Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Webový nástroj pro tvorbu, editaci, vizualizaci a analýzu Markovských modelů

Plzeň 2019

Luděk Kaňák

Místo této strany bude zadání práce.

Prohlášení

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

V Plzni dne 7. května 2019

Luděk Kaňák

Poděkování

Rád bych poděkoval Ing. Richardovi Lipkovi, Ph.D. za cenné rady, věcné připomínky a vstřícnost při konzultacích a vypracování diplomové práce.

Abstract

The goal of this thesis is to create a web tool for work with Markov chains with discrete and continuous time. The tool is written in Typescript and uses the Cytoscape graph library for model visualization and interaction. The tool allows an easy creation of chains, both by adding states and transitions sequentially, and by using a descriptive language. It is possible to perform a simulation over the created chains, which is visualized by animation and its progress can be observed.

Abstrakt

Diplomová práce se zabývá tvorbou webového nástroje pro práci s Markovskými řetězci s diskrétním a spojitým časem. Nástroj je napsaný v jazyce Typescript a pro vizualizaci a interakci s modelem využívá grafovou knihovnu Cytoscape. Nástroj umožňuje jednoduché vytváření řetězců jak pomocí postupného přidávání stavů a přechodů, tak s použitím popisovacího jazyka. Nad vytvořenými řetězci je pak možné provádět simulaci, která je vizualizována pomocí animace a je tak možné pozorovat její průběh. Dále nástroj umožňuje výpočet stacionárního rozdělení Markovských řetězců bez absorpčních stavů.

Obsah

1 Úvod **9**

2 Markovské modely **10**

2.1 Popis Markovských modelů 10

2.1.1 Markovský řetězec s diskrétním časem 10

2.1.2 Markovský řetězec se spojitým časem 11

2.2 Analytické řešení 11

2.2.1 Ověření silné souvislosti grafu 11

2.2.2 Markovský řetězec s diskrétním časem 12

2.2.3 Markovský řetězec se spojitým časem 13

2.3 Simulace Markovských retězců 15

2.3.1 Markovský řetězec s diskrétním časem 15

2.3.2 Markovský řetězec se spojitým časem 16

3 Existující nástroje **18**

3.1 Webové aplikace 18

3.2 Markov2 18

3.3 Matlab 19

3.4 Octave 20

3.5 PRISM 21

3.6 Storm 22

3.7 Shrnutí nástrojů 22

4 Požadavky na aplikaci **24**

4.1 Prostředí pro běh aplikace 24

4.2 Vizualizace modelu 24

4.3 Modifikace grafu 25

4.4 Vytváření modelu 25

4.5 Simulace Markovských řetězců 25

4.6 Analytické řešení 26

4.7 Přepínání modelů 26

4.8 Dotazování na vlastnosti modelu 26

4.9 Další funkce 27

5 Návrh nástroje 28
5.1 Výběr technologie 28
5.1.1 HTML 28
5.1.2 CSS 28
5.1.3 Javascript 29
5.1.4 Typescript 31
5.1.5 Dart 32
5.1.6 CoffeeScript 32
5.1.7 Shrnutí 33
5.2 Výběr knihovny pro vizualizaci grafů 33
5.2.1 vis.js 34
5.2.2 D3.js 34
5.2.3 Cytoscape.js 35
5.2.4 Shrnutí 35
5.3 Akce nad grafem 36
5.4 Vytvoření grafu 36
5.4.1 Způsoby pro vytvoření grafu 36
5.4.2 Layout pro uzly 37
5.4.3 Jazyk pro tvorbu grafu 37
5.4.4 Editor 37
5.5 Zobrazení informací o uzlech 38
5.6 Změna typu Markovského řetězce 39
5.7 Návrh implementace 40

6 Implementace nástroje 41
6.1 Organizace zdrojového kódu 41
6.1.1 Konfigurační soubory 41
6.1.2 Zdrojové kódy aplikace 41
6.2 Architektura implementace 42
6.2.1 Prezentační vrstva 43
6.2.2 Aplikační vrstva 44
6.2.3 Knihovna Cytoscape.js 45
6.3 Popis implementace 46
6.3.1 Inicializace aplikace 46
6.3.2 Akce nad grafem 46
6.3.3 Simulace 48
6.3.4 Animování simulace grafu 49
6.3.5 Stylování grafu 50
6.3.6 Vizualizace pravděpodobnosti 52
6.3.7 Analytické řešení 53

6.3.8 Tvorba grafu . . . 54
6.3.9 Navigační lišta . . . 57
6.3.10 Kontextové menu . . . 58
6.3.11 Skupiny uzlů . . . 59
6.3.12 Formuláře . . . 59

7 Testování **62**
7.1 Jednotkové testy . . . 62
7.1.1 Konfigurace prostředí . . . 62
7.1.2 Vytvoření testů . . . 62
7.1.3 Vyhodnocení . . . 64
7.2 Funkční testování . . . 64
7.2.1 Scénář pro testování . . . 64
7.2.2 Modely pro ověření simulace a analytického řešení . . 67
7.2.3 Vyhodnocení . . . 72
7.3 Uživatelské testování . . . 73
7.4 Omezení . . . 74
7.5 Další prohlížeče . . . 75

8 Závěr **76**

Literatura **77**

A Příručka **79**
A.1 Tvorba grafu manuálně . . . 79
A.1.1 Přidání elementu . . . 79
A.1.2 Smazání elementu . . . 80
A.1.3 Editace hodnot hran . . . 80
A.2 Tvorba grafu pomocí kódu . . . 80
A.2.1 Příklady pro vytvoření grafu . . . 81
A.3 Ovládání simulace . . . 82
A.4 Přepnutí typu Markovského řetězce . . . 83
A.5 Skupiny stavů . . . 84
A.6 Export a import grafu . . . 85

B Zprovoznění vývojového prostředí **86**

1 Úvod

Markovské modely byly pojmenovány po ruském matematikovi Andreji Markovovi, který se zabýval teorií stochastických procesů. Markovské modely jsou stochastické modely, jejichž hlavní vlastnost je bezpaměťovost. To znamená, že budoucí stav modelu je závislý pouze na současném stavu bez ohledu na předchozí stavy.

Markovské modely mají uplatnění v celé řadě oborů, jako je například chemie, fyzika, biologie a ekonomie. Jako konkrétní příklady využití lze uvést například modelování systémů s náhodnými příchody požadavků (lidé ve frontě, vozidla v křižovatce), vědecké modely (fungování enzymů, modelování burzy, pagerank), generování textů, rozpoznávání a analýza řeči[18].

Cílem diplomové práce je navrhnout a vytvořit webový nástroj pro práci s Markovskými modely. Nástrojem půjde vytvářet, editovat a vizualizovat Markovské modely. Nad vytvořenými modely by pak mělo být možné provádět jejich analýzu a simulaci.

Ve druhé kapitole diplomové práce jsou popsány Markovské modely a jejich rozdělení. Dále jsou zde uvedeny možnosti a algoritmy pro analytické řešení a pro simulaci. Ve třetí kapitole jsou představeny již existující nástroje pro práci s Markovskými řetězci a ke každému je uveden stručný popis. Ve čtvrté kapitole jsou vytvořeny a sepsány požadavky na aplikaci. Pátá kapitola je určená pro návrh nástroje. Při návrhu nástroje je vybírána technologie pro tvorbu nástroje a knihovna pro práci s grafem. Dále je zde navržen způsob tvorby, interakce a vizualizace grafu. Šestá kapitola je zaměřená na popis implementace nástroje včetně popisu architektury aplikace a konfiguračních souborů pro provoz aplikace. Poslední část je věnována testování aplikace. Při testování se ověřuje funkčnost aplikace a správné chování modelů (analytické a simulační řešení). Navíc je ověřena funkčnost vytvořeného řešení pomocí uživatelského testování.

2 Markovské modely

2.1 Popis Markovských modelů

Markovský model je stochastický model pro modelování systémů bez paměti. Využívá se v případě, kdy se předpokládá náhodné chování založené na pravděpodobnosti přechodu z jednoho stavu do dalšího stavu. Přechod závisí pouze na aktuálním stavu bez ohledu na předchozí události. Tato nezávislost následujícího stavu na předchozím stavu se označuje jako Markovská vlastnost [8]. Kromě Markovské vlastnosti musí být možné identifikovat stavy a přechody mezi nimi v modelovaném systému.

Existuje několik typů Markovských modelů (viz tabulka 2.1 [12]). Rozlišují se podle toho, zda systém je se spojitým nebo diskrétním časem, a dále zda má systém spojité nebo diskrétní stavy.

	Diskrétní stavy	**Spojité stavy**
Diskrétní čas	Markovské řetězce	Skryté markovské modely
Spojitý čas	Markovské rozhodovací procesy	Částečně pozorovatelné Markovské rozhodovací procesy

Tabulka 2.1: Typy Markovských modelů

Diplomová práce bude dále zaměřená pouze na Markovské modely s diskrétními stavy. Konkrétně tedy na Markovské řetězce s diskrétním časem a Markovské rozhodovací procesy, které se také často nazývají Markovské řetězce se spojitým časem.

2.1.1 Markovský řetězec s diskrétním časem

Markovský řetězec s diskrétním časem je popsaný maticí pravděpodobností přechodu `P` a vektorem pravděpodobností `A`. Prvky matice `P` jsou hodnoty v rozmezí od 0 do 1 a jejich součet v jednom řádků musí dát hodnotu 1. Počet řádků a sloupců matice `P` je dán počtem stavů a hodnota udává pravděpodobnost přechodu mezi stavy v diskrétním čase.

Pravděpodobnostní vektor `A` uchovává v určitém čase ke každému stavu pravděpodobnost, která se mění při dalším kroku v diskrétním čase a znamená aktuální pravděpodobnost výskytu v daném stavu [3].

2.1.2 Markovský řetězec se spojitým časem

Markovský řetězec se spojitým časem je popsaný pomocí matice intenzit přechodů `Q` a vektorem pravděpodobností `A` [3]. Matice intenzit přechodů má počet řádků a sloupců daný podle počtu stavů v řetězci. Hodnoty mimo diagonálu jsou intenzity přechodů mezi dvěma stavy a jsou nezáporné. Hodnoty na diagonále jsou součty intenzit v řádku a výsledný součet má záporné znaménko.

Pro tento typ řetězů je náhodná doba setrvání v aktuálním stavu daná exponenciálním rozdělením, jehož parametrem je součet intenzit výstupních hran ze stavu.

Vektor `A` opět uchovává pravděpodobnosti stavů stejně jako v případě Markovských řetězců s diskrétním časem.

2.2 Analytické řešení

Analytické řešení slouží k nalezení stacionárního rozdělení, které popisuje chování řetězce v ustálené podobě po dostatečně dlouhé době [8]. Pro existenci stacionárního rozdělení je nutné, aby intenzity popř. pravděpodobnosti přechodů Markovského řetězce byly konstantní[8].

Markovský řetězec může obecně obsahovat absorpční stavy. Absorpční stav je stav, ze kterého nevede žádná hrana do dalšího stavu a zároveň je možné se do něj dostat z jakéhokoliv neabsorpčního stavu[8]. Pokud model obsahuje absorpční stavy, graf popisující model není silně souvislý. V případě grafu bez absorpčních stavů je graf silně souvislý a je tedy možné se dostat z každého stavu do jakéhokoliv jiného stavu v modelu.

Pokud Markovský řetězec obsahuje alespoň jeden absorpční stav, neexistuje netriviální řešení pro nalezení stacionárního rozdělení. U tohoto typu řetězců se řeší například pravděpodobnost uvíznutí v daném uzlu po určité době nebo střední hodnota počtu kroků do pohlcení [8].

Analytické řešení v této práci bude zaměřené pouze na nalezení stacionárního rozdělení Markovských řetězců bez absorpčních stavů. Postup pro nalezení stacionárního rozdělení bude popsán v následujících kapitolách 2.2.2 a 2.2.3.

2.2.1 Ověření silné souvislosti grafu

Jak bylo řečeno v úvodu kapitoly 2.2, graf neobsahuje absorpční stavy, pokud je graf silně souvislý. Graf je silně souvislý, pokud existuje sled pro každé dva vrcholy(stavy) v grafu[7].

Pro ověření silné souvislosti existuje několik algoritmů založených na prohledávání grafu do hloubky(DFS)[7]. Ověření u těchto algoritmů probíhá v lineárním čase. Vstupem pro algoritmy je orientovaný graf.

Kosarajův algoritmus

Kosarajův algoritmus používá dvě fáze algoritmu DFS. První fáze prohledávání se aplikuje na původní graf. Pokud nebyly algoritmem navštíveny všechny vrcholy, není graf silně souvislý. V opačném případě, kdy byly navštíveny všechny vrcholy, se provede druhá fáze prohledávání do hloubky, ale tentokrát nad grafem, kde je směr všech hran obrácen. V případě, že byly navštíveny všechny vrcholy, lze graf prohlásit za silně souvislý, jinak graf není silně souvislý[7].

Tarjanův algoritmus

Tarjanův algoritmus používá pouze jednu fázi algoritmu DFS. Vrcholům se při prohledávání přiřazuje index podle pořadí svého nalezení a jsou vkládány do zásobníku. Při návratu z rekurze se každému vrcholu přiřadí vrchol s nejnižším indexem, na jaký je možné dosáhnout. Všechny vrcholy se stejným cílovým uzlem (indexem), jsou ve stejné komponentě. Pokud jsou všechny uzly ve stejné komponentě, je graf silně souvislý[7].

Gabowův algoritmus

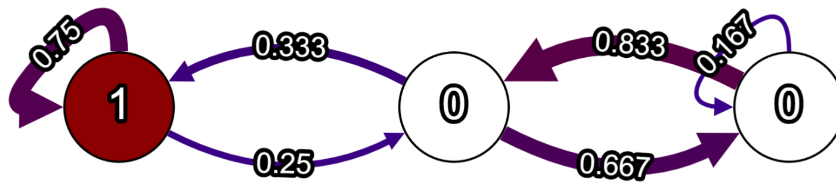
Gabowův algoritmus stejně jako Tarjanův algoritmus provádí pouze jednu fázi algoritmu DFS, ale používá dva zásobníky. První zásobník obsahuje vrcholy, které ještě nebyly přiřazeny do žádné komponenty. Druhý zásobník uchovává aktuální vyhledávací cestu[16].

2.2.2 Markovský řetězec s diskrétním časem

Stacionární rozdělení π pro Markovský řetězec s diskrétním časem bez absorpčních stavů se vypočítá podle rovnice 2.1[8], kde P je matice pravděpodobností přechodu (viz kapitola 2.1.1) a π je vektor (π_1, π_2, π_3, ...) stacionárního rozdělení.

$$\pi = \pi P \tag{2.1}$$

Jako příklad bude uveden výpočet pro Markovský model popsaný grafem 2.1. Z grafu je vidět, že se jedná o silně souvislý graf. Graf tedy neobsahuje žádné absorpční stavy a je možné najít stacionární rozdělení π.



Obrázek 2.1: Graf ukázkového Markovského řetězce

Graf 2.1 je popsán maticí (2.2) pravděpodobností přechodu P.

$$P = \begin{bmatrix} \frac{1}{4} & \frac{3}{4} & 0 \\ \frac{1}{3} & 0 & \frac{2}{3} \\ 0 & \frac{5}{6} & \frac{1}{6} \end{bmatrix} \tag{2.2}$$

Pro nalezení stacionárního rozdělení se v první řadě vytvoří soustava rovnic 2.3.

$$\begin{aligned} \pi_1 &= \frac{1}{4}\pi_1 + \frac{1}{3}\pi_2 \\ \pi_2 &= \frac{3}{4}\pi_1 + \frac{5}{6}\pi_3 \\ \pi_3 &= \frac{5}{6}\pi_2 + \frac{1}{6}\pi_3 \end{aligned} \tag{2.3}$$

Pro jednoznačné řešení je nutné do soustavy přidat normalizační podmínku $\pi_1 + \pi_2 + \pi_3 = 1$, kterou se nahradí jedna rovnice ze soustavy. Po nahrazení rovnice a úpravě vznikne soustava 2.4.

$$\begin{aligned} -\frac{3}{4}\pi_1 + \frac{1}{3}\pi_2 &= 0 \\ \frac{3}{4}\pi_1 - \pi_2 + \frac{5}{6}\pi_3 &= 0 \\ \pi_1 + \pi_2 + \pi_3 &= 1 \end{aligned} \tag{2.4}$$

Vyřešením soustavy se získá stacionární rozdělení ve tvaru:

$$\pi_1 = \frac{20}{101}, \pi_2 = \frac{45}{101}, \pi_3 = \frac{36}{101} \tag{2.5}$$

2.2.3 Markovský řetězec se spojitým časem

Podobně jako u Markovských řetězců s diskrétním časem, tak i u řetězců se spojitým časem lze nalézt stacionární rozdělení. Pro nalezení stacionárního rozdělení se používají tzv. Kolmogorovovy diferenciální rovnice (2.6) [8]. Levá strana je tvořena vektorem derivací pravděpodobnostních stavů $p'(t)$ a na

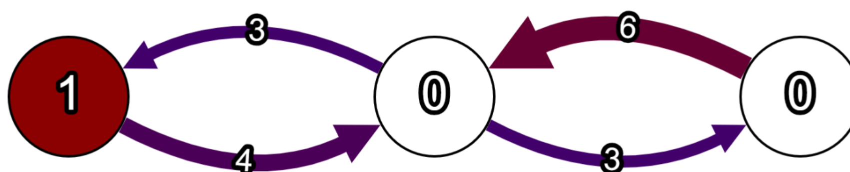
pravé straně je vektor pravděpodobnostních stavů $p(t)$ a matice intenzit přechodů Q.

$$p'(t) = p(t)Q \tag{2.6}$$

Soustava 2.6 lineárních diferenciálních rovnic 1. řádu je tvořená rovnicemi, kde pro každý stav je jedna rovnice. Rovnice pro i-tý stav má na levé straně derivaci pravděpodobnosti $p_i'(t)$ a pravá strana se skládá ze součtu příspěvků intenzit vstupujících nebo vystupujících z uzlu (viz 2.7) [18].

$$\begin{aligned} +p_k(t)\lambda_{k,i} &\quad \text{vstupní hrana do stavu i} \\ -p_i(t)\lambda_{i,k} &\quad \text{výstupní hrana ze stavu i} \end{aligned} \tag{2.7}$$

Pro graf 2.2 s maticí přechodů s intenzitami 2.8 vzniknou diferenciální rovnice 2.9.



Obrázek 2.2: Graf ukázkového Markovského řetězce se spojitým časem

$$P = \begin{bmatrix} 0 & 4 & 0 \\ 3 & 0 & 3 \\ 0 & 6 & 0 \end{bmatrix} \tag{2.8}$$

$$\begin{aligned} p_1'(t) &= -4p_1(t) + 3p_2(t) \\ p_2'(t) &= 4p_1(t) - 3p_2(t) - 3p_2(t) + 6p_3(t) \\ p_3'(t) &= -6p_3(t) + 3p_2(t) \end{aligned} \tag{2.9}$$

Kolmogorovovy diferenciální rovnice počítají s Markovskými řetězci, které můžou obsahovat absorpční stavy. V takovém případě jsou pravděpodobnosti $p_i(t)$ dané funkcemi. Pokud jsou řetězce bez absorpčních stavů, existují hodnoty limitních pravděpodobností p_i, jejichž derivace jsou nulové. Pro dostatečně velké t ($t \to \infty$) vznikne z diferenciální rovnice 2.6 lineární rovnice ($p_i'(\infty) = 0$) ve tvaru 2.10.

$$0 = pQ \tag{2.10}$$

Vznikne soustava lineárně závislých rovnic [18] a je třeba přidat normalizační podmínku(viz rovnice 2.11), jako v případě hledání stacionárního rozdělení u Markovských řetězců s diskrétním časem v kapitole 2.2.2.

$$\begin{aligned} 0 &= -4p_1 + 3p_2 \\ 0 &= 4p_1 - 3p_2 - 3p_2 + 6p_3 \\ 1 &= p_1 + p_2 + p_3 \end{aligned} \tag{2.11}$$

Po vyřešení soustavy rovnic bude výsledek vektoru p se stacionárním rozdělím vypadat následovně, viz 2.12.

$$p_1 = 0.333, p_2 = 0.444, p_3 = 0.222 \tag{2.12}$$

2.3 Simulace Markovských retězců

Simulování Markovských řetězců je vhodné pokud není snadné nebo možné nalézt stacionární řešení analytickým způsobem. Další využití simulace je pro sledování chování Markovského řetězce v průběhu času.

Pro simulování se používají rozdílné metody v závislosti na diskrétním, resp. spojitém čase. Metody budou popsány v následujících kapitolách.

2.3.1 Markovský řetězec s diskrétním časem

Monte Carlo

Pro simulaci Markovských řetězců s diskrétním časem je možné použít metodu typu Monte Carlo [15]. Monte Carlo je stochastická metoda a je založena na pseudonáhodných číslech.

