains detailed description of the game rules and the user interface including the design process. The created game supports one-dimensional cellular automata containing from two up to ten states and is able to work with common and totalistic transition functions. The goal of the game is to uncover values of all cells that were generated by the cellular automaton, according to the rules of the transition function that are known to the user. It is possible to set game and visual parameters variedly. In addition, the game provides two tools for creating new cellular automata. The first is a visual editor, which allows the user to create new automata or correct the existing ones. The second is a generator that can deduce rules and values of the default generation of cells from which the specific picture can be generated.

Keywords: Cellular automaton, Logic game, User interface

Chtěla bych velmi poděkovat svému vedoucímu Mgr. Otakaru Trundovi, že mou práci ochotně vedl a že vždy, když jsem potřebovala, mi se vším poradil.

Obsah

1 Úvod

1.1 Motivace

Celulární automaty (CA) slouží k simulaci vývoje buněk a mají velmi zajímavou vlastnost. Pomocí jednoduchých pravidel dokážou generovat různě složité vzory, například pravidelné obrazce. Navíc lze s celulárními automaty dobře experimentovat. Jednotlivým stavům můžeme přiřazovat různé interpretace a do přechodové funkce můžeme ukrýt logická pravidla. Nabízí se tedy otázka, zda by bylo možné navrhnout logickou hru založenou na celulárním automatu.

1.2 Cíl práce

Cílem této práce je vytvořit logickou hru založenou na celulárním automatu. Je tedy potřeba se seznámit se základy problematiky celulárních automatů a navrhnout způsob, jak celulární automat do hry zakomponovat. Následujícím úkolem je vymyslet konkrétní pravidla hry a poté tuto hru implementovat. Hra by měla být přiměřeně obtížná a uživatelsky přívětivá.

Dále by hra měla být co nejvíce variabilní. Měla by uživateli nabízet možnost přizpůsobit si grafické parametry dle vlastních potřeb a měla by být schopna používat více než jeden konkrétní celulární automat. Měl by tedy být popsán způsob, jak do hry přidat nový celulární automat, a bylo by vhodné navrhnout nějaké uživatelsky přívětivé nástroje pro snazší tvorbu nových automatů.

1.3 Struktura práce

V kapitole *Úvod* se seznámíme s tématem, kterého se tato práce týká a se základními pojmy. Následně v kapitole *Celulární automaty* se dozvíme něco o teorii celulárních automatů. Poté v kapitole *Analýza* projdeme všechny zvažované možnosti a navrhneme hru. V kapitole *Nástin architektury* se krátce zmíníme o nejdůležitějších třídách a metodách práce a v kapitole *Řešení* popíšeme problémy, které se během implementace vyskytly. V kapitole *Závěr* shrneme informace o vytvořené hře a nastíníme možnosti případného rozšíření.

1.4 Slovníček

Jelikož se práce zabývá celulárními automaty, budou v následujících kapitolách rozebírány problémy spojené s jejich implementací v rámci logické hry. Pro lepší porozumění jednotlivým částem automatu zde přidávám slovníček pojmů, které budu používat (Tabulka 1.1).

Pojem	Popis
Buňka	Nejmenší jednotka reprezentující konkrétní stav CA.
Stavy automatu	Množina všech unikátních hodnot, kterých mohou buňky automatu nabývat.
Pravidlo	Definuje nový stav konkrétní buňky na základě jejího aktuálního stavu a stavů jejích sousedů.
Přechodová funkce	Soubor všech pravidel automatu.
Automat[1]	Množina všech stavů a přechodová funkce.
Výchozí generace buněk	Generace buněk, ze které jsou generovány všechny ostatní generace, jako jediná není vypočtena pomocí přechodové funkce.
Vývoj buňky	Posloupnost stavů, kterých buňka nabývá v jednotlivých generacích.

Tabulka 1.1: Používané pojmy

[1] Takto může celulární automat chápat laik, formální definice bude zmíněna v kapitole Celulární automaty.

2 Celulární automaty

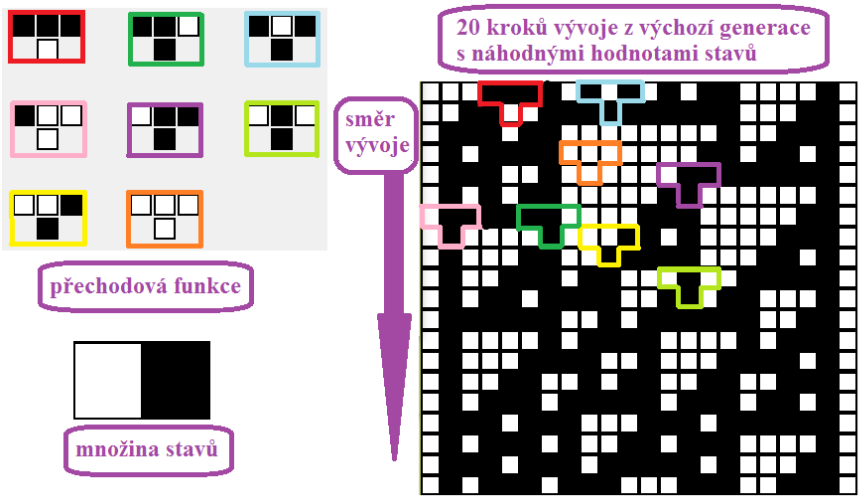
V této kapitole popíšeme celulární automaty a představíme jejich základní vlastnosti. Seznámíme se s jejich různými typy a vytipujeme si nějaká zajímavá pravidla, která mohou být použitelná ve hře.

2.1 Popis celulárních automatů

Celulární automaty jsou diskrétní matematické objekty, které dokážou simulovat vývoj buněk v čase. Obecně je můžeme formálně definovat jako pětici $(N, g, \sum, \varphi, f)$, kde

- $N \in \mathbb{N}$: rozměr automatu
- g : funkce, která pro každou buňku vrátí uspořádanou (multi)množinu stavů jejích sousedů
- $\sum$: neprázdná konečná množina stavů, kterých mohou buňky nabývat
- φ : přechodová funkce určující změnu stavu konkrétní buňky na základě jejího dosavadního stavu a stavů buněk z jejího okolí
- f : funkce pro volbu stavů výchozí generace buněk.

Obrázek 2.1 zobrazuje části jednorozměrného celulárního automatu s *okolím* velikosti 1. *Stav* konkrétní buňky se mění pouze na základě jejího *stavu* a *stavů* buněk v jejím *okolí*. Hodnotu nového *stavu* definuje *přechodová funkce*.



Obrázek 2.1: Popis definice automatu

Během generování běží diskrétní čas, při každém tiku hodin změní naráz všechny buňky svůj *stav*, čímž vytvoří novou generaci. Celý vývoj automatu s pevně danou přechodovou funkcí lze ovlivnit pouze změnou hodnot stavů výchozí generace buněk. Přesto však CA dokážou vytvořit velmi zajímavé, komplexní vzory. Navíc lze výsledky tvorby celulárních automatů dobře vizuálně reprezentovat pomocí mřížky pro zobrazení jednotlivých buněk a pomocí barev jakožto jednotlivých *stavů*.

Celulární automaty lze rozdělit do různých kategorií podle jejich konkrétních vlastností. Nyní zmíníme základní kritéria, podle kterých jsou CA obvykle tříděny.

Počet stavů automatu

Množina *stavů* $\sum$ každého celulárního automatu musí obsahovat alespoň 2 odlišné *stavy*. Nic však nebrání tomu, aby měl automat *stavů* více, jedinou podmínkou je, aby byl jejich počet konečný. Čím větší množinu *stavů* automat má, tím komplexnější má *přechodovou funkci*.

Rozměr automatu

Nejjednodušším typem automatu je jednorozměrný CA, jehož jednu generaci si lze představit jako pás buněk. Přidáme-li automatu druhý rozměr, jednu generaci buněk bude reprezentovat dvojdimenzionální mřížka. V případě vícerozměrného automatu by jednotlivé generace buněk tvořily ještě komplexnější objekty.

Typ přechodové funkce

Základní přechodová funkce obsahuje pro každou možnou uspořádanou n-tici stavů automatu jedno pravidlo. Alternativně mohou být n-tice chápány jako neuspořádané anebo pro konkrétní n-tice nemusí být výsledek *přechodové funkce* vůbec specifikován.

Velikost okolí

Buňka CA obvykle mění svůj *stav* na základě svého aktuálního *stavu* a *stavů* svých sousedů. Podle rozměru automatu je těchto sousedů různý počet, obvykle jimi bývají všechny buňky, které „fyzicky“ sousedí s danou buňkou. Pokud tedy do *přechodové funkce* zahrneme jednoho nejbližšího souseda z každé strany buňky, říkejme této množině buněk *okolí* velikosti 1. Pokud bychom však chtěli do *přechodové funkce* zahrnout i sousedy sousedů těchto buněk, rozšířili bychom

analogicky *okolí* na velikost 2. *Okolí* buňky navíc nemusí být ve všech směrech stejně velké, je možné použít i různá nepravidelná okolí.

Tvar mřížky

Obvykle se buňky zobrazují v pravoúhlé mřížce, ve které jsou reprezentovány jako čtverečky. Lze však pro vykreslování buněk použít i složitější útvary, jako například šestiúhelníky. To zároveň způsobí, že buňky poté mohou mít více sousedů, aniž by došlo ke zvětšení rozměru automatu či jeho *okolí*.

Způsob chování okrajů mřížky

V teorii celulárních automatů je plocha s buňkami nekonečná. Pokud však používáme CA v informatice, potřebujeme použít pouze konečnou plochu. Pak ale nastane komplikace, jelikož okrajové buňky nemají vždy jednoho či více svých sousedů, což komplikuje generování. Tento problém lze řešit čtyřmi základními způsoby.

První možností je při výpočtu, ve kterém chybí informace o nějakém *stavu*, použít místo něj vždy jeden pevně daný *stav*. Odlišnou možností je použít vždy *stav* totožný se *stavem* okrajové buňky, která se výpočtu účastní. Tím docílíme toho, že místo každého chybějícího souseda bude použita odlišná hodnota.

Další dva způsoby, které zmíníme, nabízí možnost, jak simulovat či zobrazit nekonečnou plochu s buňkami. Prvním způsobem je považovat za chybějící buňku na okraji hrací plochy vždy okrajovou buňku na opačném kraji. Hrací plochu si poté lze představit jako plášť válce v případě jednorozměrného CA, případně jako torus pro dvourozměrný CA. Druhý způsob zobrazení spočívá v tom, že pokud vezmeme CA s výchozí generací buněk, která jen v konečném úseku obsahuje buňky s různými *stavy* (všechny ostatní buňky nekonečné řady mají jeden pevně daný *stav*), můžeme vždy zobrazovat pouze tu část vývoje automatu, kterou tyto buňky ovlivní (*stavy* zbytku hrací plochy zůstávají neměnné). Vždy, když dojde při generování poprvé ke změně *stavu* okrajové buňky, rozšíříme zobrazovanou část CA o tuto buňku.

2.2 Jednorozměrné celulární automaty

Jednorozměrné CA jsou nejjednodušším typem celulárních automatů. Jednotlivé generace vývoje tvoří řady buněk, v nichž každá má sousedy pouze vlevo a vpravo. Celý vývoj buněk lze proto zobrazit najednou jako dvourozměrnou mřížku, v níž každý řádek reprezentuje jednu generaci.

Označme symbolem $c_i(t)$ stav i-té buňky v generaci t vývoje a symbolem φ přechodovou funkci CA. Následně pak lze stav i-té buňky v generaci $(t+1)$ pro CA s *okolím* velikosti 1 vyjádřit matematicky jako

$$c_i(t+1) = \varphi[c_{i-1}(t),\ c_i(t),\ c_{i+1}(t)],$$

což lze zobecnit pro CA s *okolím* velikosti o jako

$$c_i(t+1) = \varphi[c_{i-o}(t), \dots,\ c_{i-1}(t),\ c_i(t),\ c_{i+1}(t), \dots,\ c_{i+o}(t)].$$

Elementární celulární automaty

Elementárními celulárními automaty rozumíme jednorozměrné dvoustavové celulární automaty s *okolím* velikosti 1. Celkem jich existuje 256, z nichž některé jsou vůči sobě jen stranově či bitově inverzní. Některé z nich mají velmi zajímavé vlastnosti, které se využívají při generování náhodných čísel či při šifrování. Zabýval se jimi *Wolfram* (Wolfram, 2002), který také navrhl jejich značení pomocí čísel. Tento systém spočívá v tom, že bílým buňkám je přiřazena hodnota 0 a černým hodnota 1 a následně jsou pravidla přechodové funkce seřazena tak, jak ukazuje Obrázek 2.2. Převedeme-li binární číslo, které získáme přečtením hodnot výsledků přechodové funkce zleva doprava, do desítkové soustavy, získáme číslo identifikující dané pravidlo. Na příkladu z obrázku tedy dostaneme $01010011_2 = 83_{10}$.

Obrázek 2.2: Wolframovo pravidlo 83

Wolframova klasifikace tříd celulárních automatů

Celulární automaty lze třídit na základě složitosti vzorů, které generují. Tuto klasifikaci navrhl *Wolfram* a shrnul ji *Schiff* (Schiff, 2008 str. 70). Jsou rozlišovány 4 základní třídy, které však nejsou striktně dané, některé CA mohou stát na rozhraní více z nich. Ačkoli (stejně jako *Schiff* (Schiff, 2008)) tuto klasifikaci zmiňuji v podkapitole o jednorozměrných CA, lze ji aplikovat i na CA vyšších dimenzí.

První třída (Class I)

Tato třída reprezentuje CA s nejjednodušším chováním. Bez ohledu na hodnoty stavů výchozí generace se vždy všechny buňky ustálí na stejných hodnotách stavů. Důsledkem toho jsou ztraceny všechny informace, které se v automatu během vývoje vyskytovaly. Zástupci této třídy jsou například *Wolframova pravidla 94* (Obrázek 2.3), *220* a *249*.

Druhá třída (Class II)

Ve druhé třídě se setkáme s automaty, které produkují periodicky se opakující vzory. V jednotlivých částech automatu se opakující se vzory mohou vyskytovat poměrně nezávisle. Při generování na konečném kruhovém hřišti může být generovaný vzor navíc silně ovlivněn šířkou hřiště (počtem buněk ve výchozí generaci). Do této třídy lze zahrnout *Wolframova pravidla 90* (Obrázek 2.4) a *150*.

