Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Spolehlivostní modelování pohotových systémů

Plzeň, 2006 Marek Paška

Abstract

This work is based on previous work by Radek Hoštička. In the previous work a language for description of Markov models without absorptive states was designed. In this work a language for evaluation of Markov models is designed. Evaluation means obtaining of higher order information from asymptotic state probabilities of models, such as mean queue length in a queueing system. The language is patterned after SQL and is called MMQL—Markov Model Query Language. A proof of concept implementation was developed and usability of the query language was demonstrated on examples. Part of the implementation is also a simple and cross–platform graphical user interface that covers both creating Markov model in descriptive language and evaluation using query language.

Obsah

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 17. května 2006 Marek Paška

Poděkování

Chtěl bych poděkovat svému vedoucímu Stanislavu Rackovi za jeho cenné rady a připomínky.

1 Úvod

Matematické modelování

Matematický model je abstraktní model, který používá matematického jazyka k popisu chování systému. Matematické modely se používají především v přírodních vědách (fyzika, chemie, biologie), ale též v ekonomii, sociologii a politologii.

Modelovaný systém obvykle představuje nějaký objekt reálného světa. Při popisu reálného objektu matematickým jazykem se (často nevyhnutelně) dopouštíme (i značného) zjednodušení. Obvykle do modelu zahrnujeme jenom ty vlastnosti (atributy) modelovaného systému, které považujeme za důležité, ostatní z různých důvodů zanedbáváme. Rozhodnutí, které vlastnosti jsou důležité a které zanedbatelné, je odvislé od účelu konstrukce modelu. Žádoucí vlastností modelů je též obecnost — model nepředstavuje jeden objekt, ale celou třídu objektů.

Modelováním je celý proces konstrukce a využití modelu. Modelování nám umožňuje získat nové poznatky o modelovaném objektu. To je obzvláště cenné, pokud je originální objekt nějakým způsobem nedosažitelný a experimentování s ním nemožné nebo příliš nákladné.

Markovské modely v kontextu této práce

Jednou z metod, které umožňují matematické modelování reálných objektů, jsou i markovské náhodné procesy. Modely markovských náhodných procesů dále zjednodušeně nazýváme markovskými modely.

Tato práce navazuje na práci Radka Hoštičky [HOŠ99], který ve své diplomové práci vytvořil jazyk pro popis markovských modelů. Tato práce má za cíl vylepšit vyhodnocování markovských modelů, k tomuto účelu je vytvořen speciální dotazovací jazyk.

Přehled kapitol

Kapitola 2 se zabývá teoretickými poznatky o markovských náhodnýc procesech. V kapitole 3 je stručně popsána předchozí práce Radka Hoštičky. Je popsán jeho jazyk pro popis markovských modelů a formáty datových souborů. V kapitole 4 je návrh dotazovacího jazyka pro získávání užitečných dat z markovských modelů. Tato kapitola je těžištěm této práce. Kapitola 5 popisuje realizaci navrženého dotazovacího jazyka. Kapitola 6 je věnována příkladům.

2 Teoretická část

2.1 Markovské náhodné procesy

Nejprve vysvětlíme některé používané pojmy. Obejdeme se přitom bez exaktních definic a důkazů, které lze v případě potřeby nalézt ve speciální literatuře, např [Mand85], [Havr86], [Triv82].

Náhodný proces je každá funkce $X(t)$, jejíž hodnota při každé hodnotě argumentu je náhodná veličina. Jako argument t budeme dále s ohledem na aplikace uvažovat výhradně čas. Definičním oborem D argumentu t je (spojitá) množina nezáporných reálných čísel. Realizací náhodného procesu $X(t)$ rozumíme funkci $x(t)$ získanou konkrétním pokusem. Náhodný proces $X(t)$ můžeme rovněž chápat jako množinu všech možných realizací $x(t)$.

Náhodná posloupnost. Definiční obor argumentu D obsahuje konečný nebo spočetný počet hodnot, tedy $D = \{t_k\}, k = 0, 1, 2, ...$ Náhodná funkce $X(t)$ přechází v náhodnou posloupnost $X(t_k) = X_k$.

Markovský řetězec je speciální náhodná posloupnost. Pravděpodobnost, že člen posloupnosti X_k nabude určitou hodnotu, je ovlivněna pouze hodnotou předchozího členu posloupnosti X_{k-1}. Bez ztráty na obecnosti můžeme dále uvažovat diskrétní čas t_k jako posloupnost nezáporných celých čísel, tedy $t_k = k$. V aplikacích představují hodnoty členů X_k čísla stavů modelovaného diskrétního systému. Jejich definičním oborem je například množina přirozených čísel $E = \{1, 2, ..., n\}$. Řetězec lze popsat maticí přechodů mezi stavy

$$P(k) = \begin{bmatrix} p_{11}(k) & \dots & p_{1n}(k) \\ \vdots & \ddots & \vdots \\ p_{n1}(k) & ... & p_{nn}(k) \end{bmatrix}, \sum_{j=1}^{n} p_{ij}(k) = 1$$

kde $p_{ij}(k)$ je pravděpodobnost přechodu ze stavu i do stavu j v diskrétním čase k. Jinak řečeno — je–li hodnota $X_k = i$, pak pravděpodobnost, že $X_{k+1} = j$, je $p_{ij}(k)$. Součet pravděpodobností v řádku musí být 1.

Diskrétní markovský proces. Množina argumentu D je spojitá (spojitý čas t), množina funkčních hodnot E je diskrétní. Uvažujme opět $E = \{1, 2, ..., i, ..., n\}$. Hodnota náhodné funkce $X(t)$ má tedy význam čísla stavu a n je počet možných stavů. Ke změně stavu může dojít v libovolném časovém okamžiku, tedy pro jakékoliv t. Chování procesu po přechodu do stavu i (doba setrvání ve stavu, příští stav) nijak nezáleží na minulosti (posloupnosti stavů, kterými proces prošel). Jedná se o důležitou vlastnost markovských náhodných procesů označovanou jako

absence paměti. Diskrétní markovský proces je vhodným matematickým modelem pro stochastické systémy s diskrétní množinou stavů a spojitým časem.

Uvažujme, že v čase t_1 je náhodný proces ve stavu i. Pravděpodobnost přechodu do stavu j v časovém okamžiku $< t_1, t_2 >$ záleží zřejmě v obecném případě na hodnotách t_1 a t_2 a tedy prvky matice přechodů jsou funkce $p_{ij} = p_{ij}(t_1, t_2)$. U *homogenních* diskrétních markovských procesů jsou pravděpodobnosti přechodů pouze funkcí časového rozdílu $\Delta t = t_2 - t_1$ a tedy $p_{ij} = p_{ij}(\Delta t)$. Dále budeme pracovat výhradně s homogenními markovskými procesy.

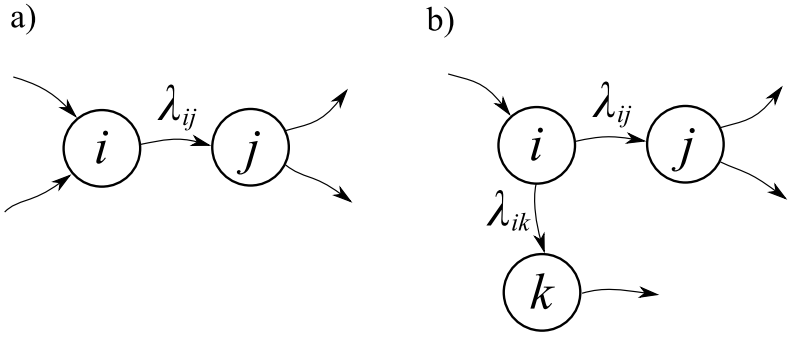
Důležitou veličinou u diskrétních markovských procesů je intenzita přechodu ze stavu i do stavu j, kterou označíme λ_{ij}. Intenzita pravděpodobnosti přechodu (zkráceně intenzita přechodu) je definována vztahem

$$\lambda_{ij} = \lim_{\Delta t \to 0} \frac{p_{ij}(\Delta t)}{\Delta t}$$

Pro homogenní markovské procesy je λ_{ij} konstantní. Pravděpodobnost přechodu ze stavu i do stavu j v dostatečně malém časovém intervalu Δt je pak

$$p_{ij}(\Delta t) = \lambda_i \Delta t \tag{1}$$

V aplikacích markovských náhodných procesů nás zpravidla zajímá s jakou pravděpodobností se proces vyskytuje v čase t ve stavu i. Tuto pravděpodobnost označíme $p_i(t)$. Způsob jejího určení předvedeme na několika jednoduchých příkladech, které umožní pochopit obecný postup výpočtu. Budeme uvažovat konkrétní markovský náhodný proces s daným počtem stavů n. V čase $t = 0$ se tento proces vyskytuje ve stavu i a tedy $p_i(0) = 1$. Jediný stav různý od i, do kterého má přechod nenulovou intenzitu je stav j. Uvažovaná situace je na obrázku 1a.



Obrázek 1: Fragmenty grafu přechodů

Podmíněná pravděpodobnost přechodu ze stavu i do j v malém časovém intervalu $< t, t + dt >$ je podle vztahu (1) dána jako $\lambda_{ij} dt$. Podmínkou je, že proces se v čase t nachází ve stavu i. Nepodmíněná pravděpodobnost přechodu je

pak dána součinem podmíněné pravděpodobnosti přechodu a pravděpodobnosti podmínky, tedy $p_i(t)\lambda_{ij}dt$. Přechodem uskutečněným v uvažovaném časovém intervalu $< t, t + dt >$ se zmenšuje pravděpodobnost $p_i(t + dt)$ o $p_i(t)\lambda_{ij}dt$. Tyto úvahy vedou ke vztahu

$$p_i(t + dt) = p_i(t) - \lambda_{ij}p_i(t)dt$$

Po úpravě postupně dostaneme

$$\frac{p_i(t + dt) - p_i(t)}{dt} = -\lambda_{ij}p_i(t)$$

$$p_i'(t) = -\lambda_{ij}p_i(t)$$

Získanou lineární diferenciální rovnici prvního řádu s konstantními koeficienty řešíme pro počáteční podmínku $p_i(0) = 1$ a známým postupem dostaneme výslednou pravděpodobnost setrvání ve stavu i.

$$p_i(t) = e^{-\lambda_{ij}t}$$

Pravděpodobnost, že v čase t již proces nebude ve stavu i (tedy že náhodný čas odchodu ze stavu je menší než t) je zřejmě distribuční funkcí $F_i(t)$ náhodné doby setrvání procesu ve stavu i.

$$F_i(t) = 1 - e^{-\lambda_{ij}t}$$

Jedná se o distribuční funkci exponenciálního rozdělení s parametrem λ_{ij}.

Zcela analogickým postupem dostaneme pro modifikovanou situaci znázorněnou na obrázku (1b) diferenciální rovnici.

$$p_i'(t) = -(\lambda_{ij} + \lambda_{ik})p_i(t) = -\lambda_i p_i(t)$$

Bez dalšího odvozování je možné učinit důležitý závěr, že náhodná doba setrvání ve stavu i má exponenciální pravděpodobnost rozdělení s parametrem λ_i, kde λ_i je součet intenzit všech odchodů ze stavu i. Střední doba setrvání ve stavu (tj. střední hodnota exponenciálního rozdělení) i je pak $T_i = 1/\lambda_i$.

Na tomto místě zdůrazníme, že základním předpokladem využitelnosti markovského náhodného procesu jako reálného systému (kromě nezávislosti chování systému na jiném stavu než posledním) je exponenciální pravděpodobnostní rozdělení doby setrvání v každém stavu systému (resp. exponenciální pravděpodobnostní rozdělení mezi časovým bodem příchodu do stavu a vznikem události inicializující jednotlivé přechody). Jinak řečeno: nachází-li se model v čase t ve stavu i, je podmíněná pravděpodobnost realizace přechodu do jiného stavu j (tj. vzniku příslušné události) v navazujícím malém časovém intervalu dt stále stejná (tj. nezávislá na čase t) a daná vztahem $\lambda_{ij}dt$.

Hrana vedoucí do stavu i například ze stavu h by zřejmě byla v diferenciální rovnici reprezentována prvkem $\lambda_{hi}p_h(t)$ s kladným znaménkem. Pro každý stav uvažovaného náhodného procesu (uzel grafu přechodů) můžeme tedy formulovat lineární diferenciální rovnici prvního řádu s konstantními koeficienty. Matematickým popisem časového vývoje pravděpodobností stavů diskrétního markovského náhodného procesu je pak soustava lineárních diferenciálních rovnic v normálním tvaru s konstantními koeficienty, doplněná vektorem počátečních pravděpodobností stavů. Tato soustava se v teorii náhodných procesů nazývá druhý systém Kolmogorových rovnic a lze ji maticově zapsat ve tvaru

$$\boldsymbol{p}'(t) = \boldsymbol{p}(t)\boldsymbol{\Lambda} \tag{2}$$

V uvedené maticové rovnici je $\boldsymbol{p}(t)$ řádkový vektor pravděpodobností stavů a $\boldsymbol{p}'(t)$ řádkový vektor derivací pravděpodobností stavů.

$$\boldsymbol{p}(t) = [p_1(t), p_2(t), ..., p_n(t)]$$

$$\boldsymbol{p}'(t) = [p_1'(t), p_2'(t), ..., p_n'(t)]$$

Čtvercová matice koeficientů soustavy (2) se nazývá matice intenzit přechodů. Obecný prvek λ_{ij} matice $\boldsymbol{\Lambda}$ představuje pro $i \neq j$ intenzitu přechodu ze stavu i do j. Řádkový součet prvků matice je nulový a hodnota prvku na diagonále (pro $i = j$) je zápornou hodnotou součtu ostatních prvků v řádku, tedy $-\lambda_i$. Řešení soustavy (2) vyžaduje znalost vektoru počátečních pravděpodobností stavů $p(0)$.

Pro další výklad je významný pojem absorpční stav. Pokud v grafu přechodů nevede ze stavu i žádná hrana do jiného stavu, je stav i absorpční. Pokud se proces do tohoto stavu dostane, nikdy se již nedostane do žádného jiného stavu.

2.2 Markovské procesy s absorpčními stavy

Markovské náhodné procesy s absorpčními stavy se využívájí zejména k modelování přechodových dějů. „Život“ modelu je časově omezený a končí dosažením absorpčního stavu. Typickou aplikací jsou spolehlivostní modely neobnovovaných systémů, tj. systémů, jejichž život končí neopravitelnou poruchou. Absorpční stavy modelu představují stavy, ve kterých je systém porouchaný jako celek. Použití markovských modelů ve spolehlivostních aplikacích je omezeno předpokladem, že intenzity poruch a oprav jsou konstantní pro všechny prvky systému a náhodné doby do poruchy prvků a náhodné doby oprav prvků mají exponenciální pravděpodobnostní rozdělení.

2.3 Markovské procesy bez absorpčních stavů

Pokud nemá markovský proces absorpční stavy, je graf přechodů silně souvislý (pro každé dva uzly i a j existuje cesta z i do j). Pro takové markovské procesy

je možné určit limitní (též asymptotické, ustálené) hodnoty pravděpodobností stavů $p_1(\infty)$, $p_2(\infty)$, ..., $p_n(\infty)$ jednodušším způsobem, než řešením soustavy diferenciálních rovnic (2).

