Vizualizace skákajících konečných automatů

Ladislav Kollár

Bakalářská práce
2023

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Ladislav Kollár**
Osobní číslo:	**A20224**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Vizualizace skákajících konečných automatů**
Téma práce anglicky:	**The Visualisation of Jumping Finite Automata**

Zásady pro vypracování

1. Popište teorii týkající se skákajících konečných automatů (dále SKA).
2. Do popisu zahrňte SKA obousměrné i jednosměrné, deterministické i nedeterministické.
3. Ve vhodném programovacím jazyce sestavte simulátor různých variant SKA.
4. Simulátor musí umožňovat jednoduché zadávání SKA ve zvolené variantě, jejich editaci a názornou vizualizaci jejich činnosti.

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. H. CHIGAHARA, S. Z. FAZEKAS, a A. YAMAMURA: „One-way jumping finite automata", International Journal of Foundations of Computer Science, díl 27, 2016, str. 391-405.
2. E. CSUHAJ-VARJÚ, C. MARTÍN-VIDE a V. MITRANA: „Multiset automata" v Multiset processing — mathematical, computer science, and molecular computing points of view, C. S. CALUDE, G. PAUN, G. ROZENBERG a A. SALOMAA, Ed., Lecture notes in computer science, díl 2235, Berlín: Springer, 2001, str. 69–83.
3. ČERNÁ, Ivana, a kol.: Automaty a formální jazyky I. Učební text FI MU. Fakulta informatiky, Masarykova univerzita, Brno: 2002. Dostupné z: http://is.muni.cz/elportal/estud/fi/js06/ib005/Formalni_jazyky_a_automaty_I.pdf
4. A. MEDUNA a P. ZEMEK: „Jumping finite automata", International Journal of Foundations of Computer Science, díl 23, 2012, str. 1555-1578.
5. M. SIPSER, Introduction to the Theory of Computation, 2. vyd., Boston: Thomson Course Technology, 2006.

Vedoucí bakalářské práce: **Ing. Pavel Martinek, Ph.D.**
Ústav matematiky

Datum zadání bakalářské práce: **2. prosince 2022**
Termín odevzdání bakalářské práce: **26. května 2023**

L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 7. prosince 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ladislav Kollár, v.r.

Ve Zlíně, dne …….

podpis studenta

ABSTRAKT

Konečné automaty jsou základní modely používané v informatice a teorii výpočtů využívané k analýze a řešení různých výpočetních problémů. Tradiční konečné automaty však mají určitá omezení ve své vyjadřovací schopnosti, zejména při řešení složitých problémů s nelokálními závislostmi. Toto vedlo k vývoji skákající konečných automatů jako rozšíření tradičních modelů, které poskytují větší výpočetní možnosti.

Tato bakalářská práce se zabývá skákajícími konečnými automaty, což jsou rozšířené modely konečných automatů umožňujících přeskakování libovolného počtu symbolů v řetězci. Skákající konečné automaty poskytují zvýšenou výpočetní sílu a jsou vhodné pro řešení problémů s nesousedními závislostmi.

V teoretické části práce se zaměřujeme na formální popis skákajících konečných automatů, včetně jejich definice, pravidel přechodů a vlastností. Analyzujeme rozdíly mezi skákajícími konečnými automaty a tradičními konečnými automaty, definujeme modely skákajících konečných automatů a jejich vlastnosti.

V praktické části práce vyvíjíme demonstrační aplikaci nazvanou *JFA Simulator*, která umožňuje vizualizaci a experimentování se skákajícími konečnými automaty. Aplikace umožňuje uživatelům zadávat zápis skákajících konečných automatů a sledovat jejich výpočetní chod. Díky grafickému zobrazení automatu si uživatelé mohou lépe představit a porozumět výpočtům skákajících konečných automatů.

Výsledkem práce je teoretický popis skákajících konečných automatů a jejich implementace v podobě demonstrační aplikace *JFA Simulator*. Naše práce poskytuje užitečný nástroj pro zkoumání a experimentování s těmito rozšířenými modely automatů, a přispívá tak k lepšímu porozumění jejich výpočetních možností.

Klíčová slova: konečné automaty, skákající konečné automaty, vizualizace, demonstrační aplikace

ABSTRACT

Finite automata are basic models used in computer science and computational theory to analyze and solve various computational problems. However, traditional finite automata have some limitations in their expressive power, especially when solving complex problems with nonlocal dependencies. This has led to the development of jumping finite automata as an extension of traditional models to provide greater computational capabilities.

This thesis deals with jumping finite automata, which are extended models of finite automata that allow skipping of any number of symbols in a string. Jumping finite automata provide increased computational power and are suitable for solving problems with nonadjacent dependencies.

In the theoretical part of the paper, we focus on the formal description of jumping finite automata, including their definition, transition rules and properties. We analyze the differences between jumping finite automata and traditional finite automata, and define models of jumping finite automata and their properties.

In the practical part of the work, we develop a demonstration application called *JFA Simulator* that allows visualization and experimentation with jumping finite automata. The application allows users to enter the notation of jumping finite automata and observe their computational operation. With a graphical representation of the automaton, users can better visualize and understand the computation of jumping finite automata.