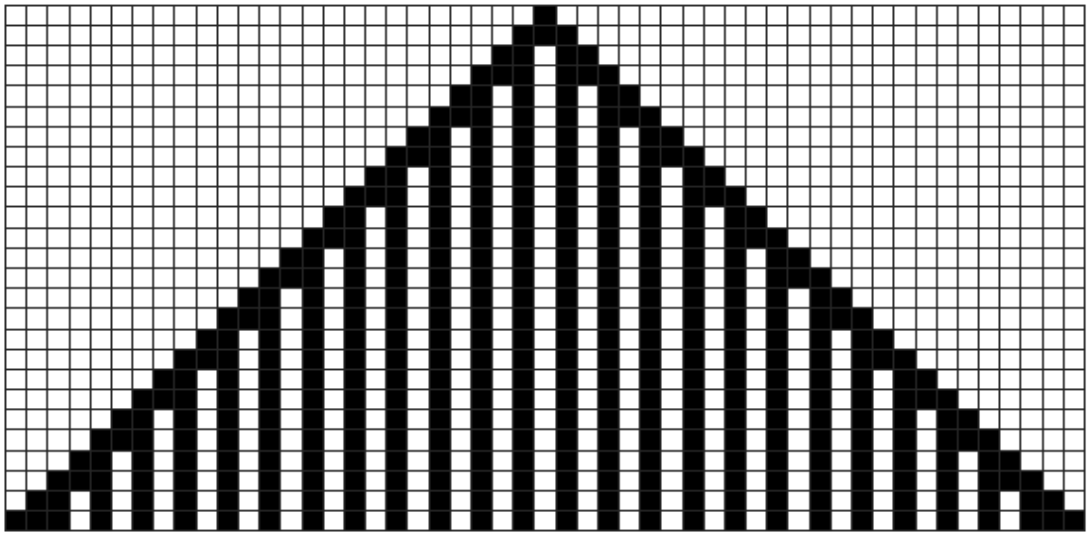
Třetí třída (Class III)

Třetí třída zahrnuje automaty s nepravidelnými, chaotickými vzory, které obvykle obsahují drobné trojúhelníkové útvary. Jsou velmi citlivé na hodnoty výchozí generace buněk, jejichž drobná změna může radikálně změnit celý vytvořený vzor. Pro tyto vlastnosti je tato třída CA využívána při studii náhodnosti. Elementárními celulárními automaty, které patří do této třídy, jsou například *Wolframova pravidla 30* a *126* (Obrázek 2.5).

Čtvrtá třída (Class IV)

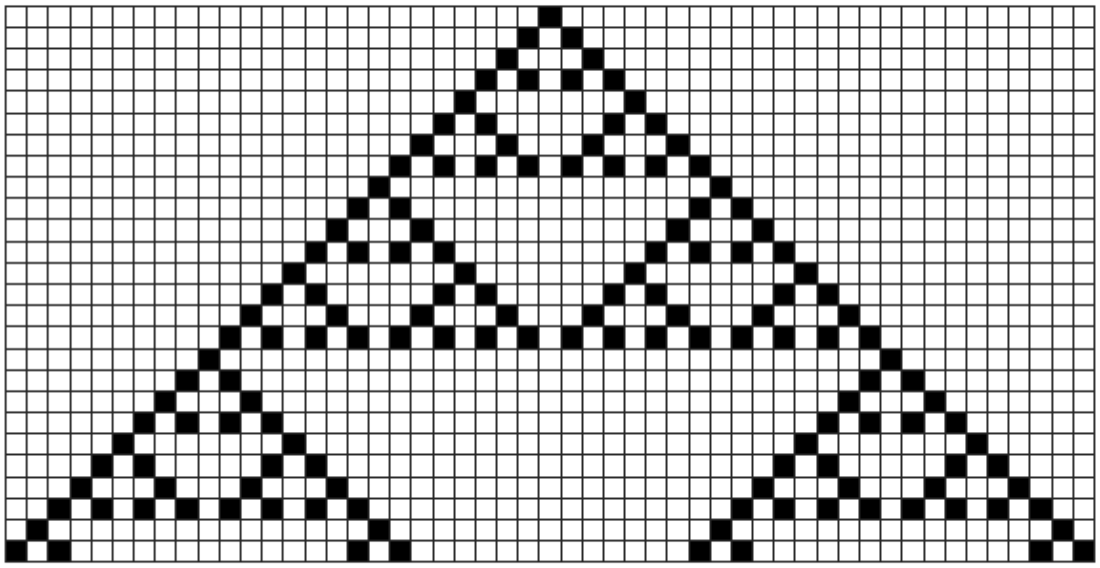
Poslední třída CA sdružuje automaty, které generují pohybující se a mezi sebou interagující lokální struktury. Vzniklé vzory se vzhledově nachází na pomezí druhé a třetí třídy, tedy mezi periodickými a chaotickými. Tyto CA

dokážou přenášet informace v čase a díky tomu by podle *Wolframa* (Wolfram, 1984) mohly být použity k univerzálnímu komplexnímu počítání. Automaty, které mají tyto vlastnosti, jsou kupříkladu *Wolframova pravidla 110* a *137* (Obrázek 2.6).



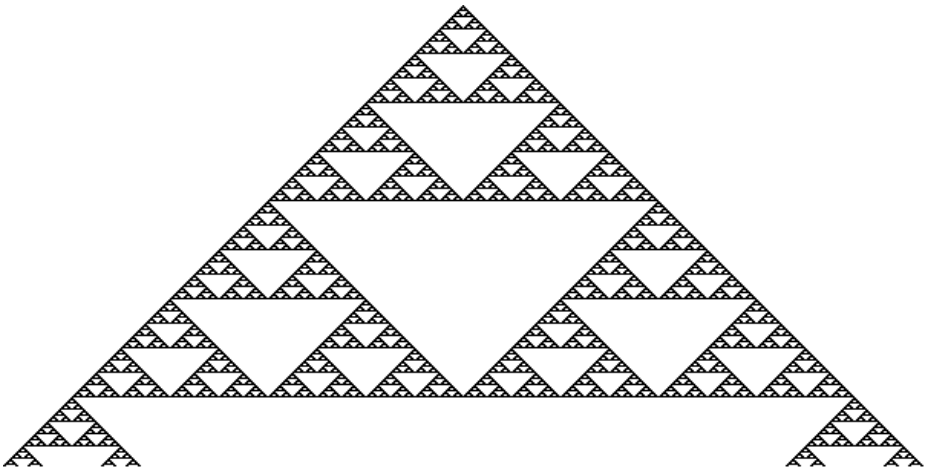
(Wolfram Research, 2003)

Obrázek 2.3: Wolframovo pravidlo 94



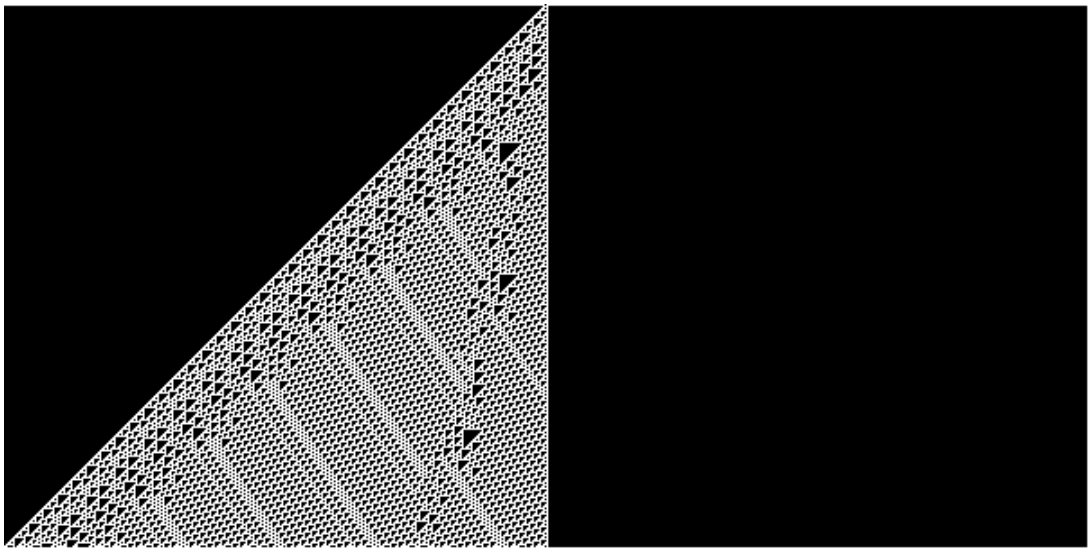
(Wolfram Research, 2003)

Obrázek 2.4: Wolframovo pravidlo 90



(Wolfram Research, 2003)

Obrázek 2.5: Wolframovo pravidlo 126



(Wolfram Research, 2003)

Obrázek 2.6: Wolframovo pravidlo 137

Zajímavá pravidla

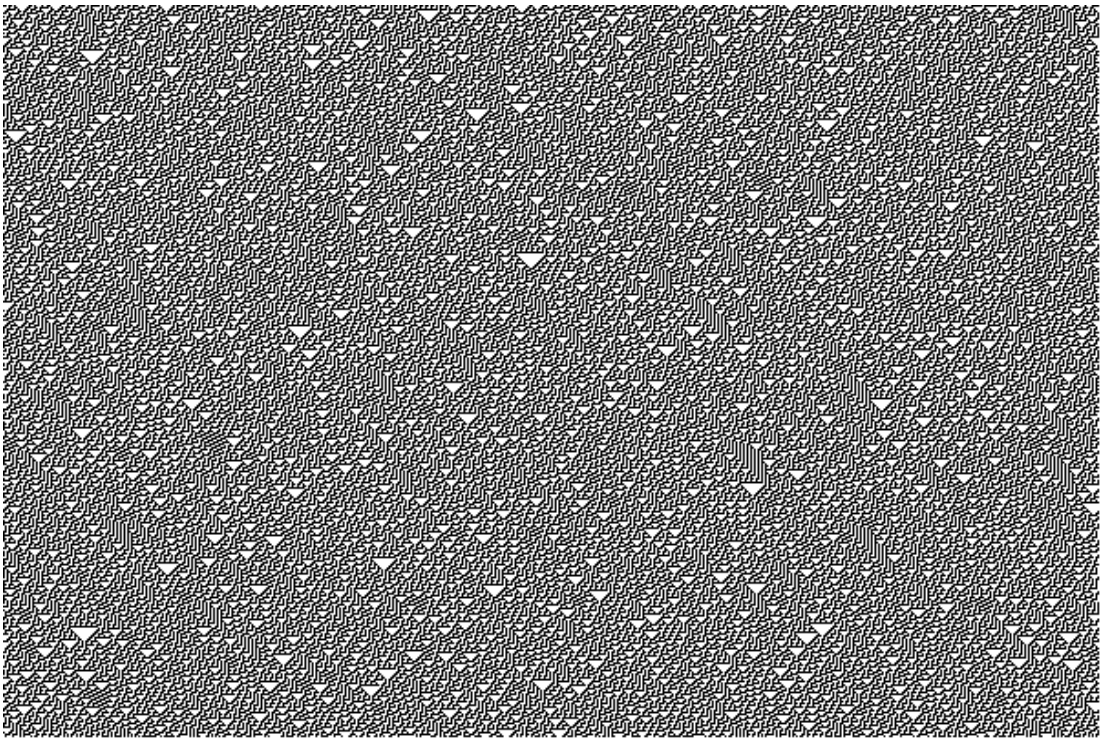
Pravidla, která se v kontextu této práce jeví jako zajímavá, odpovídají *třetí* či *čtvrté třídě Wolframovy klasifikace*. Tato pravidla totiž produkují bohaté, různorodé vzory. Krátce se tedy zmíníme o několika zajímavých elementárních celulárních automatech.

Wolframovo pravidlo 30 (viz Obrázek 2.7) generuje pozoruhodně chaotický vzor (Wolfram, 2002), ve kterém se vyskytuje mnoho různě velkých trojúhelníků. Bylo dokázáno, že sekvence barev získaná ze dvou přilehlých buněk není periodická

(Jen, 1990). Pro tyto vlastnosti se toto pravidlo někdy používá jako generátor náhodných čísel (Weisstein, 2017).

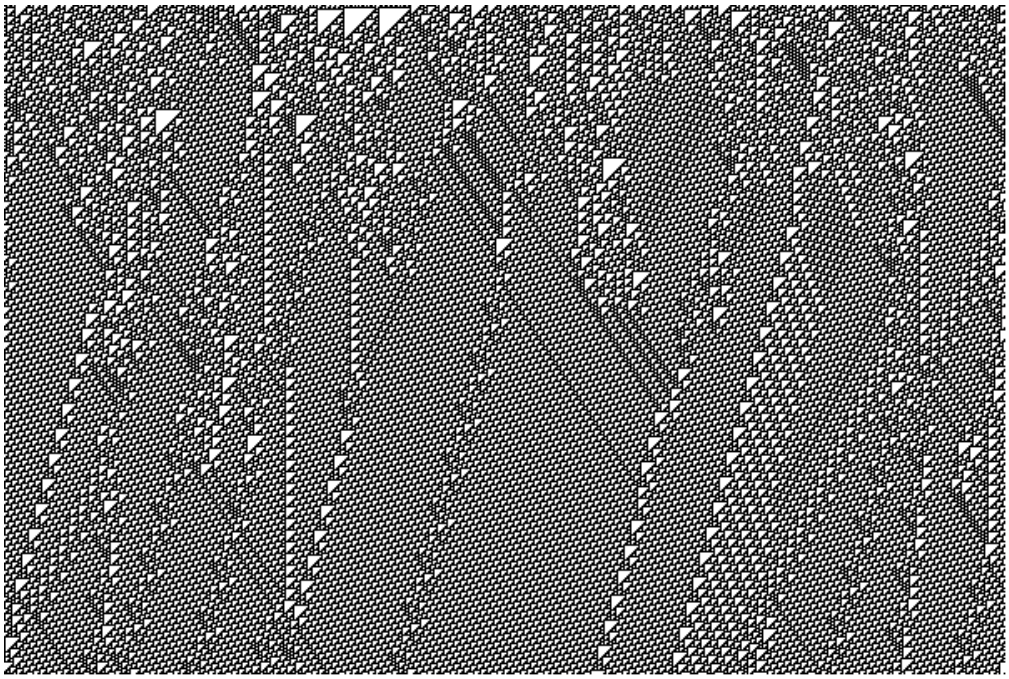
Vzor *Wolframova pravidla 110* (viz Obrázek 2.8) má poněkud jiný ráz, přesto je stále velmi chaotický. *Wolfram* a *Cook* dokázali, že je toto pravidlo *turingovsky kompletní* (Wolfram, 2002), (Cook, 2004) a že tedy může simulovat libovolný algoritmus.

Další zajímavé pravidlo zobrazuje Obrázek 2.9. Opět obsahuje spoustu trojúhelníkovitých útvarů a vzor je celkově chaotický. Zajímavou vlastností tohoto pravidla je, že je samo sobě zrcadlovým obrazem.



