VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

GENEROVÁNÍ PROTIPŘÍKLADŮ PŘI ANALÝZE MARKOVOVÝCH MODELŮ

COUNTER-EXAMPLE GENERATION IN THE ANALYSIS OF MARKOV MODELS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MARTIN MOLEK**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. MILAN ČEŠKA, Ph.D.**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Molek Martin**
Obor: Informační technologie
Téma: **Generování protipříkladů při analýze Markovových modelů**
Counter-Example Generation in the Analysis of Markov Models
Kategorie: Formální verifikace

Pokyny:

1. Seznamte se s problematikou generování protipříkladů v kontextu formální analýzy a verifikace Markovových modelů.
2. Nastudujte existující metody pro generování protipříkladů a jejich implementace v nástrojích PRISM a STORM.
3. V rámci jedno z těchto nástrojů implementujte generování protipříkladů pomocí metody hledání k-nejkratších cest.
4. Experimentálně ověřte funkcionalitu a praktickou užitečnost implementovaných metod na vhodné sadě modelů. Zaměřte se rovněž na efektivitu implementace v kontextu velikosti verifikovaných modelů.

Literatura:

- E. Abraham et al. "Counterexample Generation for Discrete-Time Markov Models: An Introductory Survey". In *SFM'14,* LNCS, pages 65-121, Springer, 2014
- M. Kwiatkowska, G. Norman and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *CAV'11*, LNCS, pages 585-591, Springer, 2011.
- Ch. Dehnert, S. Junges, J.P. Katoen, and M. Volk. A Storm is Coming: A Modern Probabilistic Model Checker. In *CAV'17*, LNCS, pages 592-600, Springer, 2017.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body ze zadání a funkční implementace základních metod pro generování protipříkladů.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Češka Milan, RNDr., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Tato práce se zabývá generováním protipříkladů v kontextu verifikace pravděpodobnostních systémů. Protipříklady jsou generovány nad Markovovými modely (přesněji DTMC). Specifikace vlastností modelu jsou zadávány pomocí logiky PCTL, která je v této práci popsána. Pro generování protipříkladů byly použity dva různé algoritmy (*Best-first search* a *Recursive Enumration Algorithm*). Práce obsahuje popis implementace algoritmů do verifikačního nástroje *STORM*. Výsledky experimentů ukazují, že REA je schopen pracovat s modely obsahující miliony stavů.

Abstract

This thesis deals with generating counterexamples in context of probabilistic systems. Counterexamples are generated for Markov models (specifically DTMC). Definitions of model properties are given by logic PCTL. Two algorithms (*Best-first search* and *Recursive Enumration Algorithm*) are used to generate these counterexamples. Thesis describes implementation of algorithms into verification tool *STORM*. The results of experiments show that REA is capable of handling models containg millions of states.

Klíčová slova

formální verifikace, protipříklady, Markovovy modely, pravděpodobnostní modely, DTMC, k-nejkratších cest, pravděpodobnostní temporální logika, STORM

Keywords

formal verification, counterexamples, Markov chains, probabilistic models, DTMC, k-shortest paths, Probabilistic Computation Tree Logic, STORM

Citace

MOLEK, Martin. *Generování protipříkladů při analýze Markovových modelů*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Milan Češka, Ph.D.

Generování protipříkladů při analýze Markovových modelů

Prohlášení

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

.....
Martin Molek
16. května 2018

Poděkování

Rád bych poděkoval svému vedoucímu Milanovi Češkovi za odborné vedení a rady při vypracování této práce. Dále bych chtěl poděkovat Christianu Denhertovi, vývojáři programu STORM, za ochotné zodpovídání mých dotazů.

Obsah

Kapitola 1

Úvod

Příval nových a vylepšování stávajících technologií neodmyslitelně patří k trendu posledních let. Firmy se navzájem popohánějí k rychlému pokroku a často nezbývá dostatek času na důkladné ověření správné funkčnosti nových produktů. Produkty pak mohou obsahovat chyby, ať už softwarové nebo hardwarové [11]. Pomoci této situaci se snaží techniky z oboru *formální verifikace*. Namísto testování produktu lidmi je produkt převeden na *formální model*. Ve chvíli, kdy je model sestaven, s ním může začít pracovat počítač, který dokáže provádět kontrolu specifikovaných vlastností mnohem rychleji než lidé. Role člověka se tedy přesouvá z testera na tvůrce testů.

Tento proces však skýtá několik problémů, které je ještě třeba vyřešit. Prvním z nich je převod systému na model. Existuje několik možností, jak může model vypadat. My budeme pracovat s modely označovanými jako *Markovovy modely*, přesněji podskupinou *Discrete-Time Markov Chains* (DTMC). Tyto modely se skládají z množství stavů/uzlů, které jsou vzájemně propojeny. Díky propojením se můžeme mezi stavy pohybovat a zkoumat, jak často, případně jakým způsobem se do příslušných stavů dostat.

Při formální verifikaci musíme i námi zkoumané vlastnosti popisovat formálně. *Probabilistic Computation Tree Logic* (PCTL) [8] nám poskytuje syntaxi a sémantiku formulí, pomocí kterých jsme schopni jednoznačně sepsat požadavky, které by měl model splňovat.

Po specifikování vlastností přijde vyhodnocení počítačem a výpis výsledků. V jakém formátu jsou nám výsledky poskytnuty? Záleží na vlastnosti, kterou zkoumáme – výsledek může být typu splňuje/nesplňuje, nebo vyjádřen číselně, například splňuje na 73 %. Tato práce se zabývá třetí možností vyjádření výsledku, a sice vyjádřením pomocí *protipříkladu* [1].

Úlohou protipříkladů je poskytnout přehled o tom, které části modelu přispívají k tomu, aby byla porušena některá z očekávaných vlastností. Protipříklad není jednoduchý kvantitativní výsledek, ale rozsáhlejší informace v podobě množiny cest. O Markovových modelech, PCTL a protipříkladech pojednává teoretická kapitola 2.

I když jsou dnešní počítače velmi výkonné, musíme vytvářet verifikační programy, které používají rychlé algoritmy, protože velikost modelů může dosáhnout řádu milionů stavů. Pro vytváření protipříkladů byly pro tuto práci vybrány algoritmy *Best-first search* (BFS) [14] a *Recursive Enumeration Algorithm* (REA) [10]. Tyto algoritmy nebyly vytvořeny za účelem hledání protipříkladů, avšak dá se jejich pomocí řešit problém hledání *k-nejkratších cest*, který je převeditelný na hledání protipříkladů. Tento převod a popis algoritmů je podrobně popsán v kapitole 3.

Praktická část této práce ze zabývá implementací zmíněných algoritmů do již existujícího a stále se vyvíjejícího nástroje *STORM* [4], jehož mottem je *A Storm is coming*. Toto motto

má značit příchod nového, moderního a efektivního nástroje pro analýzu modelů. Rozbor výsledků implementace je v kapitole 5. Na několika experimentech je zde ukázáno, jak různé parametry modelu ovlivňují čas potřebný k nalezení protipříkladu.

Kapitola 2

Teorie

Nebylo by vhodné začít tuto kapitolu algoritmy na generování protipříkladů v Markovových modelech bez uvedení základních pojmů spojených s touto problematikou. V této kapitole si postupně projdeme teoretické základy, počínaje Markovovými modely a termíny, které jsou s nimi spojeny. Dále přejdeme ke způsobu zápisu, kterým můžeme Markovovy modely reprezentovat, a ukážeme, co o modelech můžeme zjišťovat pomocí jazyka PCTL. Poté už si budeme moci definovat protipříklad v kontextu Markovových modelů a zadefinovat vlastnosti, které bude zapotřebí brát v úvahu po zbytek této práce.

2.1 Markovovy modely

Obecný model vyjadřuje s určitou mírou abstrakce nějaký konkrétní systém. Příkladem obecného modelu může být autoatlas, který převádí (konkrétní) silnice na čáry na papíře. V této práci se budeme věnovat modelům, jejichž původní systém zahrnuje neurčitost či náhodné jevy. Může se jednat o modely her, kde se hází kostkou, programy, kde počítač generuje náhodné číslo, nebo chování lidí v supermarketu.

Jedním ze způsobů reprezentace takovýchto modelů jsou Markovovy modely, které jsou pojmenovány po ruském matematikovi Andreji Markovovi. Pro tyto modely platí, že z každého stavu lze přejít s určitou pravděpodobností do jednoho nebo více dalších stavů. Stav, do kterého se přechází, může být shodný se stavem, ze kterého se vychází. Musí však být dodržena podmínka, která stanovuje, že součet pravděpodobností přechodů z každého stavu je roven jedné.

Další vlastností, která musí být splněna, je Markovova vlastnost (*Markov property*). Tato vlastnost vyžaduje, aby rozhodnutí učiněná v každém stavu byla založena pouze na tom, o jaký stav se jedná. Jinými slovy není povoleno, aby rozhodnutí záviselo na událostech předcházejících danému stavu.

V této práci se budeme zabývat modely, ve kterých události nastávají v diskrétním čase, tedy v jednotlivých krocích. Takové modely označujeme pojmem Discrete-Time Markov Chains (DTMC). Jejich protipólem jsou Markovovy modely pracující se spojitým časem – Continuous-Time Markov Chains (CTMC). Tyto modely popisují systémy, v nichž mohou události nastávat během spojitých časových intervalů.

Definice 1 (Discrete-Time Markov Chains)**.** DTMC je trojice $\mathcal{D} = (S, P, L)$, ve které:

- S je množina stavů,
- $P = S \times S \to [0; 1]$ je matice přechodů mezi stavy,
- $L = S \to 2^{AP}$ je funkce přiřazující vlastnosti jednotlivým stavům (*labeling*). AP je množina atomických propozic.

Slovně je DTMC množina stavů S, mezi kterými jsou vzájemné přechody. Tyto přechody jsou zaneseny do matice P o velikosti $S \times S$. Součet přechodů vycházejících z daného stavu, čili součet všech hodnot na řádku matice, je vždy roven jedné. L umožňuje pojmenovávat několik stavů jedním termínem.[7]

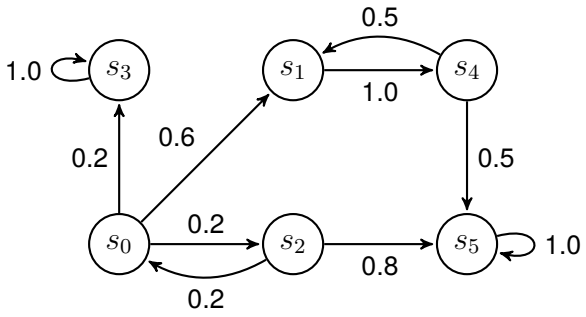
Příklad 1 (Model hry v DTMC)**.** Uvažme jednoduchou hru, ve které hráč hodí dvěma kostkami. Pokud padnou dvě stejná čísla nebo součet čtyři, hráč vyhrává. V opačném případě hráč prohrává. Model takové hry by mohl obsahovat jeden stav, ve kterém hra začíná, a z něhož by vycházely přechody do 36 stavů. Hodnota 36 předpokládá, že jsou kostky rozlišitelné (kombinace 31 a 13 jsou různé stavy). Hru by bylo možné modelovat i sloučením těchto stavů, záleží, jakou míru abstrakce modelu zvolíme. Další dva stavy by byly zapotřebí pro reprezentaci výherní a nevýherní situace. Těchto 39 stavů by tvořilo množinu S.