Dále budeme kvůli přehlednosti označovat limitní pravděpodobnosti stavů jako p_1, p_2, ..., p_n. Můžeme je určit na základě následující úvahy: pokud existují limitní hodnoty pravděpodobností stavů, musí být odpovídající limitní hodnoty jejich derivací nulové (funkční hodnota se již nemění). Pro dostatečně velkou hodnotu času ($t \to \infty$) soustava (2) přejde na soustavu lineárních algebraických rovnic pro neznámé limitní pravděpodobnosti p_1, p_2, ..., p_n. V maticovém vyjádření tedy dostaneme rovnici

$$\mathbf{0} = \boldsymbol{p}(t)\boldsymbol{\Lambda} \tag{3}$$

kde $\boldsymbol{p} = \boldsymbol{p}(\infty) = [p_1, p_2, ..., p_n]$ je hledaný vektor limitních pravděpodobností stavů a $\mathbf{0}$ je nulový vektor. Soustava (3) rovnic je lineárně závislá. Úspěšné vyřešení vyžaduje náhradu libovolné rovnice soustavy normalizační podmínkou $\sum_{i=1}^{n} p_i = 1$. Řešení opět vyjádříme maticovou rovnicí.

$$\boldsymbol{p} = \boldsymbol{v}\boldsymbol{\Lambda}_m^{-1}$$

kde $\boldsymbol{\Lambda}_m$ je matice, která vznikne z $\boldsymbol{\Lambda}$ vyplněním libovolně vybraného sloupce (označme ho m) samými jedničkami (doplněním normalizační podmínky). Vektor $\boldsymbol{v}$ má všechny prvky nulové, knomě m-tého, který je jedničkový.

Nejlepší představu o markovském náhodném procesu bez absorpčních stavů dává jeho graf přechodů. Představme si „značku“, která označuje aktuální stav systému a která se tedy pohybuje po grafu. V každém stavu, do kterého se dostane, setrvá náhodnou dobu s exponenciálním rozdělením s parametrem λ_i. Parametr má význam intenzity přechodu z odpovídajícího stavu. Střední doba T_i setrvání značky v i-tém uzlu je dána hodnotou $1/\lambda_i$. Z i-tého uzlu značka přechází náhodně některou výstupní hranou do dalšího uzlu. Pravděpodobnosti větvení jsou dány poměrem intenzit přechodů výstupních hran. Protože z každého uzlu grafu přechodů vedou cesty do všech ostatních uzlů, projde za dostatečně dlouhý časový interval značka každým uzlem. Limitní pravděpodobnosti všech stavů tedy budou mít nenulovou hodnotu.

Vzhledem k cyklickému charakteru náhodného procesu můžeme zavést další důležité veličiny. Nepodmíněná střední frekvence f_{ij} přechodů po obecné hraně grafu je průměrný počet přechodů po hraně ze stavu i do stavu j za jednotku času. Je dána vztahem

$$f_{ij} = p_i \lambda_{ij}$$

kde λ_{ij} je intenzita přechodů (podmíněná střední frekvence přechodů) po uvažované hraně grafu a p_i je limitní pravděpodobnost výchozího stavu hrany.

Střední frekvenci f_i průchodů stavem i získáme součtem frekvencí přechodů po výstupních hranách stavu i jako

$$f_i = p_i \lambda_i = \frac{p_i}{T_i}$$

kde λ_i je součet intenzit přechodů na výstupních hranách stavu i a T_i je střední doba setrvání ve stavu i.

Střední doba T_{ci} cyklu průchodů stavem i, tedy střední doba od jednoho příchodu do dalšího, je dána převrácenou hodnotou f_i.

$$T_{ci} = \frac{1}{f_i}$$

Na frekvenčním principu je rovněž možné provést přímou formulaci rovnic pro výpočet limitních pravděpodobností stavů. Pro každý uzel i se zřejmě musí rovnat střední frekvence příchodů a odchodů ze stavu. V této souvislosti se používá termín frekvenční rovnováha. Zápisem uvedené podmínky pro všechny stavy a doplněním normalizační podmínky dostaneme soustavu, kterou jsme dříve odvodili limitním přechodem z diferenciálních Kolmogorovových rovnic. Frekvenční rovnováha platí i pro libovolný řez grafu.

Markovské procesy bez absorpčních stavů lze využít například v oblasti spolehlivostních modelů obnovovaných systémů (každou poruchu lze nějak opravit, přechody jsou způsobeny událostmi typu porucha/oprava), modelů systémů hromadné obsluhy nebo modelů spolupracujících paralelních procesů. Základní podmínkou využití markovských náhodných procesů bez absorpčních stavů je opět pro každý stav exponenciální pravděpodobnostní rozdělení časových intervalů mezi časovým bodem příchodu a vznikem události iniciující přechod.

3 Současný stav

Tato práce navazuje na diplomovou práci Radka Hoštičky z roku 1999 [HOŠ99]. Radek Hoštička navrhl jazyk pro popis markovských modelů a vytvořil program Markov, který tento jazyk zpracovává. *Jazyk pro popis markovských modelů* budeme v dalším textu občas zkracovat na *popisovací jazyk*.

3.1 Jazyk pro popis markovských modelů

Jazykem pro popis markovských modelů vlastně popisujeme orientovaný graf. Každý uzel grafu reprezentuje jeden stav modelu, každá hrana z uzlu a do uzlu b reprezentuje přechod ze stavu a do stavu b.

Každá hrana je ohodnocena intenzitou přechodu. Každý uzel je (nepovinně) ohodnocen celým číslem, toto číslo přiřazuje uživatel a obvykle má nějakou návaznost na modelovaný systém — například všechny bezporuchové stavy jsou označeny jedničkou a všechny poruchové stavy jsou označeny dvojkou. Řešení modelu přiřadí každému uzlu ještě jedno ohodnocení — ustálenou (asymptotickou, limitní) pravděpodobnost stavu.

Popisovací jazyk umisťuje každý stav do n–rozměrného euklidovského prostoru. Jelikož jednotlivé souřadnice jsou celočíselné, jedná se vlastně o n–rozměrnou mřížku.

3.2 Popis příkazů

3.2.1 Přechod mezi stavy

V popisovacím jazyku se stav zapisuje jako jeho souřadnice v hranatých závorkách. Pro značení přechodu se používá symbol `->`. Za touto značkou následuje ohodnocení hrany (intenzita přechodu), což je reálné číslo. Tedy přechod s intenzitou 0,5 ze stavu [1, 1] do stavu [1, 2] se zapíše jako

```
[1,1] -> 0.5 [1,2];
```

3.2.2 Ohodnocení stavu

Pro přiřazení uživatelského ohodnocení se používá rovnítko. Stavu [1, 2] se přiřadí uživatelské ohodnocení 3 takto

```
[1,2] = 3;
```

3.2.3 Cykly

Jelikož markovské modely a jejich grafy mají často regulární strukturu, obsahuje popisovací jazyk konstrukci pro tvorbu cyklů. Každý cyklus má řídicí proměnnou,

počáteční a koncovou hodnotu. Před první iterací se do řídicí proměnné přiřadí počáteční hodnota. Každá iterace cyklu zvýší hodnotu řídicí proměnné o jedničku. Pokud hodnota řídicí proměnné dosáhne koncové hodnoty, iterování se ukončí.

Cyklus který propojí stavy 0 až 10 přechody s intenzitou 0,9 a stavy 0 až 9 ohodnotí číslem 5 vypadá takto

```
for (i; 0; 10) {
    [i]->0.9 [i+1];
    [i] = 5;
}
```

Cykly je samozřejmě možné do sebe vnořovat. Například dva vnořené cykly se použijí pro konstrukci dvojrozměrného modelu, jehož stavy tvoří jakousi trojúhelníkovou matici

```
for (i; 0; 20) {
    for (j; i, 20) {
        [i, j] -> 0.8 [i, j+1];
    }
}
```

3.2.4 Záhlaví

Před vlastním popisem přechodů musí být záhlaví. Na začátku skriptu popisovacícho jazyka musí být zadány dimenze a rozměry n–rozměrné mřížky, do které jsou stavy umísťovány

```
module model1 [200, 10];
```

Za vyhrazeným slovem `module` následuje název modelu a potom rozměry jednotlivých dimenzí. Dále popisovací jazyk umožňuje definici konstant, například

```
#define num 10
#define lambda 0.9
#define mi 1.0
#define val = 5
```

Tyto konstanty je možné používat místo číselných literálů dále ve skriptu. Tedy výše uvedený příklad cyklu je možné přepsat takto

```
for (i ; 0; num){
    [i]->lambda [i+1];
    [i] = val;
}
```

3.3 Příklad

Demonstrujme popisovací jazyk na triviálním ale úplném příkladu. Mějme model triviálního systému hromadné obsluhy. Do systému přicházejí požadavky s intenzitou $0,9\ s^{-1}$ a systém je obsluhuje s intenzitou $1,0\ s^{-1}$. Předpokládejme, že všechny časy mají poissonovské rozdělení. U tohoto systému budeme modelovat frontu — index stavu odpovídá počtu požadavků čekajících ve frontě. Fronta může být teoreticky nekonečná, my však její délku shora omezíme na 200. Tento model se popisovacím jazykem vyjádří takto

```
module bufferexample [200];
#define num 200
#define lambda 0.9
#define mi 1.0

for (i ;0; num-2){
    [i]->lambda [i+1];
}

for (i; 0; num-2){
    [i+1]->mi [i];
}
```

3.4 Program Markov

Vstupem programu Markov je skript v popisovacím jazyce. Na základě tohoto vstupu program v paměti sestaví markovský model. Z modelu sestaví soustavu lineárních algebraických rovnic a tu vyřeší. Kořeny rovnice jsou ustálené pravděpodobnosti jednotlivých stavů.

Program je napsán v jazyce C dle normy ANSI/ISO 9899-1990, díky tomu je velmi dobře přenositelný na různé počítače a operační systémy. K samotnému řešení soustavy lineárních algebraických rovnic používá knihovnu CLAPACK[1]. Díky tomu je možné řešit rozsáhlé modely, na nějakém speciálním výkonném hardwaru. (Dále viz zhodnocení výkonu a omezení 5.5, str. 42.)

K programu je též k dispozici jednoduché grafické uživatelské rozhraní naprogramované v jazyce C++ v prostředí Borland C++ Builderu. Toto prostředí je tedy k dispozici jen pro MS Windows. Grafická nadstavba obsahuje též prostředky pro vyhodnocení modelu. Tyto prostředky jsou však velmi omezené, program v podstatě umožňuje pouze zjišťovat ustálené pravděpodobnosti různých stavů a sčítat ustálené pravděpodobnosti stavů s určitým uživatelským ohodnocením.

[1] http://www.netlib.org/clapack/

3.4.1 Formáty výstupních souborů

Vstupem programu Markov je skript v popisovacím jazyce, výstupem je hned několik souborů, které obsahují jak spočtené ustálené pravděpodobnosti stavů tak i topologii modelu. Program uloží řešení do souborů o těchto příponách:

- *map* — Obsahuje mapovací vektor, který definuje jednoznačnou transformaci z prostoru stavů (n–rozměrné mřížky) do matice přechodů.
- *mtx* — Obsahuje popis matice přechodů mezi jednotlivými stavy markovského grafu.
- *val* — Obsahuje uživatelská ohodnocení stavů.
- *pbt* — Obsahuje ustálené pravděpodobnosti stavů markovského modelu, tedy hlavní výsledek výpočtu.
- *err* — Obsahuje protokol o parsování popisovacího jazyka a průběhu výpočtu.

S ohledem na přenositelnost jsou všechny soubory navrženy jako textové. Tím pádem je bezproblémová přenositelnost mezi systémy s architekturami procesoru littleendian a bigendian[2]. To umožňuje počítat a vyhodnocovat model na různých počítačích.

[2]Endianita je způsob uložení čísel v paměti, který definuje, v jakém pořadí se uloží jednotlivé bajty příslušného datového typu. Označuje se také jako pořadí bajtů. Více například http://en.wikipedia.org/wiki/Endianness.

4 Návrh dotazovacího jazyka

4.1 Motivace a cíle

Výstupem programu Markov je markovský model se spočtenými asymptotickými pravděpodobnostmi. Tento model je reprezentován jako orientovaný graf — každý stav modelu odpovídá uzlu grafu, přechody mezi stavy odpovídají hranám. Každý uzel má dvě ohodnocení, vlastní asymptotickou pravděpodobnost a dále celočíselné označení dané uživatelem. Toto označení obvykle odpovídá nějaké reálné vlastnosti modelovaného systému — například všechny stavy označené jedničkou jsou bezporuchové a stavy označené dvojkou jsou poruchové.

Každý uzel grafu je umístěn v n–rozměrné mřížce. Vždy je samozřejmě možné naskládat stavy do jednoho rozměru a indexovat jedním číslem, ale obvyklé je použít rozměrů více a jednotlivým rozměrům dát nějaký význam. Například jeden rozměr je počet zákazníků v samoobsluze a druhý délka fronty před pokladnou (viz příklad 6.2, str. 52).

Informace vyšší úrovně

Obvykle potřebujeme z vyřešeného modelu získat informace vyšší úrovně než jen pravděpodobnost nějakého stavu — střední délku fronty, střední dobu mezi poruchami atd. Typicky je to součet pravděpodobností nějaké třídy stavů (například všech poruchových) nebo mírně složitější výpočty. Například střední délka fronty v triviálním systému hromadné obsluhy se spočte jako vážený součet limitních pravděpodobností stavů, kde váhy odpovídají délce fronty v jednotlivých stavech.

Obvykle se takovéto výpočty provádějí v běžném kancelářském tabulkovém kalkulátoru. To je značně nepraktické z několika důvodů. Za prvé se v tabulkovém kalkulátoru špatně pracuje s n–rozměrnými daty, za druhé řešené modely mohou být příliš rozsáhlé (tisíce stavů).

Je tedy potřeba nějaký prostředek, který by extrahoval informace vyšší úrovně z modelu a to jednoduše a univerzálně. Tímto prostředkem bude nějaký jazyk ve kterém bude možné specifikovat potřebné výpočty z zobrazovat jejich výsledky.

Žádoucí jsou tyto vlastnosti:

- pohodlná práce v n–rozměrné mřížce
- výpis stavů (jejich pravěpodobností a uživatelského ohodnocení) daných vlastností
- řazení výpisů
- matematické operace (např. součet vážených pravděpodobností)

Možná řešení

Ke konstrukci jazyka je v zásadě možno přistoupit dvěma způsoby.

První možnost je vytvořit imperativní jazyk, ve kterém bude možné definovat potřebné výpočty. Takový jazyk by mohl být syntakticky konzistentní k jazyku pro popis markovských modelů. Umožnil by práci v n–rozměrné mřížce stejným způsobem, jaký je v popisovacím jazyce. Na druhou stranu požadované vlastnosti vyžadují vytvoření relativně komplexního jazyka. Kromě cyklů z popisovacího jazyka by byl potřeba podmíněný příkaz, práce s proměnnými pro ukládání mezivýsledků a nějaké funkce pro řazení atd.

