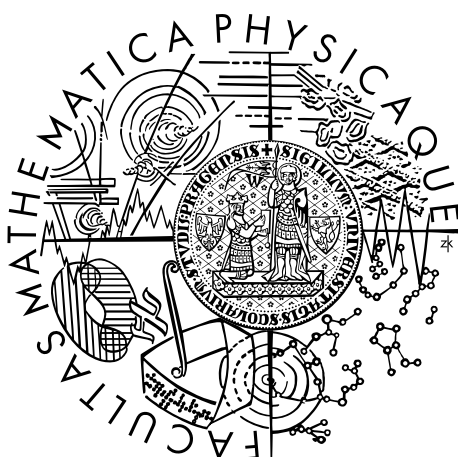


Univerzita Karlova v Praze  
Matematicko-fyzikální fakulta

## BAKALÁŘSKÁ PRÁCE



Martin Dvořák

## Šifrování pomocí celulárních automatů

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: PhD. Otakar Trunda

Studijní program: Informatika

Studijní obor: Obecná informatika

Praha 2016

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) 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 v Praze 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: Šifrování pomocí celulárních automatů

Autor: Martin Dvořák

Katedra teoretické informatiky a matematické logiky: Katedra teoretické informatiky a matematické logiky

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

Abstrakt: Abstrakt.

Klíčová slova: celulární automaty šifrování protahování klíčů

Title: Cryptography using cellular automata

Author: Martin Dvořák

Department of Theoretical Computer Science and Mathematical Logic: Department of Theoretical Computer Science and Mathematical Logic

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

Abstract: Abstract.

Keywords: cellular automata cryptography key stretching

Děkuji svému vedoucímu za ochotnou pomoc.

# Obsah

<b>Úvod</b>	<b>2</b>
<b>1 Celulární automaty</b>	<b>3</b>
1.1 Co jsou to celulární automaty . . . . .	3
1.2 Elementární celulární automaty . . . . .	3
1.3 Jiné 1D celulární automaty . . . . .	3
1.4 Dvourozměrné celulární automaty . . . . .	6
1.5 Implementace celulárních automatů . . . . .	6
<b>2 Šifrování</b>	<b>8</b>
2.1 Několik ukázek . . . . .	8
2.2 Testy . . . . .	8
2.2.1 Testy na úrovni jednotlivých výstupů . . . . .	9
2.2.2 Testy na úrovni celého zobrazení . . . . .	9
2.2.3 Použití testů . . . . .	10
2.2.4 Grafické znázornění . . . . .	10
<b>3 Nástin architektury</b>	<b>12</b>
3.1 MartinDvorak . . . . .	12
3.1.1 Cellular . . . . .	12
3.1.2 Crypto . . . . .	12
3.1.3 Testing . . . . .	12
3.2 Program . . . . .	12
<b>4 Způsoby protahování klíčů</b>	<b>13</b>
4.1 KeyExtenderSimpleLinear . . . . .	13
4.2 KeyExtenderSimpleQuadratic . . . . .	13
4.3 KeyExtenderInterlaced . . . . .	22
4.4 KeyExtenderUncertain . . . . .	22
4.5 KeyExtenderGenetic . . . . .	27
<b>Závěr</b>	<b>32</b>
<b>Seznam obrázků</b>	<b>34</b>
<b>Seznam tabulek</b>	<b>35</b>
<b>Seznam použitých zkratk</b>	<b>36</b>
<b>Přílohy</b>	<b>37</b>

# Úvod

Toto je implementačně–experimentální práce. Cílem mojí práce je vytvořit šifrovací algoritmus využívající celulární automaty. Úkolem bylo naprogramovat různé způsoby natahování šifrovacích klíčů a naměřit změřit jejich vlastnosti, za účelem zvolení správného algoritmu pro šifrovací aplikaci.

# 1. Celulární automaty

## 1.1 Co jsou to celulární automaty

Celulární automat je diskretní model, který se skládá z pravidelné mřížky buněk. Buňky se nacházejí v určitých stavech, přičemž množina stavů a pravidla pro přechod mezi stavy jsou společná pro celý automat. Celulární automat se vyvíjí diskretně v čase (celý najednou).

Specialitou celulárních automatů je to, že i velmi jednoduchá sada pravidel může vést k velmi komplexnímu chování. Celulární automaty našly využití jako modely v biologii, chemii, fyzice, ale také třeba jako nástroj při procedurálním generování terénu pro počítačové hry.

## 1.2 Elementární celulární automaty

Wofram popisuje 256 elementárních celulárních automatů. Jedná se o binární 1D automaty, kde nový stav každé buňky závisí pouze na jejím stavu a stavu přímých sousedů. Formálně by se dal přechod zapsat jako:

$$x_i(t+1) = f(x_{i-1}(t), x_i(t), x_{i+1}(t))$$

, kde

$$f : \{0,1\}^3 \rightarrow \{0,1\}$$

, tudíž existuje  $2^3 = 2^8 = 256$  takových funkcí  $f$ .

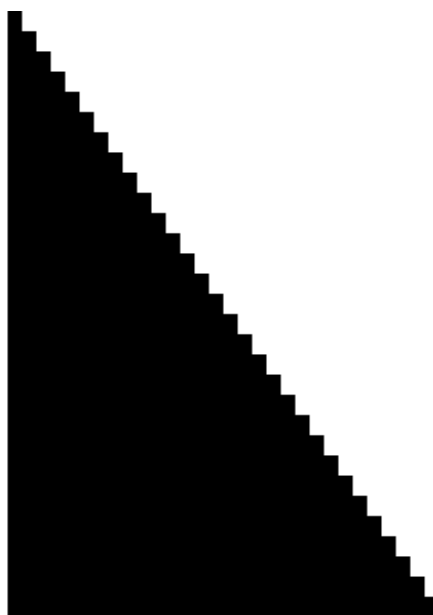
Elementární automaty číslujeme tak, že pravidlo zapsané jako osmice bitů (kde první bit je obraz (1, 1, 1) a poslední bit je obraz (0, 0, 0), řadí se lexikograficky) převedeme do dvojkové soustavy.

Ukážeme si teď několik elementárních automatů, na které se budeme v práci odkazovat. Zakreslíme vždy jejich vývoj v čase, kde čas roste směrem dolů, přičemž počáteční konfigurace se skládá se samých nul (bílých / mrtvých buněk) s jedinou jedničkou (černou / živou buňkou) uprostřed. Lze si všimnout, že budeme využívat jen automaty se sudými čísly. Ty s lichými čísly totiž zobrazují (0, 0, 0) na 1 a hned v prvním kroce zaplní neomezené množství buněk.

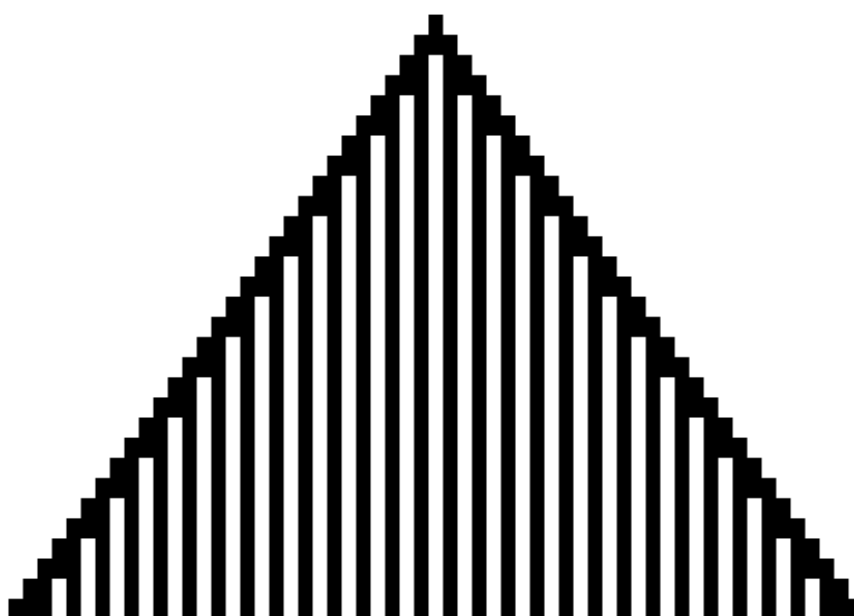
Příkladem CA s úplně jednotvárným chováním (Class I) je elementární automat číslo 220 (viz 1.1). Podobně jednotvárné chování vykazuje automat číslo 94 (viz 1.2). Zajímavější je automat číslo 90, který opakuje stejné vzory (Class II) a dokonce generuje fraktální tvary (viz 1.3). Jeho pravidlo se dá popsat tak, že ignoruje současný stav buňky a jako nový stav buňka nastaví xor mezi stavy levého a pravého souseda. Automat číslo 30 vykazuje chaotické (pseudonáhodné) chování (Class III, viz 1.4). A nakonec si prohlédneme automat číslo 110 (viz 1.5), který vykazuje určité lokální struktury, které spolu mohou složitě interagovat (Class IV).

## 1.3 Jiné 1D celulární automaty

- Nemusíme se omezovat jen na těsné sousedy.

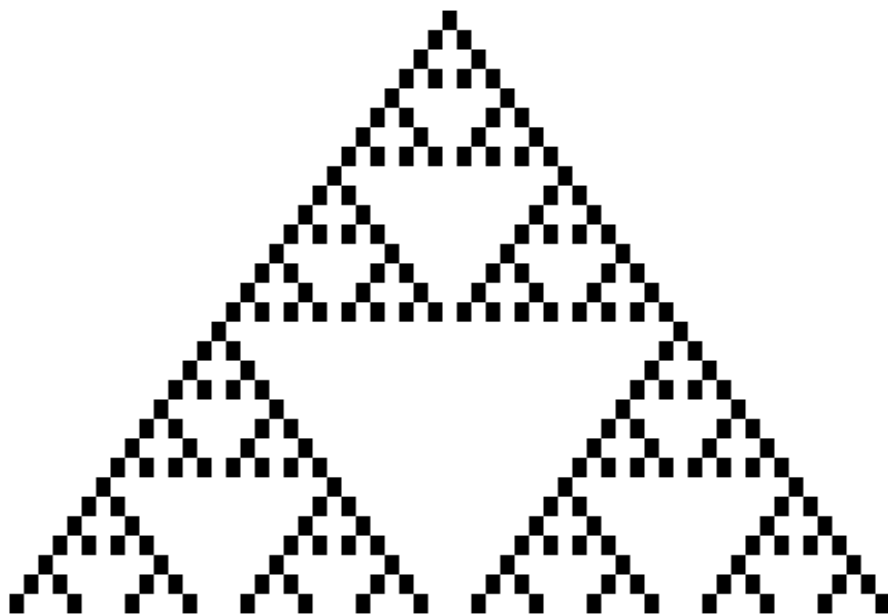


Obrázek 1.1: Elementární automat číslo 220

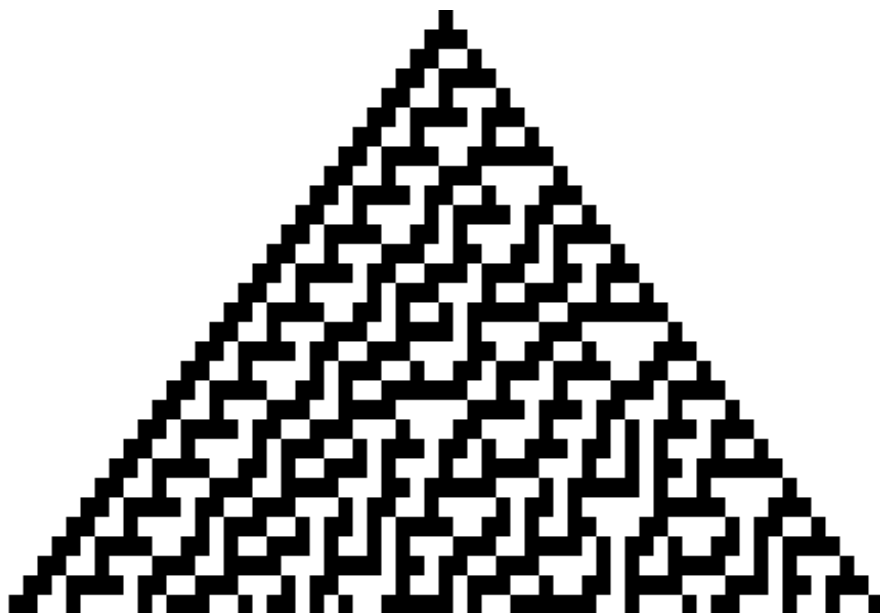


Obrázek 1.2: Elementární automat číslo 94

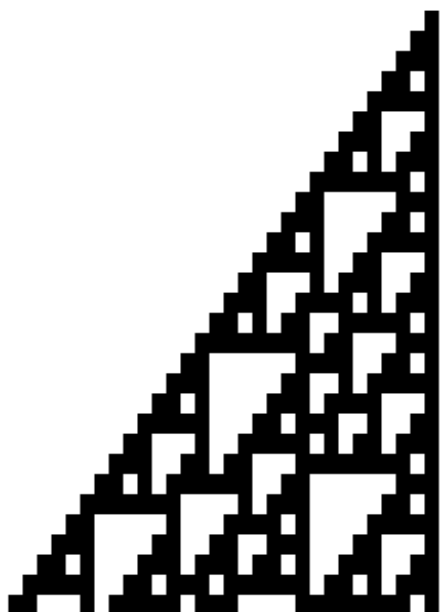




Obrázek 1.3: Elementární automat číslo 90



Obrázek 1.4: Elementární automat číslo 30



Obrázek 1.5: Elementární automat číslo 110

- Můžeme povolit více než 2 stavy.
- Významným druhem celulárních automatů jsou totalistické automaty. Nehledě na to, jak velké se používá okolí a kolik stavů buňky nabývají, v totalistických automatech se pouze číselně posčítají hodnoty hodnoty buňky s celým jejím okolím a podle součtu hodnot se přiřadí výsledná hodnota. Přechodová funkce pro totalistický automat s okolím velikosti  $o$  a počtem stavů  $s$  se tedy dá vyjádřit jako:

$$f : \{0, \dots, (s-1)(o+1)\} \rightarrow \{0, \dots, s-1\}$$

## 1.4 Dvourozměrné celulární automaty

V oblasti 2D celulárních automatů se používají hlavně totalistické automaty, protože vytváření jiných typů pravidel by bylo příliš složité. Typickým příkladem totalistického 2D automatu je Game Of Life. To je dvourozměrný automat nad binární abecedou používající 8-okolí, který se řídí pravidlem, že živá buňka přežívá, pokud má 2 až 3 živé sousedy (jinak umírá), zatímco mrtvá buňka obžije, pokud má právě 3 živé sousedy. Game Of Life se stal zdrojem mnoha různých hříček. Nicméně pro účely šifrování se nezdá být moc užitečný, protože nejde dostatečně parametrizovat.

## 1.5 Implementace celulárních automatů

V teorii se typicky uvažují celulární automaty na nekonečném hřišti. V počátečním stavu mají se však všechny nenulové hodnoty vyskytují jen uvnitř nějaké konečné oblasti buněk. Nenulové hodnoty se pak ale mohou neomezeně rozrůstat do všech stran. Rychlost tohoto rozšiřování lze zhora omezit na základě velikosti okolí aplikovaného pravidla.

V praxi implementujeme celý automat na konečném hříšti. Možnosti jsou dvě. Buď celý automat zacyklíme a jeho vývoj v čase „pokresluje nekonečnou válcovou plochu“, nebo automat na stranách ohraničíme a při aplikaci pravidel na okraji hřiště čteme nulové hodnoty za jeho okrajem.

Výhodou je snadná implementace a nízké paměťové nároky. Nevýhodou je, že se vývoj celého automatu periodicky opakuje, pokud provedeme dostatečný počet kroků. Délku této periody lze bohužel odhadnout pouze zhora.

## 2. Šifrování

Věda zabývající se šifrováním se nazývá *kryptografie*. Lámáním šifer se zase zabývá *kryptoanalýza*. Úkolem šifrování je uchovat a předat tajnou zprávu tak, aby ji mohl přecíst ten, pro koho je určena, ale už nikdo jiný. Dále se někdy za cíl dává ověřitelnost autora zprávy.

Původní čitelná zpráva se nazývá *plaintext*. Data po zašifrování se nazývají *ciphertext*. Pro převod plaintextu na ciphertext je potřeba použít šifrovací algoritmus. Ten by měl zároveň být schopen převést ciphertext zpět na plaintext (tzv. dešifrování). Protože fungování šifrovacího algoritmu se velmi snadno vyzradí, zásadní roli hraje *šifrovací klíč*. Pokud se jedná o *symetrickou kryptografii*, tak stejný klíč slouží i k dešifrování. V případě *asymetrické kryptografie* se používají dva různé klíče. Šifrovací resp. dešifrovací klíče v asymetrické kryptografii se s ohledem na jejich použití nazývají jako *veřejný* resp. *tajný* klíč.

Před zašifrováním zprávy se často provádí její *komprese*. To má za následek nejen úsporu přenosového pásma a množství práce (času) šifrovacího algoritmu, ale velkou výhodou je dosažení výrazně rovnoměrnějšího pravděpodobnostního rozdělení na prostoru plaintextů, což výrazně komplikuje kryptoanalýzu.

### 2.1 Několik ukázek

bla bla bla

V této práci se budeme věnovat vytvoření algoritmu na protahování klíčů (anglicky *key stretching*). Cílem je z krátkého klíče (který lze například v krátkém čase přenést pomocí RSA) vygenerovat dlouhý klíč (kterým lze přeXORovat celý soubor). Je to tedy podobný úkol jako naprogramovat *Key Stream Generator*, akorát my budeme dopředu vědět cílenou délku výsledného klíče.

Jako *Key Stream Generator* se dá použít například mnoho blokových šifer. U blokových šifer záleží na módu operace. Při *Cipher Block Chaining* (CBC, PCBC) či *Cipher Feedback* (CFB) módu zašifrování druhé části plaintextu záleží na výsledku zašifrování první části, tudíž to není *Key Stream Generator*. Ale při zapojení jako třeba *Counter* (CTR) vzniká proud bitů bez znalosti plaintextu, takže získáme *Key Stream Generator*. Dobrým příkladem je třeba AES-CTR.

### 2.2 Testy

Je příliš obtížné ukázat o šifrovacím algoritmu, že je doopravdy kvalitní. Jako záruka kvality se proto v praxi používá spíše jeho zveřejnění na několik let, aby ho měli šanci oponovat nejlepší odborníci. Pokud se ani po několika letech neukáže jeho slabina, je šifrovací algoritmus považován za dost dobrý. Naštěstí alespoň ty velmi špatné šifrovací algoritmy je možné rychle rozpoznat statistickými testy. Na to se zaměříme v této práci.

### 2.2.1 Testy na úrovni jednotlivých výstupů

Pro kryptografii je potřeba, aby dlouhé klíče měly vlastnosti pseudonáhodných posloupností. Pochopitelně zde není možné dosáhnout, aby všechny dlouhé klíče byly stejně pravděpodobné a dosáhli bychom tak „pravé náhodnosti“, ale můžeme alespoň otestovat konkrétní výstup, jestli se podobá náhodné posloupnosti.

Třída `Crypto.RandomnessTesting` obsahuje následující metody.

- `EntropyTest(BitArray b, byte lengthLimit)` : Testuje entropii bloků o velikosti od 1 po `lengthLimit`. Jde o sledování frekvence jednotlivých bloků. V náhodné posloupnosti by měly být všechny bloky stejné délky přibližně stejně časté. Pro krátké posloupnosti (zde pod 10 tisíc bitů) samozřejmě není možné, aby se všechny bloky (zde délky 10) vyskytly, takže jsou všechny výsledky porovnány s maximálním možným výsledkem. Výstupem je vážený průměr, kde mají testy entropií všech délek výsledky v rozsahu 0 až 1. Optimální hodnota je 1.
- `CompressionTest(BitArray b)` : Zkusí data zkomprimovat pomocí programu `gzip` s optimální úrovní komprese a vrátí poměr mezi novou a původní velikostí. O posloupnostech, které lze zkomprimovat, je známo, že nejsou dokonale náhodné. Konkrétně velikost dat po kompresi je horním odhadem na Kolmogorovskou složitost. Optimální hodnota je 1.

### 2.2.2 Testy na úrovni celého zobrazení

Skutečnost, že výstupem algoritmu je pseudonáhodná posloupnost čísel, je jistě dobrá. Ale co když algoritmus všem vstupům přiřadí stejnou pseudonáhodnou posloupnost? Nebo jeden konkrétní bit na vstupu neovlivní výsledek? Takovou nekvalitu musí objevit druhá skupina testů.

Budeme zde zkoumat vlastnosti algoritmů jako vlastnosti celého zobrazení z krátkých klíčů do dlouhých klíčů. Protože si budeme často klást otázky typu „Jak moc se liší výstup A od výstupu B?“, tak by bylo vhodné zavést nějaké hodnocení, ideálně s vlastností metriky. Porovnávat budeme vždy jen výstupy stejné délky. Oblíbenými metrikami pro řetězce jsou *Hammingova vzdálenost* a *Levenshteinova vzdálenost*. Hamming měří počet pozic, na kterých se řetězce liší. Levenshtein měří minimální počet potřebných změn k tomu, abychom z jednoho řetězce dostali ten druhý. Jako změnu je možné provést záměnu znaku, smazání znaku, nebo dopsání znaku na libovolné místo.

Hammingova vzdálenost je triviálně horním odhadem na Levenshteinovu vzdálenost. V případě náhodných binárních řetězců je jejich hodnota občas stejná, ale někdy může být Levenshtein výrazně nižší. Například když zrotujeme řetězec o jednu pozici, tak Levenshtein dává vzdálenost 2, zatímco Hamming může dát hodně vysoké číslo (až délka řetězce). Významná pro nás bude rychlost výpočtu. Hammingovu vzdálenost lze triviálně určit v lineárním čase, ale výpočet Levenshteinovy vzdálenosti zabere čas kvadratický. Že to rychleji nejde, se nelze divit, protože Levenshtein vlastně spouští prohledávání prostoru editací. Díky dynamickému programování to lze provést alespoň v tom kvadratickém čase.

Program ještě obě vzdálenosti vydělí délkou řetězce, aby výsledky měly hodnotu v rozmezí 0 (shodné řetězce) až 1 (například řetězec samých nul porovnan se řetězcem samých jedniček). Střední hodnota pro dvojici náhodných binárních posloupností je u Hamminga

triviálně 0,5. Nesrovnatelně těžší je odhadnout střední hodnotu Levenshteinovy vzdálenosti. Posloupnost středních hodnot Levenshteina se vzrůstající délkou vstupu roste jako subaditivní posloupnost. Relativní hodnota proto může pro delší vstupy pouze klesat. Limitní hodnotu se zdá být příliš těžké odhadnout, ale orientační experimenty i diskuze na internetu naznačují, že můžeme počítat s hodnotou kolem 0,29.

Třída `Crypto.FunctionTesting` obsahuje následující metody. Podle nastavení v konstruktoru mohou všechny testy používat buď Hammingovu, nebo Levenshteinovu vzdálenost. Dále uváděno podle Hamminga.

- `TestBitChange(IKeyExtender algorithm, int ratio)` : Testuje, jak velká část bitů výstupu se změní při změně jednoho bitu vstupu. Metoda sampuluje náhodné vstupy a pro každý z nich zkouší změnit zvlášť všechny bity. Optimální hodnota je 0,5.
- `TestAverageDistance(IKeyExtender algorithm, int ratio)` : Testuje průměrnou vzdálenost výstupů příslušející dvěma různým náhodně zvoleným vstupům. Optimální hodnota je 0,5.
- `TestLargestBallExactly(IKeyExtender algorithm)` : Testuje, jaká největší koule se dá vměstnat do prostoru výstupů tak, aby neobsahovala žádný vygenerovatelný dlouhý klíč. Zkouší úplně všechny vstupy na malém prostoru a ty natahuje na dvojnásobek. Motivací je, že pokud zobrazení nazaplní prostor dostatečně rovnoměrně, pak to rozpoznáme tak, že se do prostoru vejde velká koule.
- `TestLargestBallApprox(IKeyExtender algorithm)` : Testuje to samé, ale používá delší vstupy, které už nezvládá vyzkoušet všechny, takže je sampuluje náhodně.