V přechodové matici tohoto modelu je 36 přechodů s pravděpodobnostní 1/36, dále pak 36 „jedničkových“ přechodů z hodů do výhry, nebo případně prohry. Další dva přechody musí jít z výhry a prohry, aby byla dodržena vlastnost, že součet pravděpodobností vycházejících přechodů je roven jedné. Pokud tyto přechody povedou na začátek hry, znamená to, že hra může mít neomezeně kol. Pokud povedou samy do sebe, znamená to, že hra po jednom hodu končí.

Atomických propozic můžeme najít mnoho. Zajímavá je v tomto případě propozice *dvojice*, která zahrnuje množinu stavů se stejnými čísly na obou kostkách (11, 22, 33, ...). Druhou propozicí je propozice *čtyři*, která zahrnuje množinu stavů o součtu čtyři (13, 22, 31). Užitečná by mohla být i propozice *konec* zahrnující stav výhry a prohry.

I když je možné popisovat Markovovy modely pomocí této trojice, pro názornou demonstraci se velmi často používají grafy. V grafech je množina stavů reprezentována uzly s popisem stavu a přechodová funkce je reprezentována orientovanými hranami mezi uzly.

Příklad 2 (Discrete-Time Markov Chain reprezentovaný grafem)**.** Přechody vycházející z každého stavu mají vždy celkový součet jedna.



Obrázek 2.1: DTMC

První užitečnou strukturou při práci s Markovovými modely je cesta. Cestou označujeme řadu přechodů mezi stavy. Na rozdíl od cest v teorii grafů je za cestu považována i taková řada přechodů, v níž se stejný přechod vyskytuje vícekrát.

Definice 2 (Cesta)**.** Cesta σ je posloupnost přechodů mezi stavy $s_0 \cdot s_1 \cdots s_i$, přičemž $\forall i > 0 : P(s_{i-1}, s_i) > 0 \wedge s_i \in S$. Délka této cesty $|\sigma|$ je rovna počtu přechodů na této cestě, tedy $|\sigma| = i - 1$. Pravděpodobnost cesty je rovna součinu přechodů $P(\sigma) = \prod_{0<i} P(s_{i-1}, s_i)$.

Příklad 3 (Cesty)**.** Pro demonstraci si uveďme tři cesty v DTMC z 2.1.

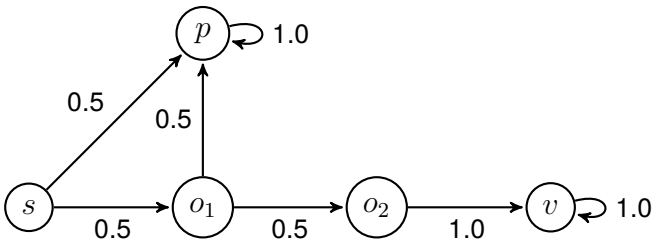
$\sigma_0 = s_1 \cdot s_4 \cdot s_5$	$\lvert\sigma_0\rvert = 2, P(\sigma_0) = 0.5$
$\sigma_1 = s_2 \cdot s_0 \cdot s_1 \cdot s_4 \cdot s_1$	$\lvert\sigma_1\rvert = 4, P(\sigma_1) = 0.06$
$\sigma_2 = s_0 \cdot s_3 \cdot s_3$	$\lvert\sigma_2\rvert = 2, P(\sigma_2) = 0.2$
$\sigma_x = s_0 \cdot s_3 \cdot s_2$	není cesta, protože $P(s_3, s_2) = 0$

Definice 3 (Markovova vlastnost)**.** Cesta nám umožňuje jednoduše definovat Markovovu vlastnost. Platí, že pravděpodobnost přechodu na další stav nezáleží na předchozích stavech cesty.

$$\forall \sigma_a, \sigma_b : \frac{P(\sigma_a \cdot u \cdot v)}{P(\sigma_a \cdot u)} = \frac{P(\sigma_b \cdot u \cdot v)}{P(\sigma_b \cdot u)} = P(u, v)$$

Příklad 4 (Rozbor Markovovy vlastnosti)**.** Mohlo by se zdát, že Markovova vlastnost říká, že v momentě, kdy se dostaneme do nějakého stavu, nejsme schopni určit, jaké události tomuto stavu předcházely. Při správné reprezentaci procesu jsme však schopni některé důležité informace o „historii“ stavu zjistit.

Jako příklad uvažme jednoduchou hru s mincí, v níž má hráč dva hody. Hráč vítězí, pokud mu padne dvakrát orel. Ve vítězném stavu v tedy víme, že mu předcházel počáteční stav s a dva hody orla, tedy přechody přes stavy o_1 a o_2.



Obrázek 2.2: Hra s mincí

2.2 Probabilistic Computation Tree Logic

V úvodu bylo naznačeno, že vlastnost, kterou budeme chtít zkoumat, je způsob, jakým se model dostal z jednoho stavu do jiného. Ke správnému vyjádření našeho problému můžeme využít syntaxe PCTL. [8]

Formule v PCTL je generována následující gramatikou:

$$\phi ::== \Phi U^{\leq h} \Phi \mid \Phi W^{\leq h} \Phi$$

$$\Phi ::== true \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid P_{\geq p}(\Phi)$$

První případ $\Phi U^{\leq h}\Psi$ připouští cesty, kde je Φ splněno v každém kroku cesty, před dosažením stavu/stavů Ψ (Φ *Until* Ψ). Maximální počet kroků (hops) omezuje hranice $h \in \mathbb{N}_0^+$. Pokud je $h = \infty$, můžeme toto omezení vynechat. V tom případě se jedná o formuli bez ohraničení (unbounded until-formula) a tento typ formulí bude přijímat má implementace. Symbol $\trianglerighteq$ zastupuje množinu operátorů $\{<, \leq, \geq, >\}$ a $p \in <0; 1>$ je pravděpodobnostní hranice události.

Druhá formule $\Phi W^{\leq h}\Psi$ připouští případy, kde Φ musí platit dokud nenastane Ψ, nebo musí Φ platit do nekonečna. Toto omezení je slabší (*Weak*) než původní požadavek U, protože nám dává možnost nikdy nedosáhnout stavu Φ a přesto formuli vyhovět. Vzhledem k tomu, že má implementace přijímá pouze until formule $\Phi U\Psi$, je třeba poukázat na fakt, že formule jsou mezi sebou navzájem převeditelné. Konkrétně:

$$P_{\geq p}(\Phi\ W\ \Psi) \equiv P_{\leq 1-p}\big((\Phi\ \wedge\ \neg\Psi)\ U\ (\neg\Phi\ \wedge\ \neg\Psi)\big)$$
$$P_{\geq p}(\Phi\ U\ \Psi) \equiv P_{\leq 1-p}\big((\Phi\ \wedge\ \neg\Psi)\ W\ (\neg\Phi\ \wedge\ \neg\Psi)\big)$$

Sémantika PCTL je vyjádřena pomocí splnitelnosti.

$$\begin{aligned}
&s \models true && \\
&s \models a && \iff a \in L(s) \\
&s \models \neg\Phi && \iff not(s \models \Phi) \\
&s \models \Phi \wedge \Psi && \iff s \models \Phi \text{ and } s \models \psi \\
&s \models P_{\trianglerighteq p}(\Phi) && \iff Probability(s, \Phi) \trianglerighteq p
\end{aligned}$$

Poslední rovnice zavádí do našeho modelu počáteční stav s.[1] Často se stává, že chceme hledat vlastnosti určitého stavu a nezajímá nás, jaké stavy této vlastnosti předchází. PCTL formule se zapisuje jako $P_{\trianglerighteq p}(true\ U\ \Phi)$, pro zjednodušení se často používá $P_{\trianglerighteq p}(F\ \Phi)$.

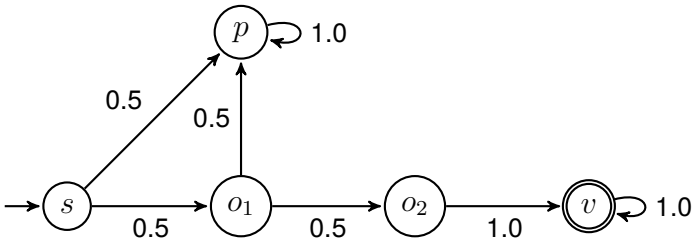
Příklad 5 (Použití PCTL)**.** Pokud bychom chtěli na modelu 2.4 zkoumat vlastnost, dosažitelnosti s_5 pouze po stavech s atomickou propozicí *sudé* a pravděpodobností alespoň 0.175, dostali bychom formuli $P_{>0.175}(sudé\ U\ s_5)$.

Příklad 6 (Zajímavá dualita v PCTL)**.** V práci [7] je zmíněna zajímavá vlastnost PCTL: $P_{>0.1}(true\ U\ \Phi) \equiv P_{<0.9}(\neg\Phi\ W\ \neg true)$. Vztah vyjadřuje na levé straně vlastnost, že s pravděpodobností 0.1 se dá dostat do Φ. Na straně pravé říká, že cesty nekončící v Φ mají pravděpodobnost menší než 0.9.

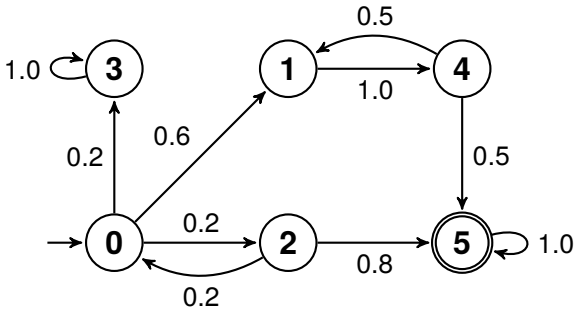
2.3 Grafické znázornění modelů

Pokud se podíváme na model hry s mincí 2.2, zjistíme, že není jasné, kde hra začíná a kde končí. V této práci bude v DTMC grafech označen počáteční stav šipkou, která by mohla symbolizovat, že „do tohoto stavu vcházíme s pravděpodobností 1.0“ a koncové stavy dvoukruhem, podobně jako přijímající (*accepting*) stavy v automatech. Počáteční stav může být vždy pouze jeden, koncových stavů (někdy označovaných jako terminální) může být několik. V takto značeném modelu 2.3 je snazší se zorientovat. Pro úplnost ještě převedeme první DTMC 2.1 na ukázkový model 2.4, který bude používán v několika dalších příkladech této práce.

[1] Někdy se uvádí DTMC jako čtveřice (S, s_0, P, L), kde s_0 značí počáteční stav. My můžeme díky PCTL hovořit o DTMC jakožto trojici.



Obrázek 2.3: Hra s mincí s označeným počátečním a koncovým stavem



Obrázek 2.4: Ukázkový model

2.4 Analýza Markovových modelů

U Markovových modelů lze zkoumat několik vlastností. Zaprvé můžeme zkoumat, jestli je vůbec možné určitého stavu v modelu dosáhnout. Algoritmus pro vyřešení tohoto problému je velmi jednoduchý a podobá se Dijkstrově algoritmu uvedenému v sekci 3.2.1. Vzhledem ke způsobu, jakým jsou modely typicky generovány, však není existence nedosažitelného stavu pravděpodobná.