Simulace začíná v počátečním stavu s a v čase $t = 0$. V následujícím kroku ($t = 1$) se vygeneruje náhodné desetinné číslo r na intervalu (0, 1] s rovnoměrným rozdělením. Nový stav se zjistí z vygenerovaného čísla r a matice pravděpodobností přechodu `P`. Z matice `P` se vezme řádek odpovídající stavu s a poté se sčítají hodnoty v řádku, dokud platí podmínka, že součet hodnot je menší než číslo r. Nový stav se pak určí podle sloupce, v němž se nacházela poslední hodnota splňující podmínku. Tento postup se opakuje pro časy $t = 2, 3, 4..., t_i$, dokud není dosaženo požadovaného času [15].

Výsledkem této simulace je pouze jedna instance simulace Markovského řetězce, která představuje pouze jednu sekvenci přechodů. Pokud se jedná

o Markovský řetězec s absorpčními stavy, musí se pro zjištění pravděpodobnosti ve stavu v čase t_i vykonat simulace vícekrát, dokud nedojde k ustálení pravděpodobností, a uchovávat si sekvence stavů.

2.3.2 Markovský řetězec se spojitým časem

Pro Markovský řetězec se spojitým časem nelze použít stejnou metodu pro simulování jako u řetězců s diskrétním časem, protože čas setrvání ve stavu má exponenciální rozdělení a pohyb s diskrétními kroky v čase není možný vykonat tímto způsobem. K simulování Markovských řetězců je tedy nutné použít jiné metody. Pro simulování se nabízí použít stochastický simulační algoritmus (Gillespieho algoritmus) nebo explicitní a implicitní tau-leaping metody [2].

Gillespieho algoritmus

Gillespieho algoritmus byl představen v roce 1976 a původním účelem byla simulace chemických a biochemických reakčních systémů. Kromě biologie se algoritmus začal uplatňovat i v dalších numerických simulací stochastických modelů [2].

Gillespieho algoritmus pro simulaci Markovského řetězce se spojitým časem vypadá následovně:

Vstup Matice intenzit přechodů `Q`, výchozí bod s, vektor pravděpodobností `A`

Výstup Aktualizovaný vektor pravděpodobností `A`

1. krok Nastavení doby simulace $t = 0$.

2. krok Zjištění intenzit λ_i výstupních hran z bodu s a součtu těchto intenzit λ.

3. krok Vygenerování dvou náhodných čísel(r_1, r_2) pomocí pseudonáhodného generátoru s uniformním rozdělením od 0 do 1.

4. krok Výpočet času τ do dalšího přechodu s exponenciálním rozdělením a střední hodnotou $1/\lambda$ podle vzorce $\tau = 1/\lambda * ln(1/r_1)$.

5. krok Výběr dalšího stavu s podle diskrétního rozdělení s pravděpodobností λ_i/λ pro každou hranu. Nový stav bude vybrán na základě sčítání pravděpodobností λ_i/λ, dokud součet nebude největší možný a zároveň menší než r_2.

Například pro $0 < r_2 < \lambda_1/\lambda$ se vybere první výstupní hrana s intenzitou λ_1, pro $lambda_1/lambda < r_2 < (\lambda_1 + \lambda_2)/\lambda$ se vybere výstupní hrana s intenzitou λ_1 a obdobně by se pokračovalo pro další hrany [2]. Nový stav s pak bude stav, do kterého vstupuje vybraná hrana

6. krok Zvýšení doby simulace $t = t + \tau$ a aktualizování vektoru pravděpodobností `A`.

7. krok Pokud bylo dosaženo požadovaného času, simulace končí, jinak algoritmus pokračuje znova od 2. kroku.

Výše popsaný algoritmus simuluje pouze jednu sekvenci přechodů, stejně jako simulace Markovských řetězců s diskrétním časem 2.3.1. Pro zjištění pravděpodobností ve stavech v čase t_i je tedy opět nutné simulaci vykonat vícekrát[2].

Tau-leaping metody

Jako další způsob pro simulaci je možné použít tau-leaping metody. Tau-leaping metody urychlují simulaci bez větší ztráty přesnosti. Jsou vhodné pro aplikace, kde je kladen důraz na časovou náročnost výpočtu [2].

Tau-leaping metody neprovádí jednotlivé kroky jako Gillespieho algoritmus, ale přeskakují několik přechodů podle časového intervalu. Pokud je dobře zvolený interval pro přeskakování, nedochází ke ztrátě přesnosti a algoritmus může být značně urychlen.

I přesto, že tau-leaping metody mohou dosáhnout rychlejší konvergence, nebudou v této práci použity. V rámci diplomové práce by se měly simulovat jednotlivé přechody a to tau-leaping metody neumožňují. Dále je u zmíněných metod nutné stanovit interval, který se liší vzhledem k řešenému problému a to také nevyhovuje požadované simulaci v diplomové práci. [2].

3 Existující nástroje

V následujících podkapitolách bude představeno několik nástrojů pro práci s Markovskými řetězci. U každého nástroje bude uveden popis a výčet informací, jako je například licence, dostupnost zdrojového kódu, dokumentace a vlastnosti pro práci s Markovskými modely. Dále bude stručně popsáno jejich použití v souvislosti s Markovskými modely.

3.1 Webové aplikace

Na internetu je možné nalézt několik online aplikací zaměřených na Markovské modely. Aplikace jsou určené převážně pro vysvětlení Markovských řetězců pomocí vizualizace. Vytváření řetězců je velmi jednoduché a nástroje neumožňují vytvářet velké a složité modely.

Jeden takový nástroj byl vytvořen na Technické univerzitě Clausthal [14]. Nástroj umí simulovat diskrétní[1] a spojité řetězce[2]. Program dovoluje vytvořit Markovské řetězce nejvýše ze šesti stavů a pravděpodobnosti, resp. intenzity přechodů se zadávají do matice sousednosti. Během simulace je možné sledovat přechody mezi stavy pomocí grafického znázornění aktuální hrany a uzlu v grafu. Dále program ukazuje aktuální pravděpodobnosti navštívení stavů v sloupcovém grafu, na kterém lze zřetelně sledovat ustálení pravděpodobností po určitém čase.

Na stránce [19] autoři Victor Powell a Lewis Lehe vytvořili stručný a srozumitelný popis Markovských modelů s názornými ukázkami simulací modelu s diskrétním časem. Simulace má snadné ovládání pro změnu rychlosti a pravděpodobnostního ohodnocení hran. Stránka navíc obsahuje jednoduchý editor pro maticový zápis grafu pro simulaci.

3.2 Markov2

Nástroj Markov2 byl vytvořen v rámci dvou diplomových prací na Západočeské univerzitě v Plzni [13] a slouží k vyhodnocení Markovských modelů bez absorpčních stavů. První diplomová práce byla zaměřená na vytvoření ná-

[1] https://www.mathematik.tu-clausthal.de/en/mathematics-interactive/simulation/markov-chain-discrete/

[2] https://www.mathematik.tu-clausthal.de/en/mathematics-interactive/simulation/markov-chain-continuous/

Licence	Zdarma, nástroj vypracován na ZČU jako dip. práce
Datum poslední verze	2016
Dokumentace	Příručka k ovládání nástroje a ukázka příkladů
Umístění zdroj. kódu	https://courseware.zcu.cz/CoursewarePortlets2/DownloadDokumentu?id=121921
Vlastnosti	- Jazyk pro popis Markovských řetězců s diskr. a spoj. časem. - Simulace Markovských řetězců - Komplexní analýza modelů pomocí dotazovacího jazyka

Tabulka 3.1: Přehled pro nástroj Markov2

stroje pro popis a řešení Markovských modelů [17]. Druhá diplomová práce se věnuje rozšíření první diplomové práce o dotazovací jazyk `MMQL` nad modelem a přidává grafické uživatelské rozhraní.

Jazyk pro popis Markovských modelů slouží k popisu orientovaného grafu, kde hrana z bodu `a` do bodu `b` je ohodnocená intenzitou. Jazyk dále umožňuje k vytváření grafů používat smyčky a definovat konstanty [17].

Dotazovací jazyk `MMQL` byl inspirován jazykem SQL a umožňuje vybírat stavy, řadit, seskupovat a provádět další složitější výpočty [13].

3.3 Matlab

Licence	Placený, třicetidenní zkušební verze
Datum poslední verze	1.9.2018
Dokumentace	Ano
Vlastnosti	- Vytvoření Markovských řetězců s disk. a spoj. časem pomocí matice pravděpodobností, resp. intenzit přechodu. - Ověření existence stacionárního rozdělení - Výpočet stacionárních rozdělení - Simulace řetězců - Vizualizace výsledku simulace

Tabulka 3.2: Přehled informací o nástroji Matlab

Matlab (tabulka 3.2) pro práci se statistickými modely používá ekonometrický toolbox, který poskytuje funkce pro modelování ekonomických dat. Součástí balíčku jsou i Markovské modely.

Sada nástrojů podporuje práci s Markovskými řetězci s diskrétním i spojitým časem. K vytvoření Markovského řetězce s diskrétním časem slouží funkce `dtmc` a pro řetězce se spojitým časem funkce `ssm`. Knihovna umožňuje nad vytvořeným modelem ověřit existenci unikátního stacionárního rozdělení pomocí funkcí `isreducible` a `isergodic`. Dále knihovna umí simulovat chování Markovských řetězců (funkce `simulate`) a výsledek simulace je možné vizualizovat funkcí `simplot`.

Matlab je možné využít i pro nalezení jedinečného stacionárního rozdělení. K tomu postačí jeho standardní operace pro řešení soustavy lineárních rovnic.

3.4 Octave

Licence	Octave i balíček queueing - GNU licence
Datum poslední verze	23.6.2017
Dokumentace	Ano
Umístění zdroj. kódu	http://hg.code.sf.net/p/octave/queueing
Vlastnosti	- Vytvoření Markovských řetězců s diskr. a spoj. časem pomocí matice pravděpodobností, resp. intenzit přechodu. - Ověření správnosti matice přechodu - Výpočet stacionárního rozdělení - Simulace řetězců - Výpočet střední doby do dosažení absorpčního stavu - Výpočet průměrného času stráveného v daném stavu

Tabulka 3.3: Přehled informací o nástroji Octave

Octave je jazyk zaměřený převážně na numerické výpočty. Jedná se o software, který je nabízen zdarma pod licencí `GNU General Public License (GPL)`. Jazyk je z velké části kompatibilní s již zmíněným Matlabem.

Octave v základu neobsahuje žádnou možnost, jak pracovat s Markovskými modely. Pro práci s nimi je nutné použít balíček `queueing`, který poskytuje funkce pro sítě front a analýzu Markovských řetězců.

Balíček `queueing` podporuje práci s Markovskými řetězci s diskrétním i spojitým časem. Funkce můžou být použity pro ověření správné matice přechodů, výpočet ustálených pravděpodobností, simulování chování řetězců

po určitý čas, výpočet střední doby do dosažení absorpčního stavu, výpočet průměrného času stráveného v daném stavu a tak dál.

K vypočtu stacionárního rozdělení a simulaci Markovských řetězců s diskrétním časem slouží funkce `dtmc`, u které je pro výpočet stacionárního rozdělení pouze jeden vstupní parametr a to matice pravděpodobností přechodu. K simulování má funkce navíc parametry udávající počet skoků a výchozí bod simulace. Obdobně je možné použít funkci `ctmc`, která slouží pro výpočet a simulaci u Markovských řetězců se spojitým časem.

3.5 PRISM

Licence	Open–source vydaný pod licencí GPL
Datum poslední verze	8.12.2018
Dokumentace	Ano, přehledná a srozumitelná.
Umístění zdroj. kódu	https://github.com/prismmodelchecker
Vlastnosti	- Jazyk pro popis Markovských řetězců s diskr. a spoj. časem. - Simulace Markovských řetězců - Komplexní analýza modelů pomocí dotazovacího jazyka

Tabulka 3.4: Přehled informací o nástroji PRISM

`PRISM`(verze 4.4) je pravděpodobnostní kontrolní model. Jedná se o bezplatný open–source nástroj pro formální modelování a analýzu systémů, které vykazují náhodné nebo pravděpodobnostní chování. `PRISM` umožňuje vytvářet a analyzovat několik pravděpodobnostních modelů, mezi které patří i Markovské modely s diskrétním a spojitým časem. [12]

Nástroj je velmi rozšířený a využívá se v mnoha různých aplikačních doménách, jako jsou například komunikační, multimediální a bezpečnostní protokoly, biologické systémy a kvantová kryptografie.

Modely se vytváří pomocí vlastního jazyka nástroje `PRISM`. Jazyk umožňuje definovat stavy a vazby mezi stavy a lze tak vytvářet složité modely. Základními prvky jazyka jsou moduly a proměnné. Model se může skládat z několika modulů, které mohou mít vlastní proměnné, anebo přistupovat ke sdíleným proměnným. Chování každého modulu je popsané sadou příkazů. Příkaz se skládá z předpokladu pro stav (například stav má hodnotu 1), po jehož splnění se vytvoří přechody mezi stavem splňujícím předpoklad a cílovým stavem s nadefinovanou hodnotou.

Nástroj umožňuje analyzovat širokou škálu vlastností modelu. K analýze se používá jazyk, který zahrnuje temporální logiku PCTL, CSL a LTL [12]. Lze tak zjistit například pravděpodobnost výskytu v daném uzlu v určitém čase, velikost fronty za určitou dobu nebo čas do přetečení dostupného bufferu.

3.6 Storm

Licence	open–source vydaný pod licencí GPL
Datum poslední verze	8.12.2018
Dokumentace	Ano. Stručná a přehledná s odkazy na dokumentace jiných použitých nástrojů jako je například PRISM
Umístění zdroj. kódu	https://github.com/moves-rwth/storm
Vlastnosti	- Jazyk pro popis Markovských řetězců s diskr. a spoj. časem. - Simulace Markovských řetězců - Komplexní analýza modelů pomocí dotazovacího jazyka

Tabulka 3.5: Přehled informací o nástroji Storm

`Storm` je open–source nástroj a slouží pro analýzu systémů zahrnujících náhodné nebo pravděpodobnostní jevy stejně jako nástroj `PRISM`. `Storm` stejně jako `PRISM` umí simulovat a analyzovat několik pravděpodobnostních modelů, mezi které opět patří i Markovské modely s diskrétním a spojitým časem.

Jako zdrojový kód pro vytvoření modelu je možné použít jazyk `PRISM` nebo některý z modelovacích jazyků jako je `JANI`, `GSPNs`, `PNML` [11].

Pro získávání vlastností modelu se používá rozšířená podmnožina jazyka `PRISM`, anebo jako alternativu pro dotazování lze využít modelovací jazyk `JANI`. `Storm` tedy umožňuje dotazování na vlastnosti podobně jako `PRISM`.

3.7 Shrnutí nástrojů

Nalezené webové nástroje slouží převážně jako materiál pro pochopení Markovských řetězců s názornou ukázkou simulace, zatímco nástroje jako je PRISM a Storm slouží ke komplexní analýze modelů pomocí optimalizované

simulace a dotazovacího jazyka na vlastnosti modelu, kde není vizuálně zobrazené chování Markovských řetězců. Obdoba dotazovacího jazyka je použita i u programu Markov2, ale dotazování se provádí pouze nad stacionárním rozdělením u modelu bez absorpčních stavů.

Programy Matlab a Octave po přidání potřebných rozšíření dokážou vytvářet a simulovat Markovské řetězce. Výsledek simulace je možné vizualizovat a k nalezení analytického řešení postačují funkce v základních programech bez rozšíření.

Ani jeden ze zmíněných nástrojů neodpovídá aplikaci, která má být vytvořena v rámci této diplomové práce, ale můžou sloužit pro inspiraci a kontrolu výsledků u vytvořené aplikace.

4 Požadavky na aplikaci

V následující části budou popsány požadavky na aplikaci, jako je běhové prostředí aplikace, vytváření a simulace Markovských modelů, interakce s modelem, dotazování na vlastnosti modelu a další funkce jako je například import a export grafu.

4.1 Prostředí pro běh aplikace

Nástroj musí fungovat jako webová aplikace, která poběží v prohlížeči na straně klienta bez potřeby dotazování na jakýkoliv server. Aplikace bude zaměřená hlavně na desktopový prohlížeč Chrome[1] od společnosti Google.

Důvodem upřednostnění Chromu oproti ostatním prohlížečům je jeho rozšíření podle statistik[2] z ledna 2019 na 70% desktopech [4]. Jelikož aplikace bude zaměřený pouze na omezený okruh uživatelů (převážně studenti IT), tak další prohlížeče jako je například druhý nejpoužívanější prohlížeč Firefox[3] (pouze 9,5% uživatelů), lze částečně zanedbat a zaměřit se primárně na prohlížeč Chrome.

4.2 Vizualizace modelu

Markovský model by měl být přehledně zobrazen jako graf pomocí uzlů a hran, kde uzel představuje stav modelu a hrana přechod z jednoho stavu do dalšího stavu.

U každého uzlu musí být zobrazena pravděpodobnost výskytu a uzel bude obarven podle velikosti této hodnoty tak, aby bylo možné snadno vizuálně rozeznat uzly s nízkou a vysokou pravděpodobností. Podobným způsobem by se měla zobrazovat hodnota u hran, která přestavuje pravděpodobnost, resp. intenzitu přechodu. Jednotlivé hrany budou také vizuálně odlišitelné podle velikosti jejich hodnoty.

[1] https://www.google.com/chrome/

[2] http://gs.statcounter.com/browser-market-share/desktop/worldwide

[3] https://www.mozilla.org/cs/firefox/new/

Dále by se uzly měly zobrazit přehledně a v rámci Markovských řetězců je vhodné, aby uzly šly uspořádat do pravidelných útvarů, jako je úsečka, trojúhelníková a nebo čtvercová síť.

Graf může nabývat velkých rozměrů, a tak je vhodné, aby šel přibližovat a oddalovat. S tím je spojen i přesun a změna náhledu na graf.

4.3 Modifikace grafu

Pouhá vizualizace grafu například z připraveného souboru nebo pomocí generátoru grafu (bude popsáno v kapitole 4.4) bez možnosti jednoduché úpravy nedostačuje a je vhodné, aby bylo možné graf upravovat.

Aplikace by tedy měla umožňovat modifikace, jako je přidávání, mazání a přemisťování stavů(uzlů). Dále by mělo být možné smazat nebo přidat hranu a k ní přiřadit hodnotu, která bude znamenat pravděpodobnost, resp. intenzitu přechodu. Při zadávání intenzity přechodu se musí kontrolovat, zda je hodnota kladné číslo. V případě pravděpodobnosti se musí navíc kontrolovat, zda zadaná hodnota i se součtem výstupních pravděpodobností z výchozího stavu nepřesahuje hodnotu 1 (viz kapitola 2.1.1). Tato přiřazená hodnota k hraně by měla jít změnit i po vytvoření hrany.

4.4 Vytváření modelu

Model bude možné vytvořit pomocí akcí zmíněných v předchozí kapitole 4.3. To je vhodné především pro tvorbu malých modelů a jejich úpravu, ale pro tvorbu větších grafů je tento způsob tvorby modelu náročný.

Část zmíněných nástrojů (kapitola 3) využívá k tvorbě grafu vlastní popisovací jazyk. Popisovací jazyk umožňuje snadno a rychle vytvářet komplexní grafy, proto by měla i navrhovaná aplikace obsahovat určitou formu popisovacího jazyka.

Popisovací jazyk by měl minimálně umožňovat vytvářet modely způsobem jako nástroj Markov2 (viz kapitola 3.2). Jedná se hlavně o definování modelu pomocí smyček, které se budou pro tvorbu grafu používat nejčastěji.

4.5 Simulace Markovských řetězců

Simulace Markovského řetězce se musí provádět po jednotlivých krocích a aktuální hrana, resp. uzel by měl být zvýrazněn. Před simulací by se měl buď automaticky vybrat výchozí bod nebo ho bude moci zvolit, resp. změnit uživatel.

Simulaci bude možné pozastavit, resetovat a urychlit, resp. zpomalit. Dále by mělo být možné vykonat skok o libovolný počet kroků dopředu a dále pokračovat v simulaci původním způsobem.

4.6 Analytické řešení

Analytické řešení se bude provádět nad vytvořeným grafem. Jelikož pro nespojitý graf neexistuje netriviální řešení, je v první řadě nutné ověřit, zda je graf silně souvislý. Pokud graf nesplňuje podmínku pro silnou souvislost, měl by být uživatel informován o této situaci. V opačném případě se provede výpočet stacionárního rozdělení a vypočítané pravděpodobnosti se zobrazí u příslušných uzlů.

4.7 Přepínání modelů

Aplikace musí umožňovat přepínání mezi Markovskými řetězci s diskrétním a spojitým časem. Uživatel musí být upozorněn na přepnutí režimu a nemožnost vrátit původní hodnoty hran, které jsou změněny přepočtem z pravděpodobností na intenzity nebo naopak.

Změna modelu se musí promítnout do chování simulace, kontroly vstupních parametrů při zadávání hodnoty u hrany a výpočtu analytického řešení.