The result of this work is a theoretical description of jumping finite automata and its implementation in the form of a *JFA Simulator* demonstration application. Our work provides a useful tool for exploring and experimenting with these extended models of automata, thus contributing to a better understanding of their computational capabilities.

Keywords: finite automata, jumping finite automata, visualisation, demonstration aplication

Poděkování, motto a čestné prohlášení, že odevzdaná verze bakalářské práce a verze elektronická, nahraná do IS/STAG jsou totožné ve znění:

Prohlašuji, že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

Tato bakalářská práce se zabývá tématikou skákajících konečných automatů, což jsou modely konečných automatů rozšířené o schopnost přeskakovat libovolný počet symbolů v řetězci. Skákající konečné automaty poskytují zvýšenou výpočetní sílu a jsou užitečné pro řešení problémů s nesousedními závislostmi. Na tuto tématiku jsme vytvořili demonstrační aplikaci určenou pro vizualizaci výpočtů skákajících konečných automatů. V práci se zabýváme deterministickým i nedeterministickým skákajícím konečným automatem, včetně jednosměrného i obousměrného modelu.

V teoretické části této práce se zabýváme formálním popisem skákajících konečných automatů, včetně všech zmíněných variant, pojmů a vlastností, které jsou potřeba k popisu. Zabýváme se tím, jak se skákající konečný automat liší od tradičního konečného automatu, čím je rozšiřován a jak se chová. Zároveň se věnujeme různým modelům skákajících konečných automatů. Vzhledem k tomu, že tato teorie je relativně nová, není v českém jazyce dostupné velké množství zdrojů.

V praktické části práce se zaměřujeme na vývoj demonstrační aplikace nazvané *JFA Simulator*, která slouží k vizualizaci chodu a výpočtů specifických modelů skákajících konečných automatů. Tato aplikace byla implementována v jazyce Python 3.10 a umožňuje uživatelům graficky zobrazovat důležité vnitřní informace modelu skákajícího konečného automatu, který uživatel definuje. Mezi tyto informace patří současný stav automatu, vstupní řetězec a informace o přečtených symbolech.

Spojením teoretického zkoumání skákajících konečných automatů s praktickou implementací demonstrační aplikace *JFA Simulator* se tato práce snaží přispět k lepšímu porozumění a aplikaci těchto výpočetních modelů.

I. TEORETICKÁ ČÁST

1 ZÁKLADNÍ POJMY

Pro pochopení teorie skákajících konečných automatů (dále JFA) je potřeba mít základní znalosti teoretické informatiky, jako jsou například formální jazyky, teorie automatů a všechny vlastnosti s nimi spojenými. V této kapitole se čtenář seznámí s teorií a vysvětlením těchto základních pojmů a termínů jako je konečný automat, stavy, přechodová funkce a deterministický i nedeterministický automat. Dále se čtenář dozví, jak se tato tematika aplikuje na model JFA a jak jsou tyto automaty dále modifikovány.

1.1 Konečný automat

Konečný automat je abstraktní výpočetní model, který může být v jednu chvíli pouze v jednom z několika konečně zadaných stavů, ve kterém na základě podnětu nějakého vstupu může zůstat anebo může přejít z daného stavu do druhého. Tuto změnu ze stavu *m* do stavu *n* nazýváme přechod [1]. Konečný automat jako stroj neobsahuje žádnou paměťovou jednotku, veškerá činnost je prováděna pomocí stavů a přechodů mezi nimi. Konečný automat je definovaný množinou stavů, počátečním stavem, množinou koncových stavů (musí být obsaženy v množině stavů) a přechodovou funkcí [2]. Model konečných automatů se dá využít v různých oblastech, nejvíce jsou používané v poli softwarového inženýrství jako metoda modelování softwarových systémů pro účely zjednodušení, optimalizaci a verifikaci funkcionality, ale také jsou používané v poli lingvistiky pro syntaktickou analýzu a pochopení jazyků, v poli umělé inteligence [3] a další.

Formální zápis konečného automatu

Konečným automatem nazýváme uspořádanou pětici $(Q, \Sigma, \delta, q_0, F)$, kde:

- Q je konečná neprázdná množina s prvky zvanými *stavy*.
- Σ je abeceda; známá také pod názvem *vstupní abeceda*.
- δ: $Q \times \Sigma \rightarrow Q$ je tzv. *přechodová funkce*.
- $q_0 \in Q$ je tzv. *počáteční stav*.
- $F \subseteq Q$ je tzv. *množina koncových stavů*.

Množina stavů

V konečných automatech se množinou stavů rozumí množina všech možných vnitřních stavů, ve kterých se automat může v daném okamžiku nacházet. Ačkoliv může být každý stav označen libovolným uskupením znaků, nejčastěji je používaný zápis $q_0, q_1, \dots, q_n$

Pětistavový konečný automat může mít množinu stavů:

$$Q = \{q_0, q_1, q_2, q_3, q_4\}$$

Vstupní abeceda

Vstupní abeceda je konečnou neprázdnou množinou symbolů, které může konečný automat přijímat jako platný vstup. Dá se chápat jako reprezentace možných podnětů, které můžou zapříčinit přechod ze stavu do stavu podle přechodové funkce.