Mnohem zajímavější je pro nás zjišťovat, jaká bude distribuce pravděpodobností v modelu po určitém počtu kroků (jednotek času). Víme, že začínáme v počátečním stavu s pravděpodobnostní jedna. Po prvním kroku budou pravděpodobnosti odpovídat hodnotám přechodům z počátečního stavu (tedy prvnímu řádku přechodové matice). Pro další kroky stačí vynásobit počáteční vektor s přechodovou maticí umocněnou na počet kroků, který nás zajímá. Máme-li vektor π_0, který obsahuje distribuci pravděpodobnosti ve stavu modelu a přechodovou matici P, bude po n krocích platit, že distribuce pravděpodobností π_n v modelu bude

$$\pi_n = \pi_0 \cdot P^n$$

Příklad 7 (Pravděpodobnost po n krocích)**.** Vezmeme-li přechodovou matici modelu 2.4 a budeme se zajímat o pravděpodobnost v jednotlivých stavech po pěti krocích, bude výpočet vypadat následovně:

$$\begin{pmatrix}1 & 0 & 0 & 0 & 0 & 0\end{pmatrix} \cdot \begin{pmatrix} 0 & 0.6 & 0.2 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0.2 & 0 & 0 & 0 & 0 & 0.8 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0.5 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}^5 = \begin{pmatrix}0 & 0.16 & 0.00 & 0.21 & 0 & 0.63\end{pmatrix}$$

V počátečním stavu budeme s nulovou pravděpodobností, ve stavu s_1 s pravděpodobností 0.16, atd.

Pokud by nás zajímalo, k jakým pravděpodobnostem jednotlivé stavy v DTMC konvergují, museli bychom vyřešit N rovnic o N neznámých (zadání těchto rovnic by vzešlo z rovnice $\pi = \pi \times P$). Gaussova eliminační metoda by sice vyřešila naši malou matici ihned, avšak pro velké matice (tisíce stavů) se používají iterativní metody. Pro náš model bychom se dobrali k tomu, že ve stavu s_3 bychom skončili s pravděpodobností $0,208\overline{3}$ a ve stavu s_5 s pravděpodobností rovnou zbytku do jedničky[2]. V ostatních stavech bychom nikdy neskončili.

Můžeme si všimnout, že ze stavu s_3 a z konečného stavu s_5 nevedou přechody do žádného jiného stavu. Takovéto stavy nazýváme *absorpční stavy*. Pravděpodobnost dosažení těchto stavů v průběhu času pouze roste.

2.5 Protipříklady v Markovových modelech

Než se pustíme do matematického popisu protipříkladu, zkusme si naznačit, co bychom intuitivně od tohoto termínu očekávali. V sekci 2.2 o PCTL jsme si popsali způsob, jakým formulovat nějaké vlastnosti modelu. Cílem protipříkladu je tuto vlastnost vyvrátit – uvést příklad, kde vlastnost není splněna.

Běžným scénářem při verifikaci modelu by mohlo být, že si uživatel nechá vypsat pravděpodobnost, že se model dostane do stavu t. Jestliže se mu zobrazí očekávaná hodnota, může s analýzou modelu skončit. Pokud je však hodnota vyšší, než očekával (například 15% vyrobených mobilů nebude fungovat), bude chtít vědět, co je příčinou. Po programu pak bude vyžadovat protipříklad jeho původní domněnky, že pouze tři procenta mobilů budou porouchaná.

2.5.1 Evidence a protipříklad

Cesty, které budeme v rámci protipříkladů hledat, budou splňovat vlastnost, že začínají v počátečním stavu s a končí v některém z koncových stavů t. Pomocí PCTL jsou takové cesty popsány vztahem $P_{\geq p}(true\ U\ t)$. Při hledání protipříkladů jsou tyto cesty velmi důležité, a proto mají vlastní pojmenování – evidence (anglicky *evidences*, což by do češtiny mohlo být přeloženo, také jako náznak či záznam). Protipříklad je takové množství evidencí, které je dostatečné k naplnění pravděpodobnosti p. [16]

Definice 4 (Evidence)**.** Evidence je konečná cesta z počátečního stavu s do některého ze stavů v t.

Definice 5 (Nejsilnější evidence)**.** Evidence σ, o které platí, že neexistuje evidence σ', která by měla vyšší pravděpodobnost, se nazývá nejsilnější evidence. $P(\sigma) \geq P(\sigma')$

Definice 6 (Protipříklad)**.** Množina C evidencí $\sigma_0, \sigma_1, \ldots, \sigma_n$, pro kterou platí, že pravděpodobnost evidencí překročí stanovenou pravděpodobnostní hranici p, tvoří protipříklad. $\sum_{0}^{n} P(\sigma_n) > p$

[2] V modelu 1 můžeme pravděpodobnost spočítat vzorcem na geometrickou posloupnost – $P(s_5) = \frac{0.2}{1-0.04}$

2.5.2 Minimální a nejmenší protipříklad

Protipříkladů s námi požadovanými vlastnostmi může být několik. Abychom z nich vybrali ty, které jsou nejvýstižnější, definujeme termíny minimální a nejmenší protipříklad.

Za minimální protipříklad považujeme ten, který obsahuje nejméně evidencí. Nejmenší protipříklad je minimální a zároveň má největší součet pravděpodobností evidencí, tedy nejvíce přesahuje námi stanovenou pravděpodobnostní hranici. Právě generováním nejmenších protipříkladů se bude zabývat tato práce.

Definice 7 (Minimální protipříklad)**.** Minimální protipříklad C obsahuje stejně nebo méně evidencí, než jakýkoli jiný protipříklad C', platí tedy $|C| \leq |C'|$.

Definice 8 (Nejmenší protipříklad)**.** Nejmenší protipříklad C je minimální a zároveň platí $P(C) \geq P(C')$.

Vzhledem k tomu, že nepoužíváme ostré nerovnosti, může se stát, že nejmenších protipříkladů bude existovat několik. Tato situace typicky nastává v symetrických modelech.

2.5.3 Existence protipříkladu

Při hledání protipříkladů mohou nastat tři situace.

- Existuje konečný protipříklad.
- Existuje nekonečný protipříklad, který vzniká asymptotickým přibližováním k námi stanovené hranici. Jednoduchým příkladem této situace je model 2.5.
- Protipříklad neexistuje. Tato situace může nastat, pokud je v modelu absorpční stav, který není konečný. Existenci protipříkladu můžeme ověřit dotázáním příslušnou formulí pro získání celkové pravděpodobnosti našich koncových stavů.

Demonstrujme si nyní použití pojmů z této a předchozí sekce. Uvažujme operátor $\xrightarrow{CE}$, který značí, že formule na levé straně generuje protipříklad na pravé straně. Pro zjednodušení na chvíli řekněme, že $C_p \equiv P_{\geq p}(true\ U\ s_5)$ a hledejme protipříklady v ukázkovém modelu 2.4. Budeme hledat evidence z počátečního stavu skrz libovolné stavy do stavu s_5 za dosažení pravděpodobnosti alespoň p.

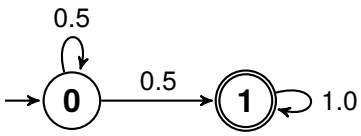
Příklad 8 (Obecný protipříklad)**.** $C_{0.2} \xrightarrow{CE} \{s_0 \cdot s_1 \cdot s_4 \cdot s_5,\ s_0 \cdot s_1 \cdot s_4 \cdot s_1 \cdot s_4 \cdot s_5\}$

Příklad 9 (Minimální protipříklad)**.** $C_{0.4} \xrightarrow{CE} \{s_0 \cdot s_1 \cdot s_4 \cdot s_5,\ s_0 \cdot s_1 \cdot s_4 \cdot s_1 \cdot s_4 \cdot s_5\}$, $|C| = 2$, $P(C) = 0.3 + 0.15 = 0.45$

Příklad 10 (Nejmenší protipříklad)**.** $C_{0.4} \xrightarrow{CE} \{s_0 \cdot s_1 \cdot s_4 \cdot s_5,\ s_0 \cdot s_2 \cdot s_5\}$, $|C| = 2$, $P(C) = 0.3 + 0.16 = 0.46$, což dokazuje, že příklad 9 není nejmenší.

Příklad 11 (Neexistující protipříklad)**.** $C_{0.9} \xrightarrow{CE} \emptyset$, řešení neexistuje vzhledem k tomu, že stav 3 „pohltí“ více než 0.2 pravděpodobnosti.

Příklad 12 (Nekonečný protipříklad)**.** Nekonečný protipříklad lze z ukázkového modelu 2.4 vygenerovat, pokud se budeme ptát na $p \geq 0.8$. Názornější protipříklad, kde $p \geq 1.0$ lze ukázat na modelu 2.5. $C_{p \geq 1.0} \xrightarrow{CE} \{s_0 \cdot s_1,\ s_0 \cdot s_0 \cdot s_1,\ s_0 \cdot s_0 \cdot s_0 \cdot s_1, \ldots\}$, $|C| = \infty$.



Obrázek 2.5: Model, který může generovat nekonečný protipříklad

2.5.4 Velikost protipříkladu

Protipříklad může být nekonečný, avšak i konečné řešení může obsahovat tisíce evidencí o tisících přechodech. Omezit shora velikost protipříkladu lze pouze u modelů, které neobsahují smyčky. Přesná hodnota maximálního počtu evidencí v modelu o N stavech bez smyček je 2^{N-2}. Počet přechodů v těchto evidencích beze smyček může dosáhnout až $N \times 2^{N-3}$.

Kapitola 3

Algoritmy na vyhledávání protipříkladů

Přestože je hledání protipříkladů problém relativně nový, dá se převést na problém hledání k-nejkratších cest[1] v grafu. V algoritmech je pouze třeba zaměnit sčítání přechodů za násobení a snažit se hledat co nejvyšší (tedy nejpravděpodobnější) řešení.

Problém hledání nejpravděpodobnějších cest je ekvivalentní problému hledání nejkratších cest, pokud zlogaritmujeme hodnoty přechodů mezi stavy. Výslednou pravděpodobnost lze v tomto případě získat umocněním základu logaritmu na délku cesty. Ekvivalence těchto dvou přístupů vychází z vlastnosti sčítání logaritmů.

$$\begin{aligned} \log_b(ac) &= \log_b a + \log_b c \\ ac &= b^{\log_b a + \log_b c} \\ \prod_{0<i} P(s_{i-1}, s_i) &= b^{\sum\limits_{0<i} \log P(s_{i-1}, s_i)} = P(\sigma) \end{aligned}$$

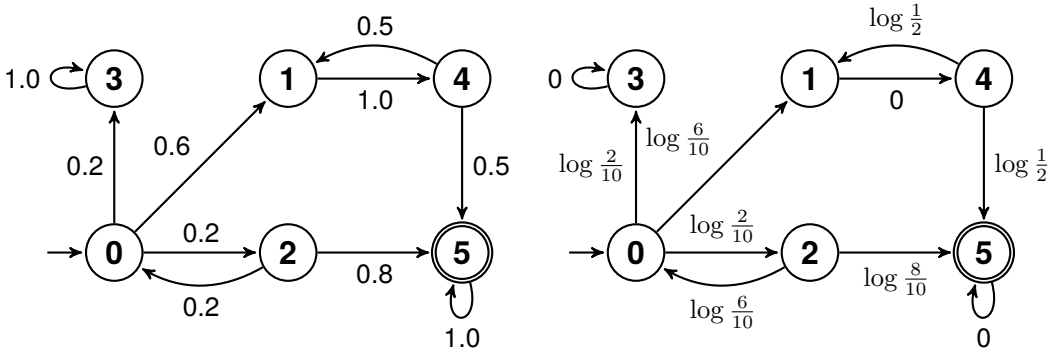
Příklad 13 (Rovnost součinů a sčítání logaritmů)**.** Výsledek bude stejný v obou modelech na obrázku 3.1.

$$0.6 \times 1.0 \times 0.5 = 0.3 = 10^{\log_{10} \frac{6}{10} \times \log_{10} 0 \times \log_{10} \frac{1}{2}}$$

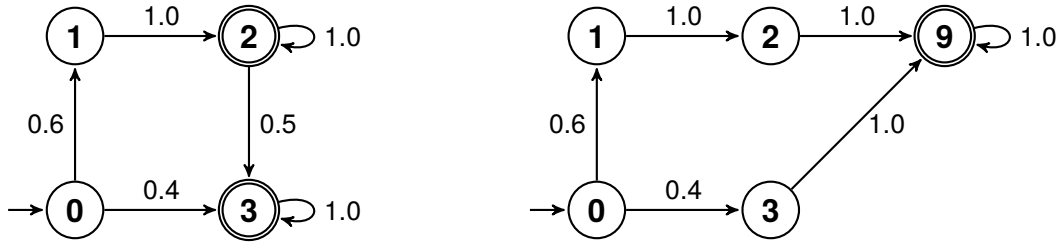
Běžné algoritmy na vyhledávání nejkratších cest očekávají jeden koncový stav, v Markovových modelech však může být koncových stavů více. Námi generovaný protipříklad tak může obsahovat evidence, které končí v několika různých stavech. Tento problém lze řešit přidáním virtuálního stavu, do kterého vedou hrany z terminálních stavů s přechodovou pravděpodobností jedna. Aby nedošlo k vygenerování cest, které vedou přes dva původně koncové stavy (a také v rámci zachování vlastností DTMC), je třeba při této změně odstranit všechny ostatní přechody vedoucí z terminálních stavů. Všem nalezeným cestám se poté musí oříznout virtuální koncový stav. Tento proces je vidět na obrázku 3.2.