Druhá možnost je vytvořit deklarativní jazyk. Pro člověka je často jednodušší popsat deklarativně představu výsledku, než přesný imperativní postup výpočtu. Zdrojová data mají regulární strukturu, lze se na ně dívat jako na tabulku. Nabízí se tedy možnost vytvořit dotazovací jazyk podobný SQL. Znalost SQL je široce rozšířena, takže zvládnutí jeho derivátu pro markovské modely by uživatelům nemělo činit potíže. Jazyk SQL dává návod jak přistupovat k vypisování údajů (`select`), výpočtům (výrazy a agregační funkce), řazení (`order by`) a seskupování (`group by`).

Zdá se, že druhý přístup bude jednodušší jak pro uživatele, tak pro implementaci a proto byl zvolen.

4.2 Dotazovací jazyk do markovských modelů (MMQL)

4.2.1 Charakteristika jazyka

Navrhněme tedy dotazovací jazyk do markovských modelů (Markov Model Query Language, dále MMQL). Jazyk je opravdu velmi podobný SQL, ovšem s tím rozdílem, že dotazovaná data nejsou v (jednorozměrné) tabulce, ale v n–rozměrné mřížce. S tím také souvisí největší rozdíl oproti SQL – MMQL obsahuje příkaz pro provádění cyklů (`for`, viz 4.3.5, str. 23). Cykly lze do sebe vnořovat (stejně jako v popisovacím jazyce) a slouží právě pro účinné procházení n–rozměrné mřížky.

Ostatní příkazy jsou přímými obdobami konstrukcí z SQL (a jejich dialektů, např. MySQL[3]) s různými úpravami a zjednodušeními. Jazyk umožňuje dotazy rekurzivně vnořovat.

Jazyk je navržen tak, aby jeho gramatika byla třídy $LL(1)$, takže je možné jednoduše (ručně) napsat parser jazyka. Parsování probíhá stylem rekurzivního sestupu.

Jazyk je navržen jako citlivý na velikost písmen (case sensitive). Tedy indentifikátory `abc` a `aBc` jsou dva různé identifikátory.

Pro jazyk je nevýznamný počet tzv. bílých znaků. Jakýkoliv počet po sobě následujících mezer, tabulátorů a znaků pro nový řádek se chová jako jeden oddělovač.

[3] http://www.mysql.com/

4.2.2 Klíčová slova

Všechna slova s předpřiřazeným významem lze v MMQL považovat za klíčová, protože význam těchto slov nelze žádným způsobem měnit. To samé lze tvrdit i o operátorech.

Slova pro vytváření jazykových konstrukcí: `select`, `as`, `from`, `for`, `where`, `group`, `order`, `limit`, `to`, `step`, `asc`, `desc`, `load`, `define`

Slova mající význam funkcí (včetně agregačních funkcí): `sum`, `mul`, `max`, `min`, `avg`, `count`, `exp`, `sin`, `cos`, `log`, `log10`, `abs`, `int`, `float`

Speciální symboly a operátory: `+ - * / % ; , " : and or not = >= <= <> > < := ( ) [ ]`

Speciální identifikátory: `p`, `val`

Předdefinované identifikátory: `e`, `pi`, `dual`

4.2.3 Dotazy a skripty

Jazyk je navržen jako dotazovací jazyk, takže stěžejní konstrukcí je dotaz. Ovšem dotaz je vždy prováděn v nějakém kontextu. Tím kontextem jsou data z modelu (modelů) a hodnoty různých předdefinovaných a předpočítaných proměnných.

Proto pomocí jazyka MMQL nevytváříme jenom dotazy, ale vlastně celé skripty. Skript jazyka MMQL se skládá z těchto částí:

1. načtení modelu
2. definice konstant
3. vlastní dotaz

Výsledkem skriptu jsou data vytvořená oním vlastním dotazem. Výsledek má formát tabulky — je to posloupnost řádek, každá řádka ze skládá z několika buněk. Buňky pod sebou tvoří sloupce.

4.3 Popis příkazů

4.3.1 Načtení modelu (příkaz load)

Skript může začínat příkazem (příkazy) pro načtení modelu. Příkaz začíná klíčovým slovem `load`, za kterým následuje umístění modelu na disku, klíčové slovo `as` a identifikátor, pod kterým se na načtený model budou odvolávat dotazy.

Umístění modelu na disku je uzavřeno v uvozovkách a nepředstavuje nějaký konkrétní diskový soubor. Řešení modelu je uloženo v několika souborech, jména konkrétních diskových souborů vzniknou tak, že se k umístění připojí příslušné přípony. Například

```
load "/home/paskma/models/model1" as m1
```

načte soubory `model1.pbt`, `model1.mtx`, `model1.val` a `model1.map` z adresáře
`/home/paskma/models` a načtený model zpřístupní pod jménem `m1` dotazům. O formátech souborů pojednává odstavec 3.4.1, str. 16.

Příkazů pro načtení modelu může být několik za sebou, odděleny jsou bílým znakem.

4.3.2 Konstanty (příkaz define)

Před vlastními dotazy můžou být uvedeny definice konstant. Technicky vzato jsou konstanty v MMQL vlastně proměnné. Jazyk MMQL a jeho současná implementace nijak nebrání modifikaci těchto proměnných, na druhou stranu ale neposkytuje žádné prostředky pro tuto modifikaci. Takže v současné implementaci lze na tyto proměnné nahlížet jako na konstanty.

Každá definice začíná klíčovým slovem `define`, poté následuje identifikátor, znak pro přiřazení a hodnota konstanty. Definice je ukončena středníkem. Hodnota konstanty může být libovolný výraz, pro tento výraz je zaručeno, že bude vyhodnocen právě jednou. Toho lze využít pro optimalizace dotazů, více viz kapitolu 4.3.10 o vnořených dotazech, str. 26.

Například:

```
define polomer := 3.8;
define obvod := 2 * pi * polomer;
```

4.3.3 Výraz

Srdcem jazyka MMQL jsou aritmetické výrazy. MMQL obsahuje jeden univerzální aritmetický výraz, který slouží jak k vlastním aritmetickým výpočtům, tak i ke konstrukci logických podmínek. Výraz se skládá z číselných literálů, identifikátorů a operátorů. Celkové pojetí se podobá aritmetickým výrazům, jak je známe z jazyka C.

Identifikátory obsažené ve výrazu můžou být tří typů:

- Předdefinované konstanty `pi` a `e`.
- Uživatelem definované konstanty.
- Speciální identifikátory `p` a `val`. Tyto identifikátory mají hodnoty ustálené pravděpodobnosti a uživatelského ohodnocení nějakého stavu.

O identifikátorech též pojednává odstavec 4.5, str. 30 o tabulce symbolů.

Aritmetické výpočty mohou probíhat nad celými nebo nad reálnými čísly (přesněji řečeno nad jejich počítačovými reprezentacemi). Při vyhodnocování

se každý výraz považuje implicitně za celočíselný, pokud je ovšem nějaký jeho podvýraz reálný, pak je i celý výraz reálný. Jedná se tedy o analogii k automatické typové konverzi z jazyka C.

Protože výrazy slouží i pro konstrukci logických podmínek, je zavedena následující konvence: pokud je výraz vyhodnocen jako nula, je nepravdivý, jinak je pravdivý. Výsledky logických výrazů jsou celočíselné, protože logické operátory `and`, `or` a `not` vždy vracejí celé číslo (a to jedničku nebo nulu). Přehled operátorů je v tabulce 1.

Operátor	Priorita	Arita	Popis
`or`	1	2	Vrátí 1, pokud je alespoň jeden z argumentů nenulový, jinak vrátí 0.
`and`	2	2	Vrátí 1, pokud jsou oba argumenty nenulové, jinak vrátí 0.
`=`	3	2	Vrátí 1, pokud se argumenty rovnají, jinak vrátí 0.
`<`	3	2	Vrátí 1, pokud je levý argument menší než pravý argument, jinak vrátí 0.
`>`	3	2	Vrátí 1, pokud je levý argument větší než pravý argument, jinak vrátí 0.
`<=`	3	2	Vrátí 1, pokud je levý argument menší než pravý argument nebo je mu roven, jinak vrátí 0.
`>=`	3	2	Vrátí 1, pokud je levý argument větší než pravý argument nebo je mu roven, jinak vrátí 0.
`<>`	3	2	Vrátí 1, pokud levý argument není roven pravému argumentu, jinak vrátí 0.
`+`	4	2	Vrátí aritmetický součet svých argumentů.
`-`	4	2	Vrátí aritmetický rozdíl svých argumentů.
`*`	5	2	Vrátí aritmetický součin svých argumentů.
`/`	5	2	Vrátí podíl svých argumentů.
`%`	5	2	Vrátí zbytek po dělení svých argumentů. Pokud argumenty nejsou celá čísla, jsou na celá čísla převedena.
`not`	5	1	Vrátí 0, pokud je argument nenulový, jinak vrátí 1.

Tabulka 1: Přehled operátorů

Dále je možné ve výrazu používat matematické funkce. Všechny funkce konvertují svůj argument na reálné číslo a reálný je rovněž výsledek. Výjimkou jsou

funkce `abs` a `pow`, které za jistých podmínek můžou vrátit celé číslo, a funkce `int`, která vrátí celé číslo vždy. Přehled funkcí je uveden v tabulce 2.

Funkce	Popis
`abs(x)`	Vrátí absolutní hodnotu čísla x.
`sin(x)`	Vrátí sinus čísla x.
`cos(x)`	Vrátí cosinus čísla x.
`exp(x)`	Vrátí e^x.
`log(x)`	Vrátí přirozený logaritmus čísla x.
`log10(x)`	Vrátí dekadický logaritmus čísla x.
`pow(x,y)`	Vrátí x^y. Obecně vrací reálné číslo. Celé číslo vrací pokud x i y jsou celá čísla a y je navíc nezáporné a výsledek lze reprezentovat jako celé číslo na daném počítači.
`int(x)`	Pokud je číslo x reálné, konvertuje ho na celé (ořízne desetinnou část), jinak vrátí svůj argument beze změny.
`float(x)`	Pokud je číslo x celé, konvertuje ho na reálné, jinak vrátí svůj argument beze změny.

Tabulka 2: Přehled funkcí

4.3.4 Sloupce (příkaz select)

Příkaz `select` uvozuje celý dotaz. Součástí dotazu můžou být i další příkazy (`group`, `where`, `order`, `limit`), ale ty jsou nepovinné.

Za klíčovým slovem `select` následuje seznam vypisovaných sloupců oddělených čárkou. Každý sloupec je výraz nebo výraz uzavřený v agregační funkci (viz 4.3.7, str. 24).

Za seznamem sloupců následuje klíčové slovo `from` a identifikátor — jméno modelu. Jméno modelu musí korespondovat s některým modelem nahraným příkazem `load` (viz 4.3.1, str. 19). Pokud pro dotaz žádný model nepotřebujeme, je možné použít speciální jméno `dual`. Například toto je korektní doaz:

```
select 1 + 1, pi * e from dual
```

Za každým sloupcem může volitelně následovat klíčové slovo `as` a identifikátor. Pokud sloupce explicitně nepojmenujeme, dostanou automatická jména: `col1` pro první sloupec, `col2` pro druhý sloupec atd. Identifikátory sloupců mají dvě funkce — za prvé slouží k zpřehlednění výpisů, za druhé se na sloupce odvoláváme při řazení výpisů (viz kapitolu 4.3.8 o řazení, str 26). Příklad pojmenovaných sloupců:

```
select 4 as ctverka, 2 + 3 as petka from dual
```

4.3.5 Cykly (příkaz for)

Cykly jsou jediná část MMQL, která nemá obdobu v SQL. Cykly v MMQL jsou přímou obdobou cyklů z jazyka pro popis markovských modelů. Typicky se jich využívá pro procházení n–rozměrné mřížky — adresaci stavů.

Cyklus má vždy jednu řídicí proměnnou, horní a dolní mez. Na začátku se do řídicí proměnné přiřadí hodnota dolní meze. Proběhne iterace, po každé iteraci se hodnota řídicí proměnné zvýší o jedničku. Iterování je ukončeno poté, co hodnota řídicí proměnné dosáhne horní meze. Podrobněji viz 4.6, str. 30 o algoritmu provedení dotazu.

Nejjednodušší příklad je výpis číselné řady (od 1 do 10).

```
select i from dual for i := 1 to 10
```

Praktický příklad může být výpis hodnot jednoduchého modelu. Předpokládejme, že model má 20 stavů, které jsou uloženy v jedné dimenzi a očíslované 0 až 19.

```
select p[i], val[i] from model1 for i := 0 to 19
```

Cyklů je možné vložit více do sebe. Přičemž vnitřní cyklus může ve výrazech pro své meze využít řídicí proměnnou vnějšího cyklu. To je velmi důležitá vlastnost (a vlastně i důvod pro existenci cyklů v MMQL), protože stavy často bývají uspořádány v dvourozměrné mřížce a tvoří jakousi trojúhelníkovou matici (viz příklad 6.2, str 52). Následující příklad prochází trojúhelníkovou matici velikosti 5.

```
select p[i, j], val[i, j] from model2 for i := 1 to 5, j := 1 to i
```

Pokud je dolní a horní mez stejná, provede se jenom jedna iterace. Vlastně se jenom inicializuje řídicí proměnná a vyhodnotí dotaz. Takovýto zápis

```
select i from dual for i := 1 to 1
```

je pak možné zkrátit na

```
select i from dual for i := 1
```

4.3.6 Podmínky (příkaz where)

Podmínky (příkaz `where`) slouží k filtrování výpisu, jedná se o přímou obdobu k `where` jazyka SQL. Za klíčovým slovem `where` následuje libovolný výraz. Pokud MMQL dotaz obsahuje podmínku, jsou výsledkem dotazu pouze takové řádky výsledné tabulky, pro které je splněna podmínka. Podmínka je splněna tehdy, pokud hodnota výrazu za `where` je nenulová. Pokud u dotazu není podmínka uvedena, zachází se s dotazem jako by byla podmínka vždy splněna.

Následující příklad vypíše posloupnost celých čísel z intervalu $[-5, 5]$, ale díky podmínce vynechá nulu.

```
select i from dual for i := -5 to 5 where i
```

Výpis sudých čísel od nuly do deseti se vyjádří následovně:

```
select i from dual for i := 0 to 10 where i % 2 = 0
```

Praktičtější příklad může být výpis indexů a ohodnocení stavů, jejichž pravděpodobnost je větší než nějaká mez:

```
select i, val[i] from model5 for i := 1 to 100 where p[i] > 0.1
```

4.3.7 Seskupování (příkaz group)

Často potřebujeme získat z modelu informace vyšší úrovně, než jenom výpis hodnot nějakých stavů. Velmi často získání takovéto informace znamená pracovat s nějakými třídami stavů (například sečíst pravděpodobnosti všech bezporuchových stavů). Takových výpočtů můžeme elegantně docílit použitím seskupování a agregačních funcí.

Součástí dotazu může být výraz pro seskupování, jedná se o analogii k příkazu `group by` z SQL. V MMQL následuje výraz pro seskupování za klíčovým slovem `group`. Tento výraz je jenom jeden, to je rozdíl oproti SQL, kde může být výrazů víc, oddělených čárkami.