(Wolfram Research, 2003)

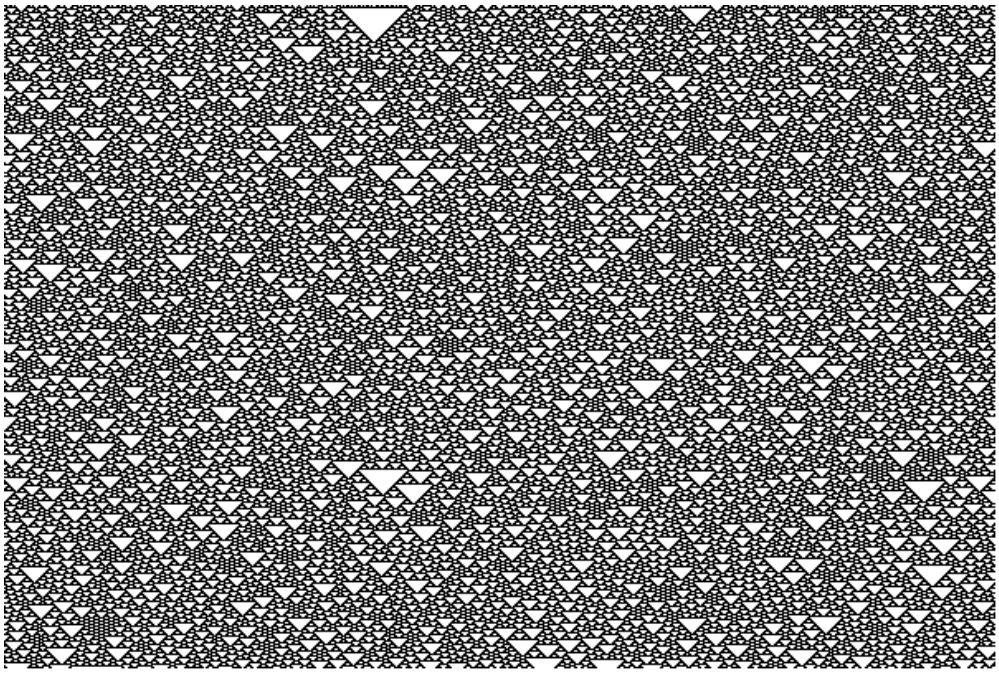
Obrázek 2.7: Wolframovo pravidlo 30



(Wolfram Research, 2003)

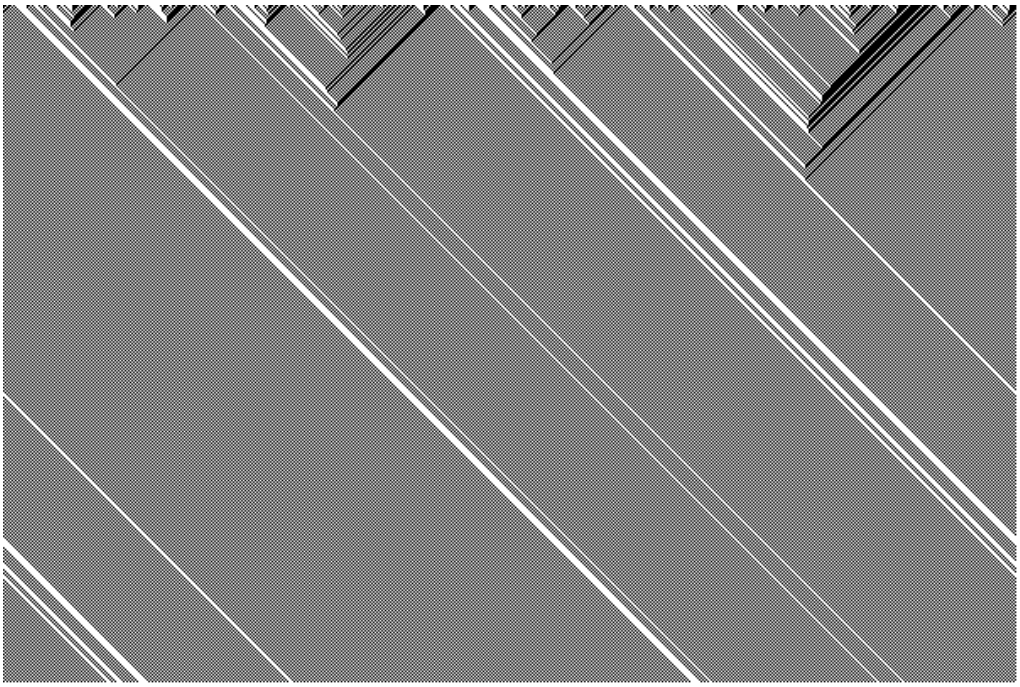
Obrázek 2.8: Wolframovo pravidlo 110

Obrázek 2.10 zobrazuje *Wolframovo pravidlo 184*, jehož vzor vykazuje poněkud odlišný typ nepravidelnosti nežli předchozí pravidla.



(Wolfram Research, 2003)

Obrázek 2.9: Wolframovo pravidlo 126



(Wolfram Research, 2003)

Obrázek 2.10: Wolframovo pravidlo 184

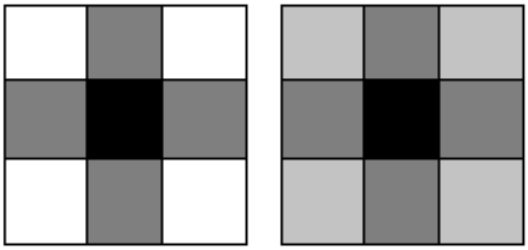
2.3 Dvourozměrné celulární automaty

Dvourozměrné celulární automaty se liší od jednorozměrných CA plochou, kterou zabírá každá jejich generace a počtem buněk v *okolí* o stejné velikosti. Vzhledem k tomu, že dvourozměrné CA mají „větší rozměr“, nabízí oproti jednorozměrným CA více přechodových funkcí, které obsahují více pravidel.

Navíc existují dva různé základní druhy *okolí* buňky (Schiff, 2008). *Von Neumannovo okolí* se skládá z buňky, jejíž stav je měněn (černá) a ze čtyř sousedních buněk (viz Obrázek 2.11). Tyto buňky se nachází v pozicích „nad“, „pod“, „vlevo“ a „vpravo“ vzhledem k prostřední buňce. *Moorovo okolí* zahrnuje všechny buňky z *von Neumannova okolí*, navíc využívá ještě další čtyři buňky, a to buňky sousedící po diagonálách (světle šedá).

Game of Life

Nejznámějším dvourozměrným celulárním automatem je tzv. *Game of Life*, jehož autorem je *John Horton Conway*. Jedná se dvoustavový dvourozměrný celulární automat, který využívá *Moorovo okolí* a většinou bývá zobrazen na periodickém hřišti (torus) (Schiff, 2008).



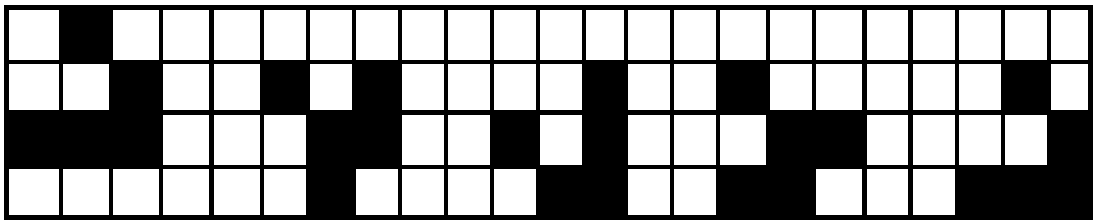
Obrázek 2.11: Druhy okolí buňky

Přechodová funkce má jednoduchá pravidla. Buňky nabývají pouze dvou stavů, z nichž stav 0 symbolizuje *mrtvou buňku* a je značen bílou barvou, černou barvou je zobrazen stav 1 symbolizující *živou buňku*. *Mrtvá buňka* se v následující generaci stane živou, pokud se v jejím *okolí* nachází přesně tři *živé buňky*. *Živá buňka* zůstává *živou*, pokud má ve svém *okolí* dvě nebo tři *živé buňky*. Ve všech ostatních případech buňka zůstává, resp. se stává *mrtvou* (Schiff, 2008).

Game of Life je příkladem dvourozměrného celulárního automatu náležícího do *čtvrté třídy Wolframovy klasifikace* (Schiff, 2008). Podle *Packarda* a *Wolframa* (Packard, a další, 1985) však nebyly nalezeny žádné jiné dvourozměrné CA, které by do této třídy patřily (kromě triviálních variant *Game of Life* (Packard, a další, 1985)).

Různé konfigurace buněk první generace způsobují odlišné chování celého vývoje. Některé konfigurace jsou stabilní a vůbec se nemění. Odlišné zas způsobují periodické střídání konečného množství konfigurací. Velmi zajímavé chování nabízí například tzv. *Glidery*, které po čtyřech generacích opakují ten samý vzor, avšak posunutý o jednu buňku dolů a doprava (viz Obrázek 2.12).

Za zmínku stojící skupinou konfigurací buněk je skupina zvaná *Garden of Eden*. Jedná se o konfigurace, pro které neexistuje jiná konfigurace, ze které by bylo možné tuto generaci vygenerovat (v dané konkrétní přechodové funkci). *Moore* (Moore, 1962) zjistil, že pokud pro nějakou konfiguraci existuje více předchozích konfigurací (které ji generují), potom jistě musí existovat nějaká konfigurace patřící do *Garden of Eden* (Moore, 1962). Tento fakt poté více studoval (Myhill, 1963).



Obrázek 2.12: Glider

2.4 Totalistické celulární automaty

Totalistické CA jsou CA s poněkud odlišnou přechodovou funkcí. Nezáleží v ní na pořadí hodnot jednotlivých buněk v *okolí*, všechna pravidla, která se liší pouze permutací stavů v okolí, dávají stejný výsledek. Celá přechodová funkce je tak jednodušší, jinak lze s těmito CA manipulovat stejně jako se všemi ostatními celulárními automaty.

3 Analýza

V této kapitole popíšeme podrobnou představu tvořené hry a nastíníme alternativy, které byly také uvažovány. Nejprve navrhneme obecná pravidla hry. Poté se zamyslíme nad různými modifikacemi a nakonec zvážíme způsob, jak co nejlépe reprezentovat data z pohledu hráčů.

3.1 Tvorba pravidel hry

Cílem této podkapitoly je shrnout možnosti, jakými lze vytvořit hru založenou na celulárním automatu. Nejprve zvolíme vhodný typ automatu a způsob, jakým bude do hry zakomponován. Následně zvážíme způsoby, jakými by hra mohla hráči napovídat, a nakonec se zaměříme na možnosti, jak hodnotit kvalitu jednotlivých řešení.

3.1.1 Rozměr automatu

Prvním úkolem práce je vymyslet způsob, jakým bude hra celulární automat využívat. Typ celulárního automatu však úzce souvisí s podstatou hry a s jejím cílem.

Jednorozměrné celulární automaty

Buňka jednorozměrného celulárního automatu mění svůj stav v čase pouze na základě svých sousedů po levé a pravé straně (přímých či nepřímých)[2]. Proto může být každá generace buněk vizuálně reprezentována jednoduchou řadou objektů symbolizujících buňky, více takových řad může poté reprezentovat několik generací vývoje. Sloupec buněk pak bude zobrazovat vývoj první buňky sloupce v čase.

V tomto případě lze tedy na jednom plošném hřišti pozorovat celý vývoj n generací buněk od počáteční generace až po n-tou. Výsledek přechodové funkce pro danou buňku a její sousedy je vždy umístěn o řádek níže, přesně pod touto buňkou[3].

[2] Přímý soused – buňka, která se nachází bezprostředně vedle dané buňky. Nepřímý soused – mezi ním a danou buňkou se nachází ještě alespoň jedna buňka.

[3] Předpokládám pravidla, ve kterých buňka mění svůj stav na základě stejného počtu levých a pravých sousedů.

Dvourozměrné celulární automaty

Dvourozměrný celulární automat již potřebuje plošné hřiště pro zobrazení jediné generace buněk. V případě zobrazení více generací najednou vzniká poměrně velký problém. Hřiště pro jednotlivé generace nelze zobrazit celistvě.

Nabízí se tedy možnost pro každou generaci buněk vytvořit nové hřiště a mezi jednotlivými hřišti nějakým způsobem přepínat. Ať už by však byl zvolen jednoduchý způsob přepínání typu mezi jednotlivými hřišti rolovat či pokud by bylo možné na jednom hřišti přepínat jednotlivé generace, hra s tímto druhem automatu by stále měla základní nedostatek – nelze sledovat celý vývoj buněk najednou.

Vícerozměrné automaty

Vícerozměrné celulární automaty mají tentýž problém jako automaty dvourozměrné, je u nich těžké zobrazit vývoj více generací najednou. Navíc s každým přibývajícím rozměrem narůstá jejich složitost, nastává problém, jak je zobrazit na dvoudimenzionálním monitoru, a pro člověka jsou jejich přechodové funkce prakticky nepředstavitelné.

Jelikož očekávám, že hra bude zobrazovat statickou hrací plochu reprezentující rovinný útvar, jednoznačně se nejvíce nabízí jednorozměrný celulární automat. Ten umožní použít obdélníkové hrací pole složené z $n * m$ buněk a do něj vygenerovat celý vývoj m generací n buněk z první řady.

3.1.2 Smysl hry

Dalším úkolem je vymyslet, jakým způsobem bude přechodová funkce jednorozměrného celulárního automatu ve hře využita.

První možností je nechat přechodovou funkci automatu utajenou. Poté ji hráč může luštit a výsledkem hry může být rekonstrukce kompletní přechodové funkce. Tato varianta hry však má hned dva problémy.

Na základě typu přechodové funkce a pomocí prvku náhody při generování výchozí generace buněk se může stát, že některá pravidla přechodové funkce nebudou v daném konečném hřišti použita. Potom by některé instance hry mohly být velmi snadné (jen málo pravidel) jiné zase naopak obtížné (všechna pravidla).

Další otázkou je, jakým způsobem by hráč měl přechodovou funkci luštit. Pokud by na počátku hry byly všechny buňky zakryty a hráč by je postupně odkrýval, ve výsledku by přechodovou funkci neluštil, ale spíše opisoval podle hodnot hřiště. Případně hlouběji přemýšlející hráč by z doposud odkrytých výsledků mohl být schopen logicky odhadnout výsledek pravidla, které by se nápadně podobalo nějakému již odkrytému. V tu chvíli by přechodová funkce musela nést jistou logickou myšlenku.

V případě, že by přechodovou funkci bylo skutečně možné logicky interpretovat, nabízela by se ještě jedna alternativa, a to nechat hráče odhalit její význam. Potom by však celé hřiště mohlo zůstat od počátku odkryté a hráč by si ho vlastně jen prohlížel. Tento nápad se také nejeví jako vhodný, už například proto, že není snadné si odpovědět, jak by hráč vlastně své řešení programu sdělil. Případně vyhrál by, pokud by vymyslel jinou rozumnou logickou reprezentaci dané přechodové funkce?