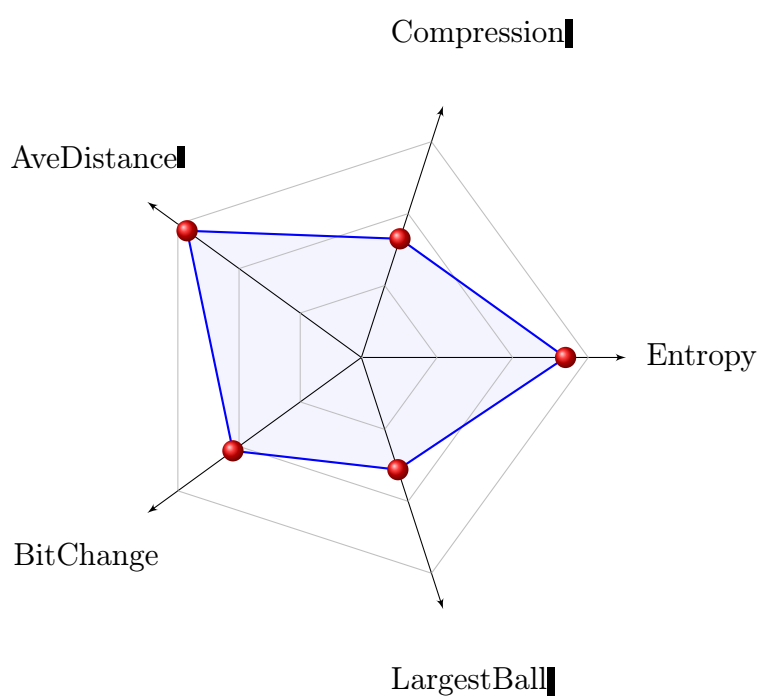
### 2.2.3 Použití testů

Všechny doposud zmíněné testy jsou obaleny ve třídě `Crypto.FunctionTestsForThesis`, kde jsou výsledky těchto testů transformovány způsobem, který zaručí, že vyšší výsledek je lepší. S výjimkou testů maximálních koulí, které jsou realizovány v této třídě trochu jinak, platí, že nejhorší hodnota je 0 a nejlepší hodnota je 1.

### 2.2.4 Grafické znázornění

Výsledky jednotlivých algoritmů ve výše uvedených testech budeme znázorňovat na diagramech jako je tento:

Pro diagramy jsou hodnoty přeškálovány. S výjimkou testů největších koulí, kde neznáme optimální hodnotu, je škálování takové, aby optimální hodnota byla 1. Tedy například když `TestBitChange` vrátí hodnotu  $x$ , pak je do diagramu zobrazeno  $\min\{2x, 2(1 - x)\}$ , aby optimum (zanesené jako hodnota 1) byl výsledek 0,5 a odchylky na obě strany „stejně vážné“.



Obrázek 2.1: Ukázkový radar chart

## 3. Nástin architektury

Tato kapitola je určitým doplněk k vývojové dokumentaci, která byla vytvořena programem Doxywizard z dokumentačních komentářů. Nesnaží se nahradit čtení této dokumentace ani přečtení ostatních kapitol této práce, ve kterých jsou podrobněji vysvětleny ty klíčové části programu.

V rámci jedné Solution ve Visual Studiu byly vytvořeny dva projekty. Projekt se jménem MartinDvorak, který měl původně být jedinou částí aplikace a měl jasně identifikovat tuto práci při jejím elektronickém odevzdávání, obsahuje veškerou logiku popisovanou v textu práci. Pomocí tohoto projektu byly prováděny veškeré experimenty. Jako druhý vznikl malý projekt se jménem Program. Ten využívá nástroje vytvořené v prvním projektu a kompiluje se na WinForms aplikaci, kterou mohou uživatelé použít k zašifrování svých souborů.

### 3.1 MartinDvorak

#### 3.1.1 Cellular

Sem bude přelita dokumentace k ročníkovému projektu.

#### 3.1.2 Crypto

Tento namespace jednak obsahuje natahovače klíčů (popsané v následující kapitole) a potom testy (popsané v předchozí kapitole). Asi zde vysvětlit Factory.

#### 3.1.3 Testing

Nejedná se o žádné Unit testy, ale o poměrně chaotickou hromadu metod, které umožňují zkoušet různé části programu.

### 3.2 Program

Okenní aplikace, umožňuje šifrovat a dešifrovat, volba vstupního a výstupního souboru, bere si IKeyExtender stanovený v projektu MartinDvorak, hešuje heslo, používá salt (a o ten je zvětšen šifrovací soubor). Heslo a salt společně určují počáteční stav celulárního automatu. Vygenerovaným dlouhým klíčem se pak přeXORuje celý soubor. Jednoduché jako facka (jenom jedna třída).



## 4. Způsoby protahování klíčů

Bylo vytvořeno několik různých algoritmů na protahování klíčů využívajících celulární automaty. Je možné si zvlášť zvolit algoritmus (instanci třídy implementující rozhraní `IKeyExtender`) a zvlášť do něj vložit libovolný binární celulární automat (instanci třídy implementující rozhraní `IBinaryCA`). Některé z algoritmů přímo generují dlouhý klíč zadané délky – ty dědí od abstraktní třídy `KeyExtenderAbstractN`. Jiné algoritmy vždy prodlouží klíč na dvojnásobek a obecné natažení realizují iterací tohoto postupu – ty jsou odvozené od abstraktní třídy `KeyExtenderAbstractD`.

Kromě opravdových algoritmů byly vytvořeny ještě dva falešné algoritmy pro účely testování. Jeden z nich jen kopíruje kratší klíč dokola. Jeho výsledek znázorňuje obrázek 4.1. Druhý generuje pseudonáhodnou posloupnost bez ohledu na vstup, což je podvod. Jak hezky vypadá výsledek podvodného generátoru, si můžete prohlédnout na obrázku 4.2.

### 4.1 `KeyExtenderSimpleLinear`

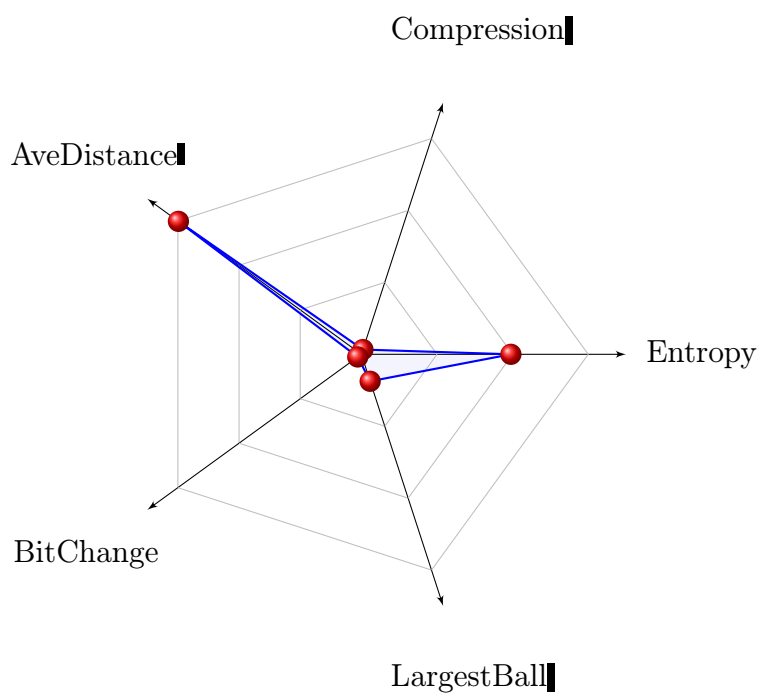
Nejjednodušší způsob, jak natáhnout klíč na dvojnásobek. Tento algoritmus použije vstup jako počáteční konfiguraci celulárního automatu. Pak udělá krok a uloží jeho stav do první poloviny dlouhého klíče (postupně ze všech buněk). Pak udělá druhý krok a načte druhou polovinu dlouhého klíče.

Kombinace jednoduchého algoritmu a automatu s jednoduchým chováním vede pochopitelně ke špatným výsledkům (viz obrázek 4.3). Ovšem výsledky automatu, který na vstupu s jedinou jedničkou generuje fraktály, není tak špatný (viz obrázek 4.5). Příjemným překvapením je ovšem výsledek 2D automatu *Amoeba Universe* (viz 4.10).

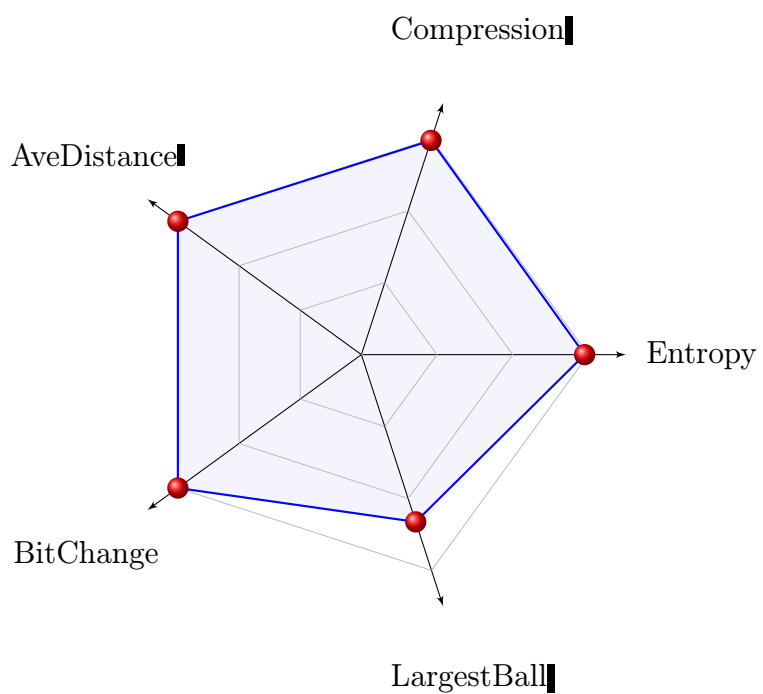
### 4.2 `KeyExtenderSimpleQuadratic`

Tento algoritmus natahuje klíč na dvojnásobek způsobem, který je mnohem lepší, ale mnohem pomalejší. Nejprve vytvoří celulární automat, který je dva a půl krát širší, než délka krátkého klíče. Ten krátký klíč uloží do jeho prostředních buněk. Tedy klíč délky  $n$  se uloží do stavu automatu s  $2,5n$  buňkami a to od  $0,75n$  po  $1,75n$ . Pak automat udělá  $2n$  kroků. Výstup se čte z prostřední buňky (1 bit po každém kroku automatu). Je to způsob, o kterém Stephen Wolfram ukázal, že generuje pseudonáhodné posloupnosti, které při volbě správného celulárního automatu splňují všechna měřítka náhodnosti. Jeho návrh používal nekonečnou plochu, ale protože nám stačí natažení na  $2n$ , tak plocha o šířce  $2,5n$  funguje stejně, jako kdyby se automat mohl rozpínat do nekonečna.

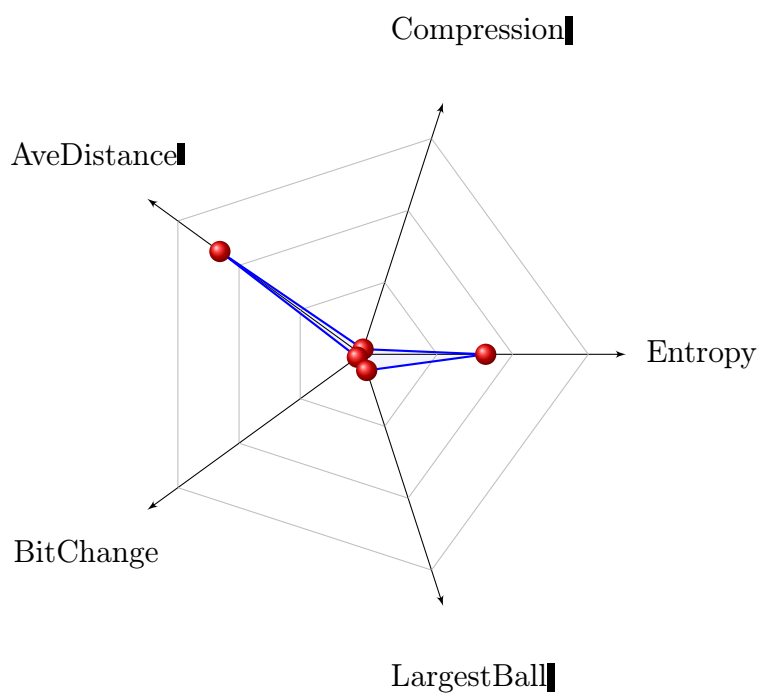
Vidíme, že použitý automatů s monotónním chováním vede k příšerně špatným výsledkům (například viz 4.11). Fraktálně na tom také nejsou dobře (třeba viz 4.13), ale automat číslo 30, který vykazuje pseudonáhodné chování, dává opravdu vynikající výsledek (viz 4.14). Použití náhodného (i když jinak kvalitně zvoleného) automatu s 2-okolím vede jen k průměrnému výsledku (jako třeba na obrázku 4.16).