Seskupování probíhá tak, že se seskupí všechny řádky výpisu, pro které nabývá seskupovací výraz sejné hodnoty. Každému řádku konečného výpisu pak odpovídá jedna hodnota seskupovacího výrazu. V následujícím příkladu je pro každou iteraci určena unikátní hodnota seskupovacího výrazu, proto je výsledek dotazu takový, jako bychom žádné seskupování nepožadovali:

```
select i from dual for i := 1 to 5 group i
```

Jak vypadají hodnoty jednotlivých sloupců, záleží na použití agregační funkce. Agregační funce vlastně spočte nějakou hodnotu ze skupiny řádek. Pro každý sloupec může být specifikována jiná agregační funkce.

Pokud neuvedeme žádnou agregační funkci, použije se implicitní. Implicitní agregační funkce vrátí poslední hodnotu, která do ní byla vložena. Ilustrujme na příkladu:

```
select i from dual for i := 1 to 5 group 1
```

V tomto příkladu se provádí seskupování dle konstantního výrazu (jedničky), všech pět řádek tedy padne do jedné skupiny. Výsledek dotazu bude jedna řádka s hodnotou 5.[4]

Agregační funkce	Popis
`min`	Vrátí nejmenší z hodnot.
`max`	Vrátí největší z hodnot.
`sum`	Vrátí součet hodnot.
`mul`	Vrátí součin hodnot.
`avg`	Vrátí aritmetický průměr hodnot.
`count`	Vrátí počet hodnot.

Tabulka 3: Agregační funkce

Seskupování používáme pro nějaké užitečné akce na skupinách řádek. Tyto akce nám zajistí agregační funkce, jejichž přehled je v tabulce 3.

Následující příklad spočte aritmetickou, harmonickou a geometrickou řadu pro i od 1 do 1000.

```
select
    min(i) as OD,
    max(i) as DO,
    sum(i) as ARIT,
    sum(1.0 / i) as HARM,
    sum(1.0 / pow(2, i)) as GEOM
from dual for i := 1 to 1000 group 1
```

Výpis stavu s největší pravděpodobností z nějakého dvojrozměrného modelu vypadá takto:

```
select max(p[i, j]) from model3
for i := 1 to 100, j := 1 to 200
```

Typickým praktickým příkladem je spočtení střední délky fronty v SHO[5]. Střední délka fronty v triviálním SHO se spočte jako vážený součet limitních pravděpodobností stavů, kde váhy odpovídají délce fronty v jednotlivých stavech. (Viz příklad 6.1, str. 48.) Předpokládejme, že máme model o 100 stavech číslovaných od 0 do 99, kde index má význam délky fronty pro určitý stav. Střední délka fronty se pak spočte:

```
select sum(i * p[i]) as Lw
from model4 for i := 0 to 99 group 1
```

[4]Jazyk MMQL ve skutečnosti nezaručuje, která hodnota do implicitní agregační funkce padne poslední, zvláště pro iterace o velkém počtu kroků můžou teoreticky pořadí narušit případné optimalizace.

[5]Systém hromadné obsluhy

4.3.8 Řazení (příkaz order)

Výpisy (zvláště delší) je třeba řadit, neboť nás typicky zajímá nějaká extrémní hodnota, například nejpravděpodobnější stav. V MMQL k řazení slouží příkaz `order`. Za klíčovým slovem `order` je seznam identifikátorů sloupců oddělený čárkami. Identifikátory sloupců mohou být explicitně (použitím klíčového slova `as`) nebo implicitně určené. Implicitní identifikátory jsou `col1` pro první sloupec, `col2` pro druhý sloupec atd. Více viz kapitolu 4.3.4, str. 22.

Identifikátory sloupců za klíčovým slovem `order` slouží ke konstrukci porovnávací funkce, která porovnává řádky výpisu. Tuto porovnávací funkci pak používá řadící algoritmus. Největší váhu má první identifikátor, pokud jsou hodnoty, na které se odvolává, stejné, použije se druhý identifikátor atd.

Za každým identifikátorem může následovat klíčové slovo `asc` nebo `desc` specifikující vzestupné nebo sestupné řazení pro příslušný sloupec. Pokud není žádné z těchto klíčových slov uvedeno, řadí se vzestupně.

Například výpis pravděpodobností stavů od nejpravděpodobnějšího po nejméně pravdepodoný vypadá následovně:

```
select p[i] as PROB from model5
for i := 0 to 99 order PROB desc
```

4.3.9 Zkrácení výpisu (příkaz limit)

Příkaz `limit` slouží k omezení délky výpisu. Tato konstrukce byla převzatá z dialektu SQL používaného databází MySQL.

Za klíčovým slovem `limit` následuje výraz. Hodnota výrazu udává maximální počet řádek ve výpisu.

Následující příklad vypíše pět nejpravděpodobnějších stavů.

```
select p[i] as PROB from model5
for i := 0 to 99 order PROB desc limit 5
```

4.3.10 Vnořený dotaz

MMQL umožňuje používat příkaz `select` i rekurzivně — vnořený dotaz. Na kterémkoliv místě, kde je číslo nebo identifikátor (obecně vhodný terminální symbol gramatiky), může být místo tohoto čísla příkaz `select`. Každý takový select se navíc může dotazovat do jiného modelu — toho lze případně využít pro různá porovnávání modelů.

Vnořený dotaz musí vrátit alespoň jeden řádek, jinak vykonání nadřazeného dotazu skončí chybou. Výsledek vnořeného dotazu se převede na skalár — použije se první sloupec prvního řádku. Následující dotaz

```
select 1 + 1 from dual where 1
```

lze tedy přepsat jako:

```
select
    select 1 from dual
    + select 1 from dual
from dual
where select 1 from dual
```

Vnořený select společně s agregačními funkcemi je poměrně mocný prostředek pro různé výpočty. Například posloupnost částečných součtů geometrické řady od jedné do sto

$$\left\{\sum_{i=1}^{N}\frac{1}{2^i}\right\}_{N=1}^{100}$$

se zrealizuje tímto dotazem

```
select
    N,
    (select sum(1.0 / pow(2, i)) from dual
        for i := 1 to N group 1)
from dual for N := 1 to 100
```

Praktičtějším příkladem může být vypsání všech stavů, které mají nadprůměrnou pravděpodobnost

```
select i, p[i]  from model6 for i := 0 to 99
where p[i] >=
    (select avg(p[j]) from model6 for j := 0 to 99 group 1)
```

Nutno podotknout, že podmínka se vyhodnocuje s každou iterací — a to platí i pro vnořený select v ní obsažený. Uvedený dotaz má tedy časovou složitost $O(n^2)$, kde n je počet stavů modelu. Pokud chceme nějaký výraz vyhodnotit jenom jednou, uložíme si jeho hodnotu do proměnné. Uvedený dotaz tedy lze jednoduše zoptimalizovat na lineární časovou složitost

```
define average := select avg(p[j])
from model6 for j := 0 to 99 group 1;

select i, p[i]  from model6 for i := 0 to 99
where p[i] >= average
```

4.4 Gramatika

Gramatika je popsána v rozšířené Backus–Naurově formě (BNF). Neterminální symboly jsou uzavřeny ve špičatých závorkách, terminální symboly jsou uzavřeny v uvozovkách. Nepovinné použití symbolu (symbolů) je označeno uzavřením mezi hranaté závorky, násobný výskyt (včetně 0–násobného) je označen uzavřením mezi složené závorky.

```
<script> ::=
    {<import>} {<variable>} <select_expression>

<import> ::= "load" <filename> "as" <ident>

<variable> ::= "define" <ident> ":=" <condition_E> ";"

<select_expression> ::=
    "select" <columns>
    "from" <ident>
    ["for" <for_conditions>]
    ["where" <condition_E>]
    ["group" <grouping>]
    ["order" <ordering>]
    ["limit" <condition_E>]

<condition_E> ::= <condition_T> <condition_E2>
<condition_E2> ::= "or" <condition_T><condition_E2> | e
<condition_T> ::= <condition_F> <condition_T2>
<condition_T2> ::= "and" <condition_F><condition_T2> | e
<condition_F> ::= <expression> | <expression> <condition_test2>

<condition_test2> ::=
    "=" <expression>
    | "<" <expression>
    | ">" <expression>
    | "<=" <expression>
    | ">=" <expression>
    | "<>" <expression>

<expression> ::= <T><E2> | +<T><E2> | -<T><E2>
<E2> ::= + <T><E2> | e
<E2> ::= - <T><E2> | e
<T> ::= <F><T2>
<T2> ::= * <F><T2> | e
```

```
<T2> ::= / <F><T2> | e
<T2> ::= % <F><T2> | e
<F> ::=
    "not" <F> | <number> | <float_number> | <ident>
    | "("<condition_E>")"
    | <select_expression>
    | <model_element>
    | <unary_function>"("<condition_E>")"
    | <binary_function>"("<condition_E>, <condition_E>")"

<model_element> ::= "p" <indexer> | "val" <indexer>
<indexer> ::= "["<indexer_body>"]"
<indexer_body> ::= <condition_E>{, <condition_E>}


<unary_function> ::=
    "abs" | "sin" | "cos" | "exp" | "log" | "log10"
    | "int" | "float"

<binary_function> ::= "pow"

<for_conditions> ::= <for_condition> {, <for_condition>}
<for_condition> ::= ident ":=" <condition_E> ["to" <condition_E>]

<columns> ::=  <column> {, <column>}
<column> ::= <column_expression> ["as" <ident>]
<column_expression> ::=
    <group_function> "(" <condition_E> ")"
    | <condition_E>

<group_function> ::= "sum" | "mul" | "max" | "min" | "avg"
<grouping> ::= <condition_E>

<ordering> ::= <order> {,<order>}
<order> ::= <ident> ["asc" | "desc"]


<number> ::= <num> {<num>}
<float_number> ::= <number> "." {<num>} ["e" ["+" | "-"] <number>]
<ident> ::= <alpha> {<alpha> | <num>}

<alpha> ::= "a" | ... | "z" | "A" | ... | "Z"
<num> ::= "0" | ... | "9"
```

Poznámka: `<filename>` představuje jméno souboru uzavřeného v uvozovkách. Toto jméno je dle konvencí operačního systému, na kterém program běží.

4.5 Tabulka symbolů

Při provádění dotazu se pracuje s tabulkou symbolů. V této tabulce jsou proměnné a jejich aktuální hodnoty. (Jak bylo uvedeno v kapitole 4.3.2, konstanty jsou v MMQL vlastně proměnné. A je tomu právě proto, že tabulka symbolů zachází se všemy symboly stejně.) Tabulka na začátku obsahuje dva symboly (`pi` a `e`) s příslušnými hodnotami (Ludolfovo číslo a Eulerovo číslo).

Tabulka je jenom jedna, globální – při práci s vnořenými dotazy je potřeba pamatovat na možnou kolizi symbolů a volbou jedinečných identifikátorů se jí vyhnout.

Další dva speciální symboly jsou `p` a `val`. Tyto symboly jsou bránami do dotazovaného modelu. Za každým z těchto symbolů vždy následují souřadnice uzavřené v hranatých závorkách. Souřadnice adresují stav v n–rozměrné mřížce. Symbol `p` obsahuje asymptotickou pravděpodobnost (reálné číslo), `val` obsahuje uživatelské ohodnocení (celé číslo) daného stavu.

4.6 Algoritmus provedení skriptu

Vyhodnocení dotazu je definováno tímto algoritmem: (Přesný kód algoritmu je uveden v příloze B.)

1 Pro každý sloupec se zinicializuje agregační funkce, pokud není definovaná, použije se implicitní agregační funkce. Pro potřeby agregace se alokují *sloty*, každý slot pro jednu hodnotu výrazu pro seskupování. Sloty jsou uloženy v hašovací tabulce, klíčem je hodnota seskupovacího výrazu. Vložení hodnoty do slotu probíha skrze agregační funkci.

2 Pokud není žádný cyklus:

  A Pokud podmínka není zadána a nebo je zadána a zároveň vyhodnocena jeko nenulová:

    - Vyhodnoť všechny sloupce a vlož do slotu 0.

  B Jdi na krok 3

  jinak:

  A Nastav řídicí proměnné všech cyklů na dolní mez, ulož tyto hodnoty do tabulky symbolů.

  B Nastav čítač průchodů na 0.

C Pokud je konec provádění cyklů (vnější čítač přesáhl horní mez), jdi na krok 3

D Pokud podmínka není zadána a nebo je zadána a zároveň vyhodnocena jako nenulová:

- Pokud je zadán výraz pro seskupování:
  - Vyhodnoť všechny sloupce a vlož do slotu daného hodnotou seskupovacího výrazu.

  jinak:
  - Vyhodnoť všechny sloupce a vlož do slotu daného čítačem průchodů.

E Inkrementuj čítač průchodů.

F Inkrementuj řídicí proměnné cyklů.

G Jdi na krok 2.C

3 Vyber všechny záznamy ze slotů a vlož do pole.

4 Pokud je zadáno řazení, seřaď záznamy v poli dle zadaných kritérií.

5 Pokud je zadáno zkrácení výpisu, zkrať pole záznamů na požadovanou délku.

5 Realizace programu

Jazyk pro dotazování do markovských modelů je pochopitelně nutné zpracovávat nějakým programem. Jelikož tento program je přímým následovníkem programu Radka Hoštičky Markov, nazvěme jej Markov2. Program Markov2 bude obsahovat všechny funkce programu Markov a ještě navíc implementaci parseru a interpretu jazyka MMQL.

5.1 Cíle

Cílem je vytvořit program, který pokryje celý proces návrhu, řešení a vyhodnocení markovského modelu. Uživatel zadá model popisovacím jazykem, pro tento model nechá spočíst ustálené pravděpodobnosti stavů a ze spočteného modelu dotazovacím jazykem vytáhne požadované informace. To všechno v jednom jednoduchém prostředí.

Program je určen primárně pro studenty. Nepočítá se s využitím programu pro nějaké konkrétní příklady z praxe nebo dokonce komerční využití. Program musí efektivně řešit typické školní příklady. „Efektivností“ je tedy v tomto případě myšleno jednoduché ovládání — aby jednoduché příklady šlo řešit jednoduše. Pokud jsou studenti nuceni pracovat se zbytečně komplexním programem, zbytečně je to odvádí od jádra problému, který se učí řešit.

Markov2 tedy bude obsahovat kód původního programu Markov, což ovlivní volbu použitých technologií.

Dalším cílem je dobrá přenositelnost, a to včetně grafického uživatelského rozhraní. Autor této práce provozuje na své pracovní stanici primárně operační systém GNU/Linux a chce logicky program vyvíjet ve svém domácím prostředí. Přestože GNU/Linux používá ne zcela zanedbatelná část studentů, velká většina používá primárně Microsoft Windows. Je tedy jasné, že program Markov2 musí být provozovatelný jak pod Linuxem, tak pod Windows.

5.2 Volba technologií

5.2.1 Volba programovacího jazyka

Jako programovací jazyk pro implementaci programu Markov2 byl zvolen jazyk C++. Původní markov je napsáv v čistém C, bylo tedy potřeba nový kód integrovat s kódem v tomto jazyce. Byly zvažovány tyto možnosti:

Čisté C

Z hlediska integrace by bylo nejjednodušší napsat nový kód také v jazyce C. Programy v C jsou obvykle velmi efektivní a rychlé, nicméně programování v něm je často neefektivní a pomalé. Programátor se musí starat o velké množství detailů, což vede k většímu počtu chyb. Přestože je v C možné do jisté míry programovat

objektově, není tento přístup jazykem podporován a proto je takováto „emulace“ objektů pracná. Obzvláště bolestivá je práce s řetězci, jazyk C v podstatě řetězce nemá, má jen pole a trochu syntaktického cukru (uvozovky) pro práci s poli znaků. Takto k chybám náchylná práce s řetězci asi nevadí nikde víc než při ručním psaní parseru.