Jinou možností je přechodovou funkci zveřejnit, aby hráč mohl do jejích pravidel kdykoli nahlédnout. S touto znalostí je však snadné odhalit celý vývoj všech buněk, je-li známá výchozí generace. Z toho vyplývá, že v případě zveřejnění přechodové funkce nesmí být zveřejněna výchozí generace buněk ani žádná jiná kompletní generace (odkrytí části hřiště pod odhalenou řadou by bylo triviální). V tomto případě by tedy úkolem hráče mohlo být postupně odhalovat jednotlivé generace (v pořadí od nejmladších po nejstarší) a určit správné hodnoty výchozí (nejstarší) generace.

Otázkou tohoto způsobu hry je, jak by hráč mohl začít, aby hodnoty prvních buněk nemusel hádat. Vhodnými buňkami, jejichž hodnoty mohou být zveřejněny, jsou buňky nejmladší generace. Jejich odkrytí ničemu nevadí, protože v hřišti mají pozice pouze výsledků přechodové funkce, ale dále se z jejich hodnot další generace neodvozují.

Dále by dle typu přechodové funkce bylo možné zvolit další buňky, které by mohly být odkryty, ale neprozradily by příliš o výchozí generaci. Vhodnými kandidáty by mohly být určité pruhy zobrazující kompletní vývoj nějakých buněk či drobné nepravidelné útvary uprostřed hrací plochy.

Jako vítěznou volím druhou variantu, kdy se hráč snaží rozluštit hodnoty buněk výchozí generace. Kompletní přechodová funkce i s případným slovním popisem, pokud její logika zasluhuje komentář, bude hráči po celou dobu hry k dispozici. Cílem hry bude rozluštění celého hřiště, což odpovídá stavu, kdy je rozluštěna výchozí generace buněk. Byla-li by výchozí generace rozluštěna bez odkrytí některých políček hřiště, odkrytí těchto polí je při znalosti přechodové funkce již triviální.

3.1.3 Nápověda

Další otázkou je, zda by hráč měl hrát jen s pomocí vlastní hlavy, či by měl mít k dispozici nějakou nápovědu nebo pomocný nástroj. Vzhledem k charakteru hry je však žádoucí, aby hra neřešila zadání sama, ale aby hráči poskytovala pouze opěrnou hůlku.

Jelikož je ve zvolených pravidlech hry kompletní přechodová funkce viditelná, mohl by si hráč „vypsat na papír“ pro každou konkrétní buňku skupinku pravidel, která by pro ni mohla být použita. Pokud by tedy pomocný nástroj uměl vytvořit tento „výpis“, neprozradil by nic tajného, pouze by uživateli urychlil manuální práci a omezil by nechtěné přehlédnutí nějakého pravidla. Hráč by poté mohl jednotlivé skupinky pravidel hlouběji analyzovat, hledat jejich průniky, a tím redukovat množství použitelných pravidel pro některé buňky. Takovýto nástroj by zaručeně usnadnil hraní, ale logickou obtížnost hry by nesnížil.

Pokud by však nápověda sama kombinovala skupinky pravidel pro sousední buňky a prováděla s nimi nejrůznější množinové operace, byla by schopna (dle míry své inteligence) odhalit spoustu situací, které nemusí být na první pohled zřejmé. Jelikož i odhalení jediné buňky může být klíčové pro rozluštění spousty dalších buněk, využitím chytré nápovědy by si hráč mohl velmi usnadnit svou situaci. Takováto nápověda by poté musela být nějakým způsobem pokutována, aby ji hráč nevyužíval příliš často.

Protože je cílem práce vytvořit dobře hratelnou logickou hru pro člověka, vyberu první možnost, tedy že hra nebude poskytovat nápovědu, ale pouze pomocný nástroj. Ten pro jednotlivá pole bude schopen vyfiltrovat pravidla, která by mohla být v daném místě platná (v aktuálním stavu skrytých a odkrytých buněk).

Dále bude mít hráč možnost si tato pravidla pro jím zvolené buňky ponechat zobrazená během hry a případně si v seznamu těchto pravidel navíc některá sám vyškrtnout. Jelikož jsem však nenalezla žádný stručný a výstižný název pro pojmenování tohoto pomocného nástroje, budu mu v rámci této práce říkat *Nápověda*.

3.1.4 Hodnocení hry

Na závěr této podkapitoly je potřeba ještě upřesnit, jak bude hra ohodnocena. Dle stanoveného cíle hry je zřejmé, že vítězem se stane každý, kdo rozluští celou hrací plochu. Hru je však možné hrát tak, jak se očekává, logickým uvažováním a rozborem možností pro jednotlivé buňky, ale i „hrubou silou“, kdy hráč hodnoty pouze tipuje. Je tedy potřeba ohodnotit i styl řešení.

Nejjednodušší variantou, jak ohodnotit způsob, kterým byla hra vyřešena, je změřit čas strávený řešením. Nabízí se však otázka, jestli rychlejší řešení je vždy to lepší. Vezměme si příklad, kdy nám na hřišti zbývá poslední neodkrytá buňka. Pokud se budeme snažit přijít na to, jakou hodnotu má tato buňka mít, můžeme přemýšlením nad tím strávit například pět minut. Pokud se však rozhodneme, že hodnotu této buňky jednoduše uhodneme, a jeden pokus nám zabere například deset sekund, za pět minut zajisté uhodneme správnou hodnotu u automatu, který bude mít až 30 různých stavů. Tento příklad tedy může hráče od logického přemýšlení spíše odradit.

Alternativou k měření času může být penalizace za chyby. Za každou chybně označenou buňku by hráč mohl dostat trestné sekundy či minuty anebo trestné body. Účelem penalizace by mělo být, aby hráče odradila od riskování, pokud to není nutné. Zároveň by ale neměla být příliš vysoká, aby po chybě mělo smysl ve hře vůbec pokračovat.

Jako zajímavé se jeví hodnocení hry založené na způsobu bodové penalizace. Aby však hráči pouze nesbírali trestné body, čímž by docházelo k porovnávání počtu chyb, kterých se dopustili, zahrneme do bodování hráčů i fakt, v jaké části hry hráč chybu udělal. Za každé korektně odkryté políčko hráč dostane určitý počet bodů. Výše odměny za jednu korektně odkrytou buňku však nebude pevně dána. Na začátku hry bude mít bodový příděl minimální hodnotu. Postupem času,

po bezchybném odkrytí určitého počtu buněk, začne bodový příděl narůstat, takže odhalení každé další buňky bude cennější. Dojde-li však k chybě, bude hráč potrestán jednorázovou pokutou odečtením určitého množství bodů, jehož výše se bude také měnit v závislosti na výši bodového přírůstku (vyšší zisk je podmíněn rizikem vyšší ztráty). Zároveň se však po chybě výše aktuálního bodového přírůstku i případné další pokuty sníží na minimum.

Příklad

Pro upřesnění uvedeme příklad. Hráč začíná hru a za každou odkrytou buňku obdrží 1 bod. Naopak za každou chybu mu bude 10 bodů odebráno. Pokud správně odkryje 5 buněk v řadě, jeho bodový příděl za správně odkrytou buňku se zvýší na 2 body. Pokuta za chybně odkrytou buňku se však také zvýší, na 20 bodů. Při odkrytí dalších 5 buněk v řadě se bodový příděl zvýší na 3 body a pokuta na 30 bodů atd. Pokud kdykoli během hry udělá hráč chybu, zaplatí pokutu v aktuální výši (např. 30 bodů), ale zároveň budou jeho bodový příděl i pokuta sníženy na minimální úroveň (v našem případě na 1 plusový a 10 minusových bodů).

3.2 Použitelné typy automatů a pravidel

V předchozí podkapitole jsme rozhodli, že ve hře použijeme jednorozměrný celulární automat, jehož přechodová funkce bude pro hráče viditelná. Jelikož však jednorozměrný celulární automat není pouze jeden, je naším dalším úkolem rozhodnout, jaký konkrétní automat či typ automatu ve hře použijeme.

Velikost okolí

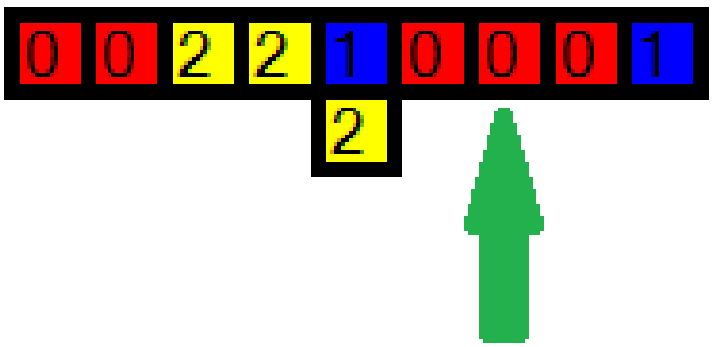
Nejprve se zamyslíme nad tím, jak velké *okolí* buňky v *přechodové funkci* použijeme. *Okolím* míníme počet buněk na jedné straně od buňky, jejíž stav měníme.

Nejběžnějším případem je *okolí* velikosti 1, tj. každá buňka mění svůj stav na základě jednoho levého a jednoho pravého souseda. V automatu s n stavy a s touto velikostí *okolí* se může objevit n^3 různých trojic buněk. Každá z těchto trojic poté reprezentuje jedno *pravidlo přechodové funkce*. Zvýšíme-li velikost *okolí* na 2, potom se počet pravidel *přechodové funkce* přirozeně zvýší na n^5, při *okolí* velikosti 3 bude pravidel n^7 atd. Pokud tedy uvážíme automat s n stavy a s velikostí *okolí* o, bude jeho *přechodová funkce* obsahovat $n^{(2o+1)}$ pravidel.

S narůstajícím *okolím* bude nutné, aby se uživatel v jednom okamžiku soustředil na větší množství buněk. V případě *okolí* velikosti 1 tvoří jedno pravidlo 3 buňky v řadě. Toto množství je rozumně přijatelné, 3 buňky vedle sebe jsou velmi dobře rozeznatelné. V případě okolí velikosti 2 bude jedno pravidlo tvořit 5 buněk v řadě. Soustředit zrak na 5 buněk naráz už je o něco obtížnější. Na více buněk se již letmým pohledem prakticky ani zaměřit nelze, například rozlišení šesté a sedmé buňky u automatu s *okolím* velikosti 4 by již vyžadovalo nepříjemné odpočítávání buněk (viz Obrázek 3.1).

Pokud bychom chtěli s *okolím* buněk experimentovat, mohli bychom navrhnout libovolná asymetrická *okolí*. Tento typ *okolí* by však stejně jako vyšší řády symetrických *okolí* zhoršoval přehlednost a zvyšoval počet pravidel automatu. Jedinými možnými přijatelně velkými asymetrickými *okolími* by tedy mohla být *okolí* taková, že by buňka měnila svůj stav na základě jedné buňky nalevo a dvou napravo (či naopak), případně pouze jedné buňky nalevo a žádné napravo (či naopak). Od tohoto asymetrického *okolí* neočekávám, že by mělo nějakou zásadní výhodu oproti použití symetrického *okolí* velikosti 1 nebo 2, proto tuto možnost rovnou zavrhuji.

Použitím pravidel s *okolím* velikosti 2 bude mít automat se stejným množstvím stavů o 2 řády více pravidel než stejný automat s *okolím* velikosti 1. Automat s *okolím* velikosti 2 tedy ztíží hru jak vizuálně, tak i množstvím použitých pravidel. Proto použijeme pouze automaty s *okolím* velikosti 1 a zvážíme jiné možnosti rozšíření hry.



Obrázek 3.1: Nepřehledně velké okolí

Počet stavů automatu

Alternativou ke zvětšení *okolí* buňky je zvýšit počet *stavů* automatu. *Elementární celulární automat* má 2 *stavy* a jeho *přechodová funkce* obsahuje 8 pravidel. Takovýchto automatů existuje pouze 256 a pouze některé z nich tvoří dost zajímavé vzory na to, aby byly použitelné v logické hře. Pokud však použijeme *stavy* 3, *přechodová funkce* bude obsahovat 27 pravidel. Automatů tohoto rozměru je již 3^{27}, což je dostatečně velké číslo na to, abychom se necítili omezeni na pouze několik zajímavých automatů.

Přidáním každého dalšího *stavu* do automatu tedy rapidně zvýšíme množství možných celulárních automatů, které lze vytvořit. Toto množství bychom sice rozšířili i zvětšením *okolí* buňky zmiňovaným výše, použití více *stavů* však slibuje možnost vytváření zajímavějších útvarů na hrací ploše.

Navíc očekávám, že počet *stavů* automatu půjde jednoduše parametrizovat a že použití automatu s vyšším počtem *stavů* nijak nenaruší kód dříve připravovaný pro automat s nižším počtem *stavů*. Důsledkem toho hra nebude muset obsahovat pevně daný počet *stavů*, ale pouze maximální počet *stavů* bude omezen nějakou rozumně zvolenou konstantou.

Typ pravidla přechodové funkce

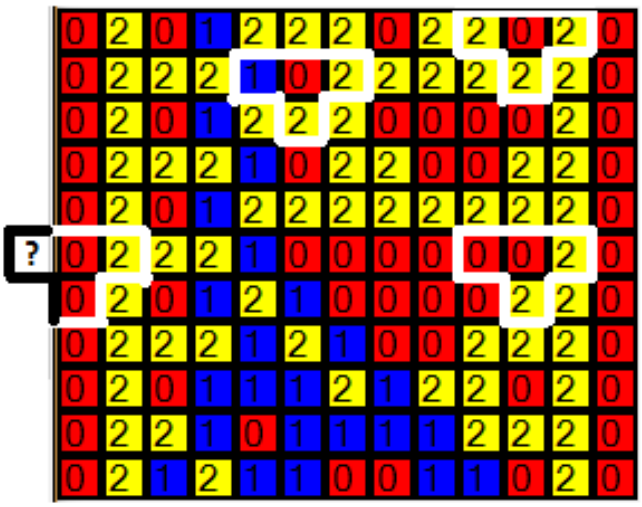
Základním typem *přechodové* funkce, kterou lze využít, je již výše nastíněný typ, kdy každá uspořádaná n-tice (v našem případě trojice) buněk má svůj unikátní výsledek. Nazývejme ji *běžnou přechodovou funkcí* a jakožto základní typ *přechodové funkce* ji určitě do práce zahrnu. Již víme, že pro n-stavový automat obsahuje n^3 pravidel. Jelikož pro každou trojici buněk musí existovat (maximálně) jeden výsledek, nelze počet pravidel pro daný automat zvyšovat.