Pro automat, který je schopný přijímat vstupní řetězce, které obsahují symboly „*A*“, „*B*“ a „*C*“, by vstupní abeceda byla:

$$\Sigma = \{A, B, C\}$$

Vstupní řetězec

Vstupní řetězec je konečná posloupnost symbolů, které bude konečný automat přijímat, zpracovávat a na základě kterých bude automat provádět přechody mezi stavy. Vstupní řetězec je přímo závislý na vstupní abecedě, jelikož vstupní abeceda definuje, které symboly je konečný automat schopný rozeznávat a zpracovávat.

Ačkoliv vstupní řetězec není součástí formálního zápisu konečného automatu, je to stále nedílná součást jeho chování a provozu, jelikož představuje konkrétní vstupní data, která automat zpracovává.

Přechodová funkce

Přechodová funkce zobrazuje dvojici aktuálního stavu konečného automatu a aktuálně čteného symbolu ze vstupního řetězce na nový stav (v případě deterministického automatu) nebo množinu stavů (v případě nedeterministického automatu). [4]

Formální zápis přechodové funkce pro deterministický konečný automat je δ: $Q \times \Sigma \rightarrow Q$, kde:

- Q je množina stavů.
- Σ je vstupní abeceda.

Počáteční stav

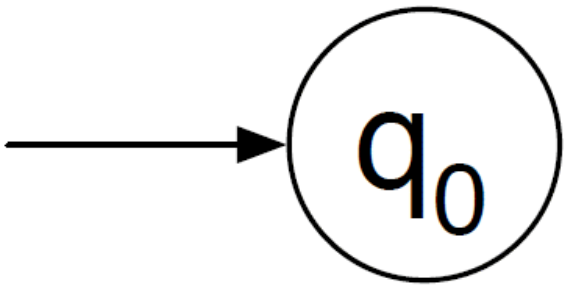
Počáteční stav je přesně jeden stav z množiny stavů, ve kterém konečný automat začíná zpracovávat vstupní řetězec.

Množina koncových stavů

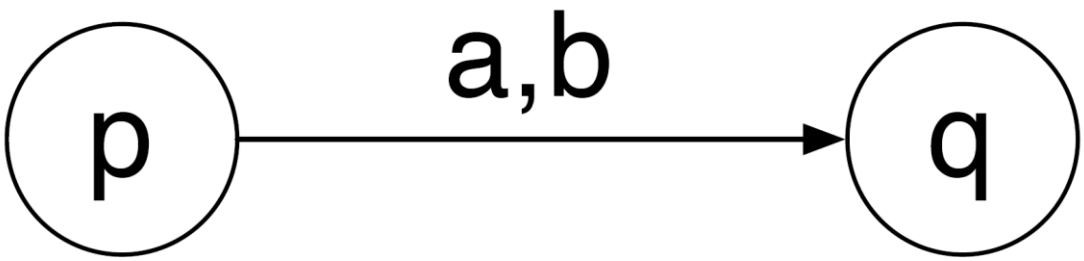
Množina koncových stavů je podmnožina množiny stavů. Koncové stavy jsou využívány pro vyhodnocování přijetí nebo odmítnutí vstupního řetězce. Pokud konečný automat zpracuje celý vstupní řetězec a skončí ve stavu, který je v této množině, vstupní řetězec přijme, jinak jej zamítne.

Pro větší přehlednost bývá konečný automat často znázorňován tzv. *stavovým diagramem*, který představuje lehce modifikovaný orientovaný graf [1]. Pro stavový diagram platí:

- Každému stavu konečného automatu odpovídá právě jeden uzel grafu, označený názvem stavu.
- Mezi uzly stavů q a p vede hrana označená symbolem a, pokud platí $\delta(p, a) = q$.
- Každý uzel, který odpovídá koncovému stavu, je označen dvojitým kroužkem.
- Do uzlu počátečního stavu je nakreslena hrana bez označení (viz. Obrázek 1).
- V případě $\delta(p, b) = q$ a $\delta(p, a) = q$, tzn. pokud přechodová funkce obsahuje přechod mezi stavy q a p čtením symbolu b a zároveň obsahuje přechod mezi stavy q a p čtením symbolu a, do stavového diagramu se často kreslí jedna hrana označená dvojicí symbolů podle přechodové funkce (viz. Obrázek 2).