Po převedení problému hledání protipříkladu na hledání nejkratší cesty je možné použít některý ze standardních algoritmů řešících problém hledání k-nejkratších cest. Podmínkou je, aby algoritmus fungoval na grafech bez smyček (*loopless*). Vhodnými kandidáty jsou

[1] Od tohoto bodu budeme evidencím říkat cesty, protože pro některé čtenáře je termín cesta zažitý a zároveň budeme několikrát hovořit o části prefixu evidence, což už nelze za evidenci považovat (zatímco prefix cesty je stále cesta).



Obrázek 3.1: Příklad převedení grafu s násobením pravděpodobností na graf se sčítáním logaritmů pravděpodobností



Obrázek 3.2: Převedení grafu s více koncovými stavy na graf s jedním koncovým stavem

Best-First Search (BFS), Recursive Enumeration Algorithm (REA), Eppsteinův algoritmus [6], nebo heuristický algoritmus od Husaina Aljazzara [2]. První dva zmíněné algoritmy byly v této práci implementovány a podrobně se jim věnují sekce 3.1 a 3.2 této kapitoly.

Pokud se budou cesty generovat od nejpravděpodobnějších, je zaručeno nalezení nejmenšího protipříkladu, protože sčítání nejvyšších hodnot převýší stanovenou pravděpodobnostní hranici s nejmenším počtem sčítanců.

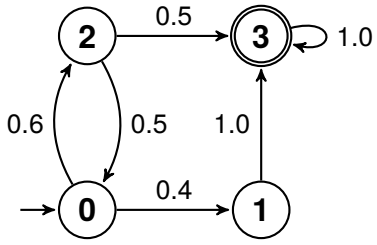
3.1 Best-first Search

Velmi jednoduchý algoritmus, který by mohl řešit problém k-nejkratších cest, je *Best-First Search algorithm*. V této práci je uveden proto, že demonstruje jeden ze špatných přístupů k řešení tohoto problému. Algoritmus sloužil v průběhu implementace jako kontrola správných řešení vzniklých z druhého implementovaného algoritmu.

Algoritmus začíná v počátečním stavu, z nějž vygeneruje cesty do všech svých následníků. Z vygenerovaných cest je vybrána ta nejlepší. Koncový stav nejlepší cesty je expandován, čímž vzniknou další cesty. Vlastnost *nejlepší* v tomto případě popisuje cestu s největší pravděpodobností. Po expanzi koncového stavu nejlepší cesty jsou známy cesty o délce jedna (z počátečního stavu) a cesty o délce dva (expandovaná původně nejlepší cesta). Algoritmus pokračuje až do chvíle, kdy by se měla expandovat cesta, jejíž poslední stav je koncový. V tomto případě je možné cestu vypsat jakožto k-tou nejkratší cestu. Cestu je třeba označit jako uzavřenou a dále ji neexpandovat. V případě dosažení celkové pravděpodobnostní hranice stanovené uživatelem může algoritmus skončit.

I když by se mohlo zdát, že algoritmus lze vylepšit tím, že by se cesta vypsala v momentě, kdy je expandováno do koncového stavu, není tomu tak. Byla by porušena vlastnost hledání

cest od nejpravděpodobnějších a nalezený protipříklad by nebyl nejmenší. Nejlépe je to vidět na modelu 3.3, kde by upravená varianta vygenerovala cestu $s_0 \cdot s_2 \cdot s_3$, i když správně je $s_0 \cdot s_1 \cdot s_3$.



Obrázek 3.3: Graf, kde by vylepšený BFS selhal

Problém, který může nastat, je zacyklení tohoto algoritmu, respektive nekonečné rozbalování jednoho stavu. Tento případ může nastat, pokud se v grafu vyskytne nekoncový absorpční stav ($P(s_i, s_i) = 1.0$). Řešením je uzavření cesty, pokud se do takového stavu dostane.

Příklad 14 (Příklad běhu BFS popsaného v této kapitole)**.** Vezmeme-li náš vzorový model 2.4 a budeme zkoumat vlastnost $P_{p>0.4}(true\ U\ \{5\})$, skončí výše popsaný algoritmus s následujícími dočasnými a vypsanými cestami.

$\sigma_0 = s_0 \cdot s_1 \cdot s_4 \cdot s_5$	$P = 0.30$, vypsáno, uzavřeno
$\sigma_1 = s_0 \cdot s_1 \cdot s_4 \cdot s_1 \cdot s_4 \cdot s_1$	$P = 0.15$
$\sigma_2 = s_0 \cdot s_1 \cdot s_4 \cdot s_1 \cdot s_4 \cdot s_5$	$P = 0.15$
$\sigma_3 = s_0 \cdot s_3$	$P = 0.20$, uzavřeno
$\sigma_4 = s_0 \cdot s_2 \cdot s_0$	$P = 0.04$
$\sigma_5 = s_0 \cdot s_2 \cdot s_5$	$P = 0.16$, vypsáno, uzavřeno

3.2 Recursive Enumeration Algorithm

Tento algoritmus byl představen v roce 1999 a jeho hlavním rozdílem oproti BFS je hledání cest od koncového stavu směrem k počátečnímu. Aby nebylo zbytečně vyhodnocováno příliš údajů, vytvoří si nejdříve algoritmus přehled o celém grafu.[10]

Než si detailněji popíšeme jednotlivé kroky algoritmu, uvádíme několik tvrzení, která jsou pro jeho správné fungování klíčová.[2]

- Nejpravděpodobnější cesta do libovolného stavu povede přes některý stav, který je jeho přímým předchůdcem.
- Nejpravděpodobnější cesta do libovolného stavu bude mít hodnotu nejpravděpodobnější cesty do svého předchůdce vynásobenou s přechodem do sebe sama.
- Druhá nejpravděpodobnější cesta do libovolného stavu povede skrz předchůdce. Bude se jednat o první nejpravděpodobnější cestu do předchůdce, který se nenachází na

[2]Popis algoritmu je změněn tak, aby odpovídal hledání nejpravděpodobnějších cest namísto nejkratších cest

první cestě, nebo druhou nejpravděpodobnější cestu, do předchůdce první nejpravděpodobnější cesty a následný přechod do sebe sama.

- K-tá nejpravděpodobnější cesta do libovolného stavu se bude vybírat z k'-tých nejkratších (a dosud nepoužitých) cest do předchůdců a následného přechodu do sebe sama.

Algoritmus je popsaný níže pseudokódem. Popisy některých funkcí, vlastností, či poznámky jsou uvedeny pod algoritmem s příslušným číslem řádku.

Algorithm 1 REA original pseudocode

```
1:  procedure GetPaths(threshold)
2:      for all state do
3:          GetFirstPath(state)
4:      k ← 1
5:      probability ← 0
6:      while probability < threshold do
7:          k ← k + 1
8:          NextPath(k; terminal)
9:      PrintPaths(terminal)

10: procedure NextPath(k; v)
11:     if k = 2 then
12:         C[v] ← {σ_1(u) · v : u ∈ Γ^{-1}(v) ∧ σ_1(v) ≠ σ_1(u) · v}
13:     if k = 2 and v = initial then
14:         goto evaluate
15:     u ← predecessor(σ_{k-1}(v))
16:     k' ← kof(σ_{k-1}(v)) + 1
17:     if not exist(σ_{k'}(u)) then
18:         NextPath(k'; u)
19:     if exist(σ_{k'}(u)) then
20:         C[v] ← C[v] ∪ σ_{k'}(u) · v
21:     if C[v] = ∅ and v = terminal then
22:         exit
23:     :evaluate
24:     if C[v] ≠ ∅ then
25:         σ_k(v) ← max(C[v])
26:         probability ← probability + prob(max(C[v]))
27:         C[v] ← C[v] ∖ max(C[v])
```

3: vysvětlení této procedury je v následující podsekci 3.2.1, řádky 2 a 3 jsou vyřešeny najednou, takže by bylo možné volat funkci GetFirstPaths
5: předpokládejme globální dostupnost proměnné *probability*, takže na řádku 26 se odkazujeme na tuto proměnnou
9: způsob tisku je řešen v kapitole 4
12: nepřidání první nejkratší cesty zajišťuje $\sigma_1(v) \neq \sigma_1(u) \cdot v$
12: funkce *kof* vrací informaci o tom, kolikátá nejkratší cesta vede do předposledního stavu cesty v parametru

17: *exist* kontroluje existenci příslušné cesty, protože cesta může být vypočtena v průběhu volání NextPath na jiný stav
22: protipříklad neexistuje
25: *max* vybere cestu s nejmenší pravděpodobností
26: *prob* zjistí pravděpodobnost cesty

3.2.1 One-to-All Dijkstra

Předchozí algoritmus 1 vyžaduje nalezení prvních nejkratších cest z počátečního stavu do všech ostatních stavů modelu. Tuto úlohu lze vyřešit použitím Dijkstrova algoritmu [5], přesněji variantou *One-to-All*. Tento algoritmus je velmi podobný algoritmu BFS popsanému výše. Jeho kroky po úpravě na pravděpodobnostní problém jsou následující:

Krok 0: Vytvoř pole $probability[v]$, které bude obsahovat hodnoty 0.
Vytvoř pole $predecessor[v]$, které bude nedefinované hodnoty.
Vytvoř seznam *states*, který bude obsahovat všechny stavy v.
Nastav hodnotu pravděpodobnosti $probability[source]$ na 1.

Dokud není seznam *nodes* prázdný, opakuj:

Krok 1: Vyber stav *node* z *nodes*, který má nejvyšší hodnotu $probability[state]$.
V prvním běhu tohoto cyklu bude $node = source$.

Krok 2: Následníkům *successor* stavu *state* nastav $predecessor[successor] = node$ a $probability[successor] = probability[state] \times P(node, successor)$, pokud je tato hodnota vyšší, než původní hodnota $probability[successor]$.

Krok 3: Odeber *state* ze *states*.

Po doběhnutí algoritmu se v polích $predecessor[v]$ a $probability[v]$ nachází na příslušném indexu informace o předchůdci tohoto stavu a pravděpodobnosti, s jakou lze ke stavu v dojít. Existuje několik modifikací a vylepšení tohoto algoritmu, jejichž podrobnější rozbor je k dispozici v kapitole 4.

K algoritmu BFS existuje také několik alternativ, v současné době je jedním z nejlepších algoritmů Thorup [15]. Za zmínku v tuto chvíli stojí fakt, že u udávání složitostí jednotlivých algoritmů se můžeme setkat nejen s počtem stavů V, počtem přechodů E, ale i maximální délkou L mezi zdrojovým a koncovým stavem.