Java a C#

Další možnost je použít pro nový kód programovací jazyk Java. Java nabízí velký programátorský komfort, dobrou práci s objekty, automatickou správu paměti. Java brání náhodnému přepisování paměti a tím vzniku obtížně odhalitelných chyb známých například z jazyka C.

V Javě ale není bezproblémová integrace s původním céčkovým kódem. Pro komunikaci mezi JVM[6] a nativním kódem se používá JNI (Java Native Interface). Spojení přes JNI není úplně jednoduché, musí se řešit rozdílné datové typy, rozdílný způsob práce s pamětí. To vyžaduje vytváření speciálního spojovacího kódu na straně jazyka C. Pro uživatele to taky představuje jisté komplikace — musel by instalovat univerzální program v Javě a k němu přidat správnou nativní knihovnu pro svůj systém. Tyto komplikace nejsou slučitelné s požadavkem na jednoduchost.

V Javě by bylo možné jednoduše naprogramovat grafické uživatelské rozhraní postavené na Swingu.

C# je ještě o něco lepší jazyk než Java, ale sdílí s ní i její nevýhody. Pro spolupráci s nativním kódem se používá P/Invoke[7], toto řešení je možná o něco jednodušší než javovská obdoba ale jedná se o naprosto stejný druh komplikací. Proti jazyku C# hovoří též fakt, že jeho současná implementace v GNU/Linuxu (Mono[8]) je poměrně mladá a nevyzrálá.

C++

Jazyk C++ umožňuje „bezešvou“ integraci s kódem v C. Používání standardní knihovny jazyka C z C++ je naprosto přirozené. Stejně tak spojování objektových souborů z C a C++ do jednoho spustitelného souboru je běžná praxe. Navíc platí, že „rozumně“ napsaný kód v C lze přímo přeložit překladačem C++. (Program Markov byl na překladač C++ přenesen s několika triviálními zásahy, které se držely v mezích jazyka C.)

C++ poskytuje vyšší míru abstrakce než čisté C. Objektové programování je zde sice na nižší úrovni než v Javě/C#, ale už prostá existence zapouzdřování, dědičnosti a polymorfismu dává jazyku dostatečnou sílu. Díky standardní ša-

[6] Java Virtual Machine
[7] Platform Invocation Services
[8] http://mono-project.com

blonové knihovně (STL[9]) má programátor připraveny k použití univerzální kontejnery. STL se též stará o řetězce, jejichž používání je odolné chybám. Díky šablonovému programování lze i zlepšit správu paměti — šablony (templates) umožňuje vytvářet „chytré“ ukazatele — třídy používající počítání referencí k automatickému uvolnění objektu, na který ukazují.

Nutno říct, že intenzivní používání STL má i své nevýhody. Především při ladění se nepříznivě projevuje to, že i (v jiných jazycích) primitivní typy jsou vytvářeny pomocí STL a mnoha vrstev abstrakce. Například řetězec je v Javě primitivní typ. V C++ je textový řetězec speciální instancí pro obecné řetězce (tedy nikoliv jen pro znaky). Jelikož šablony v C++ se vyhodnocují v době překladu, při ladění debugger není schopen takový řetězec vypsat. Stejný problém nastane, když programátor přestane používat klasické pole z C a začne používat jeho obdobu ze STL (která má zase tu výhodu, že kontroluje meze).

I přes některé komplikace je C++ vhodným kompromisem pro implementaci programu Markov2.

5.2.2 Volba grafického uživatelského rozhraní

Program Markov2 má obsahovat grafické uživatelské rozhraní. Toto rozhraní má být multiplatformní (musí být k dispozici minimálně pro Microsoft Windows a GNU/Linux). Je tedy potřeba zvolit nějakou vhodnou knihovnu. Tato knihovna navíc musí být přístupná z C++.

Jelikož vývoj byl plánován na platformě GNU/Linux, bylo potřeba zvolit nějaký GUI toolkit pro tuto platformu a vyhodnotit vhodnost použití a možnost přenesení na platformu Windows. Dále bylo potřeba vyhodnotit případné licenční nároky těchto knihoven.

Dva nejpopulárnější GUI toolkity v GNU/Linuxu jsou Qt a GTK+. Z těchto dvou možností bylo vybráno GTK+, především kvůli licenci.

Qt

Qt je toolkit vyvíjeny firmou Trolltech[10]. Qt je napsáno v C++, má velmi komfortní a výkonné API[11]. Qt je široce používáno, asi nejznámější aplikace je integrované desktopové prostředí pro GNU/Linux — KDE[12]. Díky tomu, že za Qt stojí komerční firma, má Qt kvalitní dokumentaci. Qt je k dispozici pro i pro Microsoft Windows a je na této platformě dostatečně stabilní.

S Qt se obvykle nepracuje se standardním C++, ale (převážně z historických důvodů) s jeho rozšířením od Trolltechu. Do standardního C++ se pak takový kód převede použitím preprocesoru. To je poměrně nepříjemné. Největší nevýhodou

[9] Standard Template Library
[10] http://www.trolltech.com
[11] Application Program Interface
[12] http://kde.org

Qt je jeho licence — knihovna je licencována pod agresivní licencí GPL[13]. Použití knihovny licencované GPL znamená, že i program knihovnu používající musí být licencován pod GPL. Pro program Markov ale bude vhodnější nějaká volnější akademická licence.

GTK+ a GTKmm

GTK+[14] vzniklo původně jako GUI toolkit pro editor obrázků GIMP[15]. GTK+ je vyvíjeno komunitou kolem otevřeného softwaru, sponzorováno komerčními firmami. GTK+ používá druhý velký hráč na poli GNU/Linuxových desktopů – GNOME[16]. Podobně jako Qt, i GTK+ bez problémů běhá na i na platformě Microsoft Windows.

Samotné GTK+ je napsáno v čistém C, ale je navrženo objektově. Existuje knihovna GTKmm[17], což je tenký C++ obal nad GTK+. Bylo by možné z C++ používat i holé GTK+, ale kód nad GTKmm je mnohem kratší a přehlednější a tím pádem méně náchylný k chybám. Bohužel kvalita dokumentace ke GTKmm není valná. Naštěstí lze využít toho, že GTKmm nemá žádnou funkcionalitu a všechny vlastnosti mají přímou obdobu v GTK+ — vždy lze tedy nahlédnout do dokumentace k čistému GTK+ a tato dokumentace je kvalitnější.

GTK+ i GTKmm jsou distribuovány pod licencí LGPL[18], která je oproti GPL míň agresivní. Na licenci programu používajícího LGPL knihovnu nejsou kladeny žádné nároky.

5.3 Softwarová architektura

5.3.1 Organizace zdrojových kódů

Celý program (projekt) se skládá ze tří nezávislých částí. Tomu taky odpovídá organizace zdrojových kódů — ty jsou uloženy ve třech adresářích.

Parser a interpret jazyka pro popis markovských modelů je uložen v adresáři `markov`. Jedná se o kódy z původního programu Markov, jsou přeložitelné překladačem čistého C a nemají žádné další závislosti.

Parser a interpret jazyka pro dotazování do markovských modelů je uložen v adresáři `mmql`. Tato část je napsána ve standardním C++ a závisí pouze na standardní knihovně C++.

[13]GNU General Public Licence, http://www.gnu.org/copyleft/gpl.html
[14]http://www.gtk.org
[15]http://www.gimp.org
[16]http://www.gnome.org
[17]http://www.gtkmm.org
[18]GNU Lesser General Public License, http://www.gnu.org/licenses/lgpl.html

Grafické uživatelské rozhraní je uloženo v adresáři `gui`. Tyto zdrojové kódy jsou napsány v C++ a používají funkce z kódu v ostatních adresářích. Kromě toho tento kód závisí na knihovně GTKmm (používá knihovní hlavičkové soubory).

5.3.2 Parsery

Stěžejní částí programu Markov2 jsou parser a interpret jazyka MMQL.

Jelikož gramatika jazyka MMQL je třídy $LL(1)$, byl parser programován ručně. Nebylo tedy použito některého z generátorů parserů, jakým je například Yacc nebo Bison[19]. Parsování probíhá metodou rekurzivního sestupu, což je obvyklá metoda pro gramatiky třídy $LL(1)$. Každému přepisovacímu pravidlu odpovídá jedna procedura, každá procedura generuje uzel derivačního stromu. Procedury se mezi sebou (rekurzivně) volají, podle toho, jaká data jsou na vstupu.

Parser jazyka MMQL byl rozdělen do tří částí, jako důsledek toho, že jeho programování probíhalo po částech. Při programování bylo částečně využito agilních metodik, přesněji evolučního přístupu k programování.

První část parseru je implementována ve třídě `Parser`. Tato část implementuje jenom omezenou podmnožinu gramatiky jazyka MMQL. Konkrétně jde o aritmetický výraz (viz neterminální symbol `expression` v gramatice, str. 28). Teprve až potom, co byla odladěna tato třída, aby zvládala aritmetické výrazy obsahující sčítání, odčítání, násobení, dělení a závorky, přičemž všechny operátory mají správnou prioritu i asociativitu[20], bylo přistoupeno k druhé fázi. Do odladěné třídy `Parser` se už nezasahovalo.

Druhá část parseru je ve třídě `MMQLParser`, která je potomkem třídy `Parser`. V této třídě je již implementován dotaz v jazyce MMQL. Tento parser tedy přijímá aritmetický výraz obohacený o konstrukci logických výrazů a kompletní příkaz `select`.

Třetí část parseru je ve třídě `InterpretParser`. Tato třída není potomkem třídy `Parser` ani `MMQLParser`. Tento parser zpracovává celý skript včetně konstrukcí obalující hlavní dotaz. Tedy především načítání modelu (kap. 4.3.1) a definice konstant (kap. 4.3.2). K tomuto používá několik instancí třídy `MMQLParser`.

5.3.3 Derivační stromy

Výstupem z parsování jsou derivační stromy. Uzly derivačního stromu jsou třídy, které přímo odpovídají příslušným přepisovacím pravidlům gramatiky. Důležité je, že tyto uzlové třídy využívají typovou kontrolu pro kontrolu korektní konstrukce stromu. Například pro tato přepisovací pravidla:

```
<E2> ::= + <T><E2> | e
<E2> ::= - <T><E2> | e
```

[19] http://www.gnu.org/software/bison/

[20] Nutno podotknout, že aritmetický výraz z popisovacího jazyka má gramatiku notně zjednodušenou, priority a asociativity operátorů neřeší.

je určena třída `AtomE2`, jejíž deklarace je zde:

```
class AtomE2
{
private:
    Token op;
    AtomT *t;
    AtomE2 *next;
public:
    AtomE2(Token aOp, AtomT *aT, AtomE2 *aNext);
    virtual ~AtomE2();

    Variant evaluate(Variant current);
};
```

Tedy instanci této třídy nelze zkonstruovat jinak než s parametry konstruktoru správných typů (`AtomT` odpovídající neterminálu `T` a `AtomE2` odpovídající neterminálu `E2`). Pokud je typová struktura uzlových tříd korektní (odpovídá gramatice), pak i každý derivační strom je korektní (odpovídá gramatice). Samozřejmě některé kontroly jsou dynamické, například parametr `aOp` konstruktoru musí obsahovat operátor plus nebo minus. Tato sémantická kontrola se provádí až v těle konstruktoru[21].

Interpretace

Kód pro interpretaci skriptu je obsažen v uzlových třídách derivačního stromu. Každý uzel má metodu `evaluate`, která vrátí hodnotu daného uzlu (včetně jeho případného podstromu).

5.3.4 Nejdůležitější třídy

Variant

`Variant` je datový typ, který MMQL používá v aritmetických výrazech. `Variant` uvnitř nese buď standardní typ `int` nebo `double` jazyka C++. Pro tento typ jsou přetíženy operátory aritmetické operátory, což vede k intuitivnímu použití a eliminaci chyb.

Některé metody:

- `VariantType getType()` — Vrátí typ dat, který nese aktuální instance.
- `int getInt()` — Vrátí celočíselnou hodnotu reprezentující data instance (nezávisle na skutečném vnitřním typu).

[21] V této situaci by se hodila podpora z jazyka Eiffel pro *design by contract*.

- `double getDouble()` — Vrátí typ `double` reprezentující data instance (nezávisle na skutečném vnitřním typu).

Token

`Token` reprezentuje jeden lexikální element jazyka MMQL.

Některé metody:

- `TokenType getType()` — Vrátí typ tokenu, například klíčové slovo, číslo nebo identifikátor.
- `string getValue()` — Vrátí řetězec reprezentující hodnotu tokenu. Například token typu reálné číslo může mít hodnotu 3,14159.

LexicalAnalyzer

Lexikální analyzér má jako vstup řetězec obsahující skript jazyka MMQL. Výstupem lexikálního analyzéru jsou lexikální elementy — tokeny.

Některé metody:

- `Token* getToken()` — Vrátí další token ze vstupu. Vrátí NULL pokud už na vstupu nic není, nebo nebyl žádný token rozeznán.
- `void ungetToken()` — Vrátí se o jeden token zpět ve vstupu. Je možné se vrátit jen o jeden token po zavolání metody `getToken`, opakované volání této metody nemá žádný účinek.

SymbolTable

Tabulka symbolů obsahuje pro každý symbol (typicky identifikátor) hodnotu (typu `Variant`). Vzhledem k tomu, že typické MMQL–skripty obsahují maximálně jednotky symbolů, je tabulka vnitřně implementována jako zřetězený seznam, který se prohledává lineárně. Pokud by toto lineární prohledávání (složitost $O(n)$) způsobovalo výkonnostní problémy, bude nutné vnitřní implementaci předělat na hašovací tabulku (složitost $O(1)$).

Některé metody:

- `void addSymbol(string name, Variant value)` — Přidá do tabulky nový symbol a nastaví jeho hodnotu. Pokud symbol se jménem `name` již v tabulce je, vyvolá se výjimka.
- `void setValue(string symbolName, Variant value)` — Nastaví hodnotu symbolu `name` na hodnotu `value`. Pokud symbol v tabulce není, vyvolá se výjimka.
- `Variant getValue(string symbolName)` — Vrátí hodnotu požadovaného symbolu. Pokud symbol v tabulce není, vyvolá se výjimka.

Parser

Třída `Parser` umí z textového řetězce obsahující matematický výraz vytvořit derivační strom tohoto výrazu. Více viz odstavec 5.3.2 o parserech.

Některé metody:

- `AtomE* parseE()`, `AtomE2* parseE2()`, `AtomT* parseT()`, `AtomT2* parseT2()`, `AtomF* parseF()` — Každá z těchto metod je určena pro jedno přepisovací pravidlo matematického výrazu. Poznámka: metoda `parseF` je virtuální aby mohla být předefinována pro rozšíření gramatiky.