Obrázek 1: Označení počátečního stavu



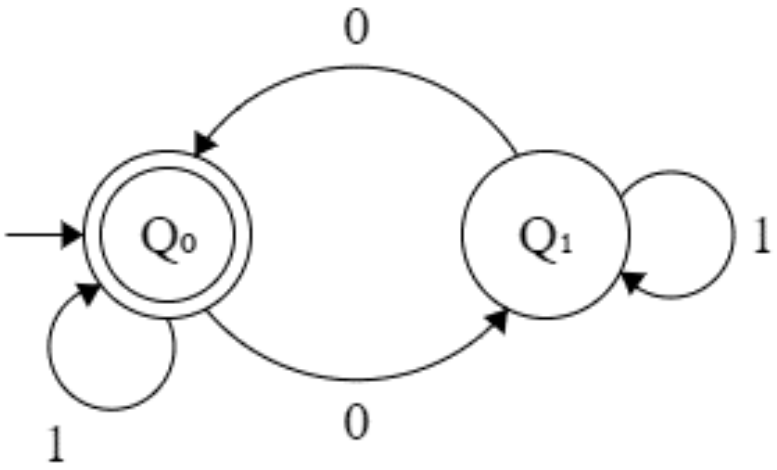
Obrázek 2: Označení dvou přechodů mezi stavy

1.2 Deterministický konečný automat

Deterministický konečný automat je model konečného automatu, u kterého platí, že pro každou kombinaci stavu q a symbolu vstupní abecedy b existuje právě jeden stav p takový, že $\delta(q, b) = p$. Determinismus se v této verzi konečného automatu projevuje deterministickým chováním, kdy pro jedno nastavení konečného automatu a jeden vstupní řetězec bude výpočet konečného automatu totožný. Pro deterministický konečný automat tedy platí, že jeden stav může mít v jednu chvíli pouze jeden přechod do jiného stavu. Pro stavový diagram deterministického konečného automatu tedy musí platit, že každá hrana vycházející ze stavu musí mít symbol, který se nevyskytuje u žádné jiné hrany vycházejícího z tohoto stavu [5].

Příklad deterministického konečného automatu: automat s binární vstupní abecedou, který přijímá řetězce, které obsahují nulový nebo sudý počet znaků “0“

- $Q = \{Q_0, Q_1\}$
- $\Sigma = \{0,1\}$
- $q_0 = Q_0$
- Přechodová funkce δ je popsána stavovým diagramem na obrázku 3
- $F = \{Q_0\}$



Obrázek 3: Příklad deterministického konečného automatu

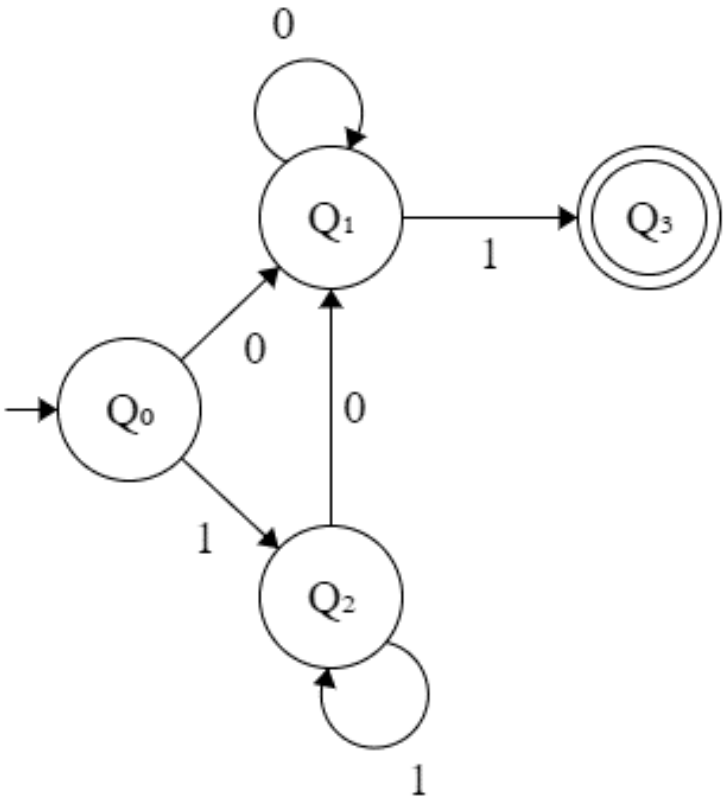
1.3 Nedeterministický konečný automat

Nedeterministický konečný automat je model konečného automatu, který se odlišuje od deterministického tím, že pro každou dvojici stavu q a symbolu b ze vstupní abecedy, nemusí existovat jediný stav p takový, že $\delta(q, b) = p$. Nedeterminismus zde znamená, že pro jedno nastavení konečného automatu a jeden vstupní řetězec se může konečný automat zachovat několika různými způsoby. Nedeterministický konečný automat může podle nastavení přejít z jednoho stavu do některého ze stavů (může jich být více), které mu předepisuje přechodová funkce. Pro stavový diagram už neplatí pravidlo, že každá hrana vycházející ze stavu musí mít unikátní symbol jako u deterministického konečného automatu.

Vzhledem k chování nedeterministického konečného automatu je tento model více komplexní a složitější na vizualizaci.