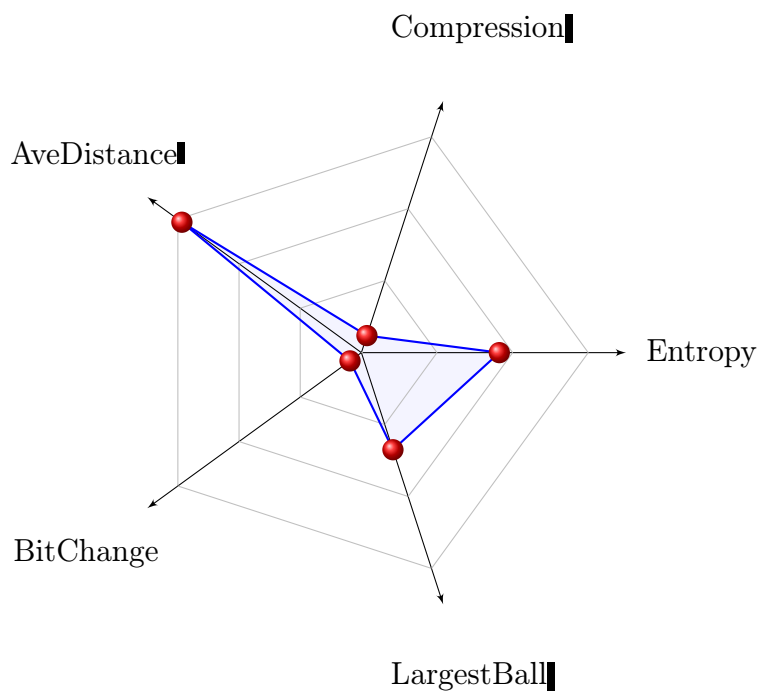
Obrázek 4.1: KeyExtenderCopy



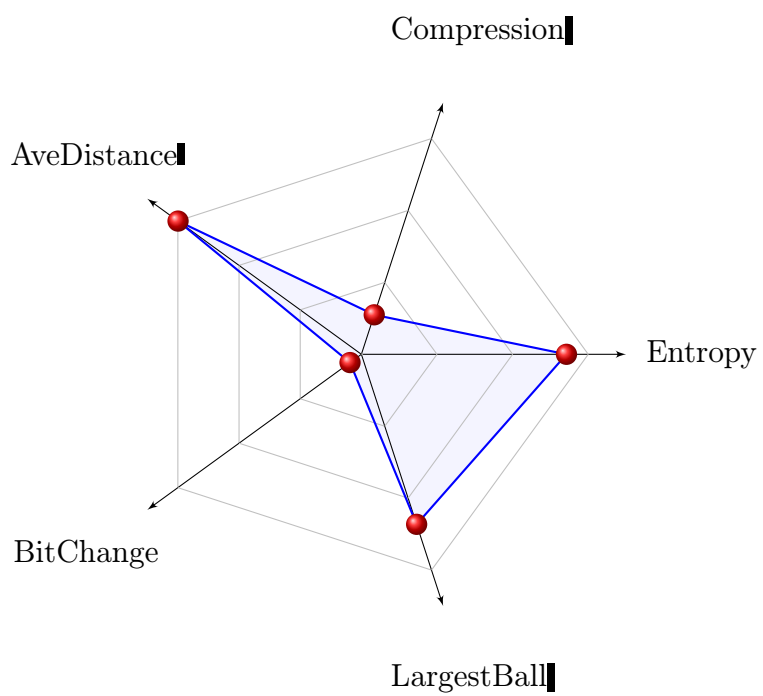
Obrázek 4.2: KeyExtenderCheating



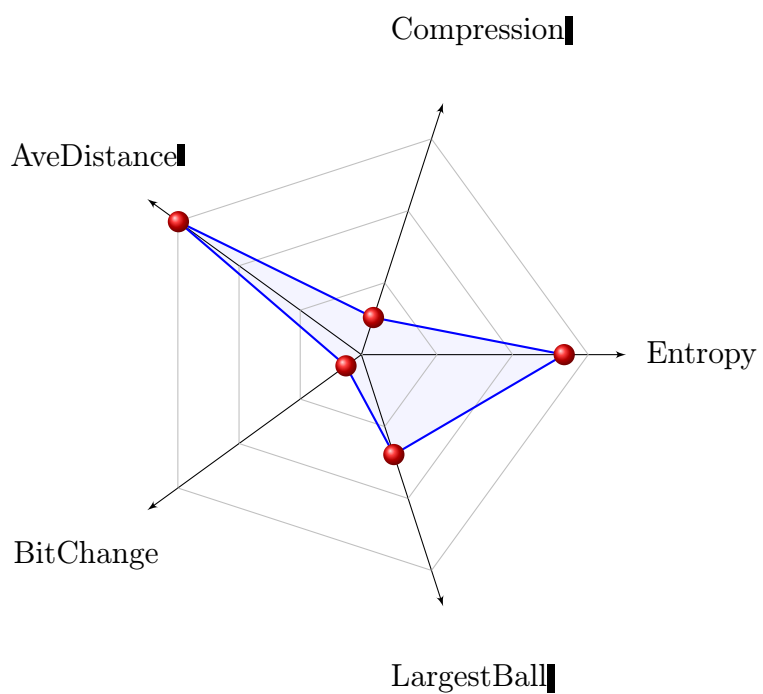
Obrázek 4.3: KeyExtenderSimpleLinear na automat 220



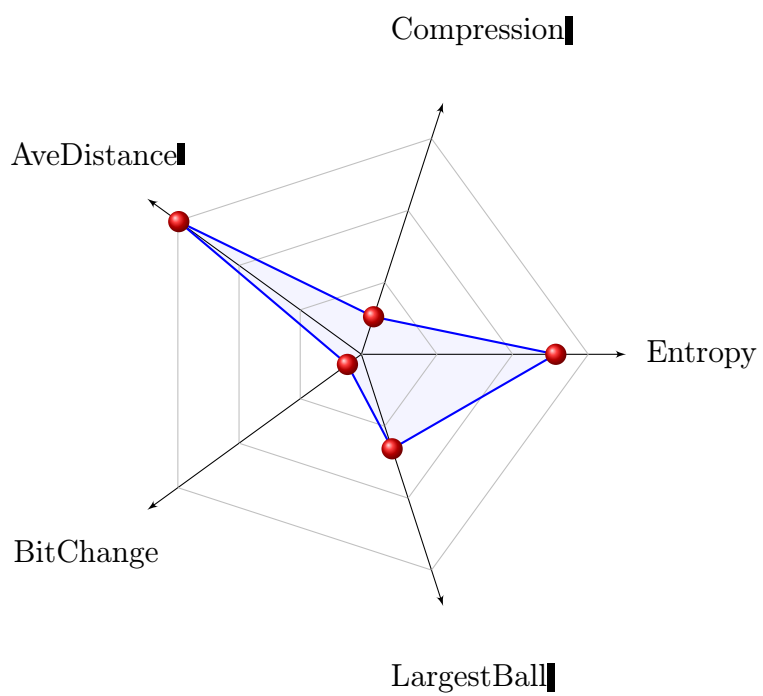
Obrázek 4.4: KeyExtenderSimpleLinear na automat 94



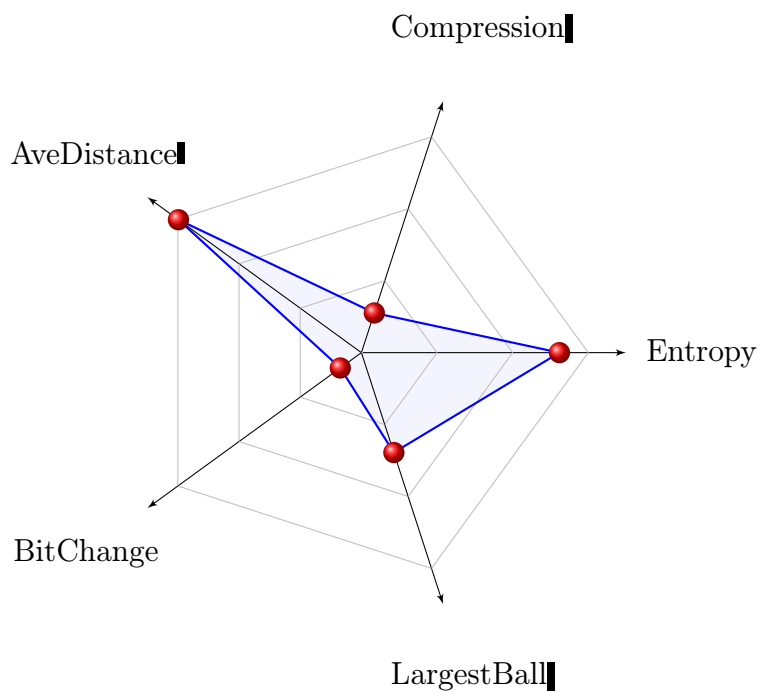
Obrázek 4.5: KeyExtenderSimpleLinear na automat 90



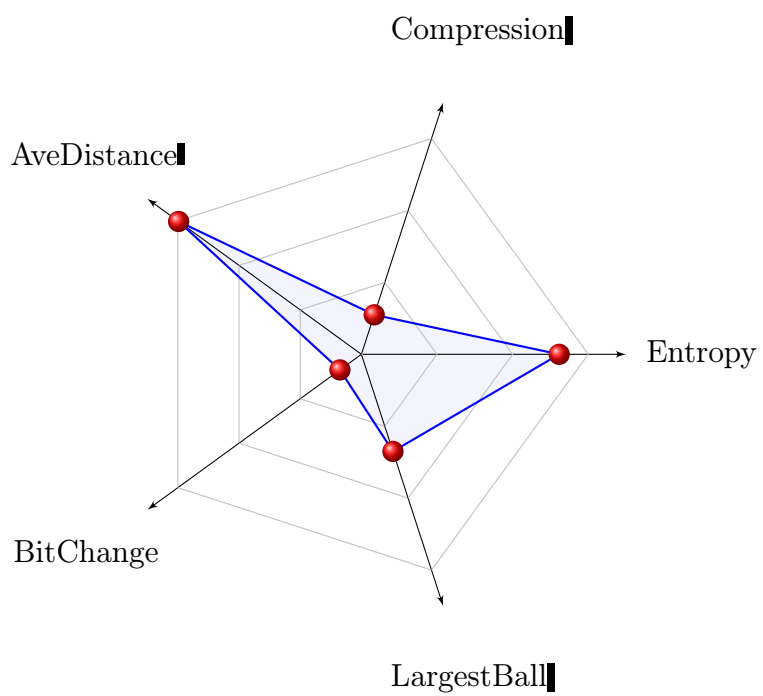
Obrázek 4.6: KeyExtenderSimpleLinear na automat 30