4.8 Dotazování na vlastnosti modelu

Během simulace nebo po výpočtu analytického řešení musí nástroj umožnit získávání vlastností modelu. V nalezených nástrojích (viz kapitola 3), jako je PRISM, Markov2 nebo Storm, se k získávání vlastností používal dotazovací jazyk. Konkrétně u Markov2 byl dotazovací jazyk zaměřen na výběr uzlů v modelu a agregaci pravděpodobností těchto uzlů.

Výběr uzlů a agregace jejich pravděpodobností výskytu je v rámci vytvářeného nástroje dostačující. Jelikož nástroj bude disponovat grafickým a interaktivním modelem, není nutné používat dotazovací jazyk. Místo dotazovacího jazyka bude tedy možné vybrat uzly v grafickém náhledu modelu a vytvářet tak skupiny uzlů.

Každé skupině by měl jít přidat název a zobrazit součet pravděpodobností uzlů ve skupině. Jednotlivé skupiny pak půjdou zobrazit jako tabulka s jejich názvem a součtem pravděpodobností. Součet se bude automaticky aktualizovat při změně pravděpodobnosti uzlu v rámci skupiny.

Další vhodnou vlastností ke sledování je změna pravděpodobnosti výskytu v průběhu času. Tato změna může být zaznamenávaná během simulace a zobrazena pro každý stav jako spojnicový graf, kde bude zaznamenaná pravděpodobnost výskytu v závislosti na počtu vykonaných kroků.

4.9 Další funkce

Nástroj by měl obsahovat funkce jako je export a import dat. Funkce budou sloužit k uložení rozdělaného grafu a k jeho opětovnému nahrání do programu. Kromě exportu a importu vytvořeného grafu bude součástí dat i informace o modelu(diskrétní nebo spojitý), skupině stavů pro analýzu a případně i kód, kterým byl graf vytvořen.

Dále by měl nástroj umožnit export stavů i s jejich posloupnostmi pravděpodobností z každého kroku simulace (viz kapitola 4.8). Data můžou uživatelům sloužit pro další zpracování mimo vytvářený program a jejich import není potřeba.

5 Návrh nástroje

V této kapitole bude popsán návrh tvorby nástroje. V první řadě budou vybrány technologie pro vytvoření nástroje na základě požadavků. Následně budou popsány existující knihovny pro vizualizaci a interakci s grafem. Z těchto knihoven bude vybrána knihovna vhodná pro implementaci nástroje. Po výběru technologie a knihovny pro vizualizaci grafu bude navrženo vykonávání akcí nad grafem. Dále bude navržen způsob, jakým se bude vytvářet graf. V poslední kapitole bude popsaný způsob vizualizace pravděpodobností navštívení uzlů.

5.1 Výběr technologie

Jedním z hlavních požadavků na program je jeho provoz ve webovém prohlížeči. Z tohoto důvodu bude výběr zaměřen pouze na technologie pro tvorbu webových aplikací. V první řadě budou popsány jazyky HTML a CSS. V dalších kapitolách budou představeny jazyky použitelné pro implementaci dynamického chování nástroje. K jazykům budou popsány i možnosti provedení jednotkových testů, které budou potřeba pro otestování aplikace. V poslední kapitole bude shrnutí jazyků i s výběrem vhodného jazyka pro implementaci.

5.1.1 HTML

HTML (HyperText Markup Language) je značkovací jazyk používaný pro tvorbu webových stránek. Definuje význam a strukturu webového obsahu pomocí značek. Značky označují například různé úrovně nadpisů, odstavce, odkazy a další prvky stránky.

V diplomové práci bude použita nejnovější verze jazyka `HTML5`. Oproti starší verzi `HTML4` se `HTML5` zaměřilo na jednodušší a rychlejší zápis značek. `HTML5` navíc umožňuje vytvořit aplikaci fungující bez připojení k internetu a je možné pracovat s lokálním úložištěm.

5.1.2 CSS

CSS(Cascading Style Sheets) je jazyk navržený organizací World Wide Web Consortium (W3C). Popisuje, jakým způsobem se budou zobrazovat elementy značkovacího jazyka (např. HTML) ve webové stránce. CSS odděluje prezentaci od obsahu a dovoluje definovat například barvu, font, layout a

nebo pozadí. Oddělení prezentace od obsahu umožňuje zobrazení stejného obsahu různým způsobem na rozdílných zařízeních, jako je například monitor, obrazovka tabletu a nebo mobilu[20].

5.1.3 Javascript

Autor/tým	Brendan Eich
Datum vzniku	červenec 1997
Poslední verze	ES9 / ES2018 (rok 2018)
Url	https://www.javascript.com/

Tabulka 5.1: Přehled k jazyku Javascript

Javascript je interpretovaný programovací jazyk využívaný hlavně při tvorbě webových aplikací. Javascript byl původně vytvořen jako skriptovací jazyk určený pouze pro použití v prohlížečích, ale rozšířil se i na další prostředí. Může sloužit pro tvorbu dynamického obsahu na stránkách, obsluhovat události, pracovat s daty z lokálního úložiště na klientské straně atd.

Jazyk je dynamicky typovaný, tzn. že většina typové kontroly je prováděna až za běhu, místo při kompilaci. Javascript umožňuje použití objektově orientovaného, imperativního a funkcionálního programovacího stylu.

Pro jednotkové testy Javascriptu existuje celá řada knihoven. Mezi hlavní knihovny patří například Jasmine.js[1], QUnit[2] a Mocha.js[3].

Jasmine.js

Licence	MIT License
Datum vzniku	10.9.2011
Poslední verze	3.3.0(25.10.2018)
Url	https://jasmine.github.io/

Tabulka 5.2: Přehled k jazyku Jasmine

[1] https://jasmine.github.io/
[2] https://qunitjs.com/
[3] https://mochajs.org/

Jasmine.js (tabulka 5.2) je open–source framework poskytovaný zcela zdarma pod licencí MIT. Jedná se o velmi populární nástroj k provádění jednotkových testů u programů napsaných v Javascriptu [10]. Vývoj nástroje je stále aktuální a velmi často dochází k vydávání verzí s novými funkcemi (viz repozitář na Githubu[4]).

Mocha.js

Licence	MIT License
Datum vzniku	22.11.2011
Poslední verze	6.0.2 (25.2.2019)
Url	https://github.com/mochajs/mocha

Tabulka 5.3: Přehled k jazyku Mocha

Mocha (tabulka 5.3) je framework určen pro Javacsript běžící v Node.js a webovém prohlížeči. Framework je také dostupný jako open–source[5] pod licencí MIT.

Nástroj Mocha se stal oblíbený kvůli jeho flexibilitě, která dává možnost vytvářet vlastní testovací knihovny, a jednoduchosti vytvářet modulární jednotkové testy [10].

QUnit

Licence	Apache v2.0
Datum vzniku	2008
Poslední verze	2.9.2 (21.2.2019)
Url	https://qunitjs.com/

Tabulka 5.4: Přehled k jazyku QUnit

QUnit (tabulka 5.4) je nástroj vytvořený od tvůrců knihovny jQuery[6] k testování vlastních produktů. QUnit je poskytován zdarma a jako open–source[7] nástroj pod licencí Apache v2.0 license[8].

QUnit umožňuje snadno vytvářet jednotkové testy a jak uvádí tvůrci, nástroj by měl testy vykonat a vyhodnotit okamžitě [9].

[4]https://github.com/jasmine/jasmine
[5]https://github.com/mochajs/mocha
[6]https://jquery.com/
[7]https://github.com/qunitjs/qunit
[8]https://www.apache.org/licenses/LICENSE-2.0

Karma

Licence	MIT license
Datum vzniku	18.3.2013
3.1.4	3.1.4 (17.122018)
Url	https://karma-runner.github.io/3.0/index.html

Tabulka 5.5: Přehled k jazyku Karma

Karma (tabulka 5.5) je nástroj pro testování vytvořený tvůrci frameworku AngularJS[9]. AngularJS nástroj poskytuje zdarma a jako open–source pod licencí MIT.

Karma spouští webový server, který testuje zdrojový kód aplikace na základě vytvořených testů. Testy jsou tedy vykonány přímo v konkrétním prohlížeči a výsledek je zaspaný do konzole.

5.1.4 Typescript

Autor/tým	Microsoft
Licence	Apache License 2.0
Datum vzniku	1. října 2012
Poslední verze	TypeScript 3.4.3 (9.4.2019)
Url	https://www.typescriptlang.org/

Tabulka 5.6: Přehled k jazyku Typescript

Typescript je programovací jazyk vyvíjený společností Microsoft. Je vydáván jako open–source pod licencí Apache License 2.0. Jedná se o nadstavbu Javascpitu, který Typescript rozšiřuje o statickou typovou kontrolu, třídy, rozhraní, přístup k atributům třídy a další prvky známé z OOP.

Pro kompilaci se používá tzv. transpiler. Transpiler převádí jeden jazyk na jiný stejně fungující kód. To umožňuje vytvářet kód v jiném jazyce bez obavy o funkčnost výsledného kódu. V případě Typescriptu se kód převede na Javascript, který již ale neobsahuje kontrolu datových typů a nedochází k žádnému zpomalení.

Jelikož se jedná pouze o nadstavbu Javascriptu, je možné použít i již existující kód napsaný v Javascriptu. Pro použití statické typové kontroly

[9] https://angularjs.org/

u Javasrciptových knihoven je nutné přidat definice typů. U velkých a používaných knihoven jsou již tyto definice vytvořené a dostupné ke stažení z Githubu[10], kde se nachází přes 4000 definicí.

Aplikaci napsanou v Typescriptu je možné testovat pomocí nástroje Mocha (viz kapitola 5.1.3). Dalším možným nástrojem k testování je například Karma, který je poskytovaný jako npm balíček[11] pro Typescript.

5.1.5 Dart

Autor/tým	Google
Licence	BSD
Datum vzniku	10.9.2011
Poslední verze	2.2 (26.2.2019)
Url	https://www.dartlang.org/

Tabulka 5.7: Přehled k jazyku Dart

Dart je objektově orientovaný programovací jazyk vytvořený společností Google. Jedná se o celkem nový open–source jazyk dostupný pod licencí BSD. Umožňuje například definovat třídy, statické typování, dědičnost a rozhraní [5].

Dart je určený převážně pro webové prohlíže. Jazyk není možné spustit v prohlížeči, ale překládá se do Javascriptu. Nejedná se o transpilaci jako v případě Typescriptu. Javascript je zde používaný jako bytekód určený pro běh v prohlížeči.

Pro jednotkové testy stačí do nástroje přidat Dart balíček pro testy. Balíček postačuje a není nutné používat knihovny třetích stran [6].

5.1.6 CoffeeScript

Autor/tým	Jeremy Ashkenas
Licence	MIT
Datum vzniku	13.12.2009
Poslední verze	2.4.1 (7.4.2019)
Url	https://coffeescript.org/

Tabulka 5.8: Přehled k jazyku Coffeescript

[10] https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types
[11] https://www.npmjs.com/package/karma-typescript

CoffeeScript je programovací jazyk, který se kompiluje do jazyka Javascript. Přidává tzv. syntaktický cukr inspirovaný jazyky Ruby, Python a Haskell, aby se zvýšila čitelnost a přehlednost Javascriptu [1].

Jelikož výsledný kód je Javascript, je možné použít pro jednotkové testy stejné knihovny jako v případě Javascriptu (viz kapitola 5.1.3).

5.1.7 Shrnutí

V kapitole bylo popsáno několik jazyků, jejichž výsledným kódem je Javascript spustitelný v prohlížeči. U všech zmíněných jazyků je možné vykonávat jednotkové testy pomocí knihoven.

Jazyky jako je Typescript, Dart nebo CoffeeScript zpřehledňují a zjednodušují zápis původního jazyka Javascript. Dále Javascript rozšiřují o další prvky používané v OOP a v případě Typescriptu a Dartu přidávají statickou kontrolu datových typů.

K tvorbě aplikace pro Markovské modely je možné použít jak samotný Javascript, tak některý ze zmíněných jazyků. Ze všech těchto nástrojů byl vybrán jazyk Typescript. Typescript je oproti Javascriptu zaměřen více na objektově orientované programování a umožňuje tak vytvořit konzistentní, čistější a jednodušší kód. Zaměření jazyka na OOP a syntaxe podobná s jazykem Java je vhodná i pro autora této diplomové práce. Další výhodou Typescriptu je statická kontrola, která je vhodná pro napovídání v kódu a automatickou kontrolu datových typů. To samé nabízí i jazyk Dart, ale co se týče knihoven, tak většina grafových knihoven (viz následující kapitola 5.2) nemá vytvořené Dart API pro snadné zacházení, zatímco pro Typescript jsou dostupné definice typů pro většinu knihoven.

5.2 Výběr knihovny pro vizualizaci grafů

V této kapitole bude popsáno několik volně dostupných knihoven pro vizualizaci síťového grafu. Existuje celá řada knihoven podporující interaktivní vizualizaci. Z možných knihoven budou probrány pouze tři hlavní open–source knihovny, jako je vis.js, D3.js a Cytoscape.js.

V závěru kapitoly bude vyhodnocení knihoven a bude vybrána knihovna, která umožní jednoduchým způsobem implementovat veškerou požadovanou funkčnost ohledně grafu.

5.2.1 vis.js

Licence	Apache 2.0 a MIT
Datum vzniku	16.4.2013
Poslední verze	v4.21.0 (12.10.2017)
Url	http://visjs.org/index.html#

Tabulka 5.9: Přehled ke knihovně vis.js

Knihovna vis.js (tabulka 5.9) slouží k zobrazení, manipulaci a interakci s velkým množstvím dynamických dat. Jedná se o open–source knihovnu vydávanou pod licencemi MIT a Apache 2.0. Poslední verze byla vydaná v roce 2017 a vývoj na knihovně byl ukončen. Na stránkách vis.js je k dispozici stručná dokumentace doplněná o příklady.

Kromě dalšího zpracování dat umí knihovna pracovat se síťovým grafy. Dovoluje zobrazit, interagovat a manipulovat s grafem. Rozhraní pro manipulaci s grafem poskytuje převážně základní funkce, jako je například přidávání a mazání uzlů, resp. hran. Výchozí layout uzlů knihovny je tvořen pomocí tzv. pružin a dále nabízí pouze hierarchické uspořádání uzlů.

5.2.2 D3.js

Licence	BSD
Datum vzniku	18.2.2011
Poslední verze	v5.9.1 (10.2.2019)
Url	https://d3js.org/

Tabulka 5.10: Přehled ke knihovně d3.js

D3.js (tabulka 5.10) je komplexní open–source knihovna (licence BSD) pro vytváření dynamické a interaktivní vizualizace dat ve webovém prohlížeči. Umožňuje velmi rychlé zobrazení a interakci nad rozsáhlými daty. Knihovna nabízí mnoho způsobů, jak data vizualizovat a dává k dispozici velké množství názorných ukázek a přehlednou dokumentaci rozhraní, které obsahuje stovky metod.

Knihovna se zaměřuje převážně na statické vizualizace, ale také umí vizualizovat a manipulovat se síťovým grafem, obsahuje mnoho možností pro uspořádání uzlů a dovoluje nastavovat velké množství vlastností ohledně vzhledu grafu.

5.2.3 Cytoscape.js

Licence	MIT License
Datum vzniku	10.9.2015
Poslední verze	v3.3.5(19.2.2019)
Url	http://js.cytoscape.org/

Tabulka 5.11: Přehled ke knihovně Cytoscape.js

Cytoscape.js (tabulka 5.11) je Javascriptová knihovna zaměřená na interaktivní vizualizaci síťových grafů ve webovém prohlížeči. Tato knihovna vznikla jako nezávislý open–source projekt od tvůrců desktopové verze programu Cytoscape.

Vývojářská komunita knihovny Cytoscape.js je stále aktivní a průběžně vydává nové verze. Ke knihovně je vytvořená přehledná dokumentace, která obsahuje téměř ke každé metodě z rozhraní ukázku kódu a vizualizaci.

Zaměření knihovny pouze na síťové grafy přináší výhody v obsáhlém rozhraní tvořící velké množství užitečných metod pro práci s grafem. Rozhraní obsahuje kromě základních metod pro interakci i metody pro animaci a rozšířené nastavení různých vlastností hran a uzlů.

Dále je knihovna optimalizována pro velký počet uzlů a umožňuje odchytávání akcí při interakci s grafem.

5.2.4 Shrnutí

Všechny tři zmíněné knihovny poskytují vhodné rozhraní pro vizualizaci a interakci s grafem. V případě vis.js se jedná o rozhraní převážně se základními metodami pro práci s grafem, zatímco D3.js dává k dispozici velmi komplexní rozhraní a možnosti pro zobrazení dat. Orientace v tak velké knihovně může být složitá a splnění požadavků na animaci a zobrazení grafu nemusí být snadné.

Nejlepší rozhraní pro síťový graf poskytuje knihovna Cytoscape.js, která navíc obsahuje metody pro animaci přechodů mezi uzly a dává k dispozici velký výběr rozložení uzlů. Další výhodou knihovny je zaměření přímo na síťové grafy, a tím přináší i snadnější a rychlejší orientaci v knihovně.

5.3 Akce nad grafem

Jedním z požadavků na aplikaci je možnost modifikace grafu (viz kapitola 4.3). Modifikací se rozumí akce jako přidávání, mazání a editace uzlů, resp. hran. Aplikace musí tyto akce poskytovat uživatelsky příjemným způsobem.

Vhodný způsob, jak by mohla aplikace umožňovat modifikace, je pomocí přepínání mezi režimy pro výběr (`Select`), přidání (`Add`) a mazání (`Delete`). V režimu `Add` a `Delete` bude možné pouze přidávat, resp. mazat uzly a hrany. Režim `Select` umožní vybírat uzly a vytvářet tak skupiny uzlů (viz kapitola 4.8). Dále bude možné v režimu `Select` ovládat simulaci Markovského řetězce a editovat hodnoty hran.

Dále půjde modifikace provádět přes kontextové menu, které se zobrazí po kliknutí na pravé tlačítko myši. Menu bude obsahovat akce v závislosti na tom, kam uživatel klikl. Pokud například uživatel vyvolá kontextové menu nad uzlem, bude možné jeho smazání nebo určení uzlu jako výchozího pro simulaci.

5.4 Vytvoření grafu

V následujících kapitolách bude popsán způsob, jakým se bude vytvářet graf. V první řadě se zváží přístupy, jak nejsnáze vytvořit graf. Dále budou popsány způsoby, jak vhodně rozmístit stavy. Poté bude detailně popsán způsob, jak se bude vytvářet graf.

5.4.1 Způsoby pro vytvoření grafu

V uvedených nástrojích (kapitola 3) se používá několik způsobů, jak vytvořit graf popisující Markovský řetězec. Jeden ze způsobů je vytváření grafu pomocí matice sousednosti. Do vyobrazené matice se přidávají hodnoty hran, ale tento způsob znemožňuje vytvářet větší grafy a zadávání je pracné.

Dalším způsobem je použití modelovacího jazyka, pomocí kterého se definují stavy a přechody. Jazyk navíc dovoluje vytvářet například smyčky a podmínky a tím je vytváření velkých modelů jednodušší a rychlejší. Použití modelovacího jazyka je pro tvorbu grafů velmi vhodné, a proto bude použit jazyk obdobným způsobem i v diplomové práci.

5.4.2 Layout pro uzly

Při použití modelovacího jazyka nastává problém v tom, jakým způsobem rozmístit uzly na plátno tak, aby graf byl přehledný a hrany se protínaly jen minimálně. K rozmístění uzlů poskytuje zvolená knihovna Cytoscape.js několik možností uspořádání.

Jednou z možností je použít uspořádání uzlů založené na pružinách (v knihovně `cose layout` a `cole layout`). Obě uspořádání byly vyzkoušeny, ale ani v jednom případě nebylo dosaženo vhodného rozložení uzlů. Tento přístup nebyl vhodný, protože je požadováno uspořádání uzlů převážně do pravidelných útvarů, jako je úsečka, trojúhelníková nebo čtvercová síť. Uspořádání pomocí pružin sice zobrazilo přehledně uzly a hrany, ale byl problém s vytvořením pravidelných útvarů.

Knihovna Cytoscape.js dále umožňuje zobrazit uzly v mřížce pomocí uspořádání `grid layout`. Uzlům lze definovat pozice v mřížce, a tím dát uzly například do úsečky.

5.4.3 Jazyk pro tvorbu grafu

Při použití uspořádání `grid layout` by bylo nejvhodnější vkládat uzly přímo do mřížky jako v případě nástroje Markov2, kde jazyk umožňuje definovat pozici stavu ve 2D mřížce.

Pokud by byla použita obdoba jazyka jako u Markov2, je třeba definovat gramatiku a vytvořit překladač. Vytvoření takového modelovacího jazyka je náročné i zbytečné, protože v případě Typescriptu lze vykonat Javascriptový kód pomocí funkce `eval()`. To znamená, že uživateli bude umožněno vytvořit graf pomocí jazyka Javascript. Pro jednoduché vkládání hran a uzlů do mřížky bude vhodné volat přímo funkci a uživateli se tak ulehčí práce s tvorbou matice.