Existují však způsoby, jak počet pravidel snížit. Prvním z nich je použít *neúplnou přechodovou funkci*. Tu snadno vytvoříme z *běžné přechodové funkce* tak, že libovolný nenulový počet pravidel zahodíme a výsledek zahozených trojic označíme jako nedefinovaný. Použijeme-li však toto pravidlo a začneme generovat náhodný vývoj buněk, může se stát, že se dostaneme do stavu, kdy budeme potřebovat použít zahozené pravidlo.

Takovéto pravidlo tedy lze použít dvěma způsoby. Ty se liší v tom, jak zvolíme výchozí generaci buněk, abychom měli jistotu, že se vyškrtnuté trojice buněk v celém vývoji nebudou vyskytovat. Budeme-li výchozí generaci vytvářet náhodně, budeme muset vždy zkoušet generovat vývoj, dokud to bude možné, a v případě neúspěchu začít s novou výchozí generací. Alternativou je, že bychom dopředu znali jednu či více použitelných výchozích generací a ty použili.

Ve druhém případě by to ale znamenalo, že by automat generoval pevně dané vzory a byl by použitelný tolikrát, kolik svých výchozích pozic „zná“. Tento druh automatu nenabízí tolik možností jako automat s *běžnou přechodovou funkcí*, avšak slibuje možnost využití specializovaných pravidel pro vygenerování velmi zajímavých pevných vzorů. Pomocí vhodně zvolených pravidel a výchozí generace může být možné vykreslovat jednoduché obrázky. Proto tento druh přechodové funkce nezavrhnu a vytvořím alespoň nějakou podporu těchto pravidel.

Jinou možností je využít *totalistickou přechodovou funkci*. Ta se liší od *běžné* přechodové funkce tím, že v přechodových pravidlech používá neuspořádané n-tice stavů (*běžná přechodová funkce* používá uspořádané n-tice). Těch je (při pevně daném počtu stavů) přirozeně méně, *přechodová funkce* je tak jednodušší. Pokud přechodové pravidlo obsahuje logický význam, ve kterém nezáleží na pořadí stavů v trojici, je jednodušší zvolit pravidlo *totalistické*. Tento druh *přechodové funkce* se jeví jako užitečný, a proto i jej do práce zahrnu.



Obrázek 3.2: Chybný výsledek přechodové funkce

Hrací plány

V teorii *celulárních automatů* je plocha s buňkami automatu chápána jako nekonečná. V každé generaci máme nekonečně mnoho buněk, které se dále a dále vyvíjí. Hrací plán mé hry však musí být pouze konečný. Proto je potřeba stanovit pravidla, jak zacházet s okrajovými buňkami.

Plochý hrací plán

Nejjednodušší možností je použít *plochý* hrací plán. V něm bude okrajovým buňkám scházet vždy po jednom sousedovi. Tuto situaci můžeme ošetřit tak, že si na místě chybějící buňky domyslíme buňku s pevně daným *stavem*.

Tento způsob je jednoduchý, přímočarý, ale má jeden drobný problém. Zaprvé hráči nemusí být zřejmé, jakou hodnotu mají okrajové buňky. Je tedy vhodné, aby tyto pomocné buňky byly zahrnuty do hrací plochy a na začátku hry byly odkryty. Zadruhé se u okrajů může *přechodová funkce* jevit jako chybná. Pakliže všechny okrajové buňky budou například ohodnoceny *stavem* 0, ale v *přechodové funkci* neexistuje pravidlo, ve kterém by z trojice buněk obsahující hodnotu 0 vzešel výsledek 0, jsou hodnoty okrajových buněk nesmyslné.

Tuto situaci navrhuji řešit následovně: Pokud kromě okrajových buněk zobrazíme i následující sloupce buněk, nejbližší buňky, které bude hráč řešit, budou buňky třetího sloupce od kraje (zleva či zprava). To způsobí, že korektně vygenerované hodnoty buněk budou odděleny od napevno ohodnoceného kraje, a kdykoli hráč bude řešit hodnotu libovolné zakryté buňky, bude se vždy opírat o buňky validní, případně odkryté.

Podíváme-li se na Obrázek 3.2, vidíme, že pokud má prostřední buňka hodnotu 0 a pravá buňka hodnotu 2, výsledkem je vždy hodnota 2. Zvýrazněné pravidlo na levém okraji tedy dává chybný výsledek, a proto musí být tato hodnota odkryta na počátku hry, aby hráče nemátla.

Chybějící buňky pravidel *plochého* hracího plánu by také mohly nabývat totožných hodnot, jako jsou hodnoty buněk na okrajích. Tuto variantu však nebudu implementovat proto, že je příliš podobná výše zmíněné variantě s pevně ohodnocenými okraji, a proto, že by mohla ještě více mást uživatele.

Cyklický hrací plán

Alternativou k plochému hracímu plánu je *cyklický* hrací plán. Ten si lze představit tak, jako kdyby se buňky generovaly na povrch válce a výchozí generace buněk tvořila kruh. Výhodou tohoto hracího plánu je, že každá buňka má vždy dva sousedy, a tedy neexistují žádné problematické okrajové buňky. Nevýhodou je, že tento plán nelze intuitivně dvoudimenzionálně zobrazit.

Nabízela by se možnost vykreslit hrací plán jako mezikruží, ale vzhledem k tomu, že by v mladších generacích vznikaly mezi buňkami čím dál větší mezery, toto řešení by zřejmě bylo esteticky špatné. Lze tedy provést rozbalení pláště válce do roviny a to, že je hrací plán cyklický, si pouze představovat.

Jelikož každému hráči může vyhovovat jiný druh hracího plánu, rozhodla jsem se, že implementuji obě varianty. Jako výchozí bude *plochý* hrací plán, protože ho považuji za intuitivnější při vytváření *neúplných* přechodových funkcí s pevně danou výchozí generací.

Shrnutí

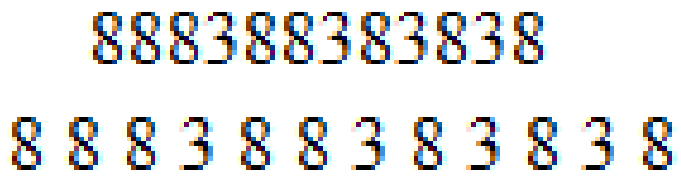
Nyní jsme stanovili, že budeme používat jednorozměrné celulární automaty s velikostí okolí 1. Počet stavů automatu bude proměnný a přechodová funkce bude moci být *běžná*, *neúplná* nebo *totalistická*. Hra bude moci být vygenerována na *plochém* nebo *cyklickém* hracím plánu.

3.3 Způsob grafické reprezentace hry

V poslední podkapitole této kapitoly je potřeba rozhodnout, jakým způsobem budou *stavy* automatu a buňky hrací plochy reprezentovány. Nejprve zvážíme všechna pro a proti znakové i grafické reprezentace *stavů* a poté rozhodneme, jakou konkrétní třídu použijeme při implementaci námi zvoleného řešení.

3.3.1 Způsob zobrazení jednotlivých stavů

Nejprve musíme rozhodnout, jak automat reprezentovat pro uživatele. Toho zajímá pouze vizuální reprezentace jednotlivých *stavů* a princip *přechodové funkce* (či výčet jejích pravidel, pokud nemá logický význam). Důležitým požadavkem je zřetelná odlišitelnost jednotlivých *stavů*.

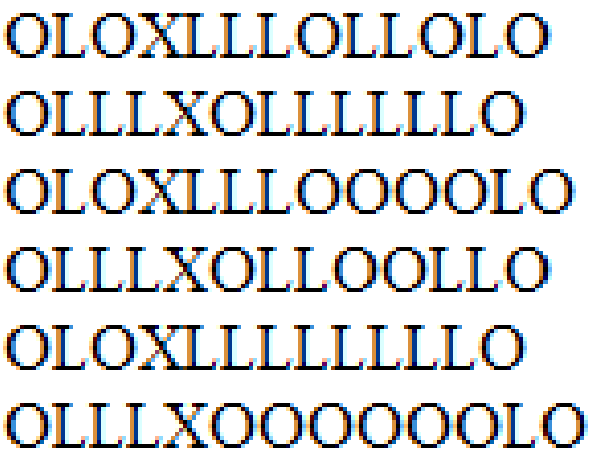


Obrázek 3.3: Špatné odlišení podobných číslic

Nejjednodušší variantou může být reprezentace pomocí číslic, ve které bude n-stavový automat obsahovat *stavy* $0 - n$. Tato reprezentace je poměrně intuitivní, ale má jednu zásadní nevýhodu. Představíme-li si hrací plochu ve tvaru obdélníka a na ní změť buněk popsaných čísly, nebude to velmi přehledné. Hráč vždy potřebuje snadno rozeznávat jednotlivé instance nějakého konkrétního *stavu*, což se mu v číselné reprezentaci nemusí dařit, protože může některé výskyty snadno přehlédnout.

Názornou ukázkou je Obrázek 3.3. Čtyři trojky ukryté mezi osmi osmičkami jsou poměrně nenápadné. I případné vložení mezer mezi jednotlivé číslice viditelnost nijak rapidně nezvýší.

Alternativou reprezentace pomocí číslic je reprezentace pomocí znaků. Těch je větší množství, a tak lze nalézt takové, které se od sebe dostatečně liší. Například znaky „O I X M E A P S U H“ by mohly být přijatelnou reprezentací desetistavového automatu. Přesto však i znakové zobrazení není zrovna esteticky ideální (viz Obrázek 3.4). Jednotlivé znaky jsou různě široké, a tak není pole na pravé straně dokonale zarovnané.



Obrázek 3.4: Reprezentace pomocí písmen

Jednoznačně uživatelsky přívětivějším způsobem zobrazení jednotlivých *stavů* je barevná reprezentace. Každý *stav* může být reprezentován jednou barvou, jedinou podmínkou pro dobré rozlišení je, aby jednotlivé barvy nebyly příliš podobné. Problémem, který u použití této reprezentace může nastat, může být obtížnější interpretace *přechodové funkce*. V případě, že má *přechodová funkce* automatu logický význam, jednotlivé *stavy* reprezentují určité hodnoty. Konkrétně v *matematické*[4] *přechodové funkci stavy* zastávají pozice čísel. V tuto chvíli však barevná reprezentace může být mírně matoucí, neboť většině lidí se přirozeně lépe pamatuje řada čísel od jedné do šesti nežli uspořádaná šestice barev *„červená, modrá, žlutá, zelená, hnědá, černá“*.

V implementaci této práce použiji možnost reprezentace *stavů* pomocí barev. Jelikož však předpokládám hojné využití *matematických přechodových funkcí*, bude každý *stav* doplněn o číselnou hodnotu pořadí *stavu*, který reprezentuje. Během hry si hráč bude moci zvolit, zda číselnou reprezentaci jednotlivých *stavů* chce nebo nechce zobrazit.

3.3.2 Možnosti vykreslení stavů na obrazovku

Jedním z klíčových implementačních rozhodnutí této práce je, jakým způsobem vykreslovat jednotlivé buňky. Spuštěná hra musí zobrazit celou hrací plochu složenou z $n * m$ buněk, dále pak p pravidel složených ze tří buněk původní generace a z jedné buňky výsledku, tj. $4p$ buněk pro zobrazení všech pravidel. Budeme-li hrát na ploše $10 * 10$ polí, při využití dvoustavového celulárního automatu bude potřeba vykreslit $10 * 10 + 8 * 4 = 132$ buněk[5]. Pro případ, že by automat měl *stavy* tři, bude nutno vykreslit dokonce 208 buněk. Pokud bychom chtěli pracovat s vícestavovým automatem, počet buněk, které budeme muset zobrazit, bude exponenciálně narůstat.

Vzhledem k obrovskému množství buněk, které hra musí zobrazovat, je přirozeným požadavkem, aby vykreslování bylo dostatečně rychlé. Při spuštění nové hry může být akceptovatelná drobná prodleva do cca 2 sekund. Během hry však již překreslování buněk nesmí hráče nijak viditelně zdržovat či rozptylovat. Pokud

[4] Takto nazývejme takové *přechodové funkce*, jejichž výsledky mohou být součet, rozdíl, medián apod. z hodnot některých buněk.

[5] $n = 10, m = 10, p = 2^3 = 8$

tedy hrací plocha obsahuje více polí, nežli se vejde do okna hry (a tím pádem je potřeba, aby okno hry bylo posuvné), je požadavkem, aby se buňky rozumně vykreslovaly právě při pohybu posuvníku.

Tlačítka

Nejjednodušší možností, jak buňky reprezentovat, je použít běžná tlačítka (třída *Button*). Jelikož má hráč na jednotlivé buňky klikat, je tato třída velmi vhodnou, neboť každé její instanci lze snadno nastavit vhodnou odezvu na událost kliknutí. Dále lze jednoduše tlačítka obarvovat, psát na ně text a měnit jim velikost, takže splňují základní požadavky, aby mohly reprezentovat buňky.

Bohužel však tato tlačítka mají jednu nepříjemnou vlastnost. Každé se vždy nachází v kolekci *Controls* nějakého formuláře, panelu apod. Vytvoříme-li instanci tlačítka, musíme ji do nějaké konkrétní kolekce vždy přidat. V našem případě bychom takhle museli vždy přidat všech $n * m + 4p$ tlačítek. Pokud bychom po ukončení hry chtěli spustit novou hru, byli bychom nuceni buď celé toto množství tlačítek opět odebrat a následně zase přidat, anebo implementovat netriviální algoritmus[6], který by uměl využít již existující tlačítka pro rekonstrukci nové hry.

Pokud bychom se tedy chtěli vyvarovat psaní rekonstrukčního algoritmu, bylo by potřeba, aby vytváření, přidávání a odebírání tlačítek bylo velmi rychlé. Toto se však při experimentálním generování nepotvrdilo, vytvoření obdélníku složeného z $22 * 10$ tlačítek trvalo cca 3,5 sekundy. Navíc se ukázalo, že i již vytvořená hra má problém s překreslováním tohoto množství tlačítek. Pokud bylo okno minimalizováno a poté obnoveno, tlačítka se opět překreslovala, což asi jednu sekundu zdržovalo hru a navíc to vypadalo velmi ošklivě. Při zmenšení okna tak, aby docházelo k rolování, se pohyb posuvníku projevoval nepříjemným trháním v místech, kde byla tato tlačítka.

V případě této reprezentace by tedy počet *stavů* a velikost hrací plochy musely být omezeny tak, aby se vždy všechny buňky potřebné ke hře vešly do okna hry bez rolování. I tak by ale musel hráč přimhouřit oči a chvilku vyčkat při obnově okna.

[6] Obtížnost tohoto algoritmu by spočívala v tom, jak by byla rekonstrukce provedena v případě, že by se změnily rozměry hrací plochy a tlačítka by musela být přeuspořádána.