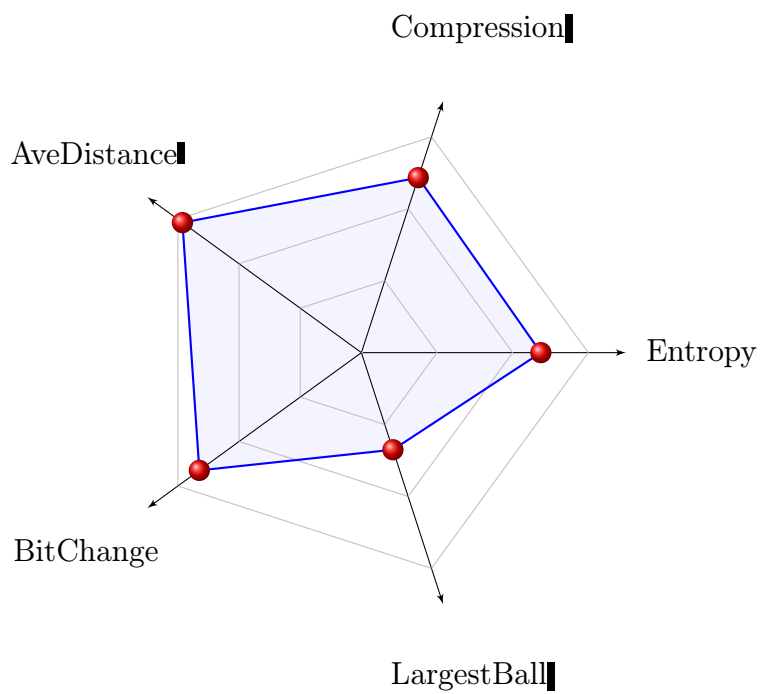
Obrázek 4.7: KeyExtenderSimpleLinear na automat 110



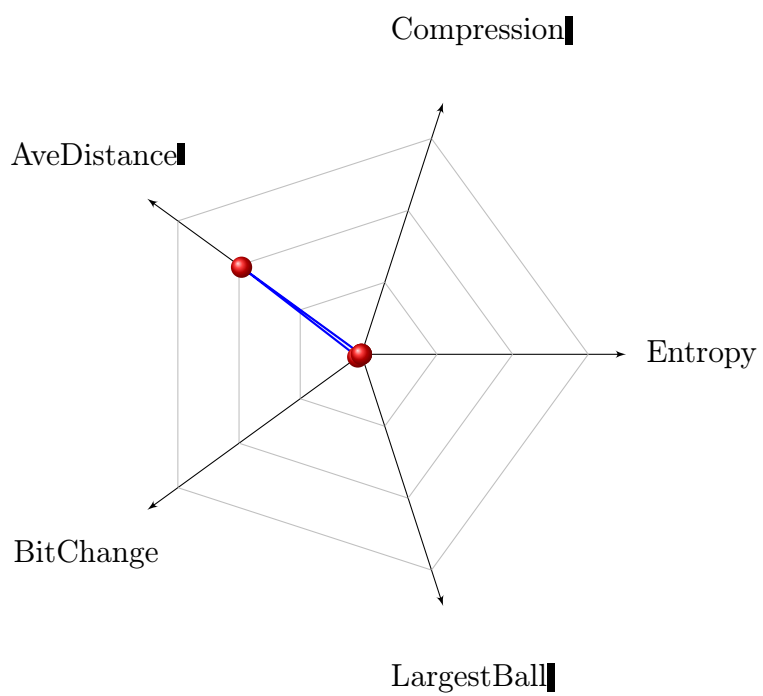
Obrázek 4.8: KeyExtenderSimpleLinear na omezený automat s 2-okolím



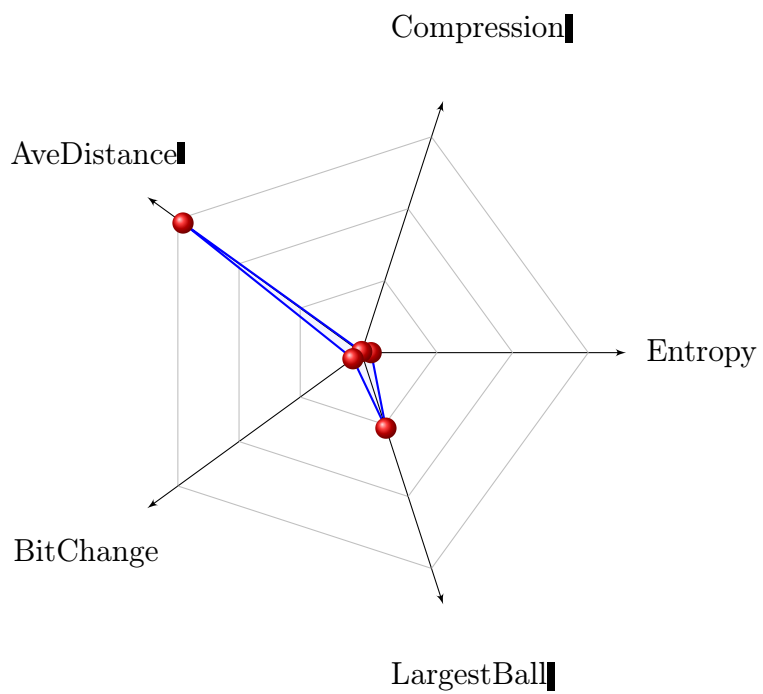
Obrázek 4.9: KeyExtenderSimpleLinear na cyklický automat s 2-okolím



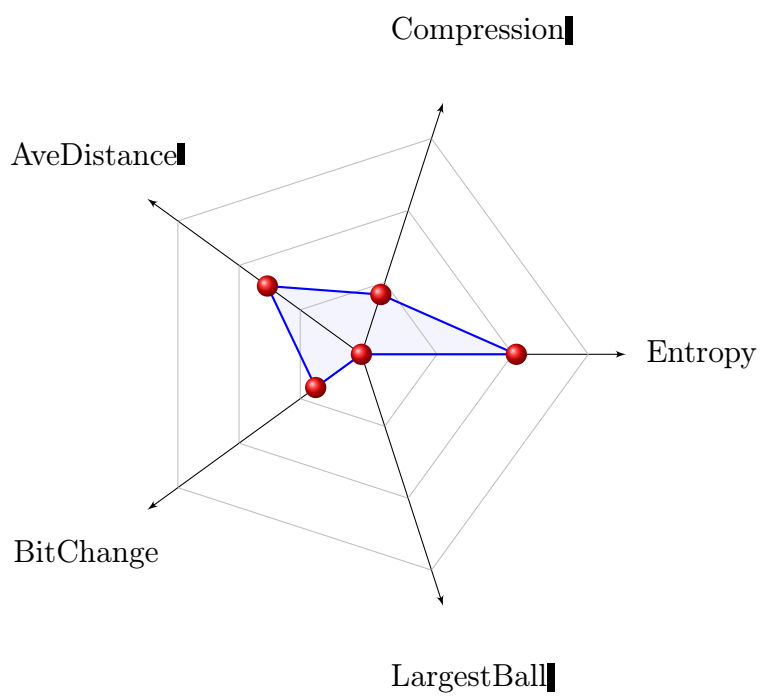
Obrázek 4.10: KeyExtenderSimpleLinear na Amoeba Universe



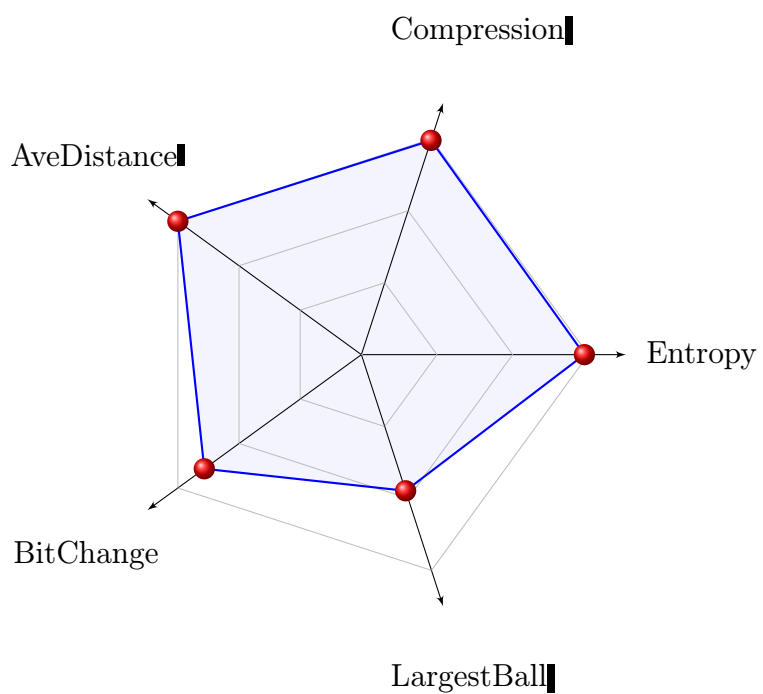
Obrázek 4.11: KeyExtenderSimpleQuadratic na automat 220