Kapitola 4

Implementace

Mezi nejznámější programy pro ověřování modelů patří **PRISM** [12]. I když umožňuje analýzu DTMC modelů (a mnoha dalších) nejen pomocí logiky PCTL, možnost generování protipříkladů v něm chybí. Program je vyvíjen převážně v jazyce JAVA a má i grafické prostředí, které umožňuje vykreslení grafů pro různé vlastnosti v námi studovaných modelech.

Mnoho nástrojů na generování protipříkladů nezahrnuje další odvětví analýzy modelů a řada starších nástrojů není udržována. Příkladem může být třeba nástroj The COMICS Tool [9].

Aktivním zástupcem a relativním nováčkem v oblasti ověřování modelů je **STORM**, jehož vývoj započal v roce 2012. Jeho jádro je psané v programovacím jazyce C++. Protipříklady v jsou implementovány pro jiné typy modelů než DTMC a autoři by rádi do svůj program o podporu DTMC protipříkladů rozšířili. Mezi své klíčové vlastnosti řadí STORM efektivitu a modularit. Vzhledem k těmto faktům jsem se rozhodl psát svůj kód jakožto doplněk do STORMu. Tato kapitola se bude věnovat implementaci algoritmů popsaných v předchozí kapitole v prostředí STORM.

4.1 STORM

Než se pustíme do samotného popisu programu a datových struktur, popíšeme si přínosy, které prostředí STORM [4] poskytuje. Jak již bylo zmíněno v úvodu této kapitoly, STORM je psán v jazyce C++, přesněji ve standardu C++14 a opravdu využívá některých novinek uvedených v této verzi C++, například šablony proměnných (*Variable templates*). Jedná se o open source projekt, který v tuto chvíli (květen 2018) obsahuje přes 110 tisíc řádků zdrojového kódu a v průběhu jeho vývoje již bylo provedeno přes šest tisíc commitů do verzovacího systému Git. STORM je možné nainstalovat na většinu Linuxových distribucí, oficiální podpora je pro nové verze Ubuntu, Debianu a macOS. STORM je narozdíl od PRISMu čistě konzolová aplikace.

4.1.1 Vstupní modely

V teoretické sekci jsme si v definici 1 řekli, že DTMC je množina stavů a přechodů mezi jednotlivými stavy. Pro popis modelu vstupním souborem je zapotřebí využít předem stanovené syntaxe. STORM podporuje několik formátů/syntaxí, které lze použít. Mezi zástupce, které mají možnost reprezentovat DTMC, patří formát PRISM (typicky s příponou *.pm*), dále pak formát JANI a nebo lze přímo zadat soubor popisující jednotlivé přechody, takzvaný *explicitní zápis*.

Příklad 15 (Explicitní a PRISM zápis)**.** Soubor popisující přechody z modelu 2.4 v explicitním zápisu nalevo a PRISM zápisu napravo.

```
dtmc
0 1 0.6
0 2 0.2
0 3 0.2
1 4 1.0
2 0 0.2
2 5 0.8
3 3 1.0
4 1 0.5
4 5 0.5
5 5 1.0
```

```
dtmc
module mymodel
  s : [0..6] init 0;
  [] s=0 -> 0.6 : (s'=1) + 0.2 : (s'=2) + 0.2 : (s'=3);
  [] s=1 -> 1.0 : (s'=4);
  [] s=2 -> 0.2 : (s'=0) + 0.8 : (s'=5);
  [] s=3 -> 1.0 : (s'=3);
  [] s=4 -> 0.5 : (s'=1) + 0.5 : (s'=5);
  [] s=5 -> 1.0 : (s'=5);
endmodule
label "end" = s=5;
```

Při definování velkých modelů o několika tisíci stavech není rozumné používat styl zápisu, který je uveden v příkladu 15, ale je výhodnější využít všech syntaktických/sémantických možností tohoto jazyka. Vytvoření modelu o milionu stavech pak může být provedeno na několika řádcích.

4.1.2 Řídké matice

Po zpracování modelu neuchovává STORM přechodovou matici jako dvourozměrné pole, protože je to velmi neefektivní způsob z hlediska paměťové náročnosti. Velikost matice pro N stavů by měla velikost $N \times N$, což by při modelu s milionem stavů vyžadovalo paměť o velikosti v řádu terabajtů (pokud informace o jednom přechodu vyžaduje 64 bitů, bylo by pro $N = 10^6$ zapotřebí zhruba 8TB paměti). U DTMC modelů je typické, že z každého stavu vedou jednotky přechodů, proto by takto uložená matice obsahovala většinu nulových hodnot. I u našeho jednoduchého modelu 2.4 je nenulovými hodnotami vyplněná pouze necelá třetina matice, a pokud bychom vytvořili model hry z příkladu 1, bude matice obsahovat přes 97 % nulových hodnot.

Problém s kvadratickým růstem paměťové náročnosti řeší způsob ukládání CSR (*compressed sparse rows*). Pro tento způsob reprezentace je zapotřebí mít tři pole - datové, sloupcové a řádkové. V **datovém** poli jsou uloženy postupně (ve směru čtení textu v knize) hodnoty přechodů. Do **sloupcového** pole ukládáme index sloupce, kterému hodnota z datového pole náleží. Z toho vyplývá, že datové a sloupcové pole mají stejnou délku. **Řádkové** pole má délku N a zaznamenává indexy první hodnoty pro každý řádek matice. Redukce paměťové náročnosti tedy klesá z $N \times N$ na $2p + N$, kde p je počet přechodů. Výhodou tohoto způsobu uložení je rychlá přístupová doba pro všechny hodnoty na řádku, v našem případě tedy pro všechny přechody z daného stavu[1].

Příklad 16 (Model 2.4 uložený způsobem CSR)**.** Indexováno on nuly.

```
Data = [0.6, 0.2, 0.2, 1.0, 0.2, 0.8, 1.0, 0.5, 0.5, 1.0]
Cols = [1  , 2  , 3  , 4  , 0  , 5  , 3  , 4  , 4  , 5  ]
Rows = [0, 3, 4, 6, 7, 8]
```

[1] Samozřejmě je možná i metoda ukládání po sloupcích, která má naopak horší přístupovou dobu při získávání dat po řádcích

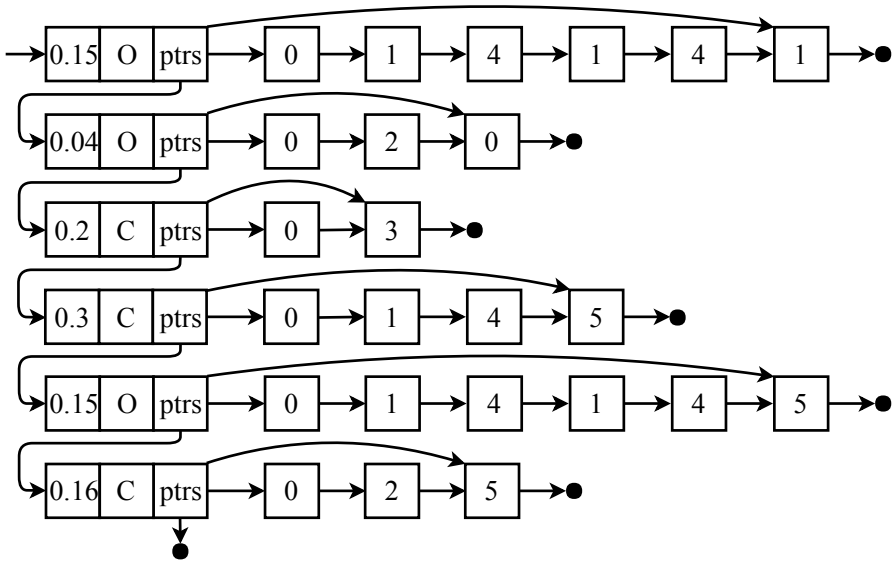
4.2 Implementace Best-First Search

Na úvod této sekce je třeba zmínit, že implementace tohoto algoritmu není optimální a pro zlepšení její efektivity by se dalo využít některých technik popsaných v následujících sekcích. I přesto by tento algoritmus nedosáhl dostatečné dobrých výsledků na to, aby „porazil“ sofistikovanější algoritmy. Výhodou této implementace je její relativní jednoduchost a možnost jejího využití pro odhalení problémů, kterých se je třeba vyvarovat při implementaci dalších algoritmů.

Z popisu algoritmu vyplývá, že je zapotřebí mít k dispozici seznam všech cest. Cesta je sama o sobě také seznam – konkrétně seznam stavů, kterými prochází. Použitou datovou strukturou je tedy seznam seznamů. Cestu lze reprezentovat jednosměrně vázaným seznamem, protože jediné, co od cesty ve výsledku očekáváme, je její výpis. Algoritmus opakovaně expanduje nejlepší cestu. Abychom mohli expanzi provést, musíme si udržovat informaci o tom, která z cest je aktuálně je nejlepší. Dále musíme znát její poslední stav, abychom mohli vytvořit další cesty. Nakonec ještě musí být dostupná informace o tom, jestli už byla cesta vytištěna, nebo se dostala do absorpčního stavu. V obou případech nesmí být cesta dále expandována, takže můžeme říci, že je *uzavřená*.

Příklad 17 (Interní reprezentace BFS)**.** Tento diagram se také vztahuje k modelu 2.4. První buňka značí pravděpodobnost cesty a druhá její stav (Open/Closed). Třetí buňka zastupuje trojici ukazatelů – na další cestu, na první prvek cesty a na poslední prvek cesty (pro rychlejší přístup).

Diagram zachycuje moment, kdy byla vytištěna cesta $s_0 \cdot s_1 \cdot s_4 \cdot s_5$, následně uzavřena cesta $s_0 \cdot s_3$ z důvodu nalezení absorpčního stavu s_3 a vytištění cesty $s_0 \cdot s_2 \cdot s_5$. Mezi těmito událostmi samozřejmě proběhlo několik expanzí. V tuto chvíli by byla expandována první cesta, protože má nejvyšší pravděpodobnost ze všech otevřených cest.



Obrázek 4.1: Interní reprezentace model BFS

V diagramu na obrázku 17 není cesta $s_0 \cdot s_1$. Je to pozitivní důsledek mé implementace. Při expanzi by měly vzniknout nové cesty a otázkou je, co s původní cestou. První možnost je nejdříve vytvořit nové cesty a nakonec původní cestu uzavřít, případně odstranit. Druhá,

mnou zvolená možnost, je k původní cestě přidat prvního následníka posledního stavu. Pro další následníky se tato cesta zkopíruje a poslední stav se zamění.

Už při zběžném testování ve fázi implementace bylo možné si všimnout viditelného poklesu rychlosti generování nových cest při zvyšování počtu dosud nalezených cest. Při analýze tohoto problému byla nalezena i příčina tohoto chování. Doba procházení seznamu při hledání nejvyšší hodnoty roste přímo úměrně s délkou tohoto seznamu. Projít tento seznam je potřeba k-krát, což znamená složitost kvadratickou složitost $O(k^2)$. Problém by se dal odstranit použitím lepší datové struktury, podobně jako v sekci 4.4, avšak pro zachování kontrastu mezi algoritmy byla ponechána původní verze se seznamem.