Příklad nedeterministického konečného automatu: automat o binární abecedě, který přijímá řetězce, které obsahují podřetězce “01“ nebo “10“

- $Q = \{Q_0, Q_1, Q_2, Q_3\}$
- $\Sigma = \{0,1\}$
- $q_0 = Q_0$
- Přechodová funkce δ je popsána stavovým diagramem na obrázku 4
- $F = \{Q_3\}$



Obrázek 4: Příklad nedeterministického konečného automatu

2 SKÁKAJÍCÍ KONEČNÉ AUTOMATY

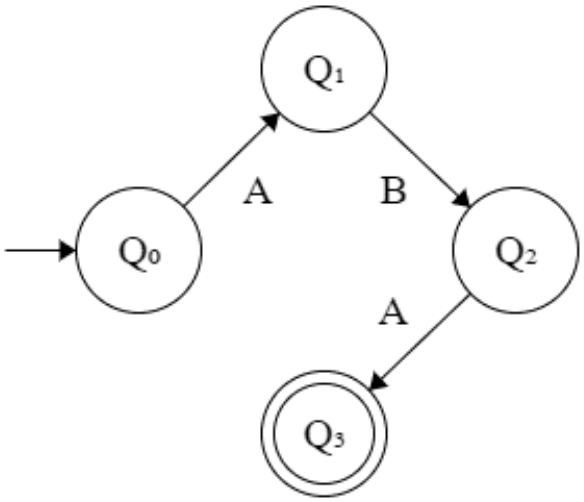
Skákající konečné automaty jsou modifikací konečných automatů, které jsou schopné číst symboly ze vstupního řetězce nespojitě v jakémkoliv pořadí, což znamená, že po přečtení symbolu a přechodu z jednoho stavu do druhého mohou přeskočit některé symboly vstupního řetězce a pokračovat ve výpočtu dále od jiného symbolu. [6] Množství přeskočených symbolů se řídí podle možného přechodu ve stavu, do kterého automat naposledy přešel. V tomto novém stavu je automat schopen přeskočit libovolné množství znaků za cílem nalezení znaku, který by povoloval další přechod do stavu jiného. Všechny přeskočené symboly ve vstupním řetězci zůstávají, za přečtené se považují pouze symboly, na základě kterých byl proveden přechod ze stavu do stavu.

Díky této speciální vlastnosti skákajících konečných automatů je potřeba znovu definovat deterministické a nedeterministické chování. Kromě těchto variací se skákající konečný automat ještě dělí podle metody přeskakování symbolů na obousměrné a jednosměrné.

Pro popis skákajících konečných automatů budeme používat stejnou notaci jako u klasických konečných automatů.

Mějme například konečný automat $K = (Q, \Sigma, \delta, q_0, F)$, kde:

- $Q = \{Q_0, Q_1, Q_2, Q_3\}$
- $\Sigma = \{A, B\}$
- $q_0 = Q_0$
- $F = \{Q_3\}$
- Přechodová funkce δ podle stavového diagramu (viz. Obrázek 5)
- Vstupní řetězec $w = "AAB"$



Obrázek 5: Stavový diagram konečného automatu K

Klasický model konečného automatu by přečetl a zpracoval symbol „*A*“ na první pozici v řetězci a provedl přechod ze stavu Q_0 do Q_1. Automat by dále měl zpracovat symbol „*A*“ na druhé pozici, ale přechodová funkce toto neumožní. Klasický konečný automat by se zde zasekl a nedokončil výpočet.

Skákající konečný automat by oproti klasickému byl schopný výpočet dokončit. Automat by přečetl a zpracoval symbol “A“ na první pozici a provedl přechod ze stavu Q_0 do Q_1. V tomto stavu by přeskočil symbol “A“ na druhé pozici, jelikož pro tento symbol není ze současného stavu definovaný přechod. Automat by přečetl symbol “B“ na třetí pozici a provedl přechod ze stavu Q_1 do Q_2. Dále by přečetl symbol “A“ na druhé pozici a provedl přechod ze stavu Q_2 do Q_3. Jelikož automat v tuto chvíli dokončil čtení celého vstupního řetězce a ukončil výpočet v koncovém stavu, vstupní řetězec by byl přijat.

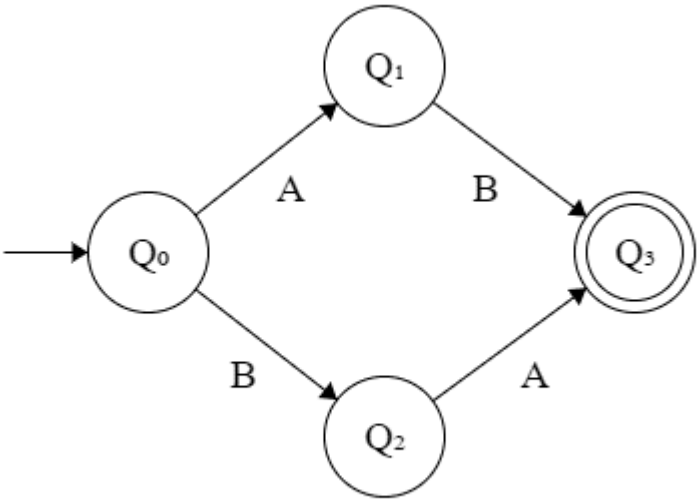
2.1 Deterministické skákající konečné automaty

Deterministický skákající konečný automat je model skákajícího konečného automatu, kde platí, že pro každý stav q existuje **maximálně jeden** stav p a **maximálně jeden** symbol $b \in \Sigma$ tak, že $\delta(q, b) = p$.[1] Tímto se odlišuje od deterministického konečného automatu, pro který platila pouze unikátnost páru symbolu b a stavu p pro $\delta(q, b) = p$. Hlavní důvod výše uvedeného omezení deterministického skákajícího konečného automatu je jeho klíčová schopnost přeskakování libovolného počtu znaků vstupního řetězce za cílem nalezení znaku pro provedení přechodu do jiného stavu. V případě, kdy existuje více odchozích přechodů do jiného stavu ze stavu současného a vstupní řetězec obsahuje nezpracované symboly pro více než jeden z těchto přechodů, skákající konečný automat má více možností, jak se zachovat.