Obrázek 4.12: KeyExtenderSimpleQuadratic na automat 94

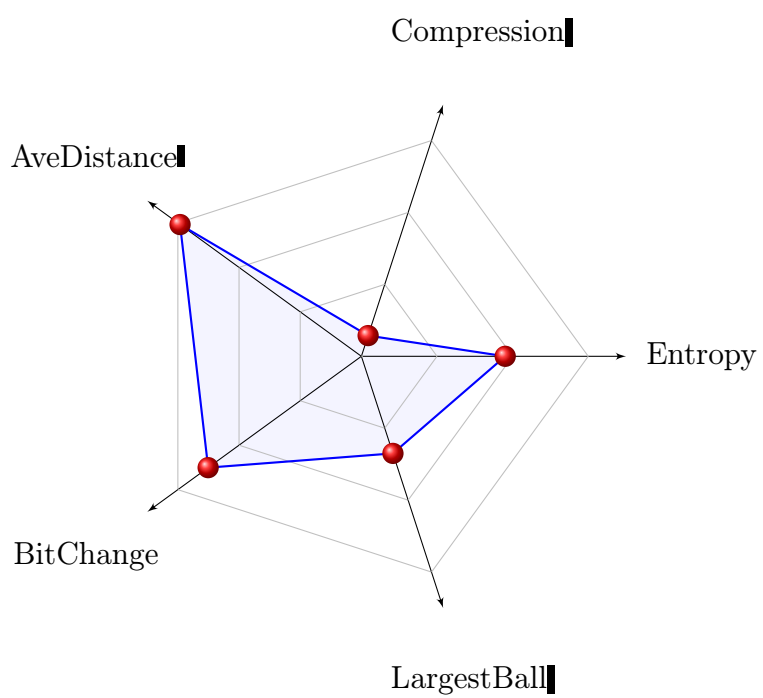


Obrázek 4.13: KeyExtenderSimpleQuadratic na automat 90

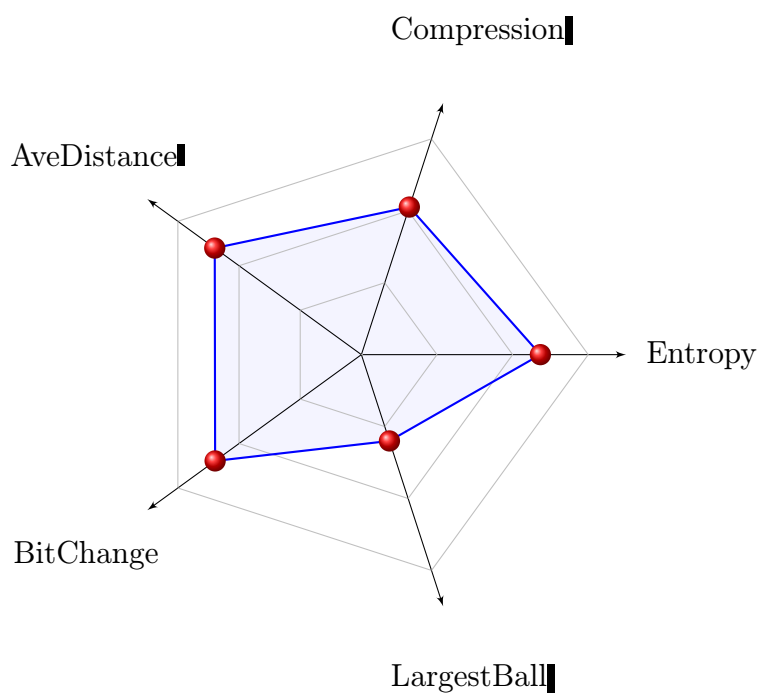


Obrázek 4.14: KeyExtenderSimpleQuadratic na automat 30

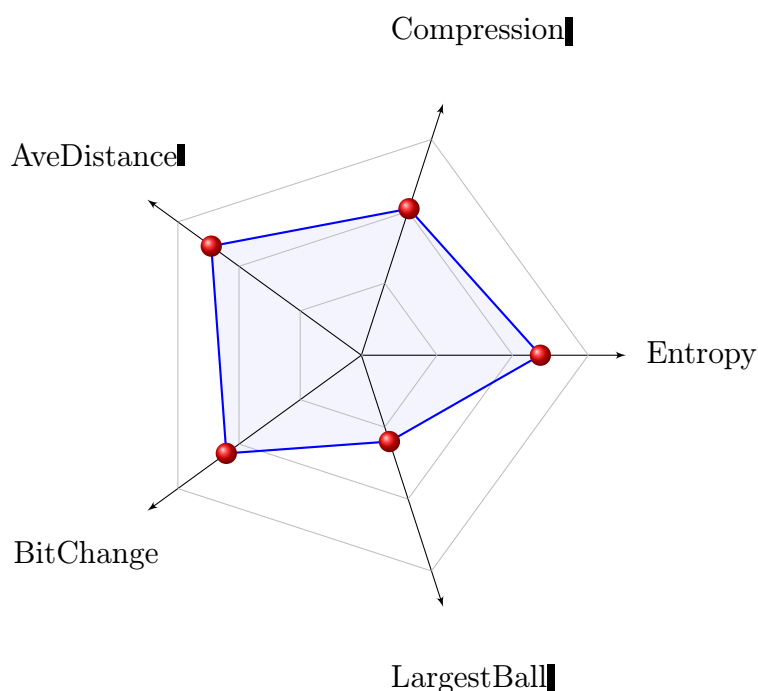




Obrázek 4.15: KeyExtenderSimpleQuadratic na automat 110



Obrázek 4.16: KeyExtenderSimpleQuadratic na omezený automat s 2-okolím



Obrázek 4.17: KeyExtenderSimpleQuadratic na cyklický automat s 2-okolím

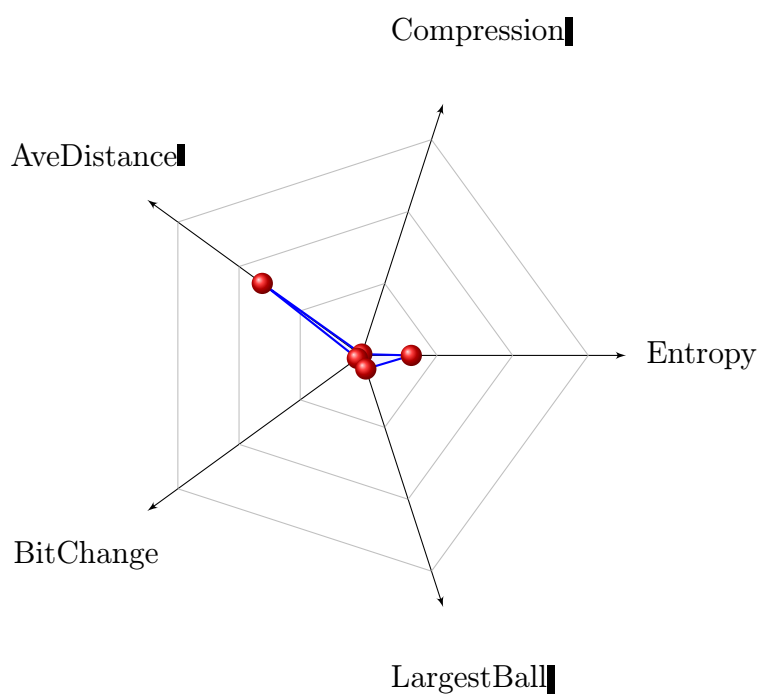
### 4.3 KeyExtenderInterlaced

Tento algoritmus se tváří jako kompromis, ale blíží se spíše první (lineární variantě). Algoritmus je parametrizován počtem řad  $p$ , ze kterých má dlouhý klíč generovat, a údajem  $q$ , kolik kroků navíc má automat vždy provést mezi generováním využívaných stavů. Pokud se spustí s parametry  $p = 2$ ,  $q = 0$ , potom generuje totožný klíč jako lineární algoritmus. Jeho časová složitost je lineární ve velikosti vstupu a ještě lineární v součinu  $p(q + 1)$ .

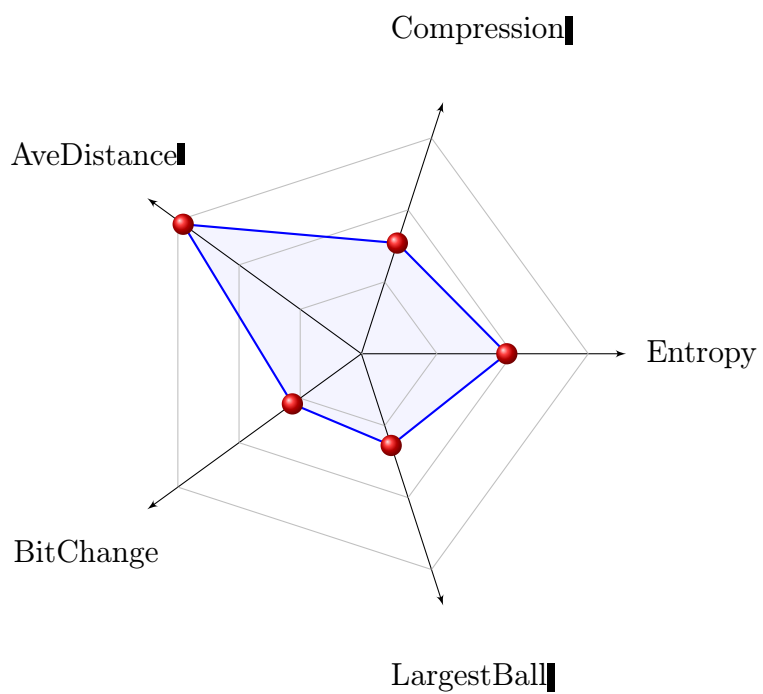
Ve srovnání s KeyExtenderSimpleLinear vede prokládání u některých automatů ke znatelnému zlepšení chování (například viz 4.22), u jiných k výraznému zhoršení chování (například viz 4.25).

### 4.4 KeyExtenderUncertain

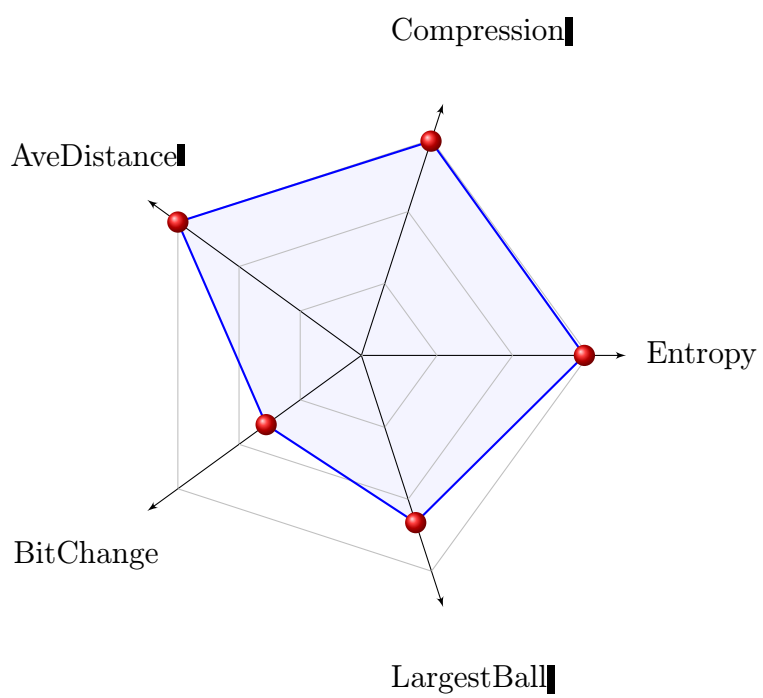
Tento algoritmus řeší častou neřest, kterou vykazují celulární automaty – různý počet 0 a 1. Algoritmus zase využívá celý stav automatu. Jsou čteny vždy dvojice bitů, přičemž dvojice 00 a dvojice 11 jsou zahazovány. Vždy, když algoritmus narazí na dvojici 01, tak pošle na výstup 0. A za každou dvojici 10 pošle na výstup 1. Počet kroků automatu, který bude algoritmus muset provést, není předem známý. Pokud se automat zasekne ve stavu, ze kterého není úniku (například samé nuly u mnoha druhů automatů, nebo také střídání 001100110011..001100 u elementárního automatu, který používá jako přechodovou funkci majorantu buňky a jejich těsných sousedů), je vyhozena výjimka. Případy, kdy se generování dlouhého klíče nepodařilo dokončit, jsou zařazeny do výsledků s hodnotou 0 u všech testů.



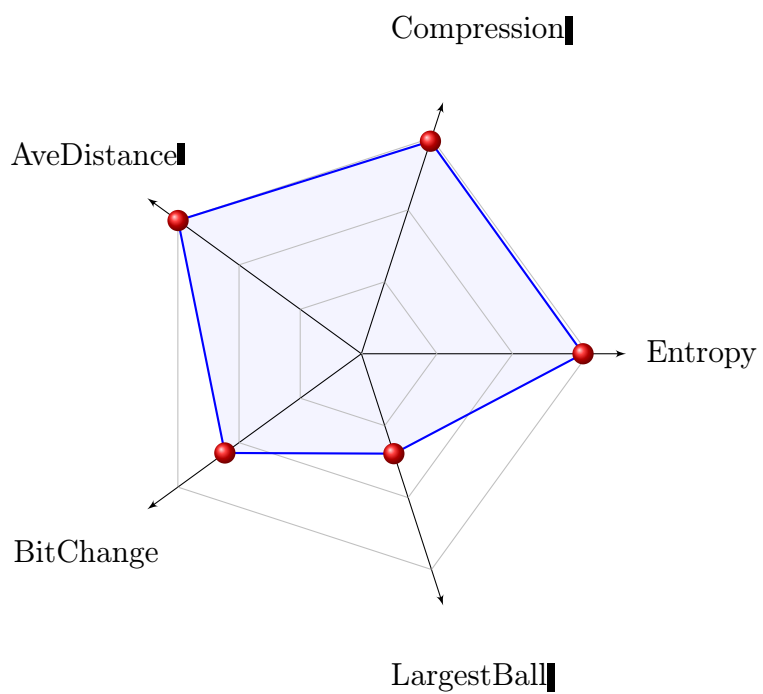
Obrázek 4.18: KeyExtenderInterlaced(10, 0) na automat 220



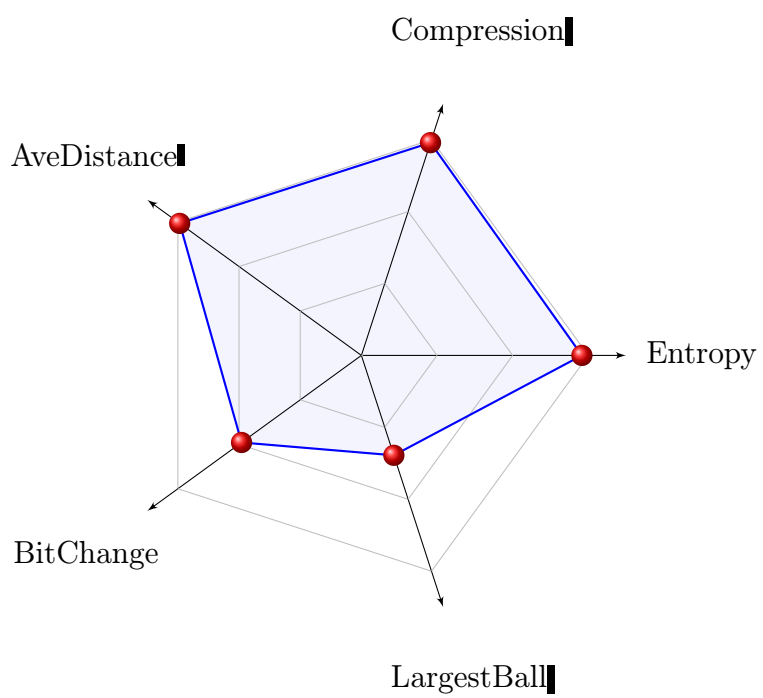
Obrázek 4.19: KeyExtenderInterlaced(10, 0) na automat 94