Kreslené objekty

Alternativou k použití tlačítek je vykreslovat útvary do tzv. *PictureBoxu*. Jednotlivé *stavy* potom lze reprezentovat jako například čtverečky. V tomto případě opět můžeme čtverečkům nastavit požadovanou velikost a libovolnou barvu. Bohužel už však nelze jednoduše přiřadit čtverečku text, který by se sám vykreslil, a co hůře, nakreslený čtvereček neumí změnit svou velikost, barvu ani reagovat na kliknutí.

Problém s vykreslováním textu můžeme poměrně snadno vyřešit. Jmenný prostor *System.Drawing* obsahuje metodu *Graphics.DrawString*, která umožňuje kreslit textové řetězce. Pokud tedy na místo, kam vykreslíme čtvereček, zároveň vykreslíme text, bude to mít stejný efekt, jako kdyby text byl součástí čtverečku.

Větší problém nastává při implementaci reakce na kliknutí na čtvereček. Čtverečky jsou součástí *PictureBoxu*, který na kliknutí reaguje jako celek, ale nedokáže určit objekt, který se na místě kliknutí nachází. Proto je potřeba udržovat *kolekci* všech čtverečků, které mají reagovat na kliknutí, a pokud ke kliknutí na *PictureBox* dojde, zjistit, zda byl nějaký z nich v ten moment pod kursorem.

Pokud chceme některý parametr čtverečku (velikost, barvu, text) změnit, musíme celý čtvereček překreslit. A jelikož pro překreslení čtverečku je potřeba překreslit celý *PictureBox*, znamená každá vizuální změna hry (např. odkrytí buňky) překreslení celé hrací plochy. Z toho vyplývá, že při této reprezentaci je potřeba, aby veškeré překreslování bylo velmi rychlé, aby nebylo poznat, že vůbec k překreslení během hry došlo.

Experimenty se stejně velkou hrací plochou (tj. $22 * 10$ buněk) ověřily, že vykreslování čtverečků je mnohem rychlejší nežli použití tlačítek. Hrací plocha byla vykreslena prakticky okamžitě, za čas pod 0,5 sekundy. Při minimalizaci a následném obnovení okna se buňky vykreslily také téměř nepostřehnutelně a při použití posuvníku nebylo vykreslování sice dokonalé, ale rozhodně mnohem lepší nežli při použití tlačítek. Při plynulém rolování *PictureBoxu* s 4 000 buněk (tj. kompletní přechodová funkce pro desetistavový automat) probíhalo rolování

téměř dokonale, při prudkém posunutí o větší vzdálenost se obrázky ustálily do 0,5 sekundy po ukončení tahu posuvníku.

Vzhledem k tomu, že se vykreslování buněk jako čtverečků do *PictureBoxu* ukázalo jako mnohem efektivnější a tím komfortnější pro hráče, volím tuto možnost reprezentace i přes to, že je potřeba napsat pomocný kód, který by u reprezentace pomocí tlačítek nebyl potřeba. Tlačítka využijeme pro zobrazení menšího množství buněk, jako například pro ovládací tlačítka, se kterými se nebude pohybovat.

4 Nástin architektury

Celá práce je psána v jazyce C# a využívá knihovnu tříd zvanou *Windows Forms*. Ke spuštění je potřeba mít na počítači nainstalován *.NET framework* verze 4.5 (či vyšší), který je k dispozici například v operačním systému *Windows 8* nebo *Windows 10*. Veškerý vytvořený kód je obsažen v *Solution CellsGame*, které nalezneme v příloze *CellsGame.zip.*

Hlavním projektem je projekt hry, taktéž zvaný *CellsGame*. Ten ke svému kompletnímu chodu potřebuje jinak samostatné projekty *CellsEditor* a *CellsRuleGenerator*. Všechny zmíněné projekty používají třídy z knihovny *CellsLib* a umí se navzájem spouštět. Spustitelné soubory odpovídající jednotlivým projektům spolu s výchozím balíkem souborů s automaty lze nainstalovat do počítače pomocí instalačního souboru *CellsGame.msi*.

Nyní se krátce zmíníme o nejdůležitějších třídách a metodách jednotlivých projektů, podrobnosti se nachází v programátorské a vývojové dokumentaci.

4.1 Knihovna CellsLib

V této podkapitole stručně představíme nejpodstatnější části knihovny *CellsLib*. Projekt knihovny obsahuje třídy využívané ve všech ostatních projektech práce. Její velkou část tvoří třídy pro reprezentaci celulárních automatů a pro práci s nimi, dále pak malé pomocné třídy a abstraktní třídy.

CellularAutomaton

Třída *CellularAutomaton* je abstraktní třídou reprezentující celulární automat. Udržuje veškeré informace o daném automatu, jako například počet stavů a pravidel přechodové funkce, barvy reprezentující jednotlivé stavy a kompletní přechodovou funkci. Jejími konkrétními potomky jsou třídy *CommonAutomaton*, *TotalisticAutomaton* a *PictureAutomaton*, které reprezentují jednotlivé druhy podporovaných automatů.

Nejdůležitější metodou této třídy je statická metoda *CreateAutomaton*. Ta zná přípony všech podporovaných typů automatů a podle přípony jména souboru, které dostává jako parametr, vytvoří správný typ automatu. V téměř celé práci jsou

automaty vytvářeny pomocí této metody, konstruktory jednotlivých automatů se přímo používají jen výjimečně.

Rule

Třída *CellularAutomaton* implementuje interface *IEnumerable<Rule>*. *Rule* je jednoduchou pomocnou třídou, jejíž instance reprezentují jednotlivá pravidla přechodové funkce. Pro *totalistické* celulární automaty existuje potomek této třídy s názvem *TotalisticRule*.

Instance pravidla se skládá z trojice stavů nazvané *Configuration* a výsledku přechodové funkce pojmenovaného *Result*. Trojice stavů je reprezentována strukturou *Triple*, která tyto tři stavy automatu udržuje.

Třída *Rule* obsahuje dvě zvláštnosti. První z nich je upravený způsob porovnávání instancí. Jsou-li dvě pravidla porovnávána, je bráno v potaz, že buňka s neznámou hodnotou je rovna libovolné platné hodnotě.

Druhou zvláštností je virtuální metoda *CreatePossibilities*, která se nachází ve třídě kvůli podpoře *totalistických* přechodových funkcí. Přechodová funkce *totalistického* pravidla poskytuje v rámci ušetření místa pouze jednu neuspořádanou trojici, která reprezentuje šest permutací tohoto pravidla. Úlohou této metody je vrátit všechny konkrétní permutace pravidla (na němž je volána), které odpovídají vzoru přijatému jako parametr.

AutomatonState

Třída *AutomatonState* má speciální význam. Napodobuje výčtový typ způsobem, který popisuje *McConnel* (McConnell, 2006). Byla vytvořena proto, že umožňuje implementaci metod, kterou *výčtový typ* neumožňuje. Díky tomu, že se jedná o třídu, lze vhodně implementovat porovnávání stavů, kontrolu mezí, metodu *ToString* či například implicitní konverze.

Třída obsahuje statické datové položky typu *AutomatonState*, které reprezentují všechny podporované hodnoty stavů. Kromě běžných hodnot stavů obsahuje i položky pro konstanty, které jsou užitečné v průběhu hry, jako například *AutomatonState.Default* nebo *AutomatonState.Undefined*.

Drawer

Abstraktní třída *Drawer* je objektem odpovědným za vykreslování. Jednotlivé části hry, které potřebují vykreslovat větší množství buněk do *PictureBoxu*, mají vlastního potomka této třídy, specializovaného na vykreslování buněk v určitém formátu. Editor pravidel *CellsEditor* obsahuje potomka této třídy zvaného *EditorDrawer*, projekt *CellsGame* má dokonce tři potomky této třídy. *GameDrawer* je specializovaný na vykreslování hrací plochy, *OverviewDrawer* vykresluje *Přehled pravidel* přechodové funkce a jeho potomek *HelpDrawer* se stará o vykreslování okének, ve kterých se zobrazují výsledky nápovědy z *Přehledu pravidel*. Projekt *CellsRuleGenerator* má také jednoho potomka této třídy, zvaného *GeneratorDrawer*.

Creator

Poslední podstatnější třídou knihovny je třída *Creator*, která obsahuje statické metody, které jsou využívány na více místech v kódu. Metody jsou zaměřeny na výpočet rozměrů *PictureBoxů* při určitých stanovených podmínkách a na kompaktnější nastavování vlastností tlačítek.

4.2 Hra CellsGame

Projekt *CellsGame* je hlavní částí celé práce. Tvoří hru jako takovou, její ovládání i pomocnou logiku. Ze spuštěné hry je navíc možné otevřít jak *Editor*, tak *Generátor pravidel*.

FCellsGame

Hlavní třídou projektu je třída *FCellsGame* reprezentující hlavní okno hry. Jako datové položky obsahuje čtyři základní třídy, které zajišťují nejdůležitější úkoly. Třídy *GamePlane* a *FRulesOverview* jsou doslova základními stavebními kameny projektu obsahujícími v sobě spoustu logiky programu. Naopak třídy *UserInterface* a *GameSettings* jsou třídami zaměřenými na design hry a vstup od uživatele.

GamePlane

Je třídou reprezentující hrací plochu a veškerou práci s ní. Má veřejné metody podávající informace o stavu hry či jednotlivých buněk, i metody, které umožňují

buňky zvýrazňovat nebo odkrývat. Hlavními částmi této třídy jsou dvourozměrné pole reprezentující buňky hrací plochy a položka typu *GameState*, která udržuje komplexní informace o stavu hry. Konkrétně tato třída reprezentuje plochý hrací plán, pro implementaci cyklického hracího plánu je k dispozici třída *GamePlaneCyclic*, která je potomkem této třídy.

GameState

Třída *GameState* je nezbytným doplňkem třídy *GamePlane,* monitorujícím aktivní hru a zachycujícím její změny. Obsahuje kompletní implementaci navrženého bodování, počítá počet provedených tahů a zaznamenává, pokud je hra ukončena.

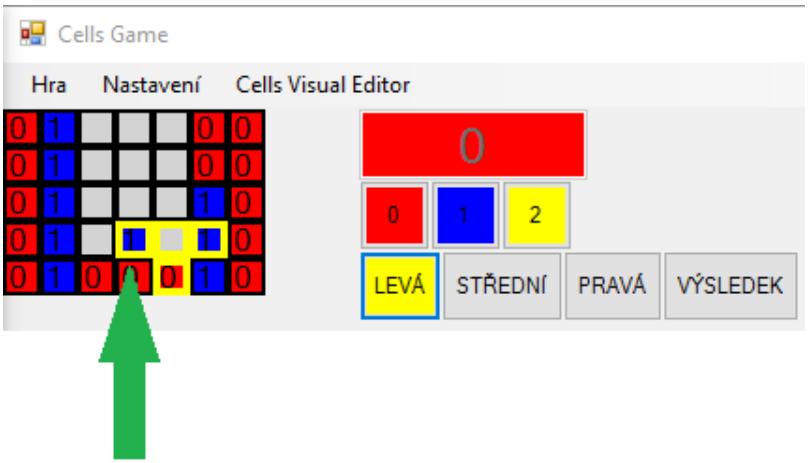
FRulesOverview

Tato třída reprezentuje okno s *Přehledem pravidel*. Stará se o nalezení pravidel, která odpovídají zadanému místu hrací plochy, o zvýraznění nalezených možností v *Přehledu pravidel* i o přenesení zvýrazněných buněk do pomocného okénka. Jako reprezentaci pomocného okénka používá vnitřní třídu zvanou *FHelpWindow*.

Ta po vytvoření instance umí udržovat zobrazená pravidla aktuální, a pokud provedené odkrytí nějaké buňky hrací plochy znemožní použití určitého pravidla zobrazeného v okénku, okénko toto pravidlo automaticky zahodí. Dále třída umožňuje uživateli měnit popisek v titulku okna a označovat pravidla jako nepoužitelná pro řešení.

UserInterface

Třída *UserInterface* zahrnuje veškerou práci zabývající se vykreslováním hry. Sama o sobě se stará o celkovou aktualizaci uživatelského rozhraní (většinou na základě dat z třídy *GameState*) a o interakci s uživatelem. Dílčí úlohy plní její dvě hlavní datové položky typů *ControlInterface* a *Drawer*. V rámci třídy *ControlInterface* ošetřuje práci s ovládacími tlačítky a v rámci třídy *Drawer* (používá potomka *GameDrawer*) vykresluje buňky hrací plochy.



Obrázek 4.1: Pohled na buňku jako na levou

GameSettings

GameSettings je velmi důležitá pomocná třída. Je potomkem třídy *CellsSettings*[7]. Kromě dat poděděných od této třídy udržuje *GameSettings* celistvě dodatečné informace o hře, kterými jsou velikost hrací plochy, použitý automat, typ hrací plochy a stav ovládacích tlačítek. Korektní změny těchto parametrů hry zajišťuje také tato třída.

HelpView

Kromě výše zmíněných základních tříd pro chod hry zmíníme ještě jednu zajímavou abstraktní třídu *HelpView* a její konkrétní potomky *LeftHelpView*, *MiddleHelpView*, *RightHelpView* a *ResultHelpView*.

Pokud má být v *Přehledu pravidel* zobrazena nápověda pro nějakou konkrétní buňku, jsou tato pravidla brána z určitého pohledu, který definuje *HelpView*. Pokud tedy uživatel zažádá o nápovědu pro nějakou buňku, může zvolit, jakou pozici v zobrazeném pravidle bude buňka mít (vybráním konkrétního potomka *HelpView*). Obrázek 4.1 zobrazuje situaci, kdy je kliknuto na modrou buňku, na kterou ukazuje šipka. Ta je chápána jako levá buňka pravidla (zvýrazněno žlutě), neboť je aktivní *LeftHelpView* (tlačítko s popiskem „LEVÁ“).

Třída má dvě veřejné metody. První z nich, *GetViewOfCell*, vrací pravidlo, jehož stavy odpovídají buňkám kolem buňky, na niž bylo kliknuto (vzhledem k typu

[7] Nachází se v knihovně *CellsLib*, ale zmiňována je pouze zde. Jedná se o jednoduchou třídu udržující informace pro vykreslování buněk, konkrétně velikost buňky, velikost mezery mezi buňkami, a zda mají být na jednotlivých buňkách vykreslovány číslice.

pohledu). Odkryté buňky pravidla v něm mají svou hodnotu, hodnota zakrytých buněk je v něm označena za nedefinovanou. Vrácené pravidlo tedy tvoří jakýsi *regulární výraz*, pomocí nějž pak *FRulesOverview* vyhledává odpovídající pravidla. Druhá metoda, *GetRuleIndices*, vrací souřadnice buněk, které vrátila *GetViewOfCell*, v rámci hrací plochy.