MMQLParser

Třída `MMQLParser` je potomkem třídy `Parser`; umí z textového řetězce obsahující dotaz v jazyce MMQL vytvořit derivační strom tohoto dotazu. Více viz odstavec 5.3.2 o parserech.

Některé metody:

- `CAtomE* parseCE()`, `CAtomE2* parseCE2()`, `CAtomT* parseCT()`, `CAtomT2* parseCT2()`, `CAtomF* parseCF()` — Tyto metody jsou určeny pro přepisovací pravidla, které gramatiku matematického výrazu rozšiřují o konstrukce pro logický výraz.
- `virtual AtomF* parseF()` — Redefinice metody z třídy `Parser`. Rozšiřuje možnost přepsání faktoru výrazu podle rozšířené gramatiky.
- `AtomEBase* parseColumn(string& columnName, GroupFunctionType& gft)`, `AtomEBase* parseGroup()` , `vector<ForCounter*> parseFor()`, `vector<OrderElement*> parseOrder()` — Tyto metody parsují různé části příkazu `select`.
- `AtomSelect *parse()` — Parsuje kompletní příkaz `select` a vrací jeho derivační strom.

Uzlové třídy derivačního stromu

Výstupem z parserů jsou derivační stromy. Stromy mají uzly různých typů, každý typ uzlu odpovídá nějakému neterminálu gramatiky. Více o derivačních stromeh viz odstavec 5.3.3.

Uzly derivačních stromů reprezentují tyto třídy, všechny jsou odvozeny od abstraktní třídy `AtomEBase`:

- `AtomE`, `AtomE2`, `AtomT`, `AtomT2`, `AtomF`, `AtomFValue`, `AtomFIdentifier`, `AtomFSubExpression`, `AtomFUnaryFunction`, `AtomFBinaryFunction` — pro strom, který generuje parser `Parser`.

- `CAtomE`, `CAtomE2`, `CAtomT`, `CAtomT2`, `CAtomF`, `AtomFIndexedIdentifier`, `AtomFNot`, `AtomFSubSelect`, `AtomSelect` — pro strom, který generuje parser `MMQLParser`

Některé metody:

- `Variant evaluate()`
  `Variant evaluate(Variant current)` — Vyhodnotí uzel (samozřejmě včetně poduzlů, tato metoda se v tom případě volá rekurzivně) a vrátí jeho hodnotu. Tuto metodu mají všechny všechny uzlové třídy. V některých případech potřebuje uzel znát hodnotu svého levého sourozence, k tomu slouží parametr `current`.
- `Rows* evaluate()` — Metoda třídy `AtomSelect`. Na rozdíl od všech ostatních uzlů nevrací skalární hodnotu, ale tabulku hodnot — výsledek dotazu v MMQL.

Třídy agregačních funkcí

Třídy pro agregační funkce mají všechny společného předka `ColumnGroupFunction` a jsou to tyto: `Groupfunction`, `GroupfunctionNone`, `GroupfunctionMin`, `GroupfunctionMax`, `GroupfunctionAvg`, `GroupfunctionSum`, `GroupfunctionMul`, `GroupfunctionCount`.

Některé metody:

- `void addValue(Variant value)` — Přidá argument do agregační funkce.
- `Variant getResult()` — Vyčíslí a vrátí konečnou hodnotu agregační funkce.

InterpretParser

Třída `InterpretParser` umí z textového řetězce obsahující skript jazyka MMQL vytvořit derivační strom tohoto skriptu. Není potomkem třídy `MMQLParser`, ale vytváří a používá jeho instance. Více viz odstavec 5.3.2 o parserech.

Některé metody:

- `void parsePreamble()` — Parsuje preambuli skriptu, tedy načtení modelu (respektive modelů).
- `void parseVariables()` —
  Parsuje definice konstant, k tomu používá `MMQLParser`.
- `Rows* go(string input)` — Zpracuje řetězec `input` obsahující MMQL–skript a vrátí výsledek.

Model

Reprezentuje jeden načtený model. Tedy množinu stavů na nějakých souřadnících (umístěných v n–rozměrné mřížce). Stavy jsou uloženy v hašovací tabulce. To umožňuje jednoduché uložení jakkoli komplikovaných modelů o libovolné dimenzi a zároveň zaručuje vysoký výkon při adresování stavů.

Některé metody:

- `void addState(const State& s)` — Přidá stav `s` do modelu.
- `State* getState(vector<int>& coords)` — Vrátí stav na souřadnicích `coords`. Pokud souřadnice neodpovídají dimenzím modelu, je vyvolána výjimka. Pokud na daných souřadnicích neleží žádný stav, vrátí NULL.
- `string getAlias()` — Vrátí alias, pod kterým je model načten. Na tento alias se odvolávají případné dotazy.

ModelReader

Načte vyřešený model z diskových souborů. Viz formáty souborů programu Markov, odstavec 3.4.1, str 16.

Některé metody:

- `Model* read(string alias, string mapFile, string probFile, string valuationFile)` — Vrátí model inicializovaný daty ze souborů `mapFile`, `probFile` a `valuationFile`. Pokud nastane nějaký problém, je vyvolána výjimka.

5.4 Testování

Už během vývoje byl program intenzivně testován. Průběžné testování bylo možné i proto, že vývoj parseru probíhal po částech. Na testování byl kladen značný důraz, protože gramatika doznala během vývoje několika změn a mohlo se lehce stát, nějaká úprava parseru mohla mít nějaký nepříjemný vedlejší efekt.

Program obsahuje několik testovacích tříd:

- `TestBase` — abstraktní třída, předek všech ostatních testovacích tříd. Poskytuje jednoduchou infrastrukturu pro provádění testů (výpisy, počítání úspěšných a neúspěšných průchodů příkladů testem).
- `VariantTest` — Testuje třídu `Variant`, především sémantiku implementovaných operátorů a práce s typy. Obsahuje 20 příkladů.
- `ParserTest` — Testuje třídu `Parser`, obsahuje testovací příklady v podmnožině jazyka MMQL, kterou tato třída zvládá. Testuje zejména sémantiku operátorů aritmetického výrazu. Obsahuje 20 příkladů.

- `MMQLParserTest` — Testuje třídu `MMQLParser`, testuje dotazy v jazyce MMQL. Obsahuje 26 příkladů, od primitivních dotazů přes testy podmínek a seskupování po vnořené dotazy.

5.5 Výkonnostní charakteristiky a omezení

Provoz programu se sestává ze dvou částí. První je řešení modelu daného popisovacím jazykem, druhá je provádění dotazů nad vyřešeným modelem.

5.5.1 Konstrukce a řešení modelu

Program konstruuje model na základě skriptu v popisovacím jazyce. Jelikož gramatika popisovacícho jazyka je třídy $LL(1)$, je čas parsování lineární. Poté následuje interpretace skriptu (délka tohoto kroku lineárně závisí na velikosti vytvářeného modelu) a sestavení modelu v paměti. Vzhledem k lineárním závislostem tato část procesu nepředstavuje výkonnostní problém.

Jak víme z teorie, řešení markovského modelu o N stavech vede na soustavu N lineárních algebraických rovnic. Program používá k řešení této soustavy knihovnu CLAPACK. Tato knihovna implementuje řešení rovnice Gaussovou eliminační metodou (přesněji používá LU–rozklad) nad obecnou maticí. To tedy znamená, že časová složitost řešení je $O(N^3)$ a paměťová složitost je $O(N^2)$.

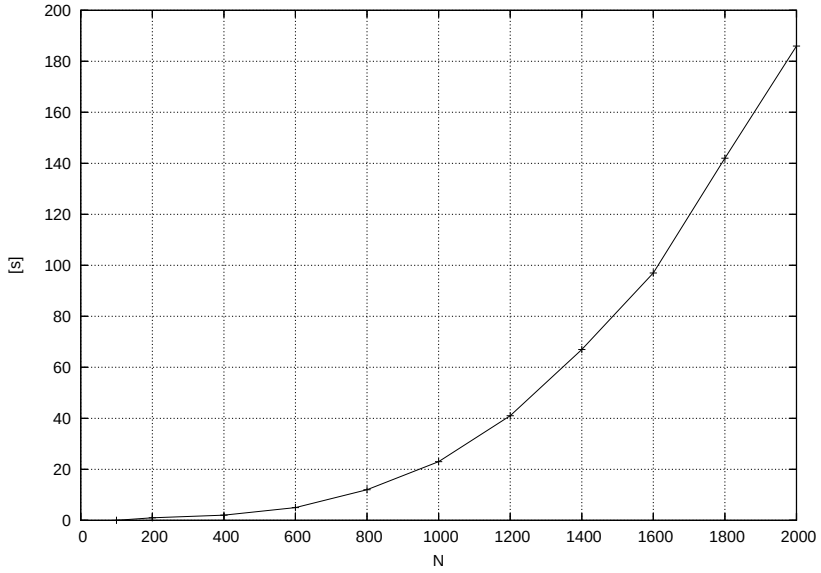
Měření

Proveďme experiment na jednoduchém příkladu. Jednoduchým příkladem budiž jednoduchý buffer (viz odst. 6.1, str. 48). Tento příklad byl řešen programem Markov2 pro různé počty stavů, pro každé takové řešení zaznamenejme spotřebovaný čas procesoru a spotřebovanou paměť. Tento experiment byl prováděn na počítači s procesorem AMD Duron 950 a s operační pamětí o velikosti 640 MB. Naměřené hodnoty jsou v tabulce 4.

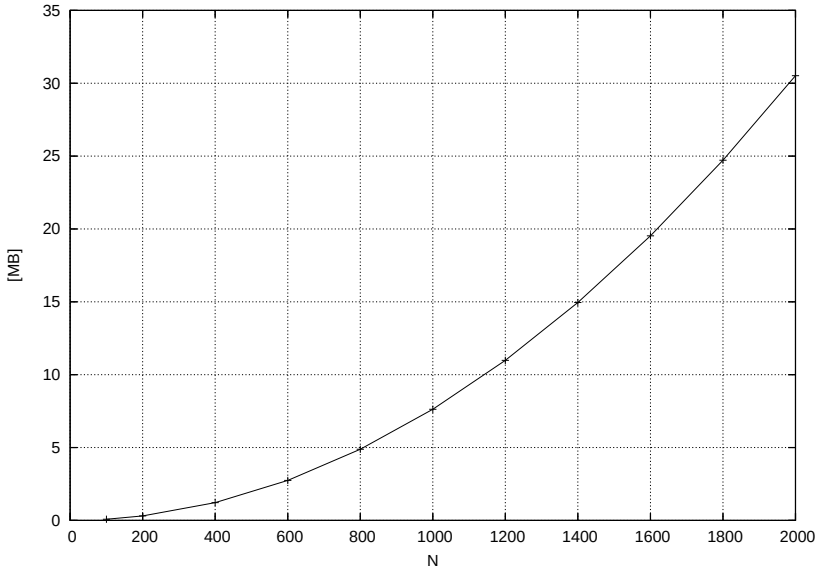
O tom, že se jedná o kubickou křivku a parabolu, nás ujistí grafy závislosti času a paměti na rozsahu modelu (obrázky 2 a 3).

Počet stavů	Čas [s]	Paměť [MB]
200	1	0.30
400	2	1.22
600	5	2.74
800	12	4.88
1000	23	7.62
1200	41	10.98
1400	67	14.95
1600	97	19.53
1800	142	24.71
2000	186	30.51

Tabulka 4: Naměřené hodnoty



Obrázek 2: Závislost spotřebovaného času na velikosti modelu



Obrázek 3: Závislost spotřebované paměti na velikosti modelu

Extrapolace

Nyní vyvstává otázka, jestli na „běžném“ počítači při řešení velkých modelů dříve dojde paměť nebo narazíme na nezvládnutelné časové nároky. Protože známe závislosti, můžeme extrapolovat naměřená data z zjistit, jak asi bude probíhat řešení obrovských modelů.

Předpokládejme, že potřebný čas se řídí vztahem (dopouštíme se jednodušení, zanedbáváme případné nižší mocniny)

$$T = K \cdot N^3$$

Konstantu K neznáme, správně bychom ji měli určit z naměřených dat metodou nejmenších čtverců, příliš velké chyby (pro naše účely) se ale nedopustíme, pokud K určíme pouze z posledního naměřeného záznamu.

$$K = \frac{T}{N^3} = \frac{186}{2000^3} = 2,325 \cdot 10^{-8}$$

Ověřme, že vzorec funguje, například pro $N = 1200$

$$T_{1200} = K \cdot N^3 = 2,325 \cdot 10^{-8} \cdot 1200^3 = 40.176\ s$$

přičemž naměřená hodnota je 41 s.

Paměťové nároky určíme jednodušeji. Program alokuje matici jako souvislý blok paměti velký N^2 prvků. Protože matice pracuje s reálnými čísly typu `double`, každá položka zabírá osm bajtů. Celková spotřebovaná paměť (v bajtech) je tedy

$$M = 8 \cdot N^2$$

Můžeme tedy přibližně určit nároky na řešení několika velkých příkladů, viz tablulku 5.

Počet stavů	Čas	Paměť
10000	6,45 hodin	762 MB
20000	2,12 dní	2,98 GB
50000	33 dní	18,6 GB

Tabulka 5: Extrapolované hodnoty času a paměti

Zdá se tedy, že i přes menší asymptotickou složitost je větší problém paměť, neboť nechat běžet výpočet desítky dní není problém, ale pořídit počítač s desítkami GB operační paměti je nákladné.

Ovšem oblast, kde nastávají potíže s výkonem, je dostatečně vzdálena oblasti předpokládaného využití programu.

5.5.2 Provádění dotazů

Doba provádění dotazu typicky závisí lineárně na počtu stavů modelu. Díky vnořování dotazů je možné napsat dotaz, který má kvadratickou (nebo libovolnou mocninnou) časovou složitost, ale takové dotazy v praxi nejsou potřeba, respektive jdou téměř vždy přepsat na lineární.

5.6 Možná vylepšení

Pro řešení soustavy rovnic se používá obecný algoritmus pro obecnou matici soustavy. Pokud by matice soustavy měla nějaké speciální vlastnosti, bylo by možné použít při řešení nějaký speciální (a tedy efektivnější) algoritmus a efektivnější způsob uložení v paměti.

5.6.1 Řídké matice

Pro model o N stavech sestavujeme soustavu N lineárních algebraických rovnic. Ustálené pravděpodobnosti stavů jsou neznámé a koeficienty matice soustavy mají význam vazeb mezi stavy (tyto vazby umožňují přechody mezi stavy). Počet vazeb mezi stavy je shora omezen — pokud by byl model reprezentován úplným grafem, počet vazeb by byl řádově N^2. Grafy markovských modelů ale tuto vlastnost typicky nemají, obvykle je každý stav svázán jen s několika sousedními stavy. Platí tedy, že počet nenulových prvků matice soustavy odpovídá $Q \cdot N$, kde Q je přirozené číslo a $Q \ll N$. Vidíme tedy, že matice soustavy je řídká.

V příkladu 6.1 je každý stav svázán s nejvýše dvěma sousedními stavy, v příkladu 6.2 má každý stav nejvýše šest vazeb.

Pro typické modely tedy lze snížit paměťovou náročnost a to z kvadratické na lineární.