Mějme například skákající konečný automat $K = (Q, \Sigma, \delta, q_0, F)$, kde:

- $Q = \{Q_0, Q_1, Q_2, Q_3\}$
- $\Sigma = \{A, B\}$
- $q_0 = Q_0$
- $F = \{Q_3\}$
- Přechodová funkce δ podle stavového diagramu (viz. Obrázek 6)
- Vstupní řetězec $w = "AB"$

[1] Ve článku [6] není deterministický JFA korektně zaveden [11], proto je v této práci přejata definice z [12], kde je definován deterministický multi-množinový konečný automat, jehož chování koresponduje s chováním deterministického skákajícího konečného automatu.



Obrázek 6: Porušení determinismu u
skákajícího konečného automatu

V tomto případě má automat ve stavu Q_0 při čtení a zpracování prvního znaku 2 možnosti výpočtu

- Přečtení znaku na první pozici; $\delta(Q_0, A) = Q_1$
- Přeskočení znaku na první pozici, přečtení znaku na druhé pozici; $\delta(Q_0, B) = Q_2$

Jelikož existují dvě možné varianty, jak se skákající konečný automat může zachovat, tento automat se nechová deterministicky.

2.2 Nedeterministické skákající konečné automaty

Nedeterministický skákající konečný automat je model skákajícího konečného automatu, kde platí, že pro každý stav q **může existovat** více než jeden stav p a jeden symbol $b \in \Sigma$ tak, že $\delta(q, b) = p$ [7]. Příklad nedeterministického skákajícího konečného automatu je na obrázku 6.

2.3 Skákající obousměrné konečné automaty

Skákající obousměrný konečný automat je typ skákajícího konečného automatu, který byl popsán výše. Tzn. automat může přeskakovat symboly vstupního řetězce v obou směrech (přeskakovat doleva i doprava) [8] [9].

2.4 Skákající jednosměrné konečné automaty

Vzhledem k problémům s definicí deterministického skákajícího konečného automatu v článku 6 (viz poznámka 1 na straně 17 této práce) přišli autoři Chigihara, Szilárd a Akihiro [10] na způsob zavedení deterministického chování založeného na skocích v řetězci pouze jedním směrem.

Deterministický jednosměrný skákající konečný automat definují stejně jako je definován běžný deterministický konečný automat, tzn. v jeho přechodové funkci existuje pro každou kombinaci stavu q a symbolu vstupní abecedy b existuje nejvýše jeden stav p takový, že $\delta(q, b) = p$. Pro jeho chování platí, že automat může přeskakovat symboly vstupního řetězce pouze v jednom směru, a to doprava[2]. Jednosměrný automat začíná číst a zpracovávat symbol zcela vlevo. Pokud má v aktuálním stavu pro čtený symbol v přechodové funkci definovaný přechod, symbol přečte a přejde do nového stavu. Pokud ne, přeskočí čtený symbol a přejde na následující symbol vstupního řetězce. Tento proces opakuje, dokud nedorazí na konec vstupního řetězce. Má-li přečtené všechny symboly, zachová se stejně jako klasický konečný automat (tj. jestli ukončí výpočet v koncovém stavu, řetězec přijme, jinak odmítne). Nemá-li přečtené všechny symboly, vrátí se na nejlevější dosud nepřečtený symbol vstupního řetězce a pokračuje dále ve výpočtu. [10]

Na základě této definice je zřejmé, že chování jednosměrného skákajícího konečného automatu je deterministické.

2.5 Jazyky přijímané skákajícími konečnými automaty

Nedeterministické skákající konečné automaty mají větší výpočetní sílu než deterministické, tj. kromě jazyků, které přijímají deterministické automaty, přijímají i další jazyky, jak vyplývá z tvrzení 2 v článku [11][3]. Z tvrzení 2 v [11] a z vlastností jazyků popsaných v [10] na schématu z obr. 1 a ve větě 9 dále plyne nesrovnatelnost jazyků přijímaných jednosměrnými skákajícími automaty a deterministickými / nedeterministickými obousměrnými skákajícími automaty.

---

[2] Autoři Chigihara, Szilárd a Akihiro sice zavádí i variantu s přeskakováním směrem doleva, ale protože nejde o principiálně odlišné chování, není tato varianta v této práci dále uvažována.