4.3 Editor pravidel CellsEditor

Dalším ze samostatných projektů je *CellsEditor*. Jedná se o editor, který umí načítat soubory s pravidly celulárních automatů a vizuálně je zobrazit. Uživateli umožňuje základní změny existujících pravidel jako změnu výsledku přechodové funkce, změnu barvy reprezentující určitý stav nebo změnu popisu pravidla. Dále umožňuje vytvoření nového automatu.

Editor může být spuštěn z projektu *hry* či z projektu *Generátoru pravidel*, ale lze jej používat zcela samostatně. Navíc nastavíme-li ve *Windows*, že soubory s pravidly chceme otvírat v tomto *editoru*, budou se soubory s automaty zobrazovat s ikonkou odpovídající ikonce *editoru* a *editor* je dokáže otevřít pouhým poklepáním na soubor.

FVisualEditor

Hlavní třídou *Editoru* je třída *FVisualEditor* reprezentující okno editoru. Do něj jsou vykreslována pravidla automatů a zároveň slouží jako uživatelské rozhraní pro úpravu automatu. Nejdůležitějšími metodami této třídy jsou *openFile* a *saveFile*, které zajišťují korektní otevření souboru v *Editoru*, resp. jeho zavření. Dále tato třída obsahuje metody pro komunikaci *editoru* s uživatelem.

IParameter

Jedná se o jednoduché rozhraní, které má jednu veřejnou metodu *GetFileName*. Ta vrací jméno souboru, který má být otevřen. Je implementováno dvěma třídami – *FRuleParameters* a *RuleDialog* – z nichž každá získává jméno souboru, které vrací, jiným způsobem.

FRuleParameters

Třída *FRuleParameters* vytváří okno, ve kterém uživatel volí parametry nového automatu. Na jejich základě je poté šablona tohoto automatu vytvořena

a otevřena v *editoru*, aby ji uživatel mohl upravovat. V metodě *GetFileName* tato třída vrací jméno souboru, který byl vytvořen.

RuleDialog

RuleDialog je třídou, která obaluje třídu *OpenFileDialog* a obohacuje ji o implementaci rozhraní *IParameter*. V metodě *GetFileName* vrací jméno souboru, který byl vybrán v *OpenFileDialogu*.

EditorSettings

Třída *EditorSettings* je potomkem třídy *CellsSettings* a udržuje všechny parametry potřebné k vykreslování buněk do *editoru*. Poslední uživatelem zvolené nastavení vždy uchovává v souboru a při dalším spuštění *editoru* toto nastavení načte. Toto nastavení *editoru* může uživatel změnit prostřednictvím třídy *FSettings*, která vizuálně zobrazuje hodnoty ze třídy *EditorSettings* a případné změny opět ukládá.

4.4 Generátor pravidel CellsRuleGenerator

Posledním samostatným projektem je *CellsRuleGenerator*. Jak název napovídá, jedná se o *Generátor pravidel*. Slouží k vyhledání pravidel, která vygenerují obrazec zadaný na vstupu. *Generátor* má vizuální uživatelské rozhraní, podporuje automaty až se čtyřmi stavy a nalezené výsledky dokáže otevřít v *Editoru pravidel*.

FGenerator

Základní třídou projektu je třída *FGenerator* tvořící hlavní okno *Generátoru*. Jejími základními stavebními kameny jsou generátory stavů *RuleGenerator* a *ResultsGenerator* a třídy *PictureMask* a *RuleCreator*, které se velkým dílem účastní výpočtu. Dalšími důležitými datovými položkami jsou třídy *GeneratorSettings* a *UserInterface*, které se starají hlavně o vzhled okna. Tato třída obsahuje nejdůležitější rekurzivní metodu celého projektu *searchSolution*, která hledá pravidla generující zadaný obrazec.

ResultsGenerator

Instance třídy *ResultsGenerator* generuje všechny možné kombinace ohodnocení řady buněk, jejíž délku dostane jako parametr konstruktoru. Používá

o jeden stav méně nežli je celkový počet stavů použitých k prohledávání. To je proto, že uživatelem obarvené buňky dostávají jednu pevnou hodnotu stavu a *Generátor* generuje jen hodnoty stavů neobarvených buněk. Potomek této třídy *SymmetricResultsGenerator* navíc generuje pouze takové hodnoty stavů, že výsledná řada je vždy symetrická. Obě tyto třídy implementují interface *IEnumerable<AutomatonState[]>* a jednotlivé řady stavů vrací jako výsledky při *enumeraci* například pomocí cyklu *foreach*.

PictureMask

Třída *PictureMask* udržuje informace o tom, které buňky mají být obarvené a které nikoli. Navíc dostane-li řadu stavů od třídy *ResultsGenerator*, dokáže do ní doplnit chybějící hodnoty obarvených buněk na správná místa.

RuleCreator

Třída *RuleCreator* má všechny potřebné informace k tomu, aby mohla rozpoznávat řady stavů, které jsou použitelné v rámci aktuálního mezivýsledku, a aby mohla ukládat aktuálně použitelné části výsledku. Dokáže také kompletní výsledek optimalizovat a uložit do souboru.

UserInterface

Třída *UserInterface* se stará o překreslování uživatelského rozhraní, k čemuž využívá potomka třídy *Drawer* jménem *GeneratorDrawer*. Jako pomocné informace jí slouží data z třídy *UserCells*, udržující informace o buňkách, jak je nastavil uživatel, a z třídy *GeneratorSettings,* obsahující informace určené hlavně pro vykreslování.

5 Řešení

V této kapitole popíšeme konkrétní řešení nejdůležitějších problémů, se kterými jsem se setkala při implementaci zadané práce.

5.1 Reprezentace automatů

Jedním z požadavků práce je, aby celulární automat, použitý ve hře, nebyl pouze jeden. Naopak by mělo být možné spouštět různé existující automaty, případně vytvářet nové a do hry je přidávat. Proto nyní popíšeme, jakým způsobem této vlastnosti hry docílíme.

Formát automatu

V předchozí kapitole na str. 31 jsme se seznámili s třídou *CellularAutomaton*, která reprezentuje libovolný celulární automat. Aby třída platný automat vytvořila, potřebuje k tomu soubor, který obsahuje všechny údaje o automatu ve správném formátu.

Soubory s automaty musí mít jednu ze známých přípon určených pro CA. Použití správné přípony je nutností pro správný chod automatu, neboť třída *CellularAutomaton* vytváří vhodnou instanci automatu podle přípony souboru a každý automat má své specifické odlišné chování. Tabulka 5.1 zobrazuje přehled všech podporovaných přípon.

Soubory mají pevně daný formát, který se u každého konkrétního automatu drobně liší. Tabulka 5.2 zobrazuje základní části formátu souborů, detaily jsou popsány ve vývojové dokumentaci.

Typ automatu	**Přípona**
Běžný celulární automat	**ccell**
Totalistický celulární automat	**tcell**
Obrázkový celulární automat	**pcell**

Tabulka 5.1: Přípony jednotlivých typů automatů

Informace	Popis
Počet stavů automatu	Na samostatném řádku.
Barvy stavů	Pro každý stav na jednom řádku.
Specifická část	U jednotlivých typů automatů se liší.
Jednotlivá pravidla	Každé pravidlo na jednom řádku.
Popis pravidla pro uživatele	Text v přirozeném jazyce.

Tabulka 5.2: Obecný formát souboru reprezentujícího automat

Možnosti rozšíření o nové automaty

Jelikož jsou automaty načítány ze souborů v konkrétním formátu, hra dokáže načíst libovolný soubor, který tento formát dodrží. Aby však nebylo jedinou možností, jak vytvářet nové automaty, ruční psaní těchto souborů, je možné nové pravidlo vytvořit v editoru pravidel (*CellsEditor*). *Editor* zobrazuje pravidla ve vizuální formě, ukládá je však ve formátu vyžadovaném potomky třídy *CellularAutomaton*. Proto jsou tedy tyto automaty ihned použitelné ve hře.

5.2 Uživatelské rozhraní

V této podkapitole popíšeme řešení problémů spojených s návrhem co nejkomfortnějšího uživatelského rozhraní.

Parametrizace hry

Desktopové aplikace pro *Windows* mívají obvykle jednodušší vzhled a „standardní“ ovládací prvky. Jelikož je však vliv vzhledu aplikace na uživatele nezanedbatelným aspektem, jak popisuje Tidwell (Tidwell, 2010), nabízí hra určité uživatelské přizpůsobení jednotlivých oken.

Hra, její pomocná okna i editor vykreslují velké množství buněk. Dle velikosti hrací plochy se jich může najednou zobrazovat například 50, ale i třeba 400. Stanovíme-li pevně danou velikost buňky, může nastat situace, kdy malé hřiště bude obsahovat „monstrózní buňky“ nebo naopak veliké hřiště bude složeno z „mikroskopických buněk“.

Pokud vykreslujeme buňku, je potřeba znát její velikost, barvu (reprezentující stav) a barvu ohraničení. Barvu stavu během hry měnit nemůžeme. Velikost čtverečku a barvu ohraničení, která je totožná s barvou čísel popisujících

pořadí stavů, však lze snadno parametrizovat. Tyto dva parametry poté ovlivní vykreslování všech buněk. Jako další vhodný parametr se jeví velikost mezery mezi stavy. Někomu může vyhovovat, pokud hrací plocha tvoří celistvou mřížku. Pokud by však hráč upřednostňoval, aby měly buňky mezi sebou drobné mezery, je mu to také umožněno. Posledním vhodným parametrem pro vykreslování buněk je barva zvýraznění buněk, ke kterým se vztahuje aktivní okénko s nápovědou. Jelikož různé automaty obsahují stavy s odlišnými barvami, napevno zvolená barva zvýraznění není úplně vhodná. Může totiž nastat situace, že barva zvýraznění bude totožná s barvou některého stavu. Proto si může hráč libovolně nastavit i tuto barvu.

Buňky *Přehledu pravidel* mohou být parametrizovány stejným způsobem. Z toho důvodu i pro toto okno existuje nastavení velikosti buněk, barvy ohraničení, a také barvy pro zvýraznění nápovědy. Dále *Přehled pravidel* umožňuje nastavení jak mezer mezi jednotlivými stavy, tak mezi celými pravidly. Nastavení stejných parametrů nabízí i *Editor pravidel* (kromě zvýraznění nápovědy, které *Editor* nepotřebuje). Ve všech výše zmíněných oknech lze navíc nastavit, zda se mají na buňkách zobrazovat číselné hodnoty jednotlivých stavů. *Generátor pravidel* umožňuje nastavení velikosti buněk, barvy ohraničení a barvy, kterou jsou označovány obarvené buňky.

Ovládací tlačítka

Jelikož není úplně jednoduché zvolit ideální místo pro umístění *ovládacích tlačítek*, může si je uživatel umístit tam, kam chce. *Ovládací tlačítka* mohou být buď umístěna přímo v okně hry, napravo od hrací plochy, anebo mohou být v samostatném okénku, které může být umístěno kdekoli na obrazovce. Mezi těmito dvěma možnostmi lze snadno přepínat v menu hry.

Klávesové zkratky

Protože je hra poměrně komplexní a nabízí spoustu možností, je potřeba do ní zahrnout obsáhlé menu. Uživatelé hry mohou být odlišní. Ti, kteří se hrou teprve začínají, potřebují, aby menu bylo přehledné, zatímco pokročilé uživatele může klikání v menu zdržovat (Sommerville, 2006). Proto jsou ke všem volbám, které se v hlavním menu nachází, přiřazeny klávesové zkratky.

Práce se spoustou oken

Vzhledem k tomu, jak byla hra navržena[8], nastal problém s množstvím zobrazených oken v jeden okamžik. Spuštěná hra vždy musí zobrazit okno s hrou. Ovládací tlačítka hry mohou být zobrazena v samostatném okénku, dále lze očekávat, že si hráč nechá zobrazené okno s *Přehledem pravidel*. Pokud si nechá zobrazit nápovědu pro nějakou buňku, může si ji přetáhnout do dalšího samostatného okénka. Počet těchto okének není omezen.

První problém většího počtu oken nastává ve chvíli, kdy otevřeme nějakou jinou aplikaci a poté chceme opět zobrazit okno s hrou. Je nepříjemné, pokud se nám zobrazí hlavní okno hry, avšak všechna ostatní okna zůstanou skryta pod aplikací. Tuto komplikaci řeší možnost přiřadit některým oknům jejich vlastníky (*Owner*). Konkrétně všem zmíněným oknům (kromě okna hry) nastavíme jako vlastníka okno hry. Díky tomu se tato pomocná okna minimalizují, obnoví, zobrazí na popředí či zavřou vždy, když totožná událost nastane u okna hry. Navíc se taková okna nikdy nedostanou „pod“ svého vlastníka, nemůže se tedy stát, že například okénko s nápovědou zmizí pod oknem hry. Nenastane tedy situace, že bychom při aktivní hře nějaké okno postrádali.

Druhým problémem, který se sice projevuje pouze nekomfortním ovládáním hry, ale stojí za zmínku, je množství oken zobrazujících se v hlavním panelu. Jelikož ve hře může nastat situace, že budeme mít najednou otevřeno například deset oken, zobrazí se všechna v hlavním panelu počítače. Navíc se z velké části bude jednat o okénka nápovědy, která, pokud si je hráč sám nepopíše, ani nemají titulek. Pro tato neoznačená okna jejich zobrazení v hlavním panelu postrádá smysl. V rámci zpřehlednění hry je tedy vhodné (a proto implementováno), že okénka s nápovědou se v hlavním panelu hry nezobrazují. Vzhledem k tomu, že zbývající okna – *Přehled pravidel* a okno s ovládacími tlačítky – jsou používána výhradně v kombinaci s hlavním oknem hry, není nutné ani tato okna zobrazovat v hlavním panelu.

V hlavním panelu se tedy zobrazí pouze hlavní okno hry. To je vlastníkem všech ostatních oken a při jeho aktivaci dojde k zobrazení/přenesení do popředí

[8] Je složena ze spousty oken: hlavní okno s hrou, přehled pravidel, okénka s nápovědou, případně navíc okno s ovládacími tlačítky, okno editoru a okno generátoru.

všech těchto oken. Pokud se jednotlivá okna překrývají, jejich pořadí není ovlivňováno. Pokud však otevřeme *Editor* nebo *Generátor pravidel*, jsou to samostatné aplikace, které na okně hry přímo nezávisí. Proto jsou spouštěny jako nové procesy, samostatně se zobrazují v hlavním panelu a jejich minimalizace, obnovení či vypnutí jsou nezávislé.

Aby se jednotlivá okénka s nápovědou nezobrazovala na náhodných místech obrazovky a zbytečně se nepřekrývala, je nastaveno, že se zobrazují nad hlavním oknem hry (za předpokladu, že hlavní okno hry zůstává na tom místě, na němž bylo při spuštění).