4.3 Upravený Recursive Enumeration Algorithm

Pokud se podíváme zpět na popis algoritmu 1, zjistíme, že spíše než s přechodovou maticí pracuje se stavy a jejich předchůdci. Dalšími parametry, které si je třeba pamatovat pro každý stav, jsou kandidátní množina a seznam k-tých nalezených cest. I když má tento algoritmus rekurzi ve svém názvu, je třeba si uvědomit, že úroveň zanoření rekurze bude odpovídat až délce nalezené cesty, což může prakticky vyústit v tisíce úrovní zanoření a nepřijatelné paměťové náročnosti. V mé implementaci je tento problém řešen využitím vlastního zásobníku.

Základním kamenem je tedy pole všech stavů o délce N, včetně virtuálního koncového stavu. Stav je tvořen strukturou obsahující seznam cest, dále seznam předchůdců a množinu předchůdců. Pro seznam, množinu i pole lze použít datového typu `std::vector`. Pro potřeby algoritmu One-to-All Dijkstra (3.2.1) je ke každému stavu přidána ještě informace o jeho navštívení. Tato informace je ukládána tak, že je nastavena nenulová pravděpodobnost první cesty stavu.

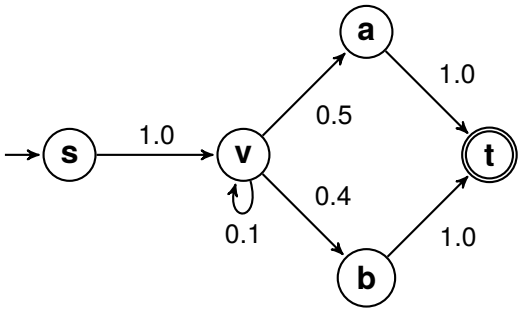
Algoritmus 1 popsaný v teoretické části této práce ukládal (na řádku 12 pseudokódu) do kandidátní množiny celou cestu. Tento způsob je nepraktický, protože zbytečně ukládá data, která lze zpětně odvodit. Namísto cesty si stačí pamatovat informaci o tom, ze kterého stavu jsme zvolili kolikátou nejpravděpodobnější cestu. Podrobněji je toto tvrzení rozvedeno v sekci 4.3.2 výpis cesty. Pravděpodobnost cesty lze taktéž dopočítat zpětně, avšak na tuto hodnotu se dotazujeme vždy při výběru nejpravděpodobnější cesty z kandidátní množiny. Pravděpodobnost je tedy z důvodu velké časové úspory ukládána na úkor drobného nárůstu paměťové náročnosti.

Příklad 18 (Hodnoty ve stavech po doběhnutí REA)**.** Chtějme $P_{>0.9}(true\ U\ t)$ z pětistavového modelu 4.2. Virtuální koncový stav je pro přehlednost vynechán[2].

Výsledná vnitřní reprezentace je vidět v tabulce 4.1. Kdybychom počítali dále, například pro $P > 0.999$, rostly by pouze seznamy cest. Pro stav t by se střídaly cesty skrz a a b, stejně tak by v kandidátní množině t bylo jednou a a jednou b.

V tabulce 4.1 je vidět, že většina kandidátních množin je prázdná. Toto pozorování si zaslouží vysvětlení. Velikost kandidátní množiny je rovna maximálně počtu předchůdců příslušného stavu. Po jejím vybrání (tedy nalezení další cesty) je tento počet ještě zmenšen o jedna. Vzhledem k tomu, že většina stavů v modelu 4.2 má pouze jednoho předchůdce a vidíme stav po nalezení cesty, je většina těchto množin prázdná.

[2]Stavy **a** a **b** mohly být v původním modelu koncové stavy a stav **t** by byl virtuální



Obrázek 4.2: Pětistavový model

Tabulka 4.1: Stavy **t**, **v**, **a** a **b** po doběhnutí příkladu 18

stav **t**

cesty			kandidáti		
stav	k-tá	ppst	stav	k-tá	ppst
b	1	0.5	a	2	0.04
a	1	0.4			
b	1	0.05			

stav **v**

cesty			kandidáti		
stav	k-tá	ppst	stav	k-tá	ppst
s	1	1.0			
v	1	0.1			
v	2	0.01			

stav **a**

cesty			kandidáti		
stav	k-tá	ppst	stav	k-tá	ppst
v	1	0.4			
v	1	0.04			

stav **b**

cesty			kandidáti		
stav	k-tá	ppst	stav	k-tá	ppst
v	1	0.5			
v	1	0.05			

4.3.1 Náhrada rekurze zásobníkem

Popis algoritmu pomocí rekurze je často používán díky své jednoduchosti. Při implementaci je však třeba brát v úvahu, jaké úkony musí počítač učinit při každém volání funkce. Typicky je nutné uložit adresu, ze které je funkce volána, poskytnout funkci parametry a následně zabrat paměť pro proměnné používané ve funkci. V této sekci je předvedena verze algoritmu 1, která si pro každé zanoření zabírá paměť pouze na dvě čísla. Algoritmus REA je zde popsán v takové podobě, v jaké je skutečně implementován. Hlavní změny oproti původnímu popisu jsou čtyři.

- Získání prvních nejkratších cest najednou (viz 3.2.1 a 4.4).
- Tisk cest při jejich nalezení.
- Ukládání cest pomocí odkazu na předchůdce.
- Využití iterace namísto rekurze.

Algorithm 2 REA implemented pseudocode

```
1: procedure GetPaths(threshold)
2:     GetFirstPaths(initial)
3:     StackPrintPath(1; terminal)
4:     k ← 2
5:     probability ← prob(σ_1(terminal))
6:     in ← true
7:     new stack
8:     while probability < threshold do
9:         if in then
10:            if k = 2 then
11:                C[v] ← {[u; 1; prob(σ_1(u))] : u ∈ Γ^{-1}(v) ∧ σ_1(v) ≠ σ_1(u) · v}
12:                if v = initial then
13:                    in ← false
14:                    continue
15:            u ← predecessor(σ_{k-1}(v))
16:            k' ← kof(σ_{k-1}(v)) + 1
17:            if exist(σ_{k'}(u)) then
18:                C[v] ← C[v] ∪ {[u; k'; prob(σ_{k'}(u) × P(u; v)]}
19:                in ← false
20:            else
21:                stack.push(v; k)
22:                v ← u
23:                k ← k'
24:        else
25:            if C[v] ≠ ∅ then
26:                σ_k(v) ← max(C[v])
27:                C[v] ← C[v] ∖ max(C[v])
28:            if C[v] = ∅ and v = terminal then
29:                exit
30:            if v = terminal then
31:                probability ← probability + prob(max(C[terminal]))
32:                StackPrintPath(k; terminal)
33:                k ← k + 1
34:            else
35:                (u; k') ← stack.pop
36:                C[u] ← C[u] ∪ {[v; k; prob(σ_k(v) × P(v; u)]}
37:                v ← u
38:                k ← k'
```

4.3.2 Výpis cest

Způsob výpisu cesty je otázkou uživatelského prostředí. Po doběhnutí algoritmu máme k dispozici počet cest (velikost k), pravděpodobnost každé z cest, její délku a samozřejmě posloupnost stavů, skrz které cesta vede. Vzhledem k tomu, že REA generuje cesty od nejpravděpodobnější a všechny její parametry jsou dostupné ihned po jejím nalezení, můžeme

si dovolit evidence vypisovat okamžitě po jejich nalezení. Přínosem tohoto řešení je možnost sledovat v reálném čase, jak rychle se evidence hledají v případě výpisu na obrazovku.

Algorithm 3 Path Printing from REA data structure

```
1: procedure RecursivePrintPath(k; state)
2:     print initial
3:     while not (k = 1 and state = initial) do
4:         RecursivePrintPath(predecessor(σ_k(state)); kof(σ_k(state)))
5:         print state

6: procedure StackPrintPath(k; state)
7:     new stack
8:     while not (k = 1 and state = initial) do
9:         stack.push(state)
10:        k' ← kof(σ_k(state)
11:        state ← predecessor(σ_k(state))
12:        k ← k'
13:    while not stack.empty do
14:        print stack.pop
```

4.4 One-to-All Dijkstra

Definice z předchozí kapitoly v sobě skrývá jeden zásadní problém. Konkrétně v kroku 1, kde je zapotřebí vybrat největší z prvků v poli. Nejspíše nejhorší způsob je průchod celého pole a hledání nejvyšší hodnoty. V tomto případě bychom se dostali ke složitosti V^2.

Lepší strukturou pro ukládání hodnot, ve kterých chceme vyhledávat nejvyšší hodnotu je halda (*heap*). Je známa modifikace původně zmiňovaného algoritmu, která ve svém inicializačním kroku nevytvoří celé pole, které obsahuje stejné hodnoty pro všechny prvky. Namísto toho pracuje s **prioritní frontou**, kterou postupně projde každý stav právě jedenkrát.

Prioritní fronta v jazyce C++ zapouzdřuje právě haldu, avšak navenek se chová přesně dle našeho očekávání – po přidání do fronty (*push*) je prvek zařazen do fronty dle své priority. Při výběru *pop* je vrácen prvek s nejvyšší prioritou [17].

Algorithm 4 One-to-All Dijkstra using priority queue

```
1: procedure GetFirstPaths(initial)
2:     new queue
3:     queue.push(initial; 1.0)
4:     while not queue.empty do
5:         currentstate ← queue.pop
6:         for all state in successor[state] do
7:             if prob(σ_1(state)) < prob(σ_1(currentstate)) × P(currentstate; state) then
8:                 σ_1(state) ← σ_1(currentstate) · state
9:                 queue.push(state)
```

4.5 Ukázka

V závěru kapitoly o implementaci si můžeme ukázat část výpisu při volání STORMu na model 2.4 a zkoumanou vlastností $P_{>0.5}(true\ U\ end)$. Úplný výpis včetně příkladu volání je k nalezení v příloze A.

```
Probability threshold: 0.5
k: 1 (0.3; 0.3)
0 -> 1 -> 4 -> 5
k: 2 (0.16; 0.46)
0 -> 2 -> 5
k: 3 (0.15; 0.61)
0 -> 1 -> 4 -> 1 -> 4 -> 5
Paths found: 3
Transitions in counterexample: 10
Reached probability: 0.61
Time for counterexample generation: 0.001s.
```

Kapitola 5

Měření a výsledky

V této kapitole již opustíme náš model o šesti stavech a posuneme se k experimentování na modelech, které jsou větší o několik řádů. Velikostí modelu je typicky míněn počet stavů, avšak jsou i další parametry, které ovlivňují obtížnost hledání protipříkladů. Stejně tak o výsledném protipříkladu můžeme říci více než kolik cest obsahuje. Pro přehled je zde seznam parametrů, které budeme zkoumat při generování protipříkladů:

- Počet stavů a přechodů v modelu
- Pravděpodobnostní hranice pro protipříklad, ze které plyne počet cest k
- Vlastnost, kterou v modelu chceme zkoumat
- Čas potřebný pro vygenerování protipříkladu
- Maximální velikost paměti zabrané programem
- Časy běhu jednotlivých částí výpočtu protipříkladu

Z tohoto seznamu jsou proměnné první tři body, proto v následujících experimentech zkusíme postupně měnit právě tyto tři parametry. Od experimentů očekáváme, že nám poskytnou informace o řádu složitosti příslušného algoritmu. Dále ukážou, jaké jsou hranice pro výpočty na běžném počítači.

5.1 Způsob testování

Na testovacím stroji byl procesor pracující na frekvenci 3,4 GHz. Osazená paměť měla velikost 16GB a žádný z experimentů nemusel odkládat data z paměti na disk. Aplikace STORM je jednojádrová a byla spouštěna z operačního systému Debian verze 9.