[3] který se sice věnuje multi-množinovým automatům, ale jejich chování koresponduje s chováním skákajících konečných automatů

II. PRAKTICKÁ ČÁST

3 TVORBA SOFTWAROVÉ APLIKACE

Abychom mohli prozkoumat chování skákajících konečných automatů a získat přehled o jejich fungování, vytvořili jsme demonstrační aplikaci pro jejich vizualizaci. Tato aplikace umožňuje uživatelům definovat skákající konečný automat pomocí grafického rozhraní a pozorovat jeho chování při zpracování vstupních řetězců. Vizualizací běhu a přechodů automatu v reálném čase mohou uživatelé hlouběji pochopit, jak tyto automaty fungují a jak je lze použít k řešení problémů v různých oblastech. V této kapitole popisujeme návrh a implementaci demonstrační aplikace a její klíčové funkce a možnosti.

Podle zadání je aplikace schopná:

- Zadání konkrétního skákajícího konečného automatu
- Editace zadaného skákajícího konečného automatu
- Vizualizace činnosti skákajícího automatu pro zadaný vstupní řetězec

Během průběžného testování aplikace byla přidána dodatečné funkce:

- Ukládání automatu z aplikace do souboru
- Načítání automatu ze souboru do aplikace

3.1 Zvolený jazyk, použité knihovny

Aplikace byla napsána v programovacím jazyce Python 3.10, GUI bylo vytvořeno skrze aplikaci Qt Designer a napojená na back-end aplikační logiku prostřednictvím PyQt5 frameworku.

V projektu byly dále využité některé zabudované Python knihovny:

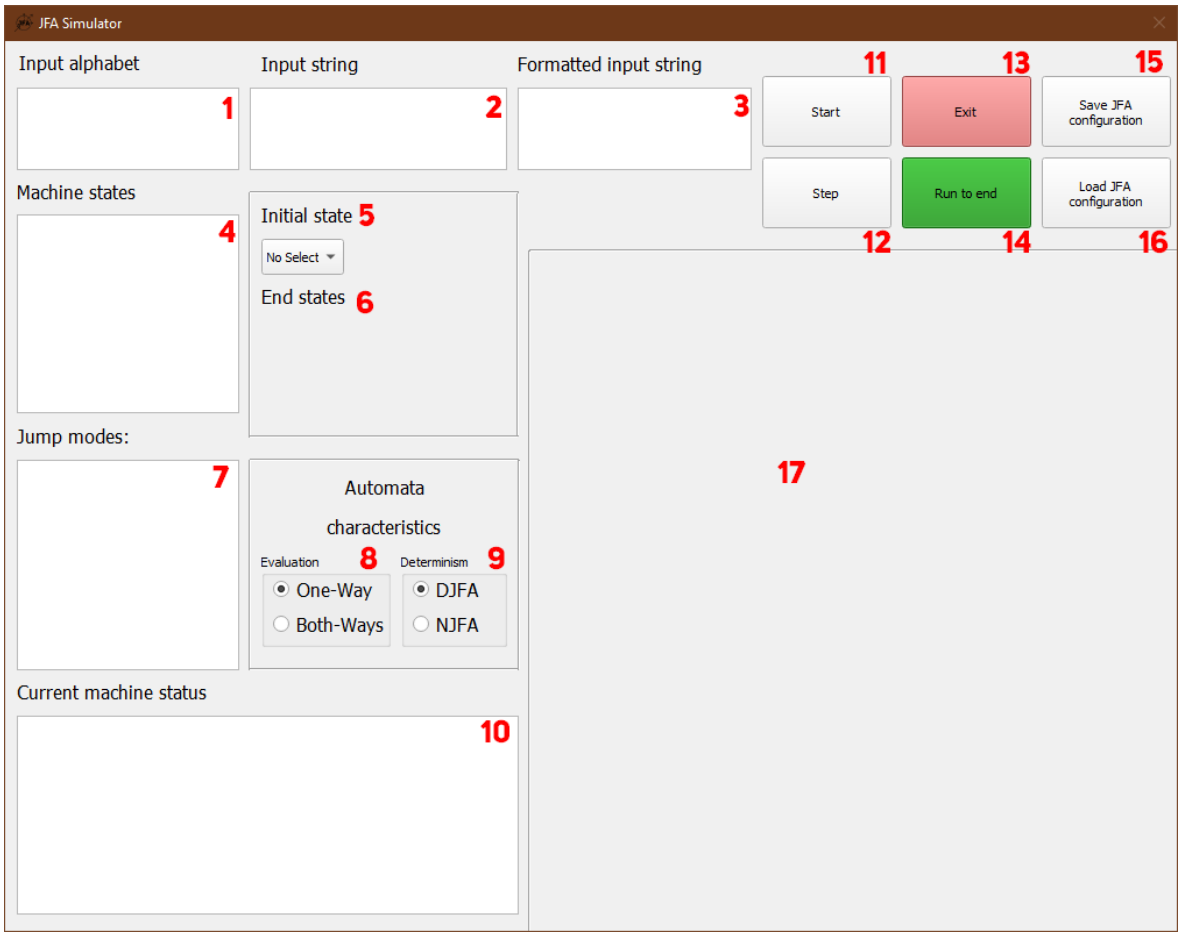
- **ctypes**
  - Knihovna pro interakci s dynamickými knihovnami a knihovnami napsanými v jazyce C/C++. Umožňuje volání funkcí, přístup k proměnným a tvoření datových typů v jazyce C/C++ v Pythonu.
  - V aplikaci je použita pouze pro editaci AppID za cílem zobrazení ikonky procesu v taskbaru
- **sys**
  - Knihovna pro přístup k systémově specifickým parametrům a funkcím (přístup do příkazové řádky, ukončení aplikace na úrovni procesu atd.).