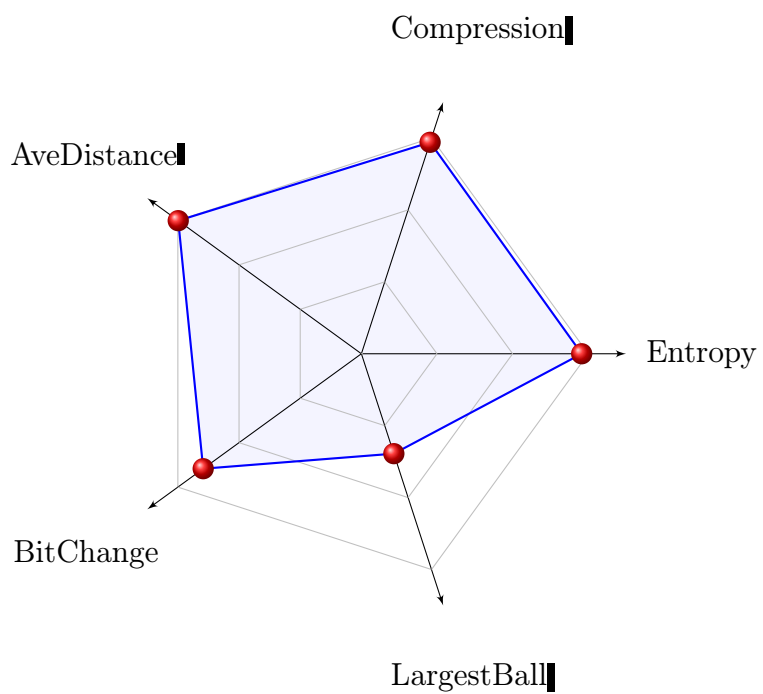
Obrázek 4.20:  $\text{KeyExtenderInterlaced}(10, 0)$  na automat 90



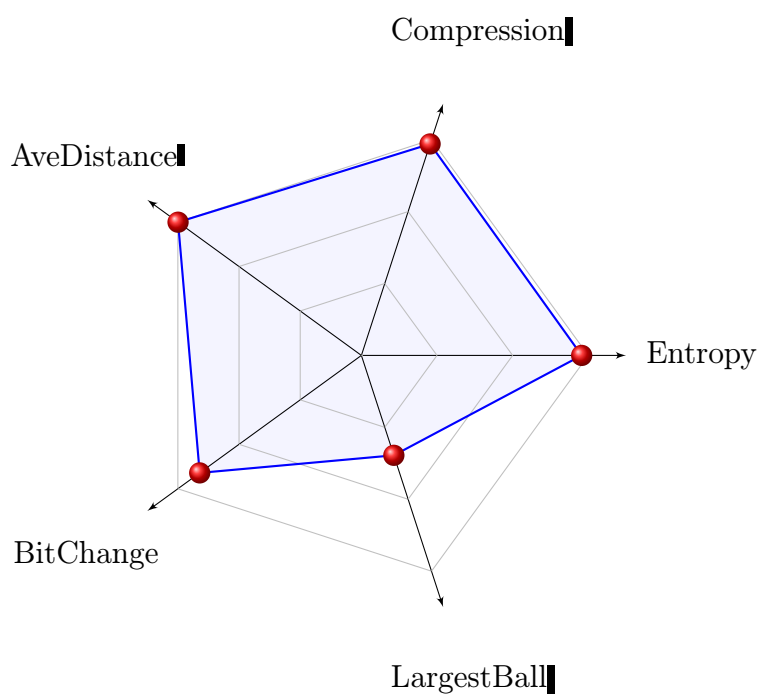
Obrázek 4.21:  $\text{KeyExtenderInterlaced}(10, 0)$  na automat 30



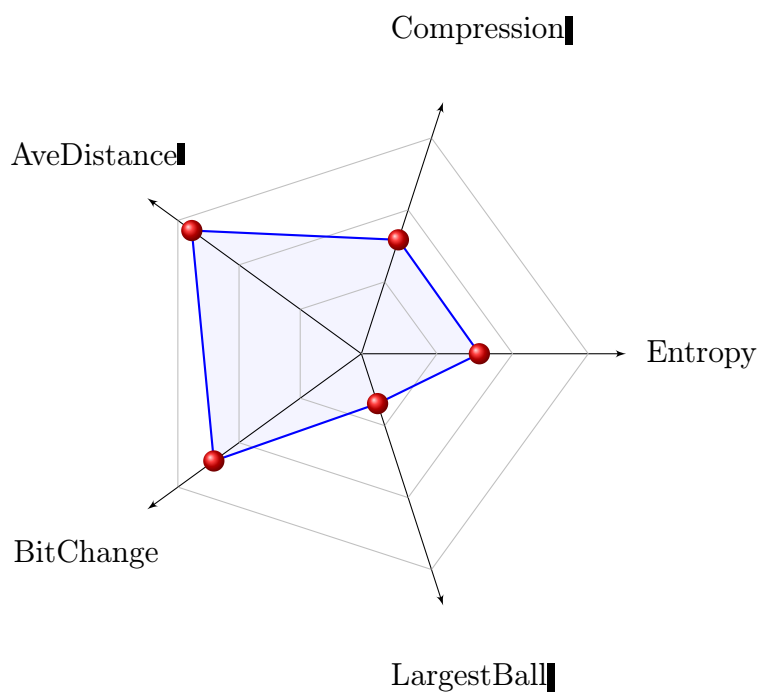
Obrázek 4.22:  $\text{KeyExtenderInterlaced}(10, 0)$  na automat 110



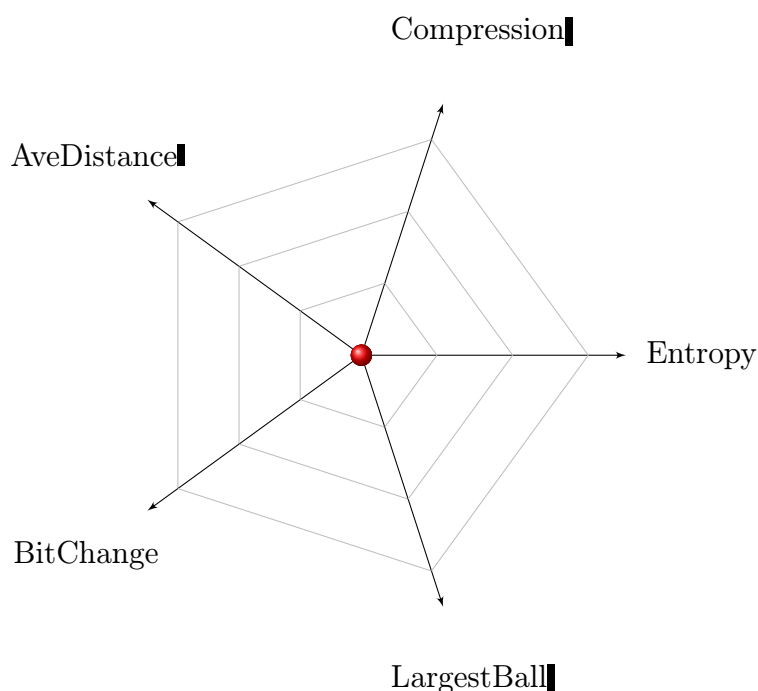
Obrázek 4.23:  $\text{KeyExtenderInterlaced}(10, 0)$  na omezený automat s 2-okolím



Obrázek 4.24: KeyExtenderInterlaced(10, 0) na cyklický automat s 2-okolím



Obrázek 4.25: KeyExtenderInterlaced(10, 0) na Amoeba Universe



Obrázek 4.26: KeyExtenderUncertain na automat 220

Tento netradiční způsob čtení hodnot vedl v pár případech k dobrému chování (třeba na obrázku 4.28).

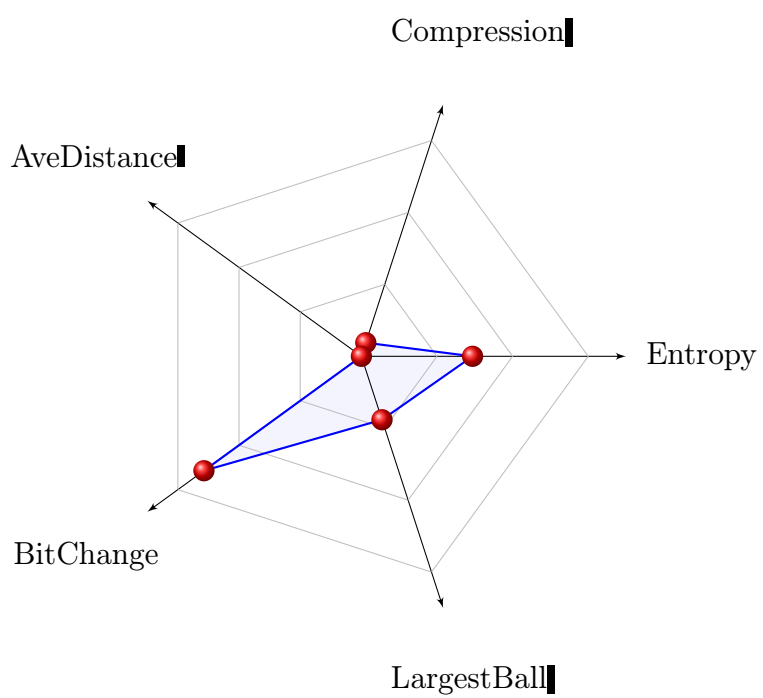
## 4.5 KeyExtenderGenetic

Nejkomplikovanější algoritmus, který tato práce obsahuje. Jako u mnoha jiných algoritmů i zde se klíč postupně natahuje na dvojnásobek, než dosáhne dostatečné délky, ale každý krok může používat jiný druh automatu a jiný dílčí algoritmus. Ke zjištění správné posloupnosti těchto natahovačů je použit genetický algoritmus. Ten tuto posloupnost optimalizuje vždy pro jeden konkrétní vstup.

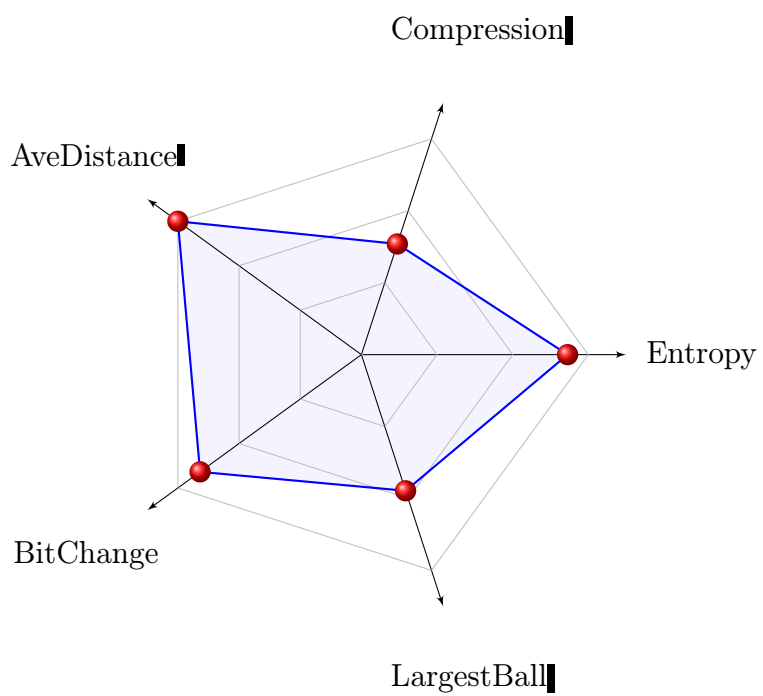
Na začátku je vygenerována náhodná populace, kde jedinci jsou sekvence natahovačů (sekvence je dlouhá  $\lceil \log_2 \frac{\text{novaDelka}}{\text{puvodniDelka}} \rceil$ ), iniciálně vygenerovaná náhodně. Pak je provedena řada iterací, která se skládá z turnajové selekce a genetických operátorů. Na konci iterace je celá původní populace nahrazena novou populací. Nejlepší jedinec z průběhu běhu celého algoritmu je zapamatován a lze k němu přistoupit i po dokončení generování dlouhé sekvence.

Jsou použity dva genetické operátory. Prvním je jednoduché křížení (One-point crossover) a druhým je náhodná mutace (výměna právě jednoho natahovače v sekvenci za náhodný). Pravděpodobnost křížení a pravděpodobnost mutace je dána konstantou v kódu, kterou se klidně můžete volně změnit. Stejně tak lze změnit velikost populace, počet iterací a selekční tlak.