5.6.2 Iterační řešení soustavy rovnic

Pro řešení velkých soustav lineárních rovnic často používá iteračních metod. Na rozdíl od přímého řešení (Gaussova eliminační metoda) tyto metody dávají pouze „přibližné“ (průběžné) výsledky. K „přesnému“ řešení se blíží (konverguje) postupným iterováním. Proces iterování je ukončen dosažením nějaké stanovené podmínky — například jednotlivé iterace už výsledek příliš neovlivňují.

Jedna z iteračních metod je Gaussova–Seidelova. Tato metoda je zvláště efektivní na řídkých maticích, neboť k výpočtu používá jen nenulových prvků matice soustavy. Celková doba výpočtu tedy odpovídá $I \cdot Q \cdot N$, kde I je počet iterací. Tento vztah neznamená, že iterační řešení takovéto soutavy má nutně lineární složitost, počet iterací I totiž závisí na rozsahu soustavy N (a často se k N blíží).

Proces konvergence lze dále urychlit různými metodami, například relaxační metodou. Více viz například [MB2000].

5.7 Grafické uživatelské rozhraní

Grafické prostředí programu se sestává z jednoho okna. Okno je rozděleno na tři „záložky“ (anglicky „tabs“) a to *Model*, *Query* a *Query Result*.

Záložka *Model* slouží k zadávání markovského modelu v popisovacím jazyce. Pro zobrazení (a editaci) aktuálního modelu je na záložce velké textové pole. Dále obsahuje čtyři ovládací prvky.

- tlačítko *Open* — Otevře dialog pro výběr souboru. Z tohoto souboru je načten model a je zobrazen v textovém poli.
- tlačítko *Save As* — Otevře dialog pro výběr souboru. Do tohoto souboru je uložen model z textového pole.
- tlačítko *Solve Model* — Vyřeší aktuální model. Před řešením musí být model uložen na disk. Řešení modelu je uloženo do stejného adresáře, kde je uložen model.
- zaškrtávací volba *Autosave* — Pokud je tato volba aktivní, je model před každým řešením automaticky uložen na disk. Nový model musí být poprvé uložen tlačítkem *Save As*.

Záložka *Query* slouží k zadání dotazu do markovského modelu. Pro zobrazení (a editaci) dotazu je na záložce velké textové pole. Dále obsahuje tři ovládací prvky.

- tlačítko *Open* — Otevře dialog pro výběr souboru. Z tohoto souboru je načten dotaz a je zobrazen v textovém poli.
- tlačítko *Save As* — Otevře dialog pro výběr souboru. Do tohoto souboru je uložen dotaz z textového pole.
- tlačítko *Run Query* — Spustí aktuální dotaz. Po skončení dotazu se přepne na záložku *Query Result*, kde je zobrazen výsledek dotazu.

Záložka *Query Result* slouží k zobrazení výsledku dotazu.

- tlačítko *Export CSV* — Otevře dialog pro výběr souboru. Do vybraného souboru je uložen výsledek dotazu ve formátu CSV (Comma Separated Values).
- tlačítko *Export for Gnuplot* — Otevře dialog pro výběr souboru. Do vybraného souboru je uložen výsledek dotazu. Výsledek zahrnuje pouze číselné hodnoty (nikoliv názvy sloupců). Takto formátovaný soubor lze načíst programem Gnuplot[22] pro případnou vizualizaci.

[22] http://www.gnuplot.info

6 Příklady

6.1 Nekonečný buffer

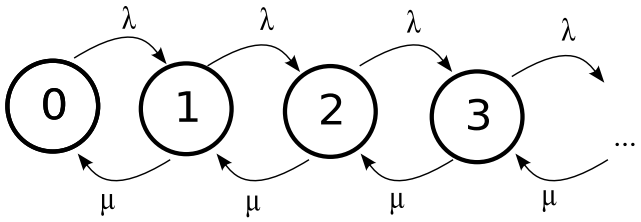
Do bufferu s neomezenou kapacitou přicházejí zprávy, doba mezi příchodem zpráv je náhodná a má exponenciální rozdělení s parametrem $\lambda = 0,9$. Zprávy jsou z bufferu vybírány (pokud tam nějaké jsou) opět náhodně, doba mezi po sobě jdoucími výběry zpráv je též náhodná a má exponenciální rozdělení s parametrem $\mu = 1,0$. S využitím markovského modelu určeme:

1 Kolik procent času při dlouhodobém sledování bude buffer prázdný.

2 Jak často (průměrná perioda) se buffer úplně vyprázdní.

3 Střední počet zpráv, které se v bufferu nachází.

6.1.1 Graf přechodů



Obrázek 4: Model nekonečného bufferu

Jedná se o regulární nekonečný graf. Číslo stavu odpovídá počtu zpráv v systému. V každém okamžiku může přijít nová zpráva (s intenzitou λ) a zařadit se do nekonečného bufferu. Stejně tak může být v každém okamžiku zpráva odebrána (s intenzitou μ).

6.1.2 Analytické řešení

Dle grafu sestavíme soustavu nekonečně mnoha lineárních algebraických rovnic.

$$0 = -\lambda p_0 + \mu p_1$$
$$0 = \lambda p_0 - \mu p_1 - \lambda p_1 + \mu p_2$$
$$0 = \lambda p_1 - \mu p_2 - \lambda p_2 + \mu p_3$$
$$\ldots$$
$$0 = \lambda p_{n-1} - \mu p_n - \lambda p_n + \mu p_{n+1}$$
$$\ldots$$

Aby bylo řešení jednoznačné, je třeba doplnit normalizační podmínku.

$$\sum_{0}^{\infty} p_n = 1$$

Jedná se o klasický učebnicový příklad, soustava rovnic má řešení ve tvaru

$$p_n = (1 - \rho)\rho^n \tag{4}$$

kde $\rho < 1$ je zatížení systému, v našem případě

$$\rho = \frac{\lambda}{\mu} = \frac{0,9}{1,0} = 0,9$$

6.1.3 Konstrukce modelu

Model je dán grafem přechodů, popíšeme tedy graf jazykem pro popis markovských modelů. Graf přechodů je nekonečný, model omezíme na 200 stavů. Skript tedy vypadá takto:

```
module bufferexample [200];
#define size 200
#define lambda 0.9
#define mi 1.0

for (i ;0; size-2){
    [i]->lambda [i+1];
}

for (i; 0; size-2){
    [i+1]->mi [i];
}
```

Takto zadaný model můžeme v programu vyřešit a tak získat ustálené pravděpodobnosti stavů.

6.1.4 Ověření modelu

Jelikož známe analytické řešení modelu, můžeme ho porovnat s numerickým řešením. Ověření provedeme dotazem jazyka MMQL, který z modelu vypíše pravděpodobnosti prvních deseti stavů a zároveň pro tyto stavy pravděpodobnosti spočte dle vzorce (4).

```
load "bufferexample" as buf

define rho := 0.9;
```

```
select
    i as StateNum,
    p[i] as ModelProb,
    (1 - rho) * pow(rho, i) as TheorProb
from buf for i := 0 to 9 order StateNum
```

Výsledek dotazu vypadá takto:

```
 StateNum  ModelProb  TheorProb
        0  0.1        0.1
        1  0.09       0.09
        2  0.081      0.081
        3  0.0729     0.0729
        4  0.06561    0.06561
        5  0.059049   0.059049
        6  0.0531441  0.0531441
        7  0.0478297  0.0478297
        8  0.0430467  0.0430467
        9  0.038742   0.038742
```

Ve sloupci `StateNum` je číslo stavu, ve sloupci `ModelProb` jsou výsledky numerického řešení a ve sloupci `TheorProb` jsou výsledky analytického výpočtu. Je vidět, že model je přesný nejméně na šest platných cifer.

6.1.5 Úkoly

Procentuální podíl času, kdy je buffer prázdný je vlastně ustálená pravděpodobnost stavu 0. Tuto hodnotu získáme skriptem

```
load "bufferexample" as buf

select p[0]*100 as PercentState0 from buf
```

jehož výsledek je

```
 PercentState0
            10
```

Takže buffer bude prázdný 10% času.

Perioda vyprázdnění bufferu je převrácená frekvence přechodu ze stavu 1 do stavu 0.

```
load "bufferexample" as buf

define mi := 1.0;

select 1/(mi*p[1]) as EmptyingFrequency from buf
```

jehož výsledek je

```
 EmptyingFrequency
           11.1111
```

K vyprázdnění bufferu v průměru dojde každých 11 sekud. To odpovídá analytickému výpočtu

$$T_{empty} = \frac{1}{f_{empty}} = \frac{1}{\mu p_1} = \frac{1}{1,0 \cdot 0,09} = 11,111\ s$$

Střední počet zpráv v bufferu je v tomto případě střední počet událostí v systému. Tento údaj získáme jako vážený součet pravděpodobností stavů, kde váha je počet událostí v daném stavu. Váhy jsou v tomto případě rovny indexům stavů. Výpočet realizujeme skriptem

```
load "bufferexample" as buf

define size := 200;

select sum(i*p[i]) as Lw from buf for i := 0 to size-1 group 1
```

jehož výsledek je

```
 Lw
  9
```

To odpovídá analytickému výpočtu

$$L_w = \frac{\rho}{1-\rho} = \frac{0,9}{1-0,9} = 9$$

6.2 Samoobsluha

Demonstrujme program na reálném příkladu — sestavme jednoduchý model samoobsluhy. Do samoobsluhy přicházejí zákazníci v poissonovském proudu s intenzitou λ. V samoobsluze je k dispozici N nákupních vozíků. Pokud není volný žádný vozík, zákazník odchází nespokojen. Pokud má zákazník k dispozici vozík, jde nakupovat. Střední doba nákupu je T_1 a řídí se exponenciálním rozložením pravděpodobnosti. Po nákupu jde zákazník zaplatit k jediné pokladně. Střední doba průchodu pokladnou je T_2 a tato doba má taktéž exponenciální rozdělení.

Určeme:

1 Střední délku fronty před pokladnou.

2 Pravděpodobnost, že když zákazník přijde do samoobsluhy, bude volných alespoň K vozíků.

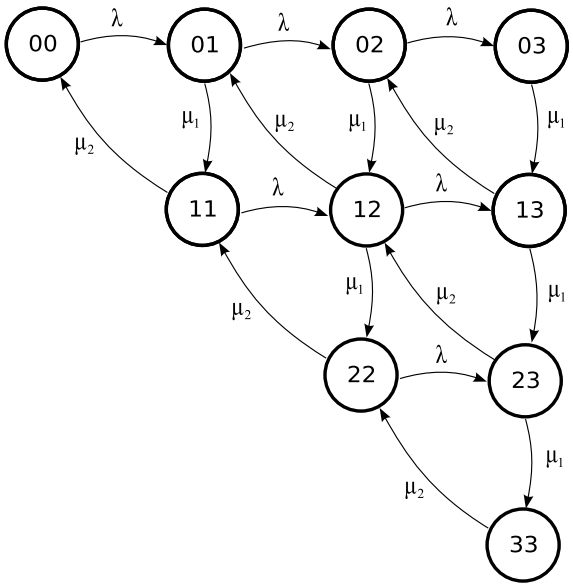
3 Podíl nespokojených zákazníků (tj těch, na které při příchodu nedostane vozík) a jak se tento podíl vyvíjí v závislosti na změnách parametrů λ a T_2. Jde o to zjistit, jaký vstupní tok prodejna zvládne. Je možné urychlit práci pokladny, ale čas nakupování samotného ovlivnit nemůžeme.

K intenzitě příchodů λ zaveďme veličiny stejného rozměru pro nakupování a placení.

$$\mu_1 = \frac{1}{T_1}$$
$$\mu_2 = \frac{1}{T_2}$$

6.2.1 Konstrukce modelu

Vytvořme markovský model. Pro pro přehlednost uvažujme, že v samoobsluze jsou jen 3 nákupní vozíky, tedy $N = 3$.



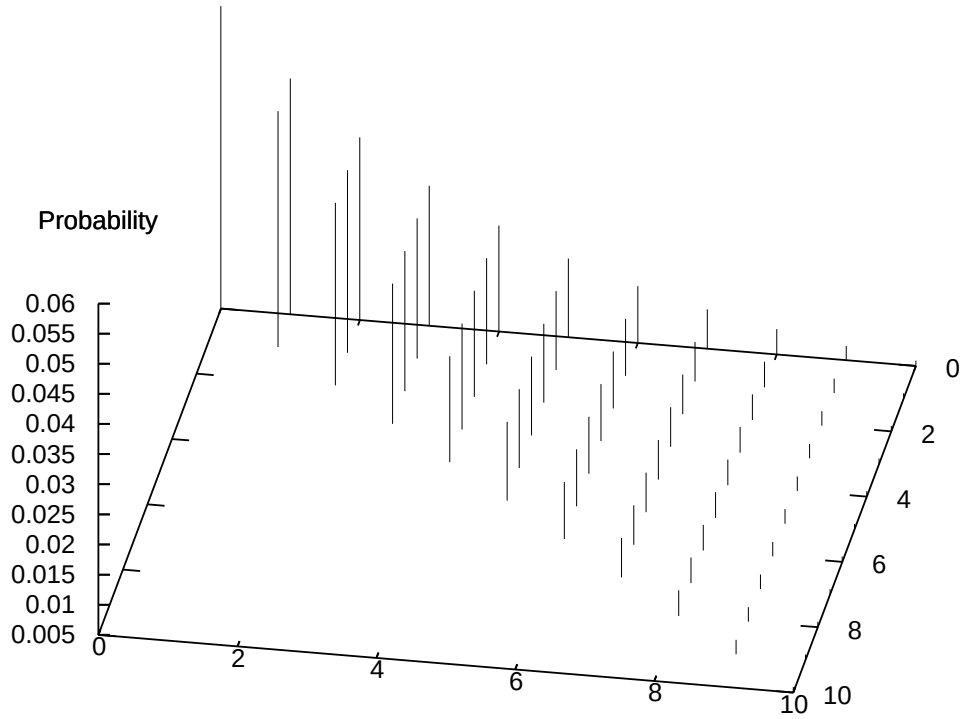
Obrázek 5: Model samoobsluhy se třemi nákupními vozíky

Graf modelu (viz obr. 5) je dvojrozměrný. Každý stav má tedy dvě souřadnice, první souřadnice má význam počtu zákazníků v samoobsluze, druhá počtu zákazníků před pokladnou.

Graf jsme zkonstruovali pro $N = 3$, dále uvažujmě, že v samoobsluze je deset nákupních vozíků (tedy $N = 10$), tím přidáme příkladu na realističnosti. Dále předpokládejme $\lambda = 8\ min^{-1}$, $\mu_1 = \mu_2 = 10\ min^{-1}$. Neboli do obchodu chodí 8 lidí za minutu a pokladna je schopna obsloužit 10 lidí za minutu. Model v popisovacím jazyce pak vypadá následovně.

```
module selfservice[11, 11];
#define N 10
#define lambda 8
#define mu1 10
#define mu2 10

for (i; 0; N-1){
    for (j; i; N-1){
        [i, j]-> lambda [i, j+1];
    }
}

for (i; 0; N-1){
    for (j; i+1; N){
        [i, j]-> mu1 [i+1, j];
    }
}

for (i; 1; N){
    for (j; i; N){
        [i, j]-> mu2 [i-1, j-1];
    }
}

for (i; 0; N){
    [i, N] = -1;
}
```

Řešením modelu dostaneme ustálenou pravděpodobnost každého stavu. Tyto pravděpodobnosti jsou zobrazeny v grafu 6. V rovině xy jsou umístěny stavy, na ose z je asymptotická pravděpodobnost stavu.