- V aplikaci je použita pro správný běh PyQt5 frameworku a čisté ukončení aplikace.

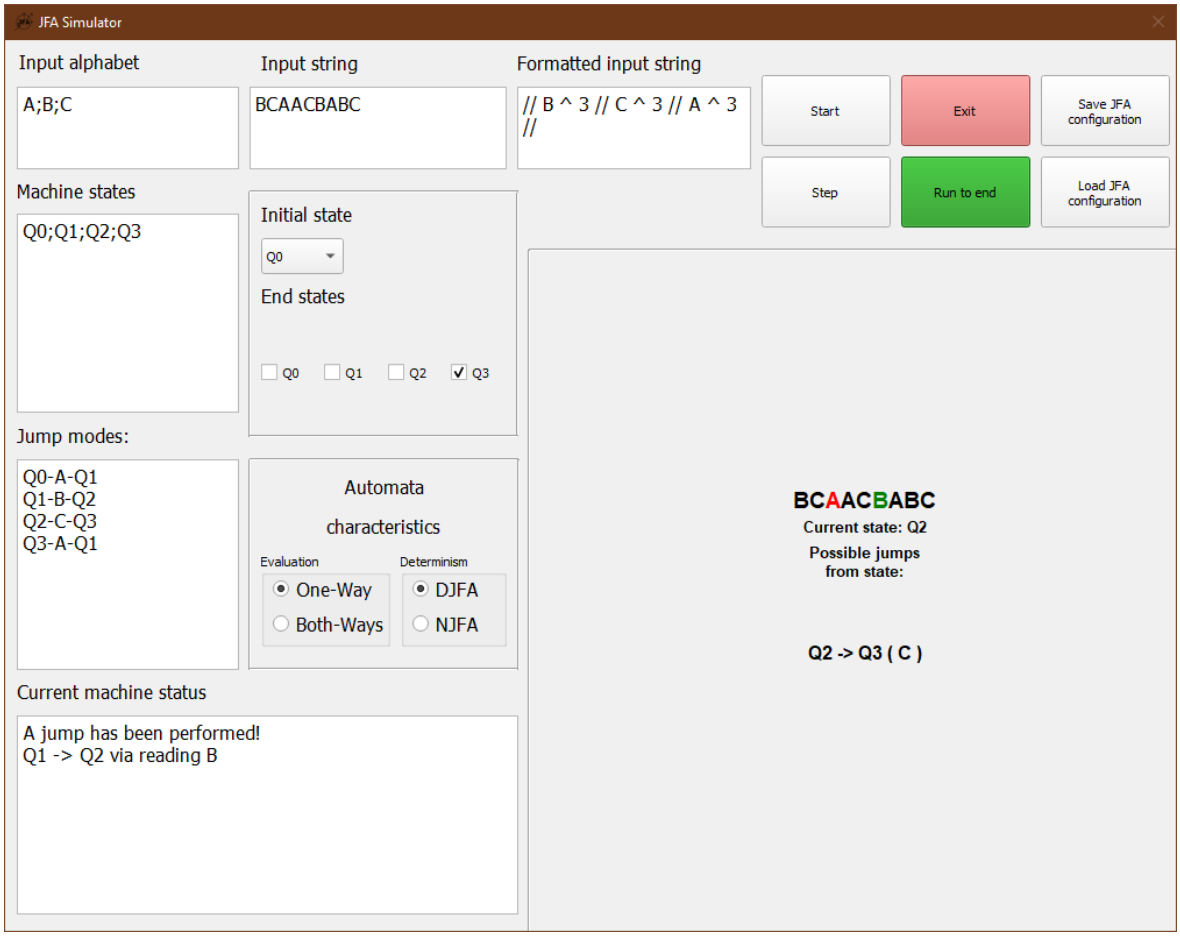
- **os**
  - Knihovna pro manipulaci s operačním a souborovým systémem (tvorba adresářů, výpis adresářů, tvorba procesů atd.)
  - V aplikaci použita pro získání cesty do složky, kde uživatel ukládá/načítá uložené konfigurace a případné vygenerování této složky, pokud ještě neexistuje.

3.2 Tvorba GUI

GUI aplikace slouží pro zadání konfigurace automatu do aplikace, výpis stavu automatu a provedených akcí, ovládání chodu automatu skrze ovládací prvky a zobrazení informací o současně vypočítávané instanci skákajícího konečného automatu. Každý element obsahuje textový *toolTip*, který se zobrazí, jakmile na elementu uživatel nechá kurzor.



Obrázek 7: Struktura GUI s číselným