Zbývá určit, co má být tím náhodným tahahovačem... Program obsahuje dvě varianty tohoto: vnitřní třídy Primitives a GoodPrimitives, které obě implementují vnitřní rozhraní IPrimitives. Třída Primitives je implementována jako singleton. Při prvním vy-

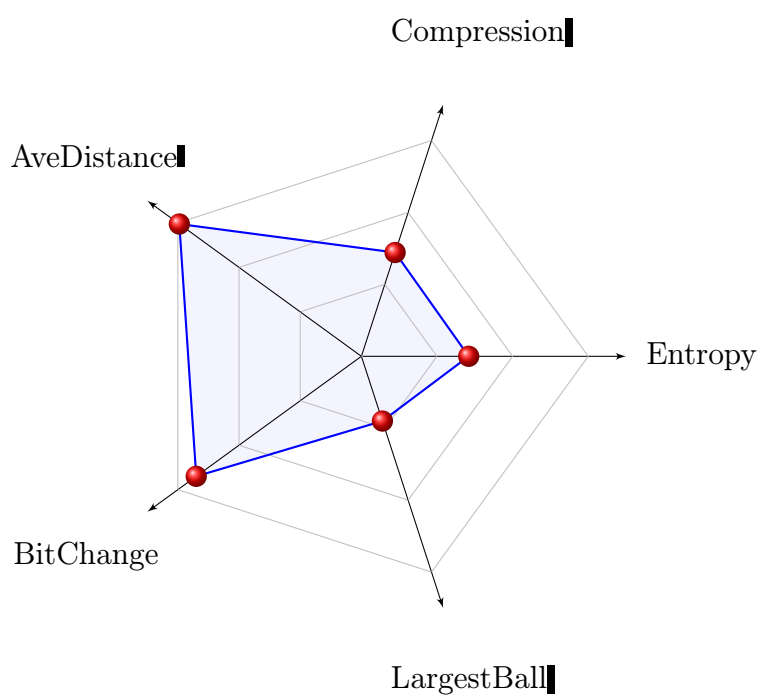


Obrázek 4.27: KeyExtenderUncertain na automat 94

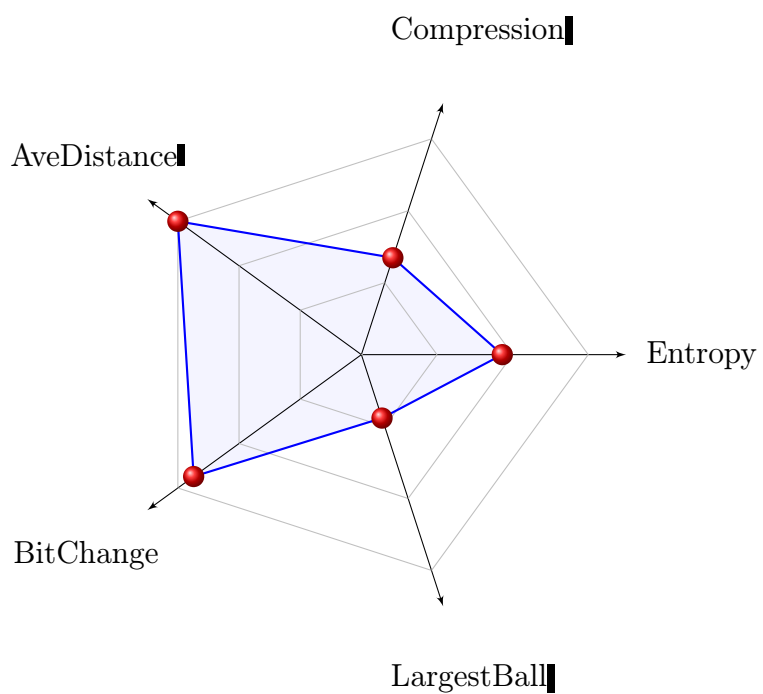


Obrázek 4.28: KeyExtenderUncertain na automat 90

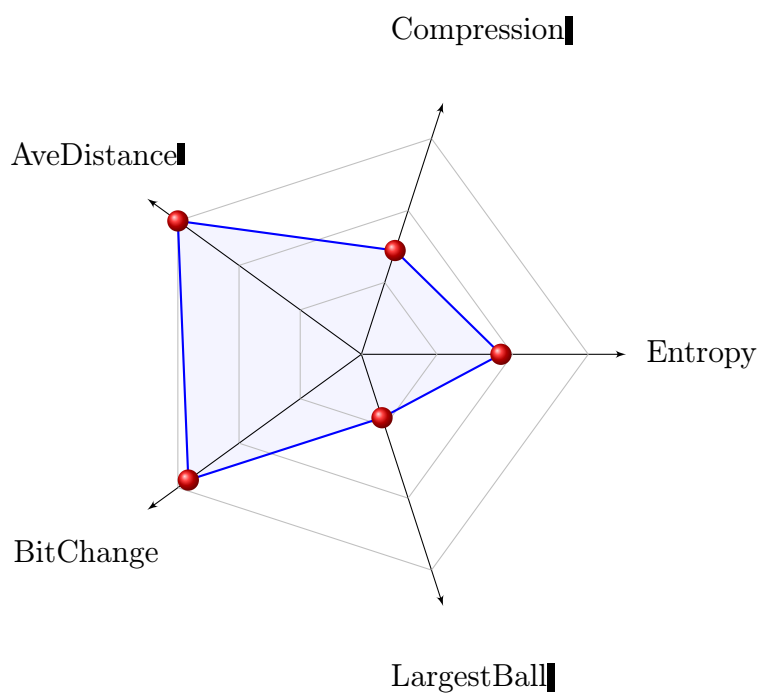




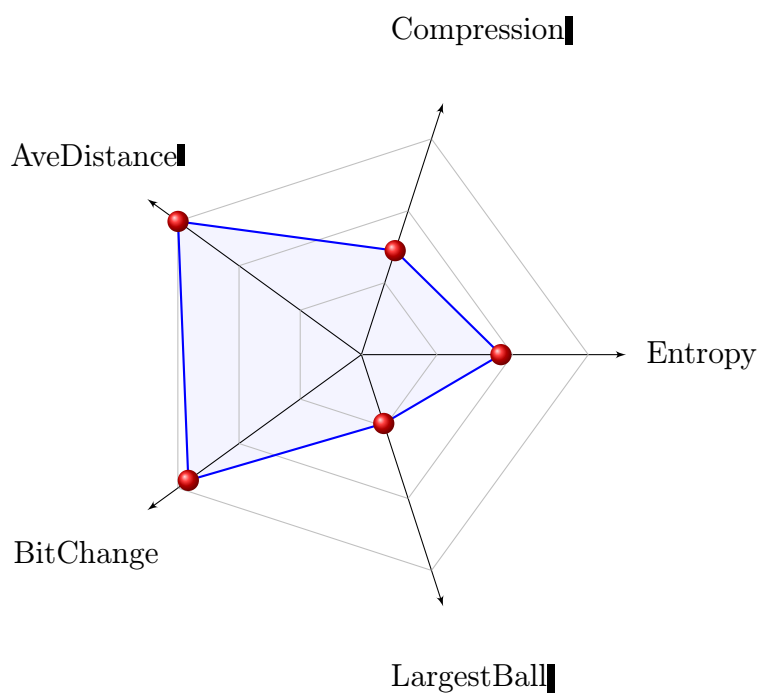
Obrázek 4.29: KeyExtenderUncertain na automat 30



Obrázek 4.30: KeyExtenderUncertain na automat 110



Obrázek 4.31: KeyExtenderUncertain na omezený automat s 2-okolím



Obrázek 4.32: KeyExtenderUncertain na cyklický automat s 2-okolím

tvoření se nejprve vygeneruje seznam binárních CA, které mohou být použity. Do seznamu je zařazeno všech 256 elementárních CA, dále 3 druhy 2D totalistických CA (Game of Life, Amoeba Universe a Replicator Universe) a k tomu 200 kusů náhodně vygenerovaných 1D automatů s přechodovou funkcí využívající buňku, její sousedy a sousedy sousedů (z toho 100 je na omezeném hřišti a dalších 100 na zacykleném hřišti). Dílčí algoritmy pro jednotlivé kroky jsou vybírány jen z těch, které mají lineární časovou složitost. S každým z výše uvedených binárních CA je vytvořen 1 KeyExtenderSimpleLinear a 6 různých KeyExtenderInterlaced.

Třída GoodPrimitives funguje jinak. Nejprve jsou vygenerována data o úspěšných natahovačích a použitých automatech během četných běhů genetického algoritmu s využitím třídy Primitives. Za tímto účelem byl mnohokrát spuštěn genetický algoritmus na náhodné klíče o velikosti 100, které natahoval na velikost 25000. Z každého běhu genetického algoritmu byla uložena ta vítězná sekvence do souboru. Celkem ten experiment běžel přes 100 hodin. Teď je možné vytvořit instanci třídy GoodPrimitives, která načte tyto natahovače s jejich automaty z určené složky a pak poskytuje právě takto získané natahovače.

# Závěr

<http://mathoverflow.net/questions/128903/expected-edit-distance?rq=1>  
[http://math.stackexchange.com/questions/375505/what-is-the-average-levenshtein-distance-  
between-two-random-binary-strings-of-le](http://math.stackexchange.com/questions/375505/what-is-the-average-levenshtein-distance-between-two-random-binary-strings-of-le)

# Seznam obrázků

1.1	Elementární automat číslo 220 . . . . .	4
1.2	Elementární automat číslo 94 . . . . .	4
1.3	Elementární automat číslo 90 . . . . .	5
1.4	Elementární automat číslo 30 . . . . .	5
1.5	Elementární automat číslo 110 . . . . .	6
2.1	Ukázkový radar chart . . . . .	11
4.1	KeyExtenderCopy . . . . .	14
4.2	KeyExtenderCheating . . . . .	14
4.3	KeyExtenderSimpleLinear na automat 220 . . . . .	15
4.4	KeyExtenderSimpleLinear na automat 94 . . . . .	15
4.5	KeyExtenderSimpleLinear na automat 90 . . . . .	16
4.6	KeyExtenderSimpleLinear na automat 30 . . . . .	16
4.7	KeyExtenderSimpleLinear na automat 110 . . . . .	17
4.8	KeyExtenderSimpleLinear na omezený automat s 2-okolím . . . . .	17
4.9	KeyExtenderSimpleLinear na cyklický automat s 2-okolím . . . . .	18
4.10	KeyExtenderSimpleLinear na Amoeba Universe . . . . .	18
4.11	KeyExtenderSimpleQuadratic na automat 220 . . . . .	19
4.12	KeyExtenderSimpleQuadratic na automat 94 . . . . .	19
4.13	KeyExtenderSimpleQuadratic na automat 90 . . . . .	20
4.14	KeyExtenderSimpleQuadratic na automat 30 . . . . .	20
4.15	KeyExtenderSimpleQuadratic na automat 110 . . . . .	21
4.16	KeyExtenderSimpleQuadratic na omezený automat s 2-okolím . . . . .	21
4.17	KeyExtenderSimpleQuadratic na cyklický automat s 2-okolím . . . . .	22
4.18	KeyExtenderInterlaced(10, 0) na automat 220 . . . . .	23
4.19	KeyExtenderInterlaced(10, 0) na automat 94 . . . . .	23
4.20	KeyExtenderInterlaced(10, 0) na automat 90 . . . . .	24
4.21	KeyExtenderInterlaced(10, 0) na automat 30 . . . . .	24
4.22	KeyExtenderInterlaced(10, 0) na automat 110 . . . . .	25
4.23	KeyExtenderInterlaced(10, 0) na omezený automat s 2-okolím . . . . .	25
4.24	KeyExtenderInterlaced(10, 0) na cyklický automat s 2-okolím . . . . .	26
4.25	KeyExtenderInterlaced(10, 0) na Amoeba Universe . . . . .	26
4.26	KeyExtenderUncertain na automat 220 . . . . .	27
4.27	KeyExtenderUncertain na automat 94 . . . . .	28
4.28	KeyExtenderUncertain na automat 90 . . . . .	28
4.29	KeyExtenderUncertain na automat 30 . . . . .	29
4.30	KeyExtenderUncertain na automat 110 . . . . .	29
4.31	KeyExtenderUncertain na omezený automat s 2-okolím . . . . .	30
4.32	KeyExtenderUncertain na cyklický automat s 2-okolím . . . . .	30

# Seznam tabulek

# Seznam použitých zkratek



# Přílohy