Časy byly měřeny přímo pomocí STORMu. Maximální zabraná paměť byla měřena pomocí programu `time`, ze kterého byl brán parametr *Maximum resident set size*. Vzhledem k tomu, že výpis protipříkladu do terminálu či do souboru na disk může být úzkým hrdlem měření, byl výpis potlačen. Pro měření se zakotveným počtem cest k byl program dočasně změněn, aby nebylo nutné hledat přesnou pravděpodobnostní hranici. Naměřené výsledky jsou průměrem několika běhů nad daným modelem, maximální odchylky byly v řádu jednotek procent.

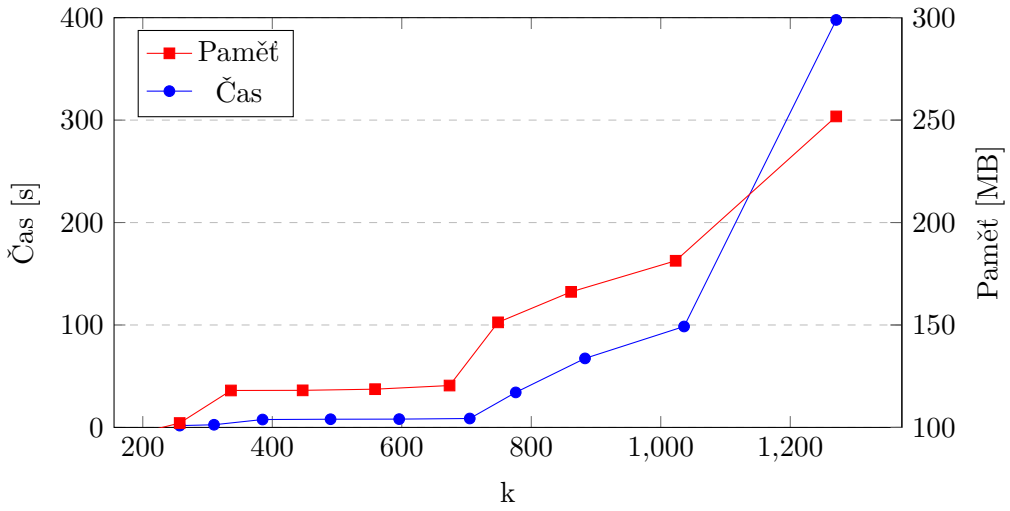
5.2 Představení modelů

První model, na kterém budeme testovat, se týká integrovaných obvodů. Přesněji se jedná o NAND multiplexing, tedy techniku, která se snaží z potenciálně nespolehlivých součástek poskládat spolehlivý integrovaný obvod. Tento model bude dále označován zkratkou nand.[13]

Druhý model se týká energetické úspornosti v discích na úkor jejich výkonu. Pokud disk poběží neustále, bude mít větší spotřebu energie, proto je výhodné přepínat ho do úspornějších režimů. V těchto režimech (ať už s nulovým nebo nižším počtem otáček ploten) není disk schopen provádět čtení a zápisy tak rychle. Pokud nestíhá, začne si ukládat požadavky do fronty. Vlastnost kterou můžeme o tomto modelu zkoumat je například počet přetečení fronty na požadavky. Tento model je nadále označován zkratkou DPM (Dynamic Power Management).[3]

5.3 Best-First Search

V prvním měření se podíváme na algoritmus BFS. Dalo se předpokládat, že tento algoritmus bude mít několik vad. Zaprvé ukládá velké množství redundantních dat (dvě cesty, které se liší pouze posledním stavem si zaberou dvakrát celé místo pro cestu v paměti). Druhou vadou je hledání nejmenších hodnot v jednosměrném seznamu. Obě tyto vady jsou příčinou kvadratické složitosti (jak časové, tak paměťové). Neúnosnou se v testovaném modelu nand o 1700 stavech stala časová náročnost. Dopředu můžeme říct, že REA zvládne stejnou úlohu vyřešit pod vteřinu.



Obrázek 5.1: Časová a paměťová závislost na zvyšujícím se počtu cest v BFS (nand-5-2)

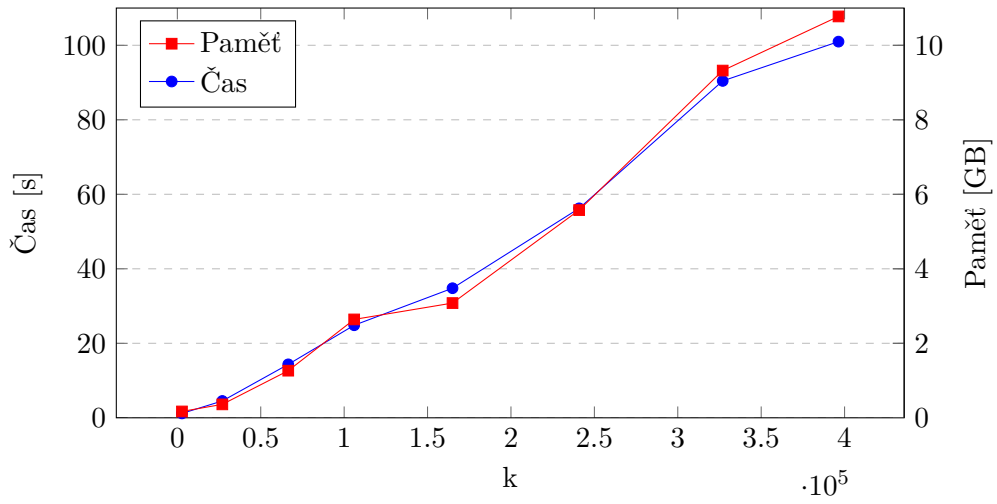
Předtím, než přejdeme k důkladnějšímu testování druhého algoritmu, můžeme říct, že pro nízká k dokáže algoritmus vytvořit protipříklad i na větších modelech.

5.4 Testování REA

Rychlost a paměťová náročnost jsou vlastnosti, které nás u nových algoritmů zajímá nejčastěji. Jak bylo řečeno v úvodu této kapitoly, tyto vlastnosti jsou ovlivněny především dvěma parametry – velikostí modelu a počtem cest.

5.4.1 Zvyšování počtu cest

První experiment s REA zachovává velikost modelu a zvětšuje pravděpodobnostní hranici, v důsledku které se zvětšuje počet cest k (osa x na následujícím grafu).



Obrázek 5.2: Časová a paměťová závislost na zvyšujícím se počtu cest v modelu nand-20-4

Pro úplnost musíme uvést, že čas pro převod modelu ze vstupního souboru na vnitřní reprezentaci trval přibližně šest vteřin a není do výsledku započítán. Co je naopak započteno je čas Dijkstrova algoritmu, který trval přibližně půl sekundy.

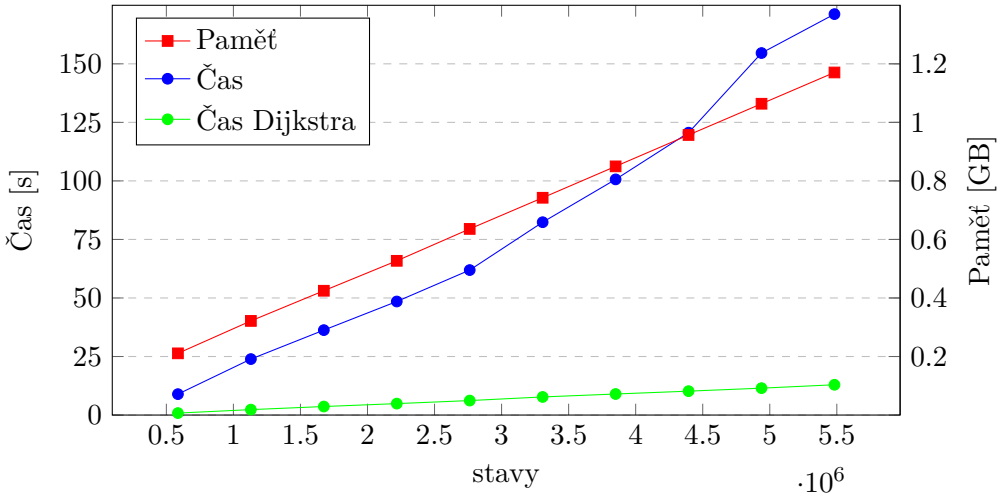
Nyní se můžeme podívat na graf 5.2 a zkonstatovat, že časová i paměťová náročnost jsou přímo úměrné velikosti protipříkladu. Tato vlastnost je očekávaná a zdůvodnění už bylo naznačeno u tabulky 4.1. V nejhorším případě je pro nově nalezenou cestu potřeba zabrat konstantní velikost paměti pro každý stav na této cestě.

V grafu je ještě jedna zajímavá věc. I když jsou zanesené body průměrem několika běhů měření, paměť zabraná u měření s 106 tisíci stavy převyšuje očekávaný lineární průběh zabírání paměti. Příčinou je pravděpodobně použití datového typu vektor pro ukládání nalezených cest pro jednotlivé stavy. Vektor v C++ má tu vlastnost, že při jeho alokaci je zabráno více místa, než kolik je v dané chvíli potřeba, a toto místo je později využito, pokud jsou přidávány další prvky. Důvodem je úspora času při případném opětovném zabírání většího úseku místa.

5.4.2 Zvětšování modelu

Druhý experiment zakotvuje počet cest k na sto tisících cestách. Zvětšuje se v tomto případě model. Toto měření nám dává příležitost zkoumat ještě dva další ukazatele. Zaprvé můžeme zjišťovat, jak rychlý je Dijkstrův algoritmus. V grafu 5.3 není zakreslen čas, za jaký převedl STORM definici modelu na svou vnitřní reprezentaci. Tato doba odpovídala

zhruba trojnásobku doby běhu REA a více informací o rychlosti tohoto převodu je možné získat na stránkách tohoto projektu[1].



Obrázek 5.3: Časová a paměťová závislost na zvyšujícím se počtu stavů v modelu DPM

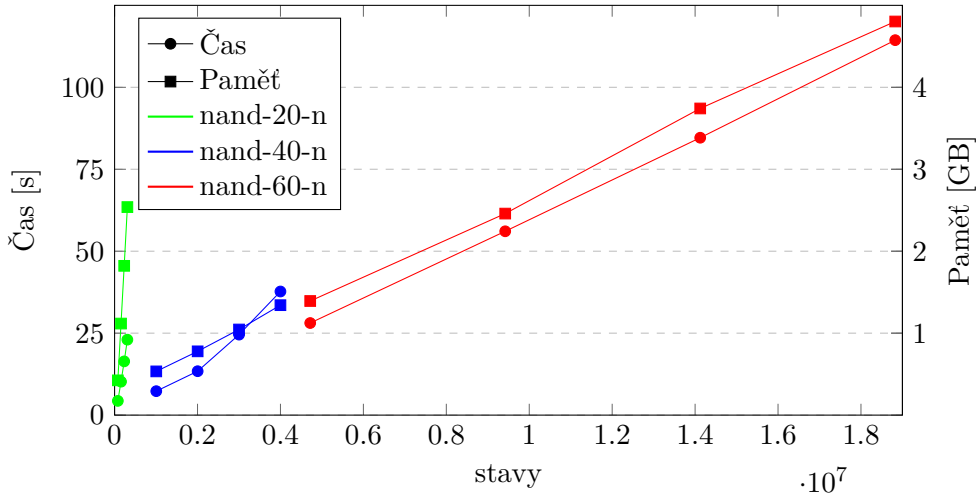
I když model nand není tak dobře škálovatelný, můžeme si v podobném grafu 5.4 ukázat naměřené výsledky až pro variantu nand-60-4, která obsahuje 18,8 milionů stavů, mezi kterými je téměř třicet miliónů přechodů. Poměr mezi počtem stavů a počtem přechodů je další ukazatel, na který je třeba poukázat. U většiny dostupných modelů, včetně těch námi používaných, je tento poměr menší než dva.