Funkce pro přidání přechodu mezi dvěma stavy bude mít jako vstupní parametry pozici výchozího a konečného uzlu v mřížce a hodnotu přechodu. Funkce bude mít tedy tvar `addTransition([vychPozX,vychPozY], [konPozX,konPozY], hodnotaHrany)`. Funkci bude moci uživatel použít kdekoliv v Javascriptovém kódu pro vytvoření grafu (například uvnitř smyčky, podmínky atd.).

5.4.4 Editor

Kód Javascriptu pro vytváření grafu je vhodné psát do editoru, který zvýrazňuje a kontroluje syntaxi. K tomu je možné použít například knihovnu Ace nebo Monaco.

Monaco

Licence	MIT license
Datum vzniku	9.6.2016
3.1.4	v0.15.6 (23.11.2018)
Url	https://microsoft.github.io/monaco-editor/index.html

Tabulka 5.12: Přehled k editoru Monaco

Monaco (tabulka 5.12) patří mezi nejlepší webové editory a obsahuje celou řadu funkcí, které jsou známé z běžných editorů, jako je například Visual Studio Code. Nevýhodou editoru je jeho velká velikost (1,23 MB).

Ace

Licence	GNU GPL
Datum vzniku	4.4.2006
3.1.4	v1.4.3 (22.2.2019)
Url	https://ace.c9.io/

Tabulka 5.13: Přehled k editoru Ace

Ace (tabulka 5.13) má velikost pouze 120 kB, ale i přesto dává k dispozici užitečné funkce, jako je automatické odsazování, zvýraznění a kontrola syntaxe. Kvůli jeho malé velikosti a dostatečných funkcí byl zvolen tento editor.

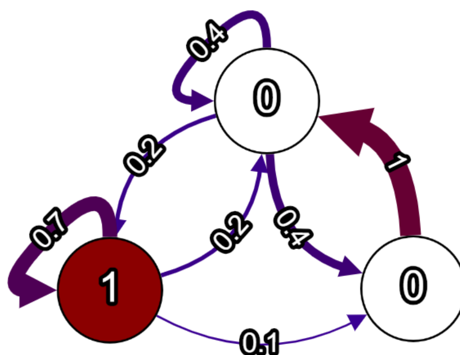
5.5 Zobrazení informací o uzlech

Během simulace se budou ukládat pravděpodobnosti navštívení uzlů v každém kroku. Tyto hodnoty je potřeba zobrazit tak, aby bylo vidět, jak se pravděpodobnosti mění v průběhu času (viz kapitola s požadavky 4.8). Takovou informaci je vhodné vizualizovat pomocí spojnicového grafu, kde na ose x budou jednotlivé kroky a na ose y budou pravděpodobnosti.

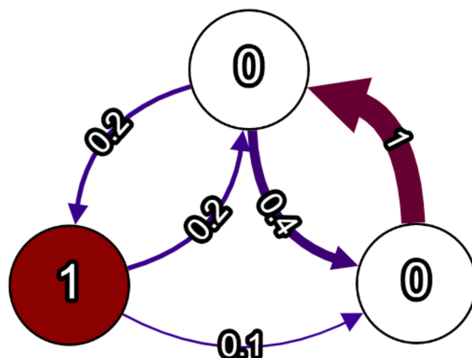
K zobrazení grafu lze použít již zmíněnou knihovnu D3.js (kapitola 5.2.2). Knihovna umí vizualizovat spojnicové grafy snadným a přehledným způsobem.

5.6 Změna typu Markovského řetězce

Při změně typu Markovského řetězce s diskrétním časem (obrázek 5.1) na Markovský řetězec se spojitým časem se hodnoty přechodů nebudou měnit. Budou odstraněny pouze přechody (smyčky), vedoucí ze stavu do něj samotného (viz obrázek 5.2).



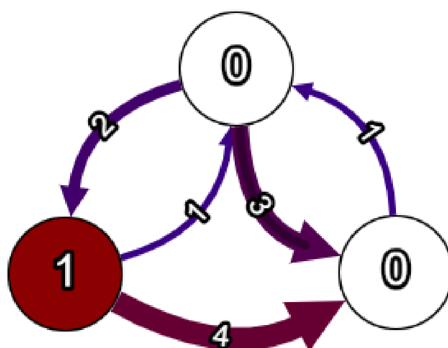
Obrázek 5.1: Původní Markovský řetězec s diskrétním časem



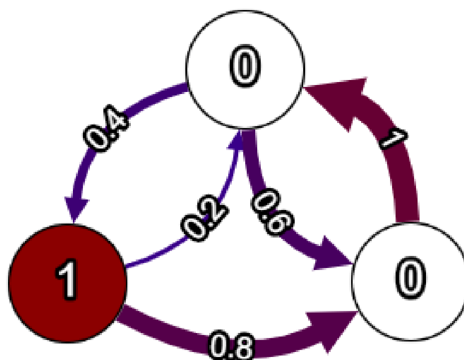
Obrázek 5.2: Převedený Markovský řetězec se spojitým časem

Při změně ze spojitého času (viz obrázek 5.3) na diskrétní čas se hodnoty přechodů přepočítají. Výsledná hodnota přechodu bude daná poměrem výstupních intenzit (viz obrázek 5.4).

Změna typu řetězce vede ke změně hodnot a při návratu k původnímu typu bude mít model rozdílné hodnoty než měl před přepnutím. Proto je vhodné uživatele informovat o nevratné změně hodnot a zobrazit mu potvrzovací dialogové okno.



Obrázek 5.3: Původní Markovský řetězec se spojitým časem



Obrázek 5.4: Převedený Markovský řetězec s diskrétním časem

5.7 Návrh implementace

Pro implementaci by bylo možné použít například MVC framework AngularJs. Ten umožňuje snadno oddělit vzhled od logiky aplikace, ale vzhledem k charakteru a malému rozsahu aplikace není nutné a ani vhodné zavádět další technologii do implementace nástroje.

I přesto by mělo být dodržené oddělení vzhledu od ostatní funkčnosti. V implementaci budou tedy odděleny třídy s HTML komponentami a jejich zobrazením od tříd s akcemi, které bude možné v rámci těchto komponent provádět.

6 Implementace nástroje

Pro implementaci nástroje byl v předchozí kapitole zvolen jazyk Typescript společně s knihovnou pro tvorbu grafů Cytoscape.js. Nástroj bude implementován v editoru Visual Studio Code s počáteční konfigurací[1] pro vývoj webové aplikace v Typescriptu.

6.1 Organizace zdrojového kódu

Kořenová složka se zdrojovými obsahuje složku `src` se zdrojovými kódy aplikace a složku `tests` s testy aplikace. Dále se zde nachází konfigurační soubory pro testy a vývoj v jazyce Typescript.

6.1.1 Konfigurační soubory

Konfigurační soubor `tsconfig.json` obsahuje konfiguraci týkající se nastavení překladače z jazyka Typescript do jazyka Javascript. Soubor `tslint.json` obsahuje pravidla pro zobrazení Typescript kódu v editoru. Další konfigurační soubor `package.json` obsahuje skripty pro sestavení a spuštění aplikace a definice jak knihoven potřebných pro vývoj (například Typescript, WebPack, Karma), tak externích knihoven pro aplikaci (Cytoscape.js, Ace).

Poslední konfigurační soubor `webpack.config.js` slouží pro nastavení Webpacku, který zpracuje kód napsaný modulárně v Typescriptu a vytvoří z něj Javascriptový balíček. Webpack tímto způsobem vytvoří aplikaci spustitelnou ve webovém prohlížeči.

6.1.2 Zdrojové kódy aplikace

Ve složce `src` se nachází veškerý zdrojový kód aplikace. Soubory se zdrojovým kódem jsou rozdělený do složek podle technologie, ve které jsou napsané (HTML, JS, CSS, TS a JSON).

Složka HTML obsahuje soubory se základní strukturou hlavních HTML komponent, jako je například navigační lišta s akcemi (`navbar.html`), editor pro tvorbu grafu pomocí popisovacího jazka (`codeEditor.html`), seznam skupin uzlů grafu (`nodeGroups.html`), panel pro ovládání simulace (`animationPanel.html`) a dialogové okno se vstupem (submitDialog.html).

[1] https://github.com/nobrainr/typescript-webpack-starter

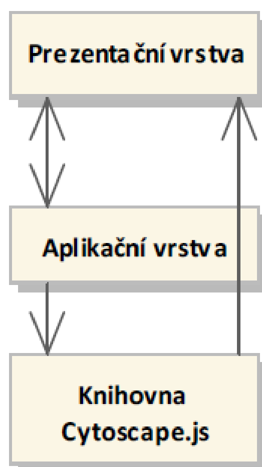
Dále obsahuje hlavní soubor `index.html`, který udává rozvržení jednotlivých komponent a importuje ze složky `JS` skripty Javascriptových knihoven, pro které neexistují definice typů. Takto importované knihovny je možné následně použít v jazyce Typescript i bez definování typů. Styly komponent jsou ve složce CSS.

Složka JSON obsahuje soubory s exporty dat z aplikace. Exportovaná data můžou být použita jako výchozí nastavení pro aplikaci.

Ve složce TS se nachází zdrojové kódy aplikace napsané v Typescriptu, které jsou rozděleny do několika podsložek. Složky `htmlComponents` a `actions` obsahují třídy oddělující vzhled (složka `htmlComponents`) od funkčnosti (složka `actions`). Dále jsou zde adresáře pro třídy týkající se simulace, editoru grafu a úpravy vzhledu grafu.

6.2 Architektura implementace

Implementaci je možné rozdělit do tří vrstev (viz obrázek 6.1). První vrstva je vrstva prezentační. Ta se stará o zobrazení jednotlivých částí aplikace v HTML. Druhá vrstva obsahuje aplikační logiku. Zajišťuje zpracování obsluh událostí z první vrstvy, nastavuje v ní data pro zobrazení a zajišťuje veškerou komunikaci s knihovnou Cytoscape.js, která se svojí provázaností s aplikací dá považovat za třetí vrstvu aplikace pro zobrazování a interakci s grafem.



Obrázek 6.1: Znázornění rozdělení aplikace na vrstvy

6.2.1 Prezentační vrstva

Prezentační vrstva zahrnuje třídy z balíčku `htmlComponents`, které představují jednotlivé komponenty, jako je navigační panel, dialog s formulářem, ovládání animace, kontextové menu a postranní panel se skupinami uzlů a s editorem pro tvorbu grafu pomocí kódu. Tyto třídy používají pro vytváření vzhledu kód napsaný přímo v HTML (viz ukázka 6.1 metody `createAnimationPanel` ze třídy `animationActions`), anebo se HTML elementy vytváří pomocí Typescriptu. Obě tyto možnosti se u většiny tříd kombinují a HTML načtené ze souboru je doplněné o další elementy Typescriptem. Stylování elementu v rámci třídy probíhá pomocí importu příslušného CSS souboru.

```
private createAnimationPanel(){
    //nacteni HTML ze souboru
    let animationPanel = require('../HTML/animationPanel.html');
    // pridani nacteneho HTML do elementu s id animationPanel
    document.getElementById("animationPanel")
        .insertAdjacentHTML('beforeend',animationPanel);
}
```

Listing 6.1: Načtení a přidáná HTML ze souboru

V prezentační vrstvě aplikace se dále přidávají obsluhy událostí, jako je například kliknutí na tlačítka nebo změna měřítka (viz ukázka kódu části metody `registerListeners` ze třídy `animationActions` 6.2). Obsluha těchto událostí probíhá v aplikační vrstvě.

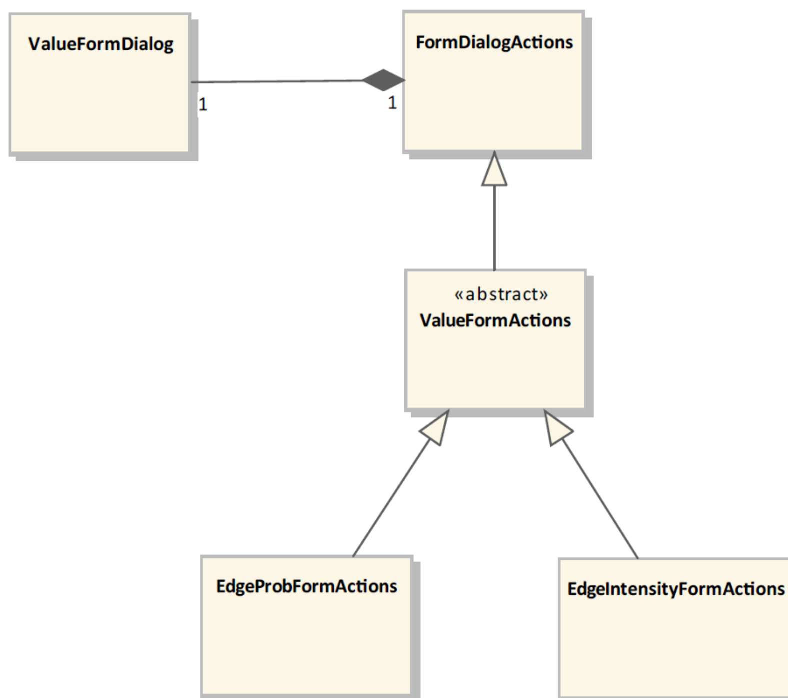
```
private registerListeners(){
    //ziskani tlacitka
    let runButton = document.getElementById("runButton");
    //nastaveni obsluhy kliknuti, ktere po kliknuti vola
    //metodu ze tridy animationActions ve druhe vrstve
    runButton.addEventListener("click",
        () => this.animationActions.runAnimation());
    ...
    let speedSlider = <HTMLInputElement>document
        .getElementById("speedSlider");
    //nastaveni obsluhy pri zmene posuvniku
    speedSlider.addEventListener("change",
        () => this.animationActions
            .changeSpeed(+speedSlider.max - +speedSlider.value));
}
```

Listing 6.2: Obsluha událostí

6.2.2 Aplikační vrstva

Aplikační vrstva obsahuje akce pro zpracování obsluh událostí z prezentační vrstvy, které jsou obsaženy v třídách z balíčku `actions`. Jedná se o třídy pro zpracování akcí z ovládání animace, navigační lišty, seznam skupin uzlů, a třídy pro zpracování formuláře. Ke komponentě z první vrstvy se tedy váže některá z těchto tříd, například komponenta pro navigační lištu (třída `NavbarMenu`) používá akce ze třídy `NavbarMenuActions`.

V případě komponenty pro dialogové okno (třída `ValueFormDialog`) existuje více tříd s rodičovskou třídou `FormDialogActions` (viz diagram 6.2), a je tak umožněno vytvářet stejné dialogové okno s různou kontrolou dat a chováním při potvrzení dialogu (bude popsáno podrobněji v kapitole 6.3.12).



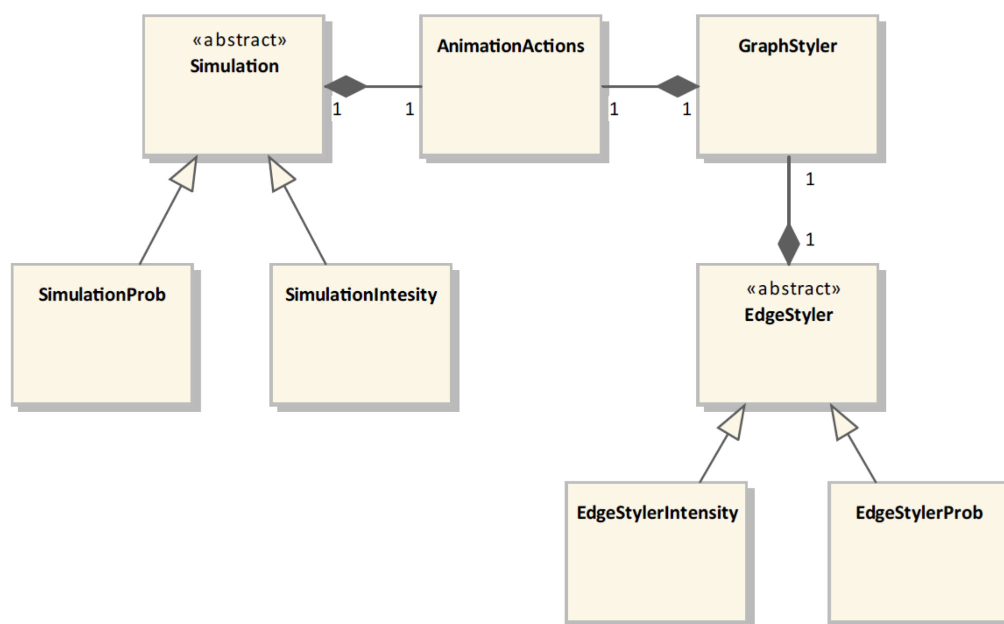
Obrázek 6.2: UML diagram tříd pro formulář

Dále se v aplikační vrstvě nachází třídy pro stylování grafu (`balíček graphStyler`), simulaci (`balíček simulation`) a vytváření grafu pomocí popisovacího jazyka (`balíček codeEditor`).

V případě stylování je nutné měnit styl hran, který se liší podle toho, jestli se jedná o Markovský řetězec s diskrétním časem anebo se spojitým časem (bude popsáno v kapitole 6.3.5). Proto třída pro stylování `GraphStyler` přistupuje k abstraktní třídě pro stylování hran `EdgeStyler`, od které dědí třídy

`EdgeStylerProbability` a `EdgeStylerProbability` (viz diagram 6.3), a tak je možné použít vhodnou třídu podle aktuálního typu Markovského řetězce.

U simulace se jedná o podobný případ jako u stylování hran. Zde je abstraktní třída `Simulation` s potomky `SimulationIntensity` a `SimulationProbability` (viz diagram 6.3). Abstraktní třídu `Simulation` obsahuje třída s akcemi pro animaci `AnimationActions` a umožňuje výběr hrany pro simulaci podle typu Markovského řetězce.



Obrázek 6.3: UML diagram tříd pro animaci

6.2.3 Knihovna Cytoscape.js

Třetí vrstvu tvoří knihovna Cytoscape.js. K této knihovně přistupuje většina tříd z aplikační vrstvy přes její instanci (`core`), což je hlavní vstupní bod do knihovny. Pomocí instance je možné například měnit layout, obsluhovat události a provádět další operace nad grafem. Knihovna si navíc uchovává veškerou strukturu a data o grafu a proto není nutné si tyto informace uchovávat v aplikační vrstvě. Zároveň knihovna sama provádí vizualizaci elementů do plátna.

Dále instance `core` poskytuje funkce pro přístup k elementům grafu, jako jsou hrany a uzly. Funkce vrací jejich kolekce a umožňují provádět nad nimi filtraci. Nad kolekcemi je pak možné provádět další operace, jako je například

zjišťování a nastavování dat jednotlivým elementům v kolekci. Zde je nutné si dát pozor na modifikace kolekcí (přidávání a odebíraní elementů), protože se úprava kolekce nepromítne do vizualizace grafu. Z tohoto důvodu je nutné provádět operace, jako je přidání a odebrání elementů přímo nad instancí `core`, anebo nad konkretním elementem v kolekci, kde element už je objekt s odkazem na instanci `core`.

Knihovna dovoluje k uzlům a hranám přiřazovat data. To se využívá například při uchování hodnoty přechodu hrany (`value`), pravděpodobnosti navštívení (`visitPercent`) a historie pravděpodobností u uzlů (`history`). Stylování uzlů i hran probíhá také pouze v rámci knihovny Cytoscape.js a elementy není možné stylovat pomocí CSS.

6.3 Popis implementace

6.3.1 Inicializace aplikace

Inicializace aplikace se provádí v souboru `index.ts`. Tento soubor je hlavním souborem pro spuštění Typescript aplikace.

V první řadě se zde se inicializuje graf pomocí metody `initGraph` ze tříd `GraphConstruction`. Metoda `initGraph` má za úkol vytvořit hlavní instanci grafu (`core`) z knihovny Cytoscape.js. Při vytváření se definuje kontejner pro zobrazení grafu, výchozí nastavení (měřítko, pozice na plátně), nastavení pro interakci s grafem a renderování grafu. Tyto vlastnosti byly nastaveny podle doporučeného nastavení[2] a následně některé hodnoty byly upraveny tak, aby interakce s grafem byla uživatelsky přívětivá a umožnila bezproblémové zacházení s grafem. Například nastavení minimálního přiblížení (minZoom) a maximálního možného oddálení (maxZoom) bylo změněno tak, aby vytvořený graf byl vždy viditelný a nezmizel při příliš velkém oddálení, resp. přiblížení.

Dále se zde inicializují hlavní třídy pro aplikaci, například se jedná o třídu pro správu skupin uzlů, ovládání animace, tvorbu navigační lišty atd. Tyto třídy budou popsány v následujících kapitolách.

6.3.2 Akce nad grafem

Akce nad grafem, jako je mazání, přidávání a editace, má na starost třída `GraphActions`. Třída se stará převážně o obsluhu událostí při interakci s plátnem pro vykreslování grafu. Naslouchá na kliknutí nebo výběr.