Velikost buněk v okénkách nápovědy

Jednotlivá okénka s nápovědou mohou obsahovat velmi odlišné počty pravidel. Jelikož mají okénka pevně danou velikost, některá mohou být poloprázdná, jiná zas naopak přeplněná, a pak je potřeba s nimi rolovat. Proto umí každé okénko měnit svou velikost nezávisle na ostatních. Pokud se myš nachází nad určitým okénkem a při stisknuté klávese CTRL dojde k pohybu kolečka myši, velikost buněk tohoto okénka se začne zvětšovat nebo zmenšovat (podle směru pohybu kolečka).

5.3 Totalistické automaty

Z podporovaných typů automatů mají *totalistické* celulární automaty poněkud odlišnou přechodovou funkci. Zatímco u ostatních typů automatů reprezentuje každé pravidlo přechodové funkce jednu konkrétní uspořádanou trojici stavů, u totalistických celulárních automatů zastupuje jedno pravidlo všechny permutace téže trojice.

Rozpoznávání pravidel přechodové funkce

Prvním problémem, který u těchto automatů nastává, je, jakým způsobem efektivně vyhledávat konkrétní pravidla v přechodové funkci. Pro získání co nejrychlejšího přístupu k přechodové funkci jsou jednotlivá pravidla ukládána do slovníku (*Dictionary*). Konkrétní uspořádané trojice jsou klíči, výsledky přechodové funkce uloženými hodnotami. Pokud však při generování buněk *totalistickým* celulárním automatem vznikne jiná permutace trojice buněk, nežli ta,

která je uvedena v přechodové funkci, nemůžeme ji použít přímo jako klíč pro přístup ke slovníku s přechodovou funkcí.

Tento problém vyřešíme poměrně snadno, pokud nalezneme nějakou vlastnost, kterou mají všechny permutace téže trojice totožnou a kterou zároveň mají trojice, které nejsou permutacemi, unikátní. Jako tuto vlastnost použijeme setřídění hodnot jednotlivých buněk. Vezmeme-li hodnoty dvou permutací téže trojice buněk a setřídíme je vzestupně, dostaneme stejný výsledek. Zároveň trojice, která není permutací námi použitých trojic, vydá po setřídění odlišnou posloupnost hodnot. Vytvoříme tedy jednoduchou metodu, která dokáže „třídit" trojice podle hodnot stavů vzestupně a její výsledky použijeme jako klíče v přechodové funkci. Metodu implementujeme tak, že hodnoty všech tří stavů okopírujeme ze stávající trojice, poté setřídíme a následně vytvoříme novou trojici, které postupně přidělíme novou hodnotu levé, prostřední a pravé buňky.

Nápověda

Nyní sice dokážeme spolehlivě používat *totalistickou* přechodovou funkci, nastává však ještě jedna komplikace. Hra byla navržena tak, že si hráč kdykoli může zažádat o zobrazení nápovědy k jím zvolené buňce. Tento proces spočívá v tom, že jsou všechna pravidla otestována, zda odpovídají zadanému *vzoru*[9], a jsou vrácena ta, která odpovídají.

V rámci vytváření této funkcionality pro automat s *běžnou* přechodovou funkcí mají celulární automaty předefinovanou metodu *Equals*. Dvě pravidla se rovnají, pokud mají stejné hodnoty jednotlivých buněk či pokud mají některé hodnoty buněk nedefinovanou hodnotu (zakryté buňky). V případě *totalistických* celulárních automatů však přichází komplikace.

Hodnoty buněk, které jsou zakryté (a tím pádem mohou nabývat „libovolného" stavu), jsou reprezentovány stavem *AutomatonState.Undefined,* který má číselnou hodnotu -1. Tento stav ale rozbije porovnávání *totalistických* trojic, které

[9] Vzorem nazývejme jakékoli pravidlo, které má místo některých hodnot uvedenu hodnotu *AutomatonState.Undefined*. Takové pravidlo můžeme chápat jako regulární výraz, proti němuž jsou testována všechna plnohodnotná pravidla.

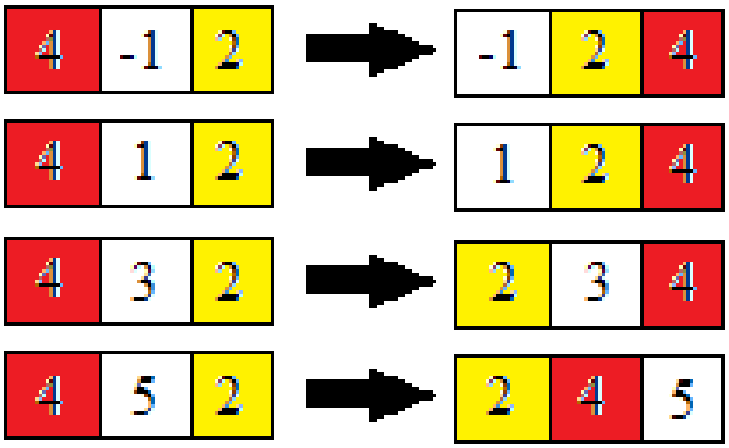
bychom jinak mohli porovnávat stejně, jako trojice *běžné*, pouze bychom je předtím setřídili způsobem popsaným výše.

Konkrétní problém zobrazuje Obrázek 5.1: Mějme trojici, jejíž hodnotu střední buňky neznáme, levá buňka je ve stavu č. 4 a pravá buňka je ve stavu č. 2. Pokud tuto trojici setřídíme, dostaneme hodnoty -1, 2, 4. My však neznáme opravdovou hodnotu střední buňky, kterou může být například 1, ale také 3 nebo 5.

Pokud bychom tedy třídili tuto trojici a znali bychom přitom hodnotu střední buňky, může se stát, že výsledné uspořádání buněk bude odlišné. Zvolení jiné číselné konstanty pro hodnotu *AutomatonState.Undefined* bohužel tento problém nevyřeší[10], a tedy tento způsob není použitelný pro porovnávání částečně zakrytých trojic.

Porovnávání

Je tedy potřeba navrhnout jiný způsob porovnávání. Jako spolehlivý se jeví postup, ve kterém vezmeme všechny platné stavy *vzoru* a kontrolujeme, zda se každý z nich nachází na alespoň jedné pozici v pravidle, se kterým *vzor* porovnáváme. Například tedy *vzor* „-1 3 -1“ obsahuje jeden platný stav 3, který se vyskytuje třeba v pravidle „3 0 1“, ale v pravidle „2 1 0“ ho nenajdeme.



Obrázek 5.1: Problém s tříděním trojice s neznámým stavem

[10] Jakákoli číselná konstanta může být při setřídění zařazena pouze na jedno místo, zatímco různé skutečné hodnoty buňky mohou buňku pokaždé zařadit jinam.

Tento algoritmus však bohužel skrývá jeden problém. Pokud porovnáváme *vzor* „-1 2 2“ s pravidlem „2 1 3“, dojdeme ke shodě. Problematický okamžik nastane, jakmile otestujeme poslední hodnotu *vzoru*. Algoritmus odpoví, že stav 2 se v pravidle nachází, bohužel už však nepoznamená, že pouze jednou. Tuto vadu naštěstí rychle opravíme. Pokud algoritmus prohlásí, že pravidlo odpovídá *vzoru*, otestujeme ještě, zda četnost každého platného stavu ve *vzoru* nepřevyšuje počet výskytů tohoto stavu v porovnávaném pravidle.

Výstup odpovídající realitě

Nyní již dokážeme posoudit, zda daný vzor odpovídá nějakému pravidlu, a není tedy problém implementovat *totalistickou nápovědu*. Bohužel však zobrazovaný výstup nebude zcela odpovídat realitě, bude ovlivněn faktem, že je automat *totalistický*. Pokud pravidlo odpovídá *vzoru*, je potřeba jej zvýraznit v *Přehledu pravidel*. Jestliže ale vyobrazený tvar pravidla neodpovídá tvaru vyskytujícímu se ve hře, může být takovýto výstup spíše matoucí.

Pro upřesnění uveďme příklad. *Vzor* „-1 2 -1“ odpovídá *totalistickému* pravidlu „2 3 4“, konkrétně jeho permutaci „4 2 3“. Je tedy vhodnější do *nápovědy* zahrnout tvar „4 2 3“ nežli setříděný tvar „2 3 4“. Naneštěstí ale pouhá permutace hodnot nezachrání všechny případy, které mohou nastat. Konkrétně v našem případě vzor „-1 2 -1“ odpovídá nejen permutaci „4 2 3“, ale také permutaci „3 2 4“. Ideálně by tedy nápověda měla zobrazit všechny přijatelné permutace pravidla.

Kvůli tomuto problému obsahuje třída reprezentující pravidla (*Rule*, *TotalisticRule*) již dříve zmiňovanou metodu *CreatePossibilities*, která v případě *běžné* přechodové funkce nemá využití. Pro *totalistické* automaty tato metoda vytvoří všechny permutace pravidla, na němž je volána, a vrátí ty, které odpovídají vzoru. Tímto způsobem je potom program schopen zobrazovat přesné tvary pravidel i pro *totalistické* celulární automaty.

5.4 Úprava neúplných obrázkových pravidel

Oproti *běžným* a *totalistickým* celulárním automatům mají *obrázkové* celulární automaty jednu komplikaci. Pro některé konfigurace buněk nemají výsledek přechodové funkce. Pakliže je obrázkový celulární automat vygenerován generátorem pravidel *CellsRuleGenerator*, nevadí to, protože výchozí generace

buněk je nastavena tak, že konfigurace, pro které neexistuje výsledek, se ve vývoji nikdy nevyskytnou.

Pokud však uživatel otevře *obrázkový* celulární automat v editoru pravidel *CellsEditor*, může pozměnit výsledky přechodové funkce nebo i hodnoty stavů výchozí generace buněk. Následně může dojít k tomu, že při generování obrázku může některé pravidlo chybět. Proti této situaci je *obrázkový* celulární automat chráněn tím, že pokud má vrátit výsledek přechodové funkce pro neznámou konfiguraci, vrací hodnotu výchozího *stavu (AutomatonState.Default* = 0). Zároveň si však pro tuto konkrétní konfiguraci uloží pravidlo s tímto výsledkem a zobrazí ho mezi pravidly v *Přehledu pravidel*.

6 Závěr

6.1 Dosažené cíle

V rámci této práce byly zváženy způsoby použití celulárního automatu k vytvoření logické hry a následně byla vybraná verze hry implementována. Byl použit jednorozměrný celulární automat o 2-10 stavech a za cíl hry byla zvolena situace, kdy se hráči podaří odkrýt všechny buňky hrací plochy. Do hry byl implementován pomocný nástroj, který dokáže pro jednotlivé buňky zobrazovat použitelná pravidla přechodové funkce.

Práce byla napsána v jazyce C#. Při vytváření byl kladen důraz na uživatelskou přívětivost hry a přehlednost. Navíc byly vytvořeny dva nástroje pro rozšiřování hry. *Generátor pravidel* umožňuje nalezení pravidel, pomocí kterých lze z dané výchozí generace buněk vygenerovat určitý obraz, *Editor pravidel* zas umožňuje úpravy libovolného již existujícího automatu či vytvoření nového automatu.

6.2 Možná rozšíření

Na úplný závěr zmíníme vhodná místa, kde by bylo možné hru rozšířit.

Jelikož existuje nepřeberné množství celulárních automatů, bylo by možné zkusit použít například mnou zavrhnuté automaty s nesymetrickým či větším okolím. Dále by mohlo být zajímavé (i když možná ne úplně snadné pro hráče) použít více pravidel a jejich přechodové funkce skládat.

Alternativou ke *Generátoru pravidel* by mohl být nástroj, který by dokázal vytvořit přechodovou funkci na základě pravidel popsaných matematickou či logickou formulí. Pro každé pravidlo přechodové funkce by vypočítal správný výsledek a ušetřil by tím uživateli práci zvláště při vytváření automatu o větším množství stavů.

Dále by bylo možné implementovat inteligentní nápovědu, která by uměla vyhodnocovat výsledky na základě dat z větší části hrací plochy či případně rozpoznávat stavy hry, které nelze jednoznačně vyřešit. Tento směr by poté mohl dále vést k vytvoření umělé inteligence schopné řešit hru jako reálný hráč.

7 Seznam použité literatury

Cook, M. 2004. *Universality in Elementary Cellular Automata.* Complex Systems, 2004.

Jen, E. 1990. *Aperiodicity in one-dimensional cellular automata.* Physica D, 1990.

McConnell, S. 2006. *Dokonalý kód: umění programování a techniky tvorby software.* Brno : Computer Press, 2006. ISBN 80-251-0849-X.

Moore, E. F. 1962. *Machine models of self-reproduction.* Proceedings Symposia in Applied Mathematics, 1962.

Myhill, J. 1963. *The converse of Moore´s Garden-of-Eden theorem.* Proceedings of the American Mathematical Society, 1963.

Packard, N. H. a Wolfram, S. 1985. *Two-dimensional cellular automata.* Journal of Statistical Physics, 1985.

Schiff, J. L. 2008. *Cellular automata : a discrete view of the world.* New Jersey : Wiley-Interscience, 2008. ISBN 997-0-470-16879-0.

Sommerville, I. 2006. *Software Engineering (8th Edition).* Addison-Wesley, 2006.

Tidwell, J. 2010. *Designing interfaces, Patterns for effective Interaction Design.* O´Reily Media, Inc., 2010. ISBN 978-1-4493-7970-4.

Weisstein, E. 2017. Wolfram MathWorld. *the web´s most extensive mathematics resource.* [Online] 11. Duben 2017. [Citace: 20. Duben 2017.] http://mathworld.wolfram.com/.

Wolfram Research, Inc. 2003. The Wolfram Atlas of Simple Programs. [Online] 27. Červen 2003. [Citace: 20. Duben 2017.] http://atlas.wolfram.com/.

Wolfram, S. 2002. *A New Kind of Science.* Champaign : Wolfram media, 2002. ISBN 1579550088.

Wolfram, S. 1984. *Universality and complexity in cellular automata.* Physica D, 1984.

8 Seznam obrázků

9 Seznam tabulek

10 Přílohy

1. **CellsGame.zip**

   Obsahuje zdrojové kódy všech částí práce – *CellsGame*, *CellsEditor*, *CellsRuleGenerator* a *CellsLib* jako *Solution* pro *Visual Studio 2015* (či vyšší). Dále obsahuje programátorskou dokumentaci v HTML a uživatelskou a vývojovou dokumentaci jako PDF/A-1a.

2. **VeraSkopkova.pdf**

   Elektronická verze této práce ve formátu PDF/A-1a.

3. **CellsGame.msi**

   Instalační soubor, pomocí nějž jsou všechny části hry nainstalovány do počítače tak, aby sloužily k reálnému použití. I pro spuštění hry z *Visual Studia* je nejprve nutné hru nainstalovat, aby byly do počítače přeneseny soubory s automaty.