Další metrika, kterou jsme ještě nerozebrali je počet přechodů, který je ve všech cestách protipříkladů (tedy součet velikostí jednotlivých evidencí). Oba naše modely jsou velmi symetrické, takže délka všech cest protipříkladu je stejná. Počet přechodů v protipříkladu je tedy přímo úměrný k. Průměrná délka cesty rostla v modelech nand od dvou set do dvou tisíc (v modelu nand-60-4), což při sto tisících cestách vychází na více než 200 milionů přechodů. Právě toto číslo je důvod, proč byl u testování potlačen výstup. Výsledný soubor by měl několik GB a rychlost zápisu na disk by neměla být při těchto měřeních směrodatná.

5.4.3 Změna hledané vlastnosti

Poslední experiment má připomenout, že cílem této práce je vytvoření nástroje, který nám poskytne protipříklad k námi dané vlastnosti. Zakotvíme nyní velikost modelu i počet hledaných cest $k = 100000$ a budeme hledat protipříklad k různým vlastnostem. Je poněkud nesmyslné vynášet hodnoty v tabulce 5.1 do grafu, vzhledem k tomu, že každý řádek hovoří o jiné vlastnosti.

[1] http://www.stormchecker.org/benchmarks.html



Obrázek 5.4: Časová a paměťová závislost na zvyšujícím se počtu stavů v modelu nand

Model	**Vlastnost**	**PC**	**Paměť** [MB]	**Čas** [s]
DPM-1000	q=qMax	202 M	211	8.914
DPM-1000	lost=lMax	203 M	751	20.258
DPM-1000	energy=eMax	601 M	1 506	30.437

Tabulka 5.1: Různé vlastnosti na stejném modelu, PC je počet přechodů v evidencích

5.5 Souhrn výsledků

V tuto chvíli je potřeba porovnat dva údaje – naměřený výsledek a očekávaný výsledek. U algoritmu BFS se očekával špatný výsledek, a ten se také dostavil. Očekávaná kvadratická časová náročnost odpovídá naměřeným hodnotám v prvním testu.

V prvních fázích vývoje Dijkstrova algoritmu byl způsob vyhledávání nejpravděpodobnější cesty implementován podobným způsobem, jako je tomu teď u algoritmu BFS. Algoritmus trpěl stejnou vadou, tedy kvadratickým prodlužováním časové náročnosti v závislosti na počtu stavů. Po přechodu na prioritní frontu tento problém zmizel.

Algoritmus REA by měl mít časouvou složitost $O(p + kN \log(p/N))$, kde p je počet přechodů, N je počet stavů a k je počet nalezených cest. Při škálování většiny modelů (včetně obou testovaných) zůstává konstantní poměr p/N, takže by výsledná složitost měla být přímo úměrná velikosti modelu (první i druhý sčítanec složitostního vztahu) a přímo úměrná počtu nalezených cest (sčítanec druhý). Lineární závislost na velikosti modelu potvrzují experimenty v sekci 5.4.2 a lineární závislost na velikosti k potvrzuje výsledek experimentu v sekci 5.4.1.

Autoři ve své práci [10] hovoří i o paměťové složitosti. Ta by měla být lineárně závislá na počtu přechodů p v modelu, protože velikost kandidátních množin je shora omezena p. Dále by měla být přímo úměrná na počtu přechodů P ve všech nalezených cestách. Složitostní vztah $O(p + P)$ potvrzuje všechny provedené testy.

Kapitola 6

Závěr

Tato práce měla teoretickou a praktickou rovinu. V teoretické rovině bylo cílem vysvětlit čtenářům pojmy spojené s formální verifikací a prací s modely. K těmto informacím bylo třeba dodat něco více o hledání protipříkladů. Omezení se na zkoumání modelů pouze typu DTMC bylo dobrou volbou. Umožnilo to vysvětlení pojmů do dostatečné hloubky a bylo možné najít logickou návaznost ve vysvětlování teorie od modelu k logice a nakonec k protipříkladům.

Při popisu algoritmů bylo použito několik pseudokódů, které jsou krokem mezi teoretickým popisem a praktickou implementací. V práci, která popisuje algoritmus REA, takovýto pseudokód chybí a pochopení algoritmu je mnohem složitější. V rovině praktické jsem se poprvé v životě setkal s projektem, který měl více než deset tisíc řádků zdrojového kódu. Dvojím neštěstím STORMu je velké množství souborů, díky které jsou však autoři schopni udržovat program velmi modulární, a nedostatečná programátorská dokumentace. Na druhou stranu musím vyzdvihnout tým, který za tímto programem stojí, jeho členové na mé e-mailové dotazy odpovídali podrobně, ochotně a v rámci několika hodin.

Před začátkem experimentování jsme si s vedoucím práce řekli, že za dostatečné budeme považovat, když bude algoritmus REA schopen pracovat s modely v řádu desetitisíců stavů. Bylo velmi příjemné zjistit, že není problém vyhodnocovat modely o milionu stavů. Velké modely na vstupu generují velké výstupy, které pro uživatele přestávají být čitelné, a proto by jako další vylepšení této práce mohlo následovat lepší grafické zobrazení protipříkladů.

Literatura

[1] Ábrahám, E.; Becker, B.; Dehnert, C.; aj.: *Counterexample Generation for Discrete-Time Markov Models: An Introductory Survey*. Cham: Springer International Publishing, 2014, ISBN 978-3-319-07317-0, s. 65–121.
URL https://doi.org/10.1007/978-3-319-07317-0_3

[2] Aljazzar, H.; Leue, S.: *K*: A heuristic search algorithm for finding the k shortest paths. Artificial Intelligence*, 2011: s. 2129 – 2154, ISSN 0004-3702.
URL http://www.sciencedirect.com/science/article/pii/S0004370211000865

[3] Benini, L.; Bogliolo, A.; Paleologo, G.; aj.: *Dynamic Power Management - IBM Disk Drive*. [Online; navštíveno 10.05.2018].
URL http://www.prismmodelchecker.org/casestudies/power_dtmc.php

[4] Dehnert, C.; Junges, S.; Katoen, J.; aj.: *A storm is Coming: A Modern Probabilistic Model Checker*. *CoRR*, 2017.
URL http://arxiv.org/abs/1702.04311

[5] Dijkstra, E. W.: *A note on two problems in connexion with graphs. Numerische Mathematik*, 1959: s. 269–271, ISSN 0945-3245.
URL https://doi.org/10.1007/BF01386390

[6] Eppstein, D.: *Finding the k shortest paths*. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, s. 154–165.

[7] Han, T.; Katoen, J. P.; Berteun, D.: *Counterexample Generation in Probabilistic Model Checking. IEEE Transactions on Software Engineering*, , č. 2, March 2009: s. 241–257, ISSN 0098-5589.

[8] Hansson, H.; Jonsson, B.: *A logic for reasoning about time and reliability. Formal Aspects of Computing*, 1994: s. 512–535, ISSN 1433-299X.
URL https://doi.org/10.1007/BF01211866

[9] Jansen, N.; Ábrahám, E.; Scheffler, M.; aj.: *The COMICS Tool - Computing Minimal Counterexamples for Discrete-time Markov Chains*. *CoRR*, 2012.
URL http://arxiv.org/abs/1206.0603

[10] Jiménez, V. M.; Marzal, A.: *Computing the K Shortest Paths: A New Algorithm and an Experimental Comparison*. In *Algorithm Engineering*, 1999, ISBN 978-3-540-48318-2, s. 15–29.

[11] Kocher, P.; Genkin, D.; Gruss, D.; aj.: *Spectre Attacks: Exploiting Speculative Execution*. *CoRR*, 2018.
URL http://arxiv.org/abs/1801.01203

[12] Kwiatkowska, M.; Norman, G.; Parker, D.: *PRISM 4.0: Verification of Probabilistic Real-time Systems*. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, *LNCS*, ročník 6806, editace G. Gopalakrishnan; S. Qadeer, Springer, 2011, s. 585–591.

[13] Norman, G.; Parker, D.; Kwiatkowska, M.; aj.: *Evaluating the Reliability of NAND Multiplexing with PRISM*. 2005.

[14] Pearl, J.: *Heuristics: Intelligent search strategies for computer problem solving*. 1984.

[15] Thorup, M.: *Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem. J. Comput. Syst. Sci.*, 2004: s. 330–353, ISSN 0022-0000.
URL http://dx.doi.org/10.1016/j.jcss.2004.04.003

[16] Tingting Han, J.-P. K.: *Counterexample Generation in Probabilistic Model Checking*.
URL http://ieeexplore.ieee.org/abstract/document/4770111/

[17] Tým GCC: *GCC 8.1 Standard C++ Library Manual*. [Online; navštíveno 25.03.2018].
URL https://gcc.gnu.org/onlinedocs/gcc-8.1.0/libstdc++/manual/

Příloha A

Ukázka výpisu

```
> ./bin/storm --prism ../resources/examples/testfiles/dtmc/example.pm
--prop "P<0.5 [true U (\"end\")]" --counterexample
--counterexample:format paths

Storm 1.2.2-alpha (dev)
Date: Fri May 4 12:00:00 2018
Command line arguments:
--prism ../resources/examples/testfiles/dtmc/example.pm
--prop P<0.5 [true U ("end")] --counterexample
--counterexample:format paths
Current working directory: /storm/build
Time for model construction: 0.018s.
-------------------------------------------------------------
Model type:     DTMC (sparse)
States:         6
Transitions:    10
Reward Models:  none
State Labels:   3 labels
   * deadlock -> 0 item(s)
   * init -> 1 item(s)
   * end -> 1 item(s)
Choice Labels:  none
-------------------------------------------------------------
Dijkstra time: 0.000s.
Virtual terminal state: 6
Probability threshold: 0.5
-------------------------------------------------------------
(probability of path; probability of all already found paths)
k: 1 (0.3; 0.3)
0 -> 1 -> 4 -> 5
k: 2 (0.16; 0.46)
0 -> 2 -> 5
k: 3 (0.15; 0.61)
0 -> 1 -> 4 -> 1 -> 4 -> 5
-------------------------------------------------------------
Paths found: 3
Transitions in counterexample: 10
Reached probability: 0.61
Time for counterexample generation: 0.001s.
```

Příloha B

Obsah CD

Přiložené CD obsahuje text této práce a zdrojové kódy včetně zdrojových kódů programu STORM.

- Přímo v kořenovém adresáři se nachází soubor `BP_xmolek00.pdf` s touto prací.
- Ve složce `\LaTeX` jsou zdrojové sobory potřebné pro vznik souboru `BP_xmolek00.pdf`. Vygenerovat pdf lze například na serveru www.sharelatex.com.
- Složka `\sources` obsahuje mé zdrojové soubory algoritmů REA a BFS.
- Složka `\STORM` obsahuje zdrojové kódy programu STORM, včetně souboru mého zdrojového kódu pro algoritmu REA a změn potřebných k tomu, aby bral STORM tento nový soubor v potaz a byl schopen generovat protipříklady.

Ve složka `STORM` jsou soubory pouze pro algoritmus REA. Autoři programu nedoporučili integrovat dva algoritmy pro stejnou úlohu do jednoho programu, a tak je pro vyzkoušení algoritmu BFS potřeba přepsat obsah souborů
`\STORM\storm\src\storm\counterexamples\MyDTMCCounterexample.cpp` / `.h` obsahy souborů
`\sources\BFS_MyDTMCCounterexample.cpp` / `.h`.

Pro kompilaci programu doporučuji nastudovat manuál umístěný na stránkách tohoto projektu www.stormchecker.org. Na stejných stránkách je taktéž umístěn manuál k používání tohoto programu. Příklad volání pro generování protipříkladů je příloze A.