[2] http://js.cytoscape.org/#getting-started/initialisation

Třída si zároveň pamatuje, ve kterém režimu se nachází (Select, Add, Remove - popsáno v kapitole 4.3 o návrhu nástroje) a podle toho se zvolí příslušná metoda.

Režim Add

Pokud se jedná o režim `Add`, je volána metoda `addElement` a při kliknutí na prázdné místo dojde k přidání nového uzlu. Pro přidání uzlu je nutné specifikovat data, která jsou nutná pro vytvoření uzlu. Jedná se pouze o identifikátor (id) a další data jako označení (label). Pozici je možné nastavit až nad vytvořeným uzlem (viz ukázka metody `addNode` 6.3). V případě, že se na místě kliknutí vyskytuje uzel, dojde k jeho zapamatování a při výběru dalšího uzlu se zobrazí formulář pro zadání hodnoty nové hrany. Přidání hrany má na starost metoda `addEdge`. Pro přidání hrany se opět specifikují data pro vytvoření, ale oproti uzlu se kromě identifikátoru musí přidat identifikátor výchozího a cílového uzlu. Navíc se zde přidává výchozí stylování hrany.

```
public addNode(event: cytoscape.EventObject){
    //vytvoreni a pridani uzlu do grafu (cy je core)
    let node = this.cy.add({ data: { id: this.nextNodeId + ""}});
    this.nextNodeId++;
    //nastaveni vychozi hodnoty na 0
    node.style("label", "0");
    //nastaveni pozice uzlu na pozici udalosti
    node.position(event.position);
}
```

Listing 6.3: Metoda pro přidání uzlu

Režim Delete

V režimu `Delete` se pro smazání zvoleného elementu (hrana, uzel) volá metoda `removeElement` (viz ukázka kódu 6.4). Pro smazání vybrané skupiny elementů slouží metoda `removeSelectedElements` (viz ukázka kódu 6.4).

```
public removeElement(event: cytoscape.EventObject){
    //overeni zda je cil element a ne platno
    if(event.target != this.cy){
        //smazani elementu
        this.cy.remove(event.target);
    }
}

public removeSelectedElements(event: cytoscape.EventObject){
```

```
    //vyfiltrovani vybranych elementu a jejich smazani
    this.cy.$(':selected').remove();
}
```

Listing 6.4: Metody pro smazání elementů

Režim Select

V režimu `Select` bylo potřeba reagovat na dvojklik pro označení uzlu jako výchozího v simulaci a pro editaci hodnot u hran, ale knihovna Cytoscape.js neumožňuje odchycení této události. Proto bylo implementováno vlastní odchycení dvojkliku tak, že se zapamatuje čas posledního kliku, a pokud přijde další klik do určitého časového intervalu, je událost považována za dvojklik a vykoná se příslušná akce (viz ukázka kódu 6.5).

```
private registerOnClickListener(){
    //doba do provedeni dalsiho kliku, aby
    //byl uznan jako dvojklik
    let doubleClickDelayMs = 350;
    let previousTapStamp: number;
    //pridani posluchace na kliknuti
    this.cy.addListener("click boxselect", event => {
        //cas udalosti
        let currentTapStamp = event.timeStamp;
        //vypocet rozdilu mezi predchozim klikem a aktualnim
        let msFromLastTap = currentTapStamp - previousTapStamp;
        //porovnani s mezi pro dvojklik
        let isDoubleClick: boolean =
            msFromLastTap < doubleClickDelayMs;
        //nastaveni noveho casu
        previousTapStamp = currentTapStamp;
        //oblsouzeni udalosti
        this.serviceClickAction(event, isDoubleClick);
    });
```

Listing 6.5: Metoda pro zpracovaní události při kliknutí na plátno

6.3.3 Simulace

V aplikaci byla vytvořena simulace Markovských řetězců s disrkétním časem (třída `SimulationProbability`) a se spojitým časem (třída SimulationIntensity). Obě třídy dědí od abstraktní třídy `Simulation` a překrývají metodu `getNextEdge` pro výběr další hrany v simulaci. V obou případech bylo vybrání hrany implementováno podle popsaných algoritmů v kapitolách 2.3.1 (Monte Carlo algortimus) a 2.3.2 (Gillespieho algoritmus).

Při vybrání hrany se z metody `getNextEdge` volá metoda `increaseNodeVisitFreq` ze třídy `Simulation`. Metoda zvýší frekvenci navštívení pro výchozí uzel. Pokud se jedná o simulaci v diskrétním čase, je frekvence navýšena o jedna. V opačném případě, kdy se jedná o simulaci ve spojitém čase, je frekvence navýšena o dobu s exponenciálním rozdělením, která byla potřeba do dalšího přechodu (viz 4. krok v kapitole 2.3.2 v popisu Gillespieho algoritmu). Dále se provede uložení všech pravděpodobností navštívení uzlů v aktuálním kroku do historie. Historii si každý uzel uchovává sám ve svých datech jako pole desetinných čísel.

Frekvence navštívení jsou uložené v asociativním poli, kde klíč je identifikátor uzlu a hodnota je velikost frekvence. Pro získání konkrétních pravděpodobností navštívení jednotlivých uzlů je nutné frekvenci uzlu vydělit celkovým součtem všech frekvencí, což se děje v metodě `getNodeVisitProbability`.

Třída `Simulation` dále umožňuje resetovat simulaci a tím smazat pole s frekvencemi a historii u všech uzlů.

6.3.4 Animování simulace grafu

Knihovna Cytoscape.js sice poskytuje metody pro práci s animací, ale pro dosažení plynulého a správného chování muselo být vyřešeno několik problémů.

Základní princip implementované animace spočívá ve volání metod pro zvýraznění uzlu (`runNodeAnimation`) a hrany (`runEdgeAnimation`) ve třídě `AnimationActions`. Metody ve standardní situaci v první řadě spustí odbarvení předchozího elementu (hrany nebo uzlu), dále se vytvoří nová animace pro aktuální element a ta je spuštěna. Po dokončení animace se zavolá metoda pro animaci dalšího elementu a takto může simulace běžet stále. Výběr další hrany a uzlu pro zvýraznění záleží na typu simulace, která je zrovna nastavena (viz předchozí kapitola 6.3.3).

Problém nastává v případě, kdy v průběhu animace bude chtít uživatel přeskočit o několik kroků dál. Animace by běžela dál a je nutné jí zastavit. K tomu slouží příznak `animationSkipped`, který zamezí dalšímu volání metody (`runNodeAnimation`), resp. (`runEdgeAnimation`).

Další problém spočíval při změně rychlosti animace, kdy při zrychlení animace se muselo počkat na dokončení animace u aktuálního elementu a až poté došlo ke změně rychlosti. Tento problém byl vyřešen zapamatováním aktuálního progresu animace a přerušením animace. Po přerušení animace se spustila nová animace se změnou rychlosti nad stejným elementem a se stejným progresem. Přerušení animace provázely další problémy, kdy objekt

pro odbarvení grafu už byl `null` a došlo k výjimce, která způsobila konec animace. V takovém případě je výjimka zachycena a animace se spustí znova.

V ukázce kódu 6.6 je vidět ošetření všech problémů a spuštění animace pro uzel. V případě animace hrany se jedná o podobný postup.

```
private runNodeAnimation(startProgress: number) {
    //konotrola zda doslo ke stisknuti tlicka pro skok
    //pokud ne, pokracuje se v animaci
    if(!this.animationSkipped) {
        //test, jestli byla animace kompletni
        if(this.prevAnimProgress == 1.0) {
            if (this.actualAnimation != null) {
                //odbarveni predchozi animace
                this.actualAnimation.reverse().rewind().play();
            }
            this.actualAnimation = this.getNodeAnimationStyle();
            //spusteni animace a nastaveni funkce ke spusteni
            //po jejim dokonceni
            this.actualAnimation.progress(startProgress).play()
                .promise("complete")
                    .then(() => this.runEdgeAnimation(0));
        } else {
            //animace nebyla kompletni pro hranu
            // a tak se spusti znova animace hrany
            this.actualAnimation = null;
            startProgress = this.prevAnimProgress;
            this.prevAnimProgress = 1.0;
            this.runEdgeAnimation(startProgress);
        }
    }
}
```

Listing 6.6: Metoda pro animaci uzlu

6.3.5 Stylování grafu

O stylování grafu se stará třída `GraphStyler` společně s abstraktní třídou `EdgeStyler` a jejími potomky `EdgeStylerIntensity` a `EdgeStylerProbability`.

Stylování uzlů

Třída `GraphStyler` má metodu pro nastavení výchozího vzhledu nazvanou `setDefaultStyle`. Tato metoda v první řadě nastavuje výchozí vzhled, jako je barva, ohraničení a velikost pro uzly a hrany. U hran se navíc nastavuje styl zakřivení, pro které se používá Beziérova křivka označovaná v knihovně

Cytoscape.js jako `bezier`. V případě, že byl u uzlu styl již změněn například při animaci, nelze mu tímto způsobem přenastavit styl, a proto je nutné každému uzlu nastavit styl zvlášť a zároveň nastavit pravděpodobnost navštívení uzlu na 0 (viz ukázka metody `setDefaultNodeStyle` 6.7).

```
private setDefaultNodeStyle(){
    this.cy.nodes().forEach((node) => {
        //nastaveni pravdepodobnosti navstvineni uzlu
        //na vychozi hodnotu 0
        node.data("visitPercent", 0);
        //nastaveni stylu uzlu
        node.style({
            backgroundColor: "white",
            borderColor:"black",
            borderWidth:1,
            label: 0,
        });
    });
}
```

Listing 6.7: Metoda nastavení výchozího vzhledu

Dále třída `GraphStyler` pomocí metody `updateNodeVisitPercents`, nastavuje barvu uzlů podle jejich hodnoty pravděpodobnosti navštívení. K určení barvy podle hodnoty se stará metoda `getColorByFrequency`, která je částečně převzatá z knihovny `lesscss`[3].

Stylování hran

K nastavení stylu u hran se používá metoda `updateEdgeStyle` z abstraktní třídy `EdgeStyler`. Metodu překrývají potomci `EdgeStylerIntensity` a `EdgeStylerProbability`, kteří se používají podle aktuálního typu Markovského řetězce.

Metoda `updateEdgeStyle` překrytá ve třídě `EdgeStylerIntensity` zjistí největší hodnotu intenzity hrany v grafu a podle ní se vypočítá poměr ostatních hran. Tento poměr je následně použit pro stylování hrany pomocí metody `getEdgeStyle` (třída `EdgeStyler`).

Ve třídě `EdgeStylerProbability` metoda `updateEdgeStyle` (viz ukázka kódu 6.8) pro každý uzel zjistí součet pravděpodobností jeho výstupních hran. Pokud součet bude menší než 1, bude vytvořena hrana(smyčka), která povede z i do tohoto uzlu s hodnotou danou zbytkem do 1. Aby došlo k zobrazení smyčky, je nutné změnit její styl z `bezier` na `haystack`, což se děje

[3]http://lesscss.org/functions/#color-operations-mix

v metodě `getEdgeStyle`, která navíc podle hodnoty pravděpodobnosti přechodu vytvoří styl pro hranu.

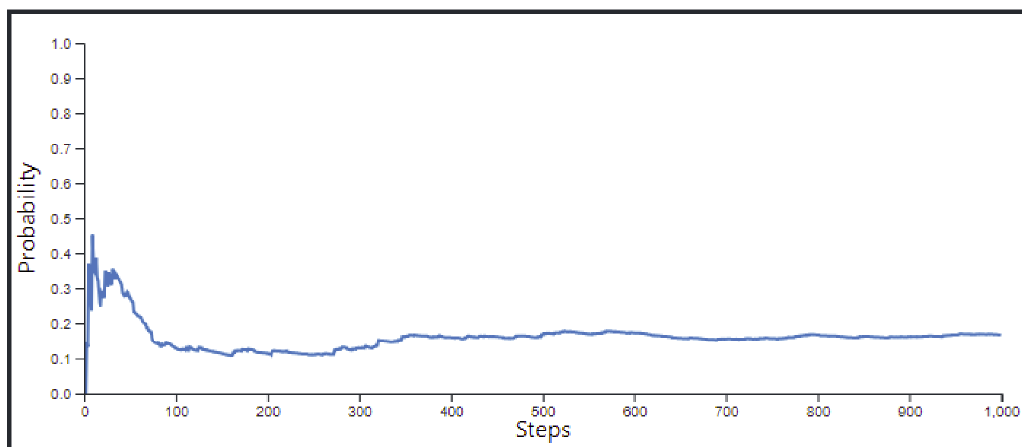
```
public updateEdgeStyle(){
    this.cy.nodes().forEach((node) => {
    //zjisteni souctu pravdepodobnosti vystupnich
    //hran uzlu node
    let probabilitySum = this.countProbSum(node);

    //vypocet zbyvajici pravdepodobnosti se zaokrouhlenim
    //na tri desetinna mista
    let reamingProb = (this.TO_DECIMAL_VALUE - probabilitySum
        * this.TO_DECIMAL_VALUE) / this.TO_DECIMAL_VALUE;
    let nodeId = node.id();
    //zjisteni hrany se stejnym vystupnim a cilovym uzlem
    let edge = this.cy.elements(
        'edge[target = "'+ nodeId +'"][source = "'+ nodeId +'"]')
    if(edge.length == 0){ //hrana neexistuje
        //pridani nove hrany se stejnym vystupnim a cilovym uzlem
        edge = this.cy.add({data: {
            id:nodeId+'o'+nodeId, source:nodeId, target:nodeId}})
    }

    //nastaveni hrane zbytek pravdepodobnosti
    let edgeStyle = this.getEdgeStyle(
        reamingProb, true, reamingProbability+"");
    edge.data("value", reamingProb);
    edge.style(edgeStyle);

    });
}
```

Listing 6.8: Metoda pro nastavení vzhledu u hrany

6.3.6 Vizualizace pravděpodobnosti

Vizualizace historie pravděpodobností během simulace se provádí pomocí knihovny D3.js ve třídě `HistoryGraph`. Třída umožňuje zaregistrovat posluchač na událost, při který je myší přejeto přes uzel. V takovém případě se zobrazí spojnicový graf s průběhem simulace pro označený uzel.

Jako data pro spojnicový graf se používají hodnoty uložené k uzlům během simulace (viz kapitola 6.3.3). Graf je zobrazený v HTML jako obrázek ve formátu SVG (viz obrázek 6.4). Pro vytvoření grafu stačí pomocí knihovny D3.js definovat velikost okna, rozsah měřítek pro osy a přiřazení hodnot na x–vou a y–vou osu. Knihovna se pak sama postará podle počtu hodnot o změnu velikosti měřítka a graf se vždy zobrazí čitelným způsobem.



Obrázek 6.4: Spojnicový graf s historií pravděpodobností navštívení

6.3.7 Analytické řešení

Při analytickém řešení, které má na starost třída `AnalyticalSolution`, se v první řadě provádí ověření silné spojitosti grafu pomocí metody `isGraph-StrongConnected`. Ověření probíhá podle algoritmu popsaném v kapitole 2.2.1, kde se používá prohledávání grafu do hloubky.

Pro prohledávání grafu byla použita funkce `depthFirstSearch` z rozhraní knihovny Cytoscape.js. V rámci této funkce se počítá počet uzlů, které byly nalezeny při prohledávání. Pokud počet nalezených uzlů je stejný jako počet uzlů v grafu, prohodí se směr hran. Změna směru hran probíhá přímo nad grafem, kdy se vytvoří kopie kolekce s hranami a z této kolekce pro každou hranu se zjistí výchozí a cílový uzel. Následně se provede záměna uzlů u hrany v původním grafu pomocí funkce `move` z rozhraní knihovny. Poté se prohledávání do hloubky provede znova. Po prohledávání je graf navrácen do původního stavu stejným postupem jako při změně hran.

V případě, že je graf silně souvislý, vytvoří se podle postupu popsaném v kapitole 2.2.2, reps. 2.2.3 soustava lineárních rovnic v maticovém tvaru. Soustava se vyřeší pomocí Gaussovy eliminační metody z knihovny pro práci s maticemi MatrixJS[4] (GNU LGPL licence). Tato knihovna nemá definované typování, a proto načtení probíhá už v souboru `index.html`. Aby bylo možné přistupovat ke knihovně, musela být ve třídě deklarována jako `JSMatrix`. To umožnilo volat funkce knihovny bez kontroly datových typů. Po výpočtu je vrácen vektor ustálených pravděpodobností.

[4]http://mech.fsv.cvut.cz/ stransky/en/software/jsmatrix/

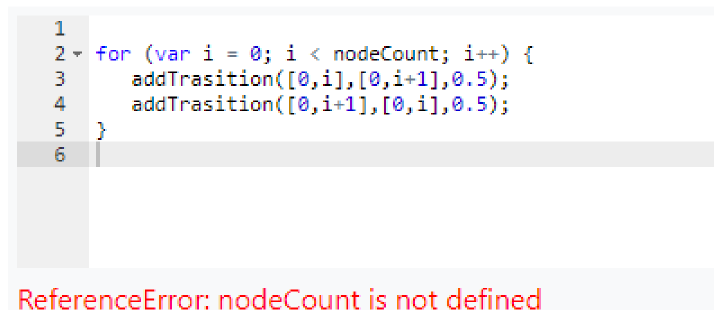
6.3.8 Tvorba grafu

O zobrazení editoru pro tvorbu grafu se stará třída `CodeEditorPanel`. Třída `CodeEditorPanel` používá editor z knihovny Ace (viz kapitola 5.4.4). Pro vytvoření editoru bylo potřeba nastavit mód pro kontrolu syntaxe u editoru Ace na Javascript. Aby bylo možné načíst mód z knihovny, bylo nutné přidat do závislostí (soubor `package.json`) `file-loader`[5]. Následně bylo možné načíst požadovaný mód a nastavit ho v editoru (viz ukázka kódu 6.9).

```
private setMode(codeEditor: ace.Ace.Editor) {
    //nacteni modu pro javascript
    let JavaScriptMode = ace.require("ace/mode/javascript").Mode;
    //nastaveni modu do sesssion editoru
    codeEditor.session.setMode(new JavaScriptMode());
}
```

Listing 6.9: Metoda pro nastavení Javasrcipt módu pro kontrolu syntaxe

Při kliknutí na tlačítko pro vytvoření grafu se volá metoda `createGraph`. V této metodě se volá metoda `createGraph` ze třídy `CodeEditor` a je zde odchycena výjimka, jejíž text se zobrazí u editoru, jak je vidět na obrázku 6.5, kde v kódu není definovaná proměnná `nodeCount`. Výjimka vzniká například, pokud je chyba v kódu nebo nastane chyba při vytváření grafu.



Obrázek 6.5: Editor s chybovou hláškou

Metoda `createGraph` ze třídy `CodeEditor` vytváří graf z kódu. V první řadě se zaobalí původní kód tak, aby bylo možné použít funkci pro přidávání hran ve tvaru popsaném v kapitole 5.4.3. Pokud tedy uživatel popíše graf skriptem 6.10, výsledný kód bude vypadat jako v ukázce 6.11, kde se nachází původní kód obalený funkcí, která obsahuje navíc list přechodů (`graphTransitions`) a funkci pro přidávání nových přechodů (`addTransition`).

[5] https://www.npmjs.com/package/file-loader

```
var nodeCount = 2;
for (var i = 0; i < nodeCount; i++) {
  addTransition([0,i],[0,i+1],0.5);
  addTransition([0,i+1],[0,i],0.5);
}
```

Listing 6.10: Skript pro vytvoření grafu

```
(function() {
    //definovani pole pro prechody
    var graphTransitions = [];

    var nodeCount = 2;
    for (var i = 0; i < nodeCount; i++) {
      addTransition([0,i],[0,i+1],0.5);
      addTransition([0,i+1],[0,i],0.5);
    }
    //funkce pro vytvoreni prechodu
    function addTransition(source, target, value) {
        graphTransitions.push(
            {source:source, target:target, value:value});
    }
    return graphTransitions;
}())
```

Listing 6.11: Výsledný kód pro zpracování

Vytvořený kód se spustí pomocí funkce `eval`, která vrátí pole s přechody. Toto pole je dále zpracováno ve třídě `GraphCreator` pomocí metody `createGraph`. Ta provede sestavení grafu z pole s přechody a zobrazí ho pomocí knihovny Cytoscape.js.