Obrázek 6: Asymptotické pravděpodobnosti stavů

6.2.2 Úkoly

Střední délku fronty před pokladnou určíme analogicky k příkladu s jednoduchým bufferem. Stavy v modelu tvoří trojúhelníkovou matici, přičemž každý řádek matice odpovídá příslušné délce fronty (pro stavy v prvním řádku je fronta prázdná). Když provedeme kondenzaci každého řádku (sečteme pravděpodobnosti stavů na řádku), získáme model analogický k předchozímu příkladu. Tuto kondenzaci a následné vážené sečtení kondenzovaných řádků zrealizujeme následujícím skriptem.

```
load "selfservice" as ss
define N := 10;

select
    sum(i * select sum(p[i,j]) from ss for j := i to N group 1)
    as Queue
from dual
for i := 0 to N group 1
```

jehož výsledek je

```
 Queue
 2.4363
```

Tedy střední délka fronty bude asi 2,44.

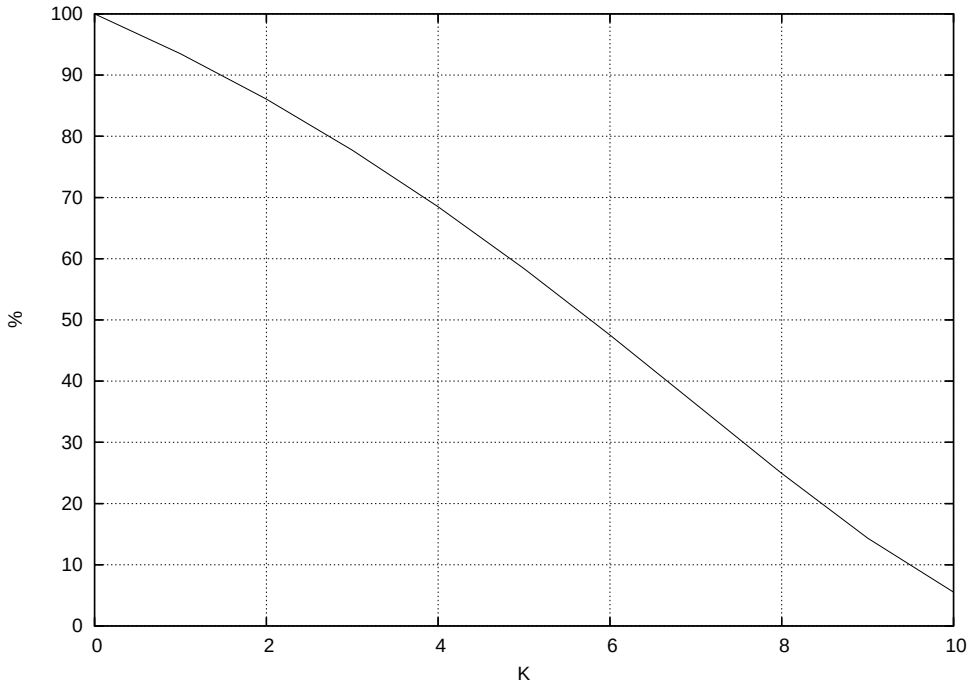
Pravděpodobnost, že je volných alespoň K vozíků. Opět vyjdeme z toho, že stavy tvoří (trojúhelníkovou) matici. Každý sloupec odpovídá nějekému počtu volných vozíků. Ve stavech v prvním sloupci (kde je jenom jeden stav) jsou všechny vozíky volné, ve stavech v posledním sloupci jsou všechny rozebrané. Abychom získali pravděpodobnost, že je *alepoň* K vozíků volných, musíme sečíst pravděpodobnosti od nultého sloupce po K–tý včetně. Pravděpodobnosti zde uvádíme v procentech.

```
load "selfservice" as ss
define N := 10;

select
    N-k as K,
    100*select sum(p[i,j]) from ss
        for j := 0 to k, i := 0 to j group 1 as PerctProb
from dual for k := 0 to N
```

výsledek dotazu je

```
 K   PerctProb
10    5.52
 9   14.34
 8   24.93
 7   36.23
 6   47.53
 5   58.37
 4   68.50
 3   77.75
 2   86.08
 1   93.48
 0  100.00
```

graficky znázorněno pak na obrázku 7.



Obrázek 7: Pravděpodobnost volných K vozíků

Tedy čtyřčlenná rodina (kde každý nakupuje na vlastní pěst) má takřka 70 procentní pravděpodobnost, že se do obchodu vměstná, naproti tomu devítičlenná skupina má tuto pravděpodobnost jen něco přes 14 procent.

Podíl nespokojených zákazníků je dán součtem pravděpodobností stavů, ve kterých jsou všechny nákupní vozíky obsazené. Těmto stavům jsme v modelu přiřadili uživatelské ohodnocení -1 (viz poslední for-cyklus v skriptu popisujícím

λ	μ_2	% nespokojených	λ	μ_2	% nespokojených
4	6	0.88	4	8	0.13
4	10	0.04	4	12	0.02
4	14	0.02	4	16	0.01
6	6	10.48	6	8	2.80
6	10	1.09	6	12	0.63
6	14	0.48	6	16	0.40
8	6	27.10	8	8	12.45
8	10	6.52	8	12	4.43
8	14	3.60	8	16	3.21
10	6	40.65	10	8	25.21
10	10	16.67	10	12	12.97
10	14	11.39	10	16	10.64
12	6	50.32	12	8	36.28
12	10	27.47	12	12	23.31
12	14	21.52	12	16	20.69
14	6	57.35	14	8	44.92
14	10	36.71	14	12	32.73
14	14	31.04	14	16	30.31

Tabulka 6: Podíl nespokojených zákazníků

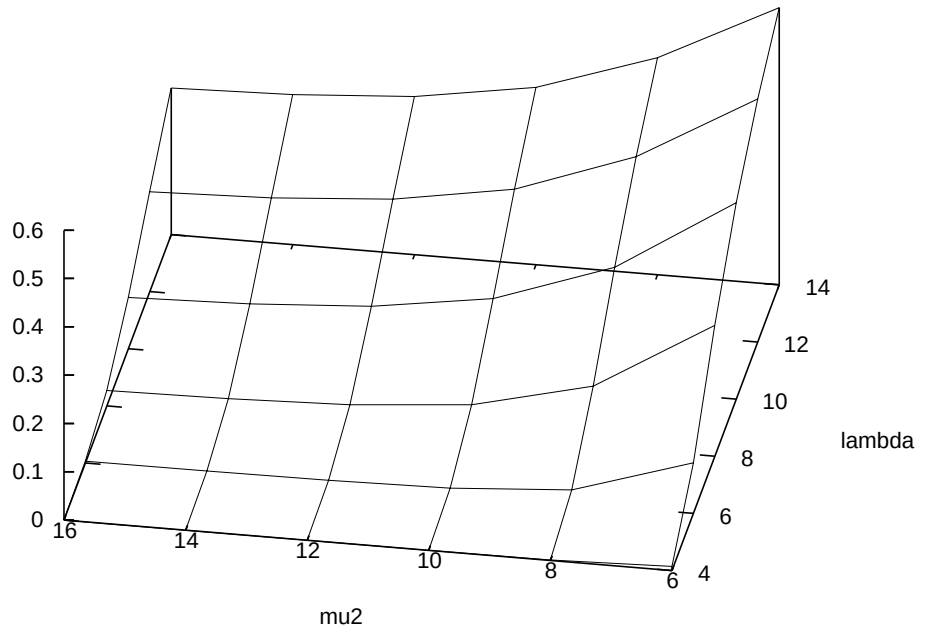
model). Můžeme tedy napsat dotaz, který „hrubou silou“ projde všechny stavy a sečte pravděpodobnosti označených stavů.

```
load "selfservice" as ss
define N := 10;

select sum(p[i,j]) from ss
for i := 0 to N, j := i to N
where val[i,j] = -1 group 1
```

Nyní nezbývá než řešit model samoobsluhy pro různá λ a μ_2 a na každé řešení aplikovat tento dotaz. Výsledeky pro λ od 4 do 14 a pro μ_2 od 6 do 16 jsou v tabulce 6.

Pro lepší představu zkonstruujme z naměřených dat graf 8.

Z grafu 8 je vidět, že se stoupajícím λ stoupá počet nespokojených zákazníků až k 60 procentům. Nespokojenost jsme schopni částečně zmírnit zrychlením práce pokladny (zvětšením parametru μ_2). Od nějaké hranice ($\mu_2 > 12$) přestane mít zrychlování pokladny kladný efekt, neboť jediné zdržování bude vlastní nakupování — a to má konstantní parametr $\mu_1 = 10$.



Obrázek 8: Podíl nespokojených zákazníků v závislosti na parametrech

6.3 Jmenné servery

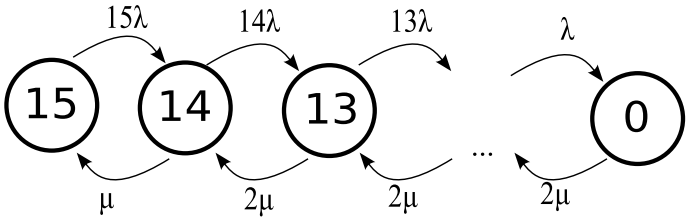
Předpokládejme že v nějaké velké počítačové síti (například Internetu) je potřeba mít nějakou službu (například překlad doménových jmen na IP–adresy) nezbytnou pro chod sítě. Tato služba běží distribuovaně na nějakém počtu (řekněme 15) serverů. Servery jsou nesmírně poruchové, každý se porouchá v průměru desetkrát do roka. Naštěstí jsou k dispozici dva týmy opravářů, každý tým je schopen provést v průměru 100 oprav za rok. Předpokládejme, že všechny časy mají exponenciální rozdělení pravděpodobnosti.

Určeme:

- Minimální počet jmenných serverů se kterými musí být síť provozuschopná, jestliže *stacionární koeficient pohotovosti* musí být alespoň 99%.

6.3.1 Graf přechodů

Číslo stavu odpovídá počtu funkčních serverů. Parametry jsou $\lambda = 10\ rok^{-1}$, $\mu = 100\ rok^{-1}$. Čím více serverů běží, tím větší je pravděpodobnost, že se některý z nich porouchá. Pokud je porouchaný jenom jeden server, opravuje jenom jeden tým, jinak předpokládáme paralelní práci obou týmů.



Obrázek 9: Model soustavy jmenných serverů.

6.3.2 Konstrukce modelu

```
module cluster [20];

#define size 15
#define lambda 10
#define mu 100

for (i ;1; size){
    [i]-> i*lambda [i-1];
}

for (i; 0; size-2){
    [i]->2*mu [i+1];
}

[size-1] -> mu [size];
```

6.3.3 Úkoly

Stacionární koeficient pohotovosti je vlastně součet ustálených pravděpodobností stavů, ve kterých je systém funkční. V dotazu určujeme koeficient pohotovosti, pokud síť potřebuje k provozu alespoň K jmenných serverů. Poté vybereme řádky takové, jejichž koeficient je větší než 0,99. Z takto vybraných řádek zobrazíme tu s nejvyšší hodnotou K[23].

```
load "cluster" as cluster

define size := 15;

select
    k as K,
    (select sum(p[i]) from cluster for i := k to 15 group 1)
```

[23]Pokud bychom vybrali řádek s nejmenší hodnotou K, dostaneme síť, která žádný jmenný server nepotřebuje a běží s koeficientem pohotovosti 1.

```
        as Coef
from dual for k := 0 to size
where
    (select sum(p[j]) from cluster for j := k to 15 group 1)
    > 0.99
order K desc limit 1
```

Výsledkem dotazu je

```
 K      Coef
 8  0.994043
```

Tedy pokud síť vystačí s osmi jmennými servery, bude stacionární koeficient pohotovosti 0,994.

7 Závěr

Cílem této práce bylo navrhnout a implementovat prostředek pro vyhodnocování modelů markovských náhodných procesů bez absorpčních stavů, zkráceně markovských modelů. Prvotní motivací bylo spolehlivostní modelování pohotových systémů, ale výsledky práce jsou obecněji použitelné (například i pro systémy hromadné obsluhy).

Vyhodnocováním modelů se myslí získávání informací vyšší úrovně — takovou informací může být například střední délka fronty v nějakém systému hromadné obsluhy. Získání takových informací vyžaduje výběr nějakých stavů modelu a nějaké operace s nimi.

Tato práce navazuje na předešlou prácí Radka Hoštičky [HOŠ99]. Radek Hoštička vytvořil nástroj pro popis a řešení markovských modelů. Pro popis modelů navrhl speciální jazyk. Součástí jeho práce byl též nástroj pro vyhodnocování vyřešených modelů, tento nástroj měl však velmi omezené možnosti, umožňoval jen výpis ustálených pravděpodobností některých stavů, popřípadě jejich součet.

Jádrem této práce bylo navržení dotazovacího jazyka do markovských modelů (Markov Model Query Language — MMQL). Tento jazyk byl inspirován jazykem SQL a obsahuje prostředky pro výběr stavů, seskupování, řazení a různé výpočty (především s využitím agregačních funkcí a vnořených dotazů). Síla jazyka byla otestována na příkladech a bylo prokázáno, že dotazy v jazyce MMQL jsou schopny získat z modelů požadované informace.

Druhotným cílem bylo vytvoření jednoduchého přenositelného grafického uživatelské rozhraní. Toto rozhraní pokrývá celou práci s modelem — jeho definice s použitím popisovacího jazyka, řešení a vyhodnocení s využitím dotazovacího jazyka. Grafické rozhraní funguje pod operačními systémy Microsoft Windows a Linux. Dle názoru autora je tento program použitelný pro výuku markovských modelů.

Seznam obrázků

Literatura

[RAC96] Racek, S.:Pravděpodobnostní modely počítačů. Skriptum ZČU, Plzeň 1999

[HOŠ99] Hoštička, R.: Markovské náhodné procesy, diplomová práce ZČU, Plzeň 1999

[MAND85] Mandel, P.: Pravděpodobnostní dynamické metody, Academia, Praha 1985

[HAVR86] Havrda, J.: Stochastické procesy a teorie informací. Skriptum ČVUT, Praha 1986

[TRIV82] Trivedy, K. S.: Probability and Statistics With Reliability, Queing and Computer Science Applications. Prentice–Hall 1982

[LMV99] Louis, D. – Mejzlík, P. – Virius, M.: Jazyky C a C++ podle normy ANSI/ISO — kompletní kapesní průvodce. Grada Publishing, Praha 1999. ISBN 80-7169-631-5. 643

[MB2000] Míka, S. – Brandner, M.:Numerické Metody I. Skriptum ZČU, Plzeň 2000

Přílohy

- A — UML model parseru (diagram tříd)
- B — Zdrojový kód algoritmu vyhodnocení dotazu
- C — Tutorial pro program Markov2
- D — CD s kompletními zdrojovými kódy programu Markov2, potřebnými knihovnami a příklady