Sestavení grafu provádí metoda `addTransitions`. Zde se nejdříve provede přidání uzlů přechodu (metoda `addNode`) a následně hrana (metoda `addEdge`). Při přidávání uzlů do grafu se specifikuje jeho pozice v mřížce a provádí se aktualizace velikosti mřížky tak, že šířka je dána nejvzdálenějším uzlem na ose x a výška nejvzdálenějším uzlem na ose y. Pozice uzlu se ukládá do dat k příslušnému uzlu (viz ukázka kódu 6.12) a je následně využívaná při umisťování uzlů do `grid layoutu`.

```
private addNode(position: number[]): string {
    let nodeId = position[0] + "x" + position[1];
    //uzel nebyl zatim pridan
    if( this.cy.getElementById(nodeId).length == 0) {
        //aktualizace velikosti mrizky
        this.updateGridLayoutSize(position[0], position[1])
        //vytvoreni uzlu
        let node = this.cy.add({data: {id:nodeId}});
        //ulozeni pozice uzlu do dat
        node.data("positionX", position[0]);
        node.data("positionY", position[1]);
    }
    return nodeId;
}
```

Listing 6.12: Metoda pro přidání uzlu

Po přidání počátečního a cílového uzlu jsou uzly spojeny hranou s hodnotou, která se ověřuje na to, zda se jedná o číslo a zda číslo je větší než nula. Pokud se jedná o Markovský řetězec s diskrétním časem, provede se navíc kontrola součtu pravděpodobností u všech výchozích hran, kde součet nesmí být větší než 1. V případě, že hodnota nesplňuje podmínky, je vyhozena výjimka se zprávou o chybě, jak je vidět na obrázku 6.6. Výjimka je opět zachycena v prezentační vrstvě a zobrazena u editoru.

```
1  var nodeCount = 7;
2  for (var i = 0; i < nodeCount; i++) {
3      addTrasition([0,i],[0,i+1],"a");
4      addTrasition([0,i+1],[0,i],0.5);
5  }
6
```

Error: Edge value between node "0x0" and "0x1" is not a number

Obrázek 6.6: Editor s chybovou hláškou při zadání hodnoty jako řetězce

Po úspěšném přidání všech uzlů a hran se provede nastavení `grid layoutu` pro rozmístění uzlů. Layout se vytváří v metodě `updateLayout`, kde se u instance grafu z knihovny Cytoscape.js provede jeho aktualizace. Při vytváření layoutu bylo použito doporučené nastavení[6], kde byla přidána velikost mřížky (`rows a cols`), rozestupy uzlů (`padding`) a pozice (`position`) daná

[6] http://js.cytoscape.org/#layouts/grid

funkcí pro rozmisťování uzlů `getPosition`. Funkce `getPosition` (viz ukázka kódu 6.13) vezme souřadnice uložené při vytváření uzlu a vrátí souřadnice upravené pro použití v layoutu.

```
private getPosition(node: cytoscape.NodeSingular):any{
    //zjisteni ulozene pozice v datech uzlu
    let positionX = node.data("positionX");
    let positionY = node.data("positionY");
    //vytvoreni souradnic pro layout
    return {row: positionX, col: positionY};
}
```

Listing 6.13: Metoda pro zjištění pozice uzlu v mřížce

6.3.9 Navigační lišta

Navigační lišta se vytváří ve třídě `NavbarMenu` a používá akce ze třídy `NavbarMenuActions`. Ve třídě `NavbarMenu` je možné přidat skupinu tlačítek, které fungují jako přepínače a volají některou z akcí. To je použité pro výběr režimu pro úpravu grafu a přepnutí mezi Markovským řetězcem s diskrétním časem a řetězcem se spojitým časem. Dále se přidávají podobným způsobem položky s akcemi pro analytické řešení, export dat, zobrazení editoru pro tvorbu grafu nebo skupin stavů.

Změna režimu

Třída `NavbarMenuActions` umožňuje měnit režim akcí pro výběr, přidávání a mazaní (viz kapitola 6.3.2) pomocí metody `changeClickAction`. Při každé této změně dojde k resetování animace a nastavení grafu do výchozí podoby.

Změna typu Markovského řetězce

Kromě přepínání režimů pro modifikaci grafu, třída `NavbarMenuActions` umožňuje přepínat mezi Markovskými řetězci s diskrétním časem (metoda `changeToProbabilityMode`) a spojitým časem (metoda `changeToIntensityMode`). Při přepnutí se přepočítají hodnoty hran podle postupu uvedeném v kapitole 5.6. Následně se změní typ simulace, podle které se vybírají další hrany v animaci. Poté se nastaví instanci třídy `GraphStyler` nová instance třídy pro vizualizaci hran (`EdgeStylerIntensity` nebo `EdgeStylerProbability`) a obdobně je změněn formulář pro přidání hrany ve třídě `GraphActions`.

Analytické řešení

Dále třída `NavbarMenuActions` obsahuje metodu pro výpočet a zobrazení ustálených pravděpodobností. Výpočet se provádí v rámci třídy `Analytical-Solution` (popsáno v kapitole 6.3.7). Pokud je místo výsledku vrácena hodnota `null`, zobrazí se upozornění, že graf obsahuje absorpční stavy a nelze tak vykonat výpočet. V opačném případě se zobrazí výsledek pomocí metody `updateNodeVisitPercents` ze třídy `GraphStyler`.

Import a export

Třída dále obsahuje metody pro export a import dat. Metoda `exportGraph` exportuje data týkající se vytvořeného grafu, kódu pro tvorbu grafu a typu řetězce. Export dat grafu umožňuje přímo knihovna Cytoscape.js pomocí funkce `json()`. Data jsou exportována a stažena v JSON formátu a pomocí metody pro import `importGraph` lze nahrát vyexportovaný graf. Import grafu lze provést opět funkcí `json()`, kde vstupním parametrem je JSON objekt.

Dále je možné exportovat cestu procházení grafu při simulaci do CSV souboru a nebo jako HTML soubor s vizualizací historie pomocí spojnicového grafu, který je vytvářen stejným způsobem jako v kapitole 6.3.6.

6.3.10 Kontextové menu

Kontextové menu se vytváří ve třídě `ContextMenu`, kde se zaregistruje posluchač na plátno pro událost kliknutí pravého tlačítka. V závislosti na tom, kam bylo kliknuto (prázdné místo, uzel, hrana), se vyvolá příslušná nabídka akcí. Při vytváření nabídky se v první řadě vytvoří pole s položkami typu ContextMenuItem, které uchovávají název akce a funkce pro vykonání při výběru akce. Následně je toto pole zobrazeno jako kontextové menu v HTML.

Umístěné nabídky bylo nutné přepočítat tak, aby se zobrazila na správném místě v plátně. Pro výpočet pozice byl použit offset levého, resp. horního okraje plátna a renderovací pozice události v rámci plátna (viz ukázka metody `setMenuPosition` 6.14).

Akce jsou v rámci nabídky obsluhovány třídou `GraphActions` (viz kapitola 6.3.2). Tyto akce nejsou závislé na režimu, ve kterém se aplikace nachází, a tak je možné například pomocí kontextové nabídky mazat hrany bez nutnosti přepínat do režimu pro mazání `Delete`.

```
private setMenuPosition(renderedPosition: cytoscape.Position){
    let container = (<HTMLElement>this.cy.container());
    this.menuElement.style.left =
        (container.offsetLeft + renderedPosition.x) + "px";
    this.menuElement.style.top =
        (container.offsetTop + renderedPosition.y) + "px";
}
```

Listing 6.14: Metoda pro nastavení pozice kontextové nabídky

6.3.11 Skupiny uzlů

Skupiny uzlů zobrazuje třída `NodeGroupsPanel` jako tabulku s názvem skupiny a součtem pravděpodobností v pravé části aplikace. Přidávání, mazání, zvýraznění a aktualizování skupin má na starost třída `NodeGroupsActions`.

Při přidání nové skupiny se zobrazí formulář pro zadání názvu skupiny. Po potvrzení se vyberou označené uzly z grafu pomocí filtru `':selected'` a vytvoří se skupina s jednoznačným identifikátorem.

Skupiny jsou aktualizované při změně pravděpodobnosti některého z uzlů grafu. K aktualizování slouží metoda `updateGroups`, která je volaná ze třídy `Graphstyler` při nastavování nových pravděpodobností v metodě `updateNodeVisitPercents`. Při aktualizaci hodnoty pravděpodobnosti skupiny se sečtou pravděpodobnosti všech uzlů ve skupině a podle výsledné hodnoty se nastaví barva pro skupinu podobným způsobem jako u uzlů (viz 6.3.5).

6.3.12 Formuláře

V aplikaci se používají tři formuláře s různou kontrolou vstupu. Jejich zobrazení má na starost třída `ValueFormDialog`, která navíc umožnuje zobrazit chybovou hlášku o nesprávném formátu vstupu.

Třída `FormDialogActions` je hlavní třída pro formuláře. Obsahuje metody pro zobrazení, potvrzení a uzavření dialogu s formulářem. Konstruktor má jako parametr funkci, která se vykoná při potvrzení dialogu. Formulář obsahuje vstupní pole s kontrolou (metoda `checkInputValue`), zda není při potvrzení pole prázdné. Pokud je prázdné, je v dialogu zobrazeno upozornění. Tento dialog se používá například při přidávání skupiny uzlů, kdy se pomocí formuláře nastavuje jméno skupiny.

Abstraktní třída `ValueEditorActions` dědí od třídy `FormDialogActions` (viz diagram 6.2). Slouží jako rodičovská třída pro třídy přidávající hodnotu k hraně (třída `EdgeProbFormActions` a `EdgeIntensityEditorActions`) a rozšiřuje metodu `checkInputValue` o kontrolu, zda vstupní řetězec je kladné

reálné číslo. Dále nastavuje funkci pro vykonání při potvrzení dialogu `changeEdgeValue`, která nastaví novou hodnotu vybrané hraně. V případě, že je dialog uzavřen bez potvrzení, nedojde k přidání nové hrany, resp. není změněna hodnota u existující hrany.

Třída `ValueEditorActions` obsahuje abstraktní metodu `getActualValue` volanou z metody `showEdgeValueForm` pro zobrazení formuláře s předem nastavenou hodnotou. Metodu `getActualValue` implementují třídy `EdgeProbFormActions` a `EdgeIntensityEditorActions`.

Formulář pro Markovský řetězec se spojitým časem

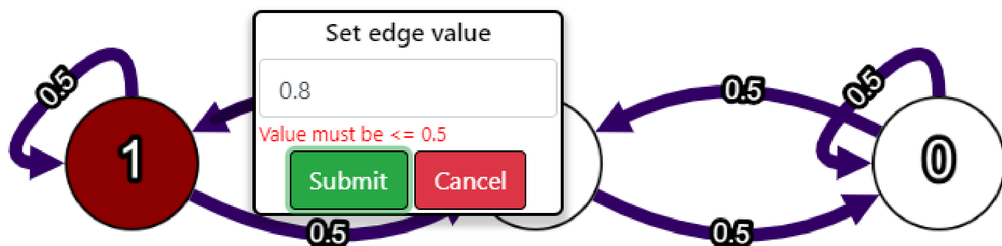
Třída `EdgeIntensityEditorActions` se používá, pokud je nastavený typ Markovského řetězce se spojitým časem. V takovém případě metoda metodu `getActualValue` zjistí hodnotu již existující hrany a tu vrátí. Pokud hrana zatím neexistuje vrátí 1 jako výchozí hodnotu.

Formulář pro Markovský řetězec s diskrétním časem

Metoda `getActualValue` implementovaná ve třídě `EdgeProbFormActions` nevrací hodnotu hrany jako v případě třídy `EdgeIntensityEditorActions`, ale vrací maximální možnou hodnotu pravděpodobnosti, kterou může hrana nabývat (viz ukázka kódu 6.15). Tato hodnota je zjištěna na základě hodnot výstupních hran výchozího uzlu. Hodnoty jsou odečteny od jedné a tím se dostane maximální možná hodnota pro novou, resp. existující hranu.

Dále se ve třídě `EdgeProbFormActions` překrývá metoda pro kontrolu vstupu ve formuláři. V metodě se přidává navíc kontrola, zda zadaná hodnota nepřekračuje maximální možnou hodnotu (viz obrázek 6.7), a dále zda má hodnota maximálně tři desetinná místa. Omezení na tři desetinná místa je kvůli problému s přesností desetinných čísel. Problém se projevuje například při výpočtu maximální možné hodnotě hrany a nebo při zobrazení hrany do stejného uzlu (viz stylování hran v kapitole 6.3.5).

```
protected getActualValue(edge: cytoscape.EdgeSingular): number {
    //nastaveni maximalni mozne pravdepodobnosti
    this.maxProbabilityValue = 1;
    //iterace pres hrany, ktere vychazi z pocatecniho uzlu
    //upravovane hrany
    edge.source().outgoers().edges().forEach(edgeTmp => {
        // nepocita se vedouci do stejneho uzlu jako vychazi
        //a hrana, ktera upravuje
        if(edgeTmp.source() !== edgeTmp.target() &&
           edge.target() !== edgeTmp.target()){
            //odecteni pravdepodobnosti prechodu
            this.maxProbabilityValue -= +edgeTmp.data("value");
        }
    });

    if(edge.data("value") == null){
        // vraceni maximalni mozne hodnoty pro novou hranu
        return this.maxProbabilityValue;
    }else{
        //vraceni hodnoty hrany
        return edge.data("value");
    }
}
```
Listing 6.15: Metoda pro zjištění hodnoty k zobrazení ve formuláři



Obrázek 6.7: Dialog s formulářem v režimu pro pravděpodobnost přechodu

7 Testování

7.1 Jednotkové testy

7.1.1 Konfigurace prostředí

Pro jednotkové testy byl použit nástroj Karma[1] společně s frameworkem pro testování Jasmin. Tato kombinace nástrojů je velmi vhodná pro otestování webové aplikace napsané v jazyce Typescript a umožnuje vykonávat jednotkové testy v závislosti na konkrétním prohlížeči.

Nástroje i jejich nastavení již bylo obsaženo v základní konfiguraci prostředí pro vývoj[2] (viz kapitola 6.1.1). Veškerá konfigurace nástrojů pro jednotkové testy se nachází v kořenové složce v souboru `karma.conf.js`, kde je nastaven prohlížeč Chrome (verze 73.0.3683), ve kterém se budou jednotkové testy vykonávat. V tomto souboru bylo dále nutné specifikovat Javascriptové knihovny využívané testovanými třídami. Jedná se tedy hlavně o knihovnu Matrix.JS.

Testy se nachází ve složce `tests` v kořenovém adresáři. Zde jsou složky `coverage` a `unit`. Složka `coverage` obsahuje funkcionalitu pro zjištění procentuálního pokrytí kódu jednotkovými testy (součást základní konfigurace). Ve složce `unit` jsou už konkrétní jednotkové testy (koncovka `.spec.ts`, o jejichž načtení se stará soubor `spec-bundle.js`, který je nastaven v konfiguraci jako hlavní testovací soubor. Spuštění testů se provádí v kořenové složce příkazem `npm test`, jenž spustí nástroj Karma a ten vykoná jednotkové testy a zobrazí report výsledků.

7.1.2 Vytvoření testů

Jednotkové testy byly vytvořeny pro ověření správné funkčnosti hlavních částí aplikace, které se netýkají vizualizace a interakce s grafem. Jedná se o testy ohledně vytváření grafu z kódu (soubor `GraphCreatorTests.spec.ts`), zjištění silné souvislosti grafu (soubor `AnalyticalSolution.spec.ts`) a porovnání simulace společně s analytickým řešením (soubor `SimulationAnalyze.spec.ts`).

[1]https://karma-runner.github.io/3.0/index.html
[2]https://github.com/nobrainr/typescript-webpack-starter

Vytváření grafu

Při testování vytváření grafu byly vytvořeny testy na kontrolu vstupních hodnot u hran (viz ukázka skriptu 7.1). Dále se testovalo správné vytvoření grafu vytvořeného pomocí skriptu.

```
//definice sady testu
describe('Graph␣creator␣tests', () => {
    let cy: cytoscape.Core;
    let transitions: Transition[];
    //provede se pred kazdym testem
    beforeEach(() => {
        cy = cytoscape({});
        transitions = [];
    });

    ...

    //test pro overeni vyhozeni vyjimky, pokud hodnota
    //pravdepodobnosti je vyssi nez 1.
    it('Probability␣value␣more␣than␣1.␣More␣edges', () => {
        //pridani prechodu
        transitions.push({source:[0,0],target:[0,1],value:0.5});
        transitions.push({source:[0,0],target:[1,0],value:0.6});
        const graphCreator = new GraphCreator(cy);
        //kontrola na vyhozeni vyjimky
        expect(function(){
            graphCreator.createGraph(transitions, true)})
            .toThrowError('Output␣probability␣for␣node␣'+
            'on␣the␣position␣"0x0"␣is␣higher␣then␣1');
    });
});
```

Listing 7.1: Část skriptu pro otestování vytváření grafu

Ověření silné souvislosti

V rámci testování správného určení silné souvislosti byly vytvořeny testy ověřující souvislost na několika rozdílných modelech grafu. Testuje se, zda je souvislost správně vyhodnocena pro silně souvislý graf(pozitivní testy) i pro graf, který není silně souvislý (negativní testy).

Grafy pro negativní testy byly vytvořeny tak, aby se otestovaly všechny varianty, i tu s opětovným prohledáváním do hloubky pro graf s inverzními hranami.

Porovnání simulace a analytického řešení

Dále se pomocí jednotkových testů testovala simulace a analytické řešení, kdy se v rámci testu porovnával výsledek obou řešení.

V testovacím skriptu se v první řadě provádí simulace s určitým počtem kroků. Následně se provede analytické řešení nad tím samým grafem a poté se postupně porovnají jednotlivé hodnoty uzlů. Jelikož při simulaci je téměř nemožné dosáhnout přesného výsledku, porovnání výsledných hodnot probíhá s tolerancí dvou procent.

Pro porovnání byly vytvořeny tři rozdílné grafy, nad kterými se prováděla simulace Markovské řetězce s diskrétním i se spojitým časem.

7.1.3 Vyhodnocení

Jednotkové testy pomohly otestovat nejdůležitější části aplikace snadným a rychlým způsobem. Během testování byla nalezena chyba při vytváření grafu, kde mohla být hraně nastavena hodnota 0 (test `Probability value is 0`). Testy navíc ověřily správnost zpracování modelů, kdy simulace i analytické řešení bylo téměř totožné.

Dále jednotkové testy usnadnily následovné funkční testování, jelikož není nutné manuálně testovat veškerou funkčnost, kterou bylo rychlejší ověřit pomocí testovacích skriptů.

7.2 Funkční testování

V rámci jednotkových testů byly otestovány hlavní části aplikace, které ale neměly na starost vizualizaci a ani interakci s grafem. Tyto části aplikace je vhodnější otestovat manuálně a vyzkoušet tak, zda vše funguje bez problémů a odpovídá požadavkům stanoveným v kapitole 4.

Pro funkční testování byl sestaven scénář (viz kapitola 7.2.1), podle kterého bude aplikace otestována. Scénář bude použit i při uživatelském testování.

7.2.1 Scénář pro testování

1. **Tvorba grafu v editoru**

    - Při vytváření grafu vyzkoušet i použití neexistujících proměnných a nastavení neplatné hodnoty (řetězec, záporné číslo, nastavení větší pravděpodobnosti než 1).

        – V chybovém případě se zobrazí chybová hláška.

    - Pokud byl skript pro vytvoření grafu zadaný správně, zobrazí se graf s jedním výchozím uzlem (červená barva a hodnota 1), dalšími uzly s bílou barvou a hodnotou 0 a hranami nastylovanými podle jejich hodnoty.

2. **Interakce s grafem**
    - Vyzkoušet přibližování, oddalování a posouvání plátna.
    - Vyzkoušet přesouvání uzlů.

3. **Akce pro uzly**
    - Vyzkoušet přidání uzlu v režimu `Add` a smazání uzlu v režimu `Delete`.
        - Při přidání by se měl zobrazit bílý uzel s hodnotou 0 a při smazání uzlu se smažou i hrany vedoucí do uzlu.
    - Akce vykonat i pomocí kontextového menu nezávisle na vybraném režimu.

4. **Akce pro hrany**
    - Vyzkoušet přidání hrany v režimu `Add`.
    - Při vytváření hrany zkusit zrušit formulář (hrana se nepřidá), zadat neplatnou hodnotu (záporná hodnota, v režimu nastavit pravděpodobnost v součtu větší než 1).
        - Při zadání neplatné hodnoty se zobrazí chybová hláška u formuláře. Pokud byla zadána platná hodnota a formulář byl potvrzen, hrana se zadanou hodnotou se zobrazí a má nastavený styl hrany podle hodnoty.
        - V případě režimu s intenzitami (Markovský řetězec se spojitým časem) se provede přenastavení ostatních hran podle nového poměru, pokud byl změněn.
    - Vyzkoušet změnu hodnoty hrany v režimu `Select`. Opět se provede v režimu s intenzitami přenastavení ostatních hran podle nového poměru, pokud byl změněn.
    - Vyzkoušet změnu hodnoty hrany v režimu `Select` a pomocí kontextového menu.

5. **Skupiny uzlů**
    - Vytvořit skupiny uzlů

- Po vytvoření se skupiny zobrazí v záložce `Node groups`. Každá skupina má součet pravděpodobností jejích uzlů a je obarvená příslušnou barvou.

- Najet na skupinu myší
  - Při najetí na skupinu myší se zobrazí uzly
- Zkusit smazaní alespoň jedné skupiny.

6. **Analytické řešení**
   - Před analytickým řešením ověřit, zda je graf silně souvislý.
     - Pokud je graf souvislý, je třeba vytvořit graf, který takový nebude (odebráním hrany a nebo několika hran). Když není graf silně souvislý, při spuštění analytického řešení se pouze zobrazí chybová hláška.
   - Po zobrazení hlášky upravit graf tak, aby byl silně souvislý a znovu spustit analytické řešení.
     - U uzlů se zobrazí vypočítané ustálené pravděpodobnosti a uzly jsou obarveny barvou podle velikosti hodnoty.
     - Analytické řešení v aplikaci musí odpovídat analytickému výpočtu podle postupu uvedeném v kapitole 2.2. Výpočet je nutný pro ověření správnosti modelu.

7. **Simulace**
   - Spustit simulaci
   - Pozastavit a znovu spustit simulaci
   - Změnit rychlost simulace
   - Skočit o několik kroků dopředu
   - Resetovat simulaci

   Během simulace je možné pozorovat změnu pravděpodobností u uzlů, kde u navštíveného uzlu se pravděpodobnost zvyšuje a u ostatních naopak snižuje. Při najetí na uzel se zobrazuje graf znázorňující změnu pravděpodobností během simulace.

   Změna pravděpodobnosti při simulaci se musí promítnout i do skupin uzlů, kde se během simulace mění jejich pravděpodobnosti navštívení.

8. **Export a import**

- Exportovat graf
  - Po vykonání exportu bude stažen soubor `graph_export.json`. Obsahuje data grafu, nastavený typ řetězce a kód z editoru.
- Importovat graf ze souboru `graph_export.json`(před importem modifikovat graf, upravit kód v editoru a přenastavit režim z intenzit na pravděpodobnosti nebo opačně)
  - Po nahrání se zobrazí graf ve výchozím stavu, nastaví se správný režim a přepíše se kód editoru. Graf i nastavení musí odpovídat stavu, který byl před exportem grafu.
- Exportovat cesty
  - Při exportu cesty (`Export path`) se stáhne soubor `path.csv` obsahující cestu, která je tvořena uzly a odpovídá cestě simulace přes tyto uzly.
- Exportovat uzly
  - Při exportu uzlů (`Export nodes`) se stáhne soubor `nodes.html`, obsahující grafy pravděpodobností pro každý uzel v grafu.

9. **Typ grafu**
   - Změnit typ grafu z intenzit na pravděpodbnosti nebo naopak.
     - Při změně se musí přepočítat hodnoty hran, podle postupu popsaném v kapitole 5.6.
   - Po změně režimu zopakovat body 1 – 9.

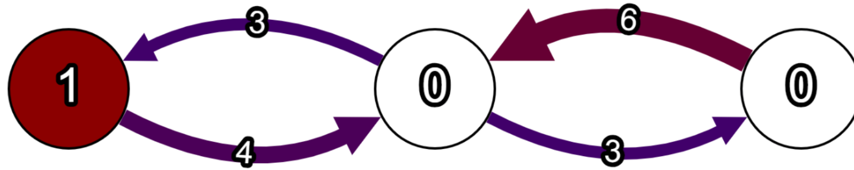
7.2.2 Modely pro ověření simulace a analytického řešení

Ověření správného výpočtu proběhlo na třech modelech Markovských řetězců s diskrétním a spojitým časem.

První model

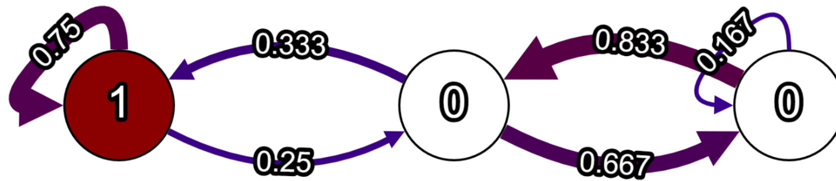
První testovaný model a výpočet analytického řešení je pro Markovský řetězec s diskrétním čase uveden v kapitole 2.2.2 (obrázek 7.1 s rovnicemi 7.1) a pro Markovský řetězec se spojitým čase v kapitole 2.2.3 (obrázek 7.2 s rovnicemi 7.2). Výsledky z vypočítaného analytické řešení, analytického řešení z aplikace a simulace (100 000 kroků) jsou uvedeny v tabulce 7.1.

Výsledky vypočítaného analytického řešení a řešení v aplikaci jsou stejné. Simulace se pro bod p_1 a p_3 liší s analytickým řešení o tisícinu, což lze požadovat za správný výsledek, protože simulace pouze konverguje k přesnému řešení.



Obrázek 7.1: Graf ukázkového Markovského řetězce se spojitým časem

$$
\begin{aligned}
0 &= -4p_1 + 3p_2 \\
0 &= 4p_1 - 3p_2 - 3p_2 + 6p_3 \\
1 &= p_1 + p_2 + p_3
\end{aligned}
\tag{7.1}
$$



Obrázek 7.2: Graf ukázkového Markovského řetězce

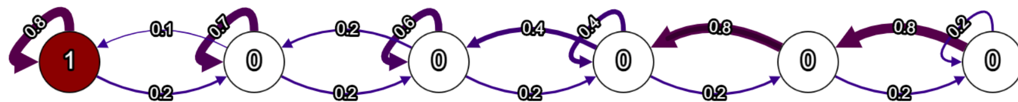
$$
\begin{aligned}
-\frac{3}{4}\pi_1 + \frac{1}{3}\pi_2 &= 0 \\
\frac{3}{4}\pi_1 - \pi_2 + \frac{5}{6}\pi_3 &= 0 \\
\pi_1 + \pi_2 + \pi_3 &= 1
\end{aligned}
\tag{7.2}
$$

Uzel	Vypočítané analyt. řešení	Analytické řešení v aplikace	Simulace
Markovský model s diskrétním časem			
p_1	0.198	0.198	0.197
p_2	0.446	0.446	0.446
p_3	0.356	0.356	0.357
Markovský model se spojitým časem			
p_1	0.333	0.333	0.335
p_2	0.444	0.444	0.447
p_3	0.222	0.222	0.219

Tabulka 7.1: Výsledky pro první model

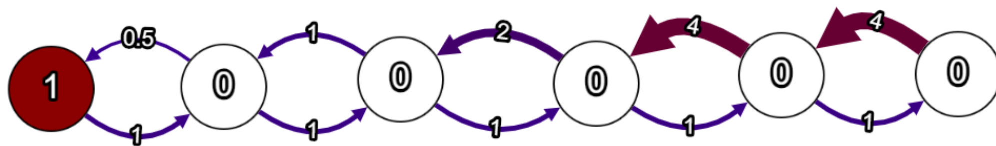
Druhý model

Další model pro ověření lze vidět na obrázku 7.3(diskrétní čas) resp. na obrázku 7.4(spojitý čas). Ověření probíhalo stejně jako u předchozího modelu. Pro nalezení stacionárního rozdělení analytickým řešením byly sestaveny rovnice(7.3 - pro Markovský řetězech s diskrétním časem, 7.4 - pro Markovský řetězech se spojitým časem), jejichž výsledek je zapsaný v tabulce 7.2 i s výsledkem simulace a analytického řešení z nástroje.



Obrázek 7.3: Graf pro druhý model s pravděpodobnostmi přechodu

$$
\begin{aligned}
-0.2p_1 + 0.1p_2 &= 0 \\
0.2p_1 - 0.3p_2 + 0.2p_3 &= 0 \\
0.2p_2 - 0.4p_3 + 0.4p_4 &= 0 \\
0.2p_3 - 0.6p_4 + 0.8p_5 &= 0 \\
0.2p_4 - 1p_5 + 0.8p_6 &= 0 \\
p_1 + p_2 + p_3 + p_4 + p_5 + p_6 &= 1
\end{aligned} \tag{7.3}
$$



Obrázek 7.4: Graf pro druhý model s intenzitami přechodu

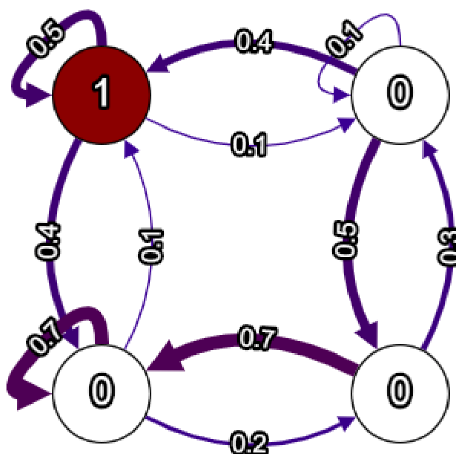
$$
\begin{aligned}
-p_1 + 0.5p_2 &= 0 \\
p_1 - 1.5p_2 + p_3 &= 0 \\
p_2 - 2p_3 + 2p_4 &= 0 \\
p_3 - 3p_4 + 4p_5 &= 0 \\
p_4 - 5p_5 + 4p_6 &= 0 \\
p_1 + p_2 + p_3 + p_4 + p_5 + p_6 &= 1
\end{aligned} \tag{7.4}
$$

Uzel	**Vypočítané analyt. řešení**	**Analytické řešení v aplikace**	**Simulace**
Markovský řetězec s diskrétním časem			
p_1	0.158	0.158	0.156
p_2	0.317	0.317	0.315
p_3	0.317	0.317	0.319
p_4	0.158	0.158	0.160
p_5	0.04	0.04	0.04
p_6	0.01	0.01	0.01
Markovský řetězec se spojitým časem			
p_1	0.158	0.158	0.156
p_2	0.317	0.317	0.318
p_3	0.317	0.317	0.318
p_4	0.158	0.158	0.158
p_5	0.04	0.04	0.039
p_6	0.01	0.01	0.01

Tabulka 7.2: Výsledky pro druhý model

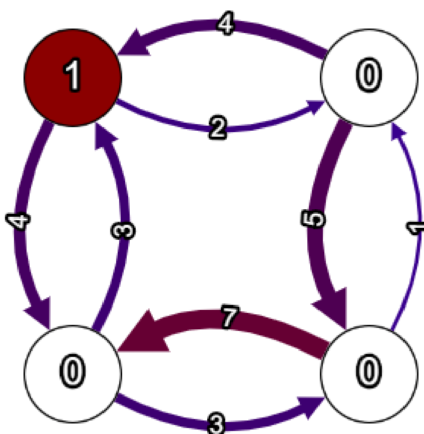
Třetí model

Třetí model je možné vidět na obrázcích 7.5 (diskrétní čas) a 7.6 (spojitý čas). Pro výpočet stacionárních rozdělení byly sestaveny rovnice 7.5 (diskrétní čas) a 7.6 (spojitý čas). Výsledek byl zapsán stejným způsobem jako u předchozích modelů do tabulky 7.3.



Obrázek 7.5: Graf pro třetí model s pravděpodobnostmi přechodu

$$\begin{aligned} -0.5p_1 + 0.4p_2 + 0.1p_3 &= 0 \\ 0.1p_1 - 0.9p_2 + 0.3p_4 &= 0 \\ 0.4p_1 - 0.3p_3 + 0.7p_4 &= 0 \\ p_1 + p_2 + p_3 + p_4 &= 1 \end{aligned} \tag{7.5}$$



Obrázek 7.6: Graf pro třetí model s intenzitami přechodu

$$
\begin{aligned}
-6p_1 + 4p_2 + 3p_3 &= 0 \\
2p_1 - 9p_2 + p_4 &= 0 \\
4p_1 - 6p_3 + 7p_4 &= 0 \\
p_1 + p_2 + p_3 + p_4 &= 1
\end{aligned} \tag{7.6}
$$

Uzel	**Vypočítané analyt. řešení**	**Analytické řešení v aplikace**	**Simulace**
Markovský model s diskrétním časem			
p_1	0.176	0.176	0.176
p_2	0.071	0.071	0.070
p_3	0.597	0.597	0.598
p_4	0.155	0.155	0.156
Markovský model se spojitým časem			
p_1	0.271	0.271	0.273
p_2	0.084	0.084	0.083
p_3	0.431	0.431	0.430
p_4	0.214	0.214	0.214

Tabulka 7.3: Výsledky pro třetí model

7.2.3 Vyhodnocení

Funkční testování aplikace bylo prováděno nad modely uvedenými v kapitole 7.2.2.

Během testování se ověřila funkčnost modelů výpočtem analytického řešení, kdy byly sestaveny rovnice a jejich výpočet byl proveden v programu Octave. Kromě ověření modelů bylo nalezeno několik chyb v aplikaci.

První nalezené chyby se týkaly vytváření grafu pomocí popisovacího jazyka. V prvním případě se jednalo o smazání původního grafu a zobrazení chybného grafu při chybě v jeho vytváření. Další chyba byla nalezena při přidání dalšího přechodu se stejným počátečním a koncovým stavem. Tato situace byla ošetřena a při jejím nastání je vypsána chybová hláška o přidání hrany se stejným počátečním a koncovým stavem.

Další problém byl nalezen při přepnutí typu Markovského řetězce z intenzit na pravděpodobnosti. Při přepnutí se projevila chyba se zaokrouhlování čísel z výpočtu. Problém byl vyřešen pozměněním výpočtu a zaokrouhlením výpočtu.

7.3 Uživatelské testování

Vytvořená aplikace byla předána třem studentům předmětu KIV/VSS, kde se jeden z okruhů týká právě Markovských řetězců. Studenti měli za úkol otestovat aplikaci podle scénáře v kapitole 7.2. Pro snadnější orientaci byli buď seznámeni s aplikací nebo dostali vytvořenou uživatelskou příručku A.

Dále měli studenti pomocí nástroje vypracovat část z jednoho úkolu zadávaného k vypracování na předmětu KIV/VSS[3].

V prvním případě student zkoušel aplikaci na úkolu číslo 0. Zde byl nekonečný buffer, do kterého přicházely a odcházely požadavky náhodně s exponenciální rozdělení. Pomocí nástroje vytvořil model s omezeným počtem stavů (100 stavů) a provedl analytické řešení, pomocí kterého zjistil kolik procent času při dlouhodobém sledování bude buffer prázdný (první stav v modelu). Vypočítaná hodnota a hodnota prvního stavu ve vytvořeném modelu se shodovaly.

Další student vypracovával úkol číslo 6. Úkol byl zaměřen na příklad s producentem a konzumentem, kde pro vyřešení bylo nutné spočítat stacionární rozdělení. Student si vytvořil model v nástroji a pomocí analytického řešení zjistil stacionární rozdělení, které se shodovalo s jeho ručně vypočítaným stacionárním rozdělením.

Třetí student měl vypracovat úkol číslo 7, kde měl vytvořit Markovský model pro SHO typu M/M/1 s omezenou délkou fronty na dva požadavky. V rámci úkolu student vytvořil model v nástroji a následně provedl výpočet pomocí analytického řešení. Hodnoty z nástroje a ručně vypočítané hodnoty se opět shodovaly.

Během testování studenti narazili na problém při mazání hran, kdy nedocházelo k překreslení modelu. Další problém byl nalezen v chybové hlášce při přidávání hrany v Markovským řetězci s diskrétním časem, kdy bylo možné zadat hodnotu 0 hraně. Dále byla pro některé studenty nejasná hláška, která se zobrazila při zadání větší pravděpodobnosti, než byla možná.

Všechny tyto chyby byly opraveny a hlášky upraveny tak, aby bylo zřetelné o jakou chybu se při zadávání jedná.

[3]https://courseware.zcu.cz/portal/studium/courseware/kiv/vss/cviceni/okruh-2.html

7.4 Omezení

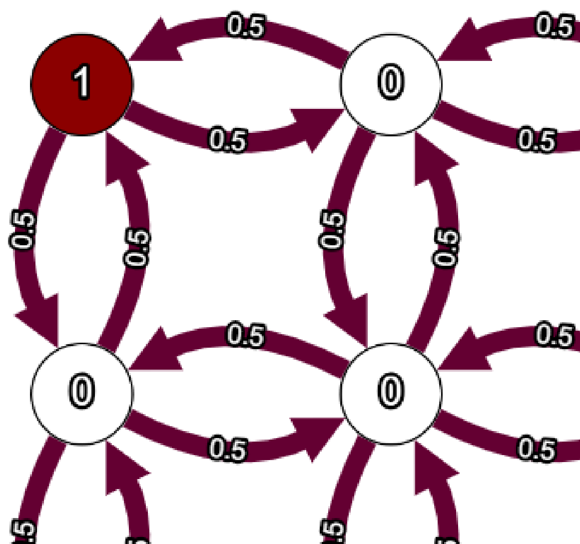
V této kapitole je shrnuté omezení aplikace jako je například velikost grafu, doba výpočtu simulace pro zadaný počet kroků a doba výpočtu analytického řešení.
Aplikace byla testována na počítači s konfigurací:

- Procesor: Intel® Core™ i7-4500U
  - Základní frekvence: 1.80 GHz
  - Max Turbo frekvence: 3.00 GHz
  - Počet fyzických jader: 2
  - Počet logických jader: 4
  - Cache: 4 MB
- Operační paměť: 8 GB DDR3 (1600 MHz)
- Operační systém: Windows 10 Pro (64bit)
- Prohlížeč: Chrome (verze 73.0.3683.86)

Graf bylo možné vytvářet v přijatelném čase (do 2 sekund), pokud obsahoval nejvýše 500 stavů spojenými hranami. Knihovna zvládala vykreslit i několik tisíc stavů, ale vytvoření trvalo několik desítek sekund a interakce s grafem již nebyla plynulá. Dále se při velkém počtu stavů ukázal problém s přibližováním, resp. oddalováním grafu, kdy v některých úrovních přiblížení graf zcela zmizel (chyba v knihovně Cytoscape).

Doba výpočtu analytického řešení závisí jak na počtu uzlů, tak na počtu hran vedoucích mezi nimi. Například doba výpočtu pro graf představující Markovský model `M/M/1` s velikostí fronty omezenou na 500 trval přibližně 5 sekund. Výpočet pro graf o stejném počtu uzlů ve tvaru mřížky (viz obrázek 7.7) trval přibližně 10 sekund.

Doba vykonání počtu kroků v simulaci je opět závislá na počtu uzlů a hran. Například odsimulování 10 000 kroků naráz trvalo pro Markovský model `M/M/1` s velikostí fronty omezenou na 10 necelé 2 sekundy.



Obrázek 7.7: Část grafu ve tvaru mřížky

7.5 Další prohlížeče

Aplikace byla po celou dobu vyvíjena a přizpůsobována pro prohlížeč Chrome (verze 73.0.3683.86). Jako další prohlížeče, kde byla aplikace na závěr testována, byly použity Mozilla Firefox[4] (verze 66.0.2) a Edge od Microsoftu[5] (verze 17.17134). Testování pro oba prohlížeče probíhalo podle scénáře popsaném v kapitole 7.2.

V aplikaci běžící v prohlížeči Mozilla Firefox nebyl nalezen žádný problém a vše fungovalo stejně jako na prohlížeči Chrome.

U prohlížeče Edge byl nalezen problém s exportem dat. Při pokusu o opravu bylo zprovozněno stahování na tomto prohlížeči, ale nastal problém v prohlížeči Chrome, kde přestal fungovat import vyexportovaného grafu z knihovny Cytoscape.js. V tomto případě byl upřednostněn používanější prohlížeč Chrome a funkce pro exportování dat nebyly na prohlížeči Edge zprovozněny.

[4] https://www.mozilla.org/cs/firefox/new/

[5] https://www.microsoft.com/cs-cz/windows/microsoft-edge

8 Závěr

Pro splnění zadání diplomové práce bylo v první řadě nutné seznámit se s problematikou Markovských řetězců, možnostmi jejich simulace a výpočtem řešení analytickou cestou. Dále byly prozkoumány existující nástroje pro práci s řetězci. Poté byly sepsány požadavky na nástroj a provedl se návrh řešení, kde se mimo jiné vybraly technologie a knihovny pro vývoj nástroje.

Pro implementaci nástroje byl vybrán jazyk Typescript a pro vizualizaci a interakci s grafem byla zvolena grafová knihovna Cytoscape. Výběr knihovny Cytoscape se ukázal jako velmi vhodný. Knihovna umožnila bezproblémovou implementaci potřebné funkčnosti a umožnila vytvářet a zobrazovat graf jednoduchým a přehledným způsobem.

Následně byl vytvořen nástroj umožňující jednoduché vytváření řetězců jak pomocí postupného přidávání stavů a přechodů, tak s použitím popisovacího jazyka. Nad vytvořenými řetězci je pak možné provádět simulaci, která je vizualizována pomocí animace, a je tak možné pozorovat její průběh. Dále nástroj umožňuje výpočet stacionárního rozdělení Markovských řetězců bez absorpčních stavů.

Při implementaci nástroje byly vytvořeny jednotkové testy pro ověření menších částí aplikace. Dále bylo provedeno funkční testování celé aplikace a ověření správné simulace a analytického řešení proběhlo na několika rozdílných modelech. Další testování bylo provedeno studenty předmětu KIV/VSS, kteří měli za úkol zpracovat alespoň jeden úkol na Markovské řetězce.

V jednotlivých fázích testování bylo nalezeno několik chyb, které byly opraveny. Dále byla ověřena funkčnost implementovaných modelů a vhodnost použití nástroje pro předmět KIV/VSS.

V rámci diplomové práce byl vytvořen webový nástroj pro práci s Markovskými řetězci s diskrétním i spojitým časem. Nástroj je vhodný pro analýzu a pochopení Markovskými řetězců. Tento nástroj má oproti nástroji Markov2, který je nyní používaný v předmětu KIV/VSS, více možností pro práci s Markovskými řetězci (např. vizualizace a simulace) a je tak vhodnou náhradou.

Literatura

[1] *CoffeeScript* [online]. 2014. [cit. 2019/02/25]. Dostupné z: https://coffeescript.org.

[2] Banks, H. et al. Simulation algorithms for continuous time Markov chain models. *Studies in Applied Electromagnetics and Mechanics*. 01 2012, 37, s. 3–18. doi: 10.3233/978-1-61499-092-5-3.

[3] Bolch, G. et al. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. Wiley-Interscience, 1998. ISBN 0-471-19366-6.

[4] *Browser Market Share Worldwide* [online]. StatCounter, 2017. [cit. 2019/02/18]. Dostupné z: http://gs.statcounter.com/.

[5] Dart. *A Tour of the Dart Language"* [online]. 2018. [cit. 2019/02/25]. Dostupné z: https://www.dartlang.org/guides/language/language-tour#important-concepts.

[6] Dart. *Dart Testin* [online]. 2017. [cit. 2019/02/25]. Dostupné z: https://www.dartlang.org/guides/testing.

[7] Dhingra, S. – S, P. – Madan, M. Finding Strongly Connected Components in a Social Network Graph. *International Journal of Computer Applications*. 02 2016, 136, s. 1–5. doi: 10.5120/ijca2016908481.

[8] Fačevicová, K. – Hron, K. – Kunderová, P. *Markovovy řetězce a jejich aplikace*. Univerzita Palackého, 2018. ISBN ISBN 978-80-244-5432-0.

[9] Foundation, J. *About QUnit* [online]. 2017. [cit. 2019/02/24]. Dostupné z: https://qunitjs.com/about/.

[10] Jack, R. *JavaScript Unit Testing for Beginners* [online]. 2016. [cit. 2019/02/24]. Dostupné z: https://designmodo.com/test-javascript-unit/.

[11] *A Storm is coming. A modern model checker for probabilistic systems.* [online]. RWTH Aachen University, 2012. [cit. 2019/01/12]. Storm model checker. Dostupné z: http://www.stormchecker.org/index.html.

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

[13] Marek, P. Spolehlivostní modelování pohotových systémů. Diplomová práce, Západočeská univerzita v Plzni, 2006.

[14] *Mathematics - Simulation* [online]. Technická univerzita Clausthal, 2019. [cit. 2019/01/15]. Dostupné z: https://www.mathematik.tu-clausthal.de/en/mathematics-interactive/simulation/.

[15] MIT – Critical – Data. *Secondary Analysis of Electronic Health Records*. Springer International Publishing, 2016. ISBN 978-3-319-43742-2.

[16] N. Gabow, H. Path-based depth-rst search for strong and biconnected components. *Inf. Process. Lett.* 05 2000, 74, s. 107–114. doi: 10.1016/S0020-0190(00)00051-X.

[17] Radek, H. Markovské náhodné procesy. Diplomová práce, Západočeská univerzita v Plzni, 1999.

[18] Richard, L. Analytické pravděpodobnostní modely, Markovské procesy. Dostupné z https://courseware.zcu.cz/CoursewarePortlets2/DownloadDokumentu?id=121909, 2016.

[19] Victor, P. – Lewis, L. *Markov Chains Explained Visually* [online]. 2014. [cit. 2019/01/15]. Dostupné z: http://setosa.io/ev/markov-chains/.

[20] W3C. *HTML and CSS - W3C* [online]. 2016. [cit. 2019/02/26]. Dostupné z: https://www.w3.org/standards/webdesign/htmlcss#whatcss.

A Příručka

Při spuštění aplikace se zobrazí v horní části navigační lišta, v pravé části editor s výchozím kódem a v levé části výchozí graf s ovládacími prvky pro simulaci.

A.1 Tvorba grafu manuálně

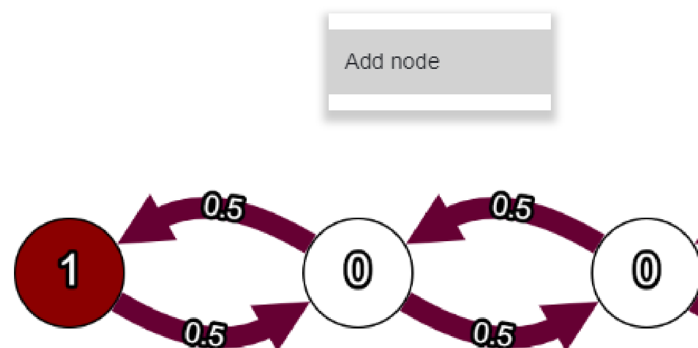
A.1.1 Přidání elementu

Pro přidání elementu se musí aplikace přepnout do režimu `Add`. To se provede v navigační liště (viz obrázek A.1).

Obrázek A.1: Výběr režimu Add

Přidání uzlu

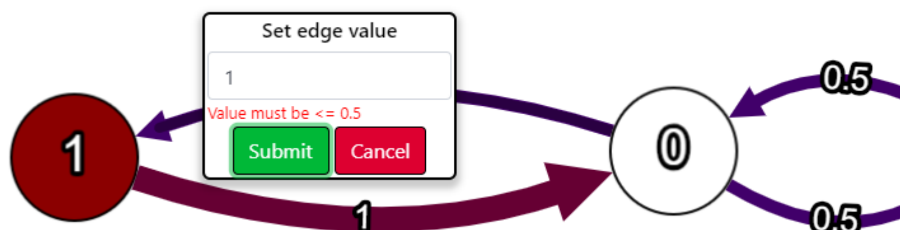
V režimu `Add` se přidání nového uzlu provádí kliknutím myši do prázdného místa. Přidání uzlu je možné i přes kontextové menu vyvolané přes pravé tlačítko myši v jakémkoliv režimu (viz obrázek A.2).



Obrázek A.2: Kontextové menu pro přidání uzlu

Přidání hrany

V režimu `Add` se pro přidání hrany musí označit dva uzly, mezi kterýma má být hrana. Po výběru druhého uzlu se zobrazí formulář pro zadání její hodnoty (hodnota musí být číslo větší než nula s maximálně třemi desetinnými místy). V případě, že jsou hodnoty hran pravděpodobnosti, je možné zadat hodnotu, která dá v součtu s ostatními výstupními hodnotami hran maximálně hodnotu 1 (viz obrázek A.3.



Obrázek A.3: Formulář pro přidání hrany

A.1.2 Smazání elementu

Elementy je možné mazat v režimu `Delete`, kde při kliknutí na hranu nebo uzel dojde k jeho smazání. Dále je možné mazat elementy přes hromadný výběr. Výběr se provádí přidržením klávesnice `CTRL` a tažením myši (stisklé levé tlačítko) přes elementy ke smazání. Po puštění tlačítka u myši dojde ke smazání označených elementů.

Smazání jde opět vykonat i přes kontextové menu.

A.1.3 Editace hodnot hran

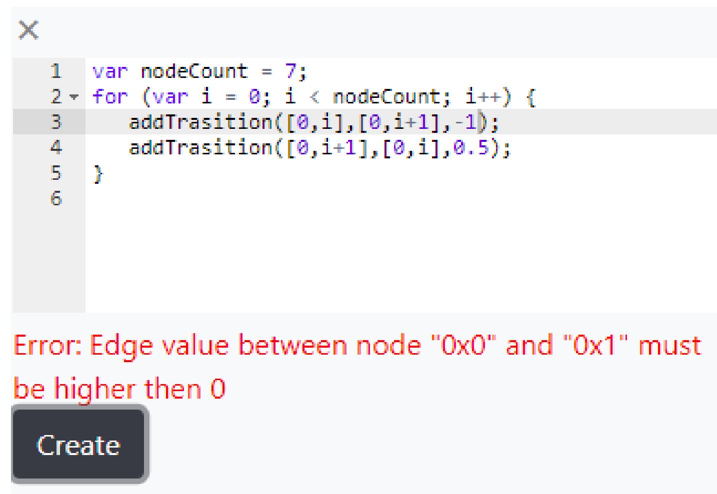
Pro editaci hran je nutné přepnout do režimu `Select`. V tomto režimu se při dvojkliku na hranu zobrazí formulář pro editaci hrany. Opět zde platí stejná pravidla jako u formuláře při vytváření hrany. Editace se musí potvrdit tlačítkem `Submit`.

A.2 Tvorba grafu pomocí kódu

Editor pro vložení kódu se zobrazí kliknutím na položku `Code editor`.

V editoru se graf definuje pomocí jazyka Javascript. Pro vkládání přechodů slouží funkce `addTransition`, kde první parametr je umístění počátečního uzlu v mřížce, druhým parametrem je pozice koncového uzlu v mřížce a třetí parametr je hodnota hrany. Volání funkce vypadá například takto: `addTransition([0,1],[1,1],0.5)`. Při vykonání této funkce se přidá počáteční stav do prvního sloupce a druhého řádku. Dále se přidá cílový stav do druhého sloupce a řádku v mřížce. Tyto stavy se propojí hranou o hodnotě 0.5.

Přiřazovaná hodnota musí být kladné číslo a v případě tvorby grafu s pravděpodobnostmi nesmí být součet hodnot výstupních hran z uzlu větší než hodnota 1. Pokud není, je vypsán uzel a nebo hrana, která má neplatnou hodnotu (viz obrázek A.4).



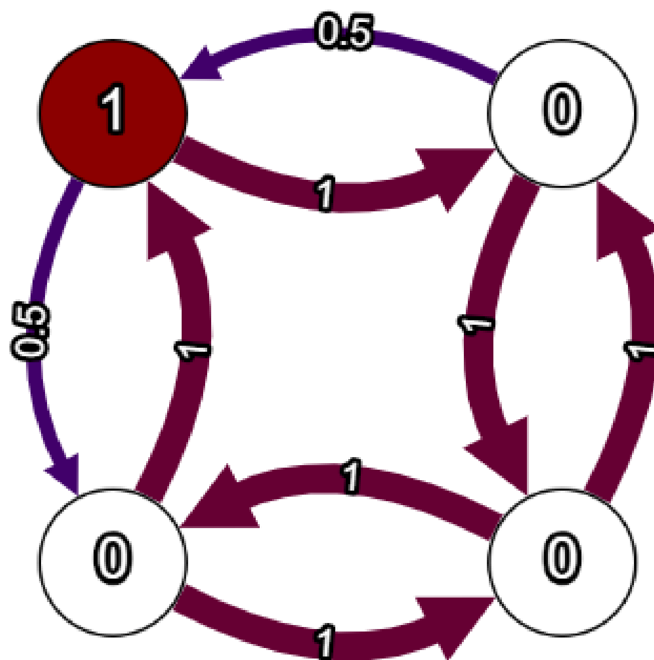
Obrázek A.4: Chybová hláška při vytváření grafu

A.2.1 Příklady pro vytvoření grafu

První model byl vytvořen pomocí skriptu A.1, kde byly použity pouze funkce `addTransition`. Výsledný model je vidět na obrázku A.5

```
addTransition([0,0],[0,1], 1);
addTransition([1,0],[0,0], 1);
addTransition([0,0],[1,0],0.5);
addTransition([0,1],[0,0],0.5);
addTransition([0,1],[1,1],1);
addTransition([1,1],[0,1],1);
addTransition([1,1],[1,0],1);
addTransition([1,0],[1,1],1);
```

Listing A.1: Skript pro vytvoření prvního modelu



Obrázek A.5: Graf k ukázce skriptu skriptu A.1

Druhý příklad je na frontu typu `M/M/3` s omezením na 6 stavů. V ukázce A.2 je napsaný kód pro zobrazení grafu, který je možné vidět na obrázku A.6.

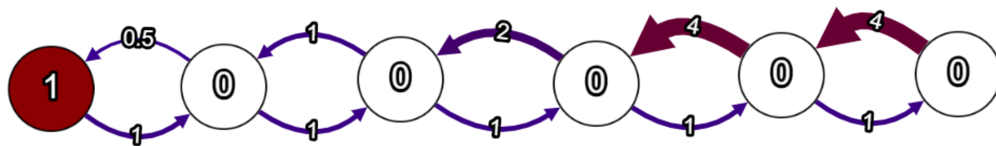
```
var alfa = 1;
var micro = 0.5;
var m = 3;
for (var i = 0; i < 5; i++) {
    addTransition([0,i],[0,i+1], alfa);
    addTransition([0,i+1],[0,i],micro);
    if(i < m) {
        micro += micro;
    }
}
```

Listing A.2: Skript pro vytvoření modelu s frontou

A.3 Ovládání simulace

Pro ovládání simulace slouží panel umístěný ve spodní části aplikace (viz obrázek A.7. Tento panel je viditelný pouze v režimu `Select`.

Tlačítkem `play` se spustí simulace. Simulace začíná z červeně označeného stavu. Výchozí stav lze změnit dvojklikem na jakýkoliv jiný stav a nebo přes



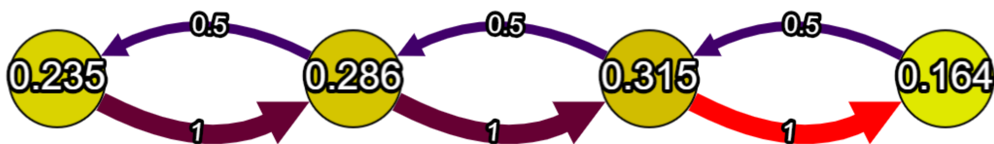
Obrázek A.6: Graf k ukázce skriptu skriptu A.2

Obrázek A.7: Ovládací panel pro simulaci

kontextovou nabídku. Tlačítkem `reset` se uvede model dop původního stavu. Rychlost simulace lze změnit pomocí posuvníku.

Tlačítko `next` provedete skok v simulaci o tolik kroků, kolik je napsáno ve vstupním políčku označeném popiskem `Steps`. Počet kroků nemá vliv na běžící simulaci, která stále běží po jednom kroku.

Aktuální přechod resp uzel je označený červenou barvou, jak je vidět na obrázku A.8.

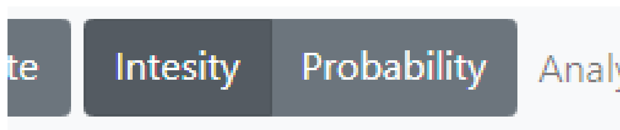


Obrázek A.8: Zvýraznění aktuální hrany

A.4 Přepnutí typu Markovského řetězce

Přepnutí mezi Markovským řetězce s diskrétním časem a spojitým časem se provádí v navigační liště (viz obrázek A.9). Pro přepnutí na Markovský řetězec s diskrétním časem slouží položka `Probability` a se spojitým časem položka `Intensity`.

Při přepnutí dochází k modifikaci hodnot hran a tato změna je nevratná. Proto se musí potvrdit přepnutí typu modelu v dialogovém okně.

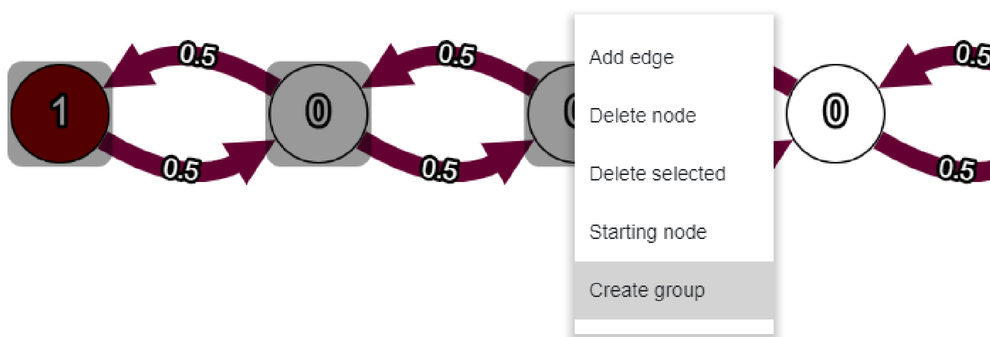


Obrázek A.9: Panel pro změnu typu Markovského řetězce

A.5 Skupiny stavů

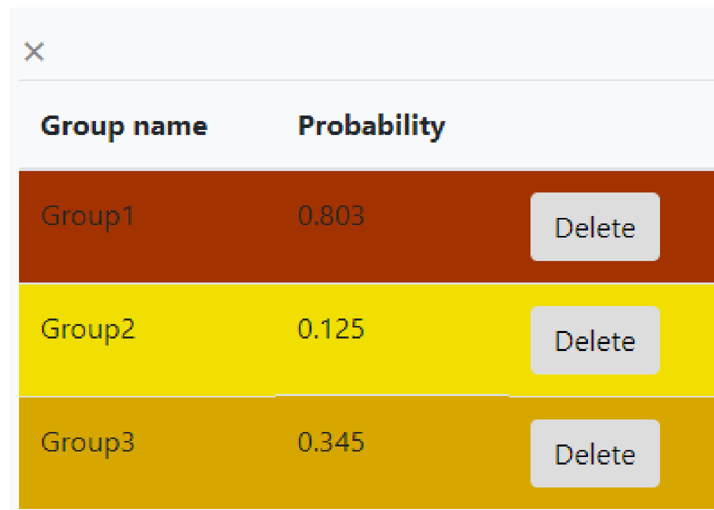
Skupinu stavů je možné přidat v režimu `Select`. Označení stavů ve skupině lze provézt přidržením klávesnice `CTRL` a tažením myši (stisklé levé tlačítko) přes stavy anebo přidržením klávesnice `CTRL` a vybráním požadovaných stavů.

Po označení uzlů se při kliknutí na uzel pravým tlačítkem myši vyvolá kontextové menu. Skupina se vytvoří pomocí položky `Create group`. Následně se objeví formulář pro zadání názvu skupiny a po ptvrzení dojde k přidání (viz obrázek A.10).



Obrázek A.10: Přidání skupiny

Vytvořené skupiny lze zobrazit při kliknutí na položku `Node groups` (viz obrázek A.11). Pravděpodobnost skupiny je součet všech obsažených stavů najetí myší na na skupinu se stavy vyznačí.



Obrázek A.11: Seznam skupin stavů

A.6 Export a import grafu

Pro export a import dat slouží položky v navigačním menu.

Export path

Provede se export cesty aktuální simulace. Cesta je popsaná identifikátory stavů.

Export nodes history

Provede export grafů s historií pravděpodobností pro každý uzel.

Export

Provedete export grafu (soubor `graph_export`, kódu všech informací, které jsou potřeba pro opětovné nahrání dat do aplikace.

Import

Import slouží pro nahrání vyexportovaných dat.

B Zprovoznění vývojového prostředí

Uvedený popis je určený pro Windows 10 64bit.

Pro vývoj aplikace je nutné nainstalovat správce balíčků pro JavaScript `npm`, který je součástí Node.js[1]. Node.js stáhnete ze stránky https://nodejs.org/en/. Poslední vyzkoušená verze Node.js byla 10.15.3 LTS.

Po nainstalování zkopírujte obsah ze složky src (obsah složky je popsaný v kapitole 6.1), která se nacházejí v kořenovém adresáři na přiloženém CD, do složky, kde chcete vyvíjet aplikaci. Následně v této složce pro vývoj spusťte příkazový řádek a v něm spusťte příkaz `npm install`. Dojde ke stažení všech potřebných balíčků pro aplikaci a vytvoří se složka `node_modules`. V této složce se nachází složka `ace-builds`, do které je nutné překopírovat soubor `ace.d.ts` ze složky `types`. Překopírování je nutné z důvodu špatného typování u knihovny `ace-builds` (chybí návratové typy u některých funkcí). Soubor `ace.d.ts` je nutné smazat ze složky `types`. Poté spusťte příkaz k sestavení aplikace pro vývoj `npm run build:dev`.

Po sestavení aplikace se vytvoří složka `dist`. Do této složky je nutné přesunout soubor Matrix.js ze složky lib.

Následně pomocí příkazu `npm start` spustíte aplikaci a server pro vývoj. Aplikace poběží na adrese `http://localhost:3000`. Port lze změnit ve složce `webpack.config.js` u položky `devServer`. Pokud takto běží server, je možné upravovat kód v textovém editoru a nebo vývojovém prostředí a při uložení souboru se změna projeví po aktualizaci webové stránky s aplikací.

Sestavení aplikace pro nasazení se provádí příkazem `npm run build:prod`. Výsledek sestavení bude ve složce `dist`. Stejně jako při sestavování aplikace pro vývoj, je i zde nutné přidat soubor Matrix.js ze složky lib. Při sestavování se ve složce vytvoří navíc nepotřebné soubory. Pro běh aplikace jsou potřebné pouze soubory: `index.html, Matrix.js a app.bundle.js`.

Jednotkové testy (viz kapitola 7.1.1) se spouští příkazem `npm test`. Spustí se okno prohlížeče, ve kterém se vykonají testy a výsledek testů je možné vidět v příkazové konzoly.

[1] https://nodejs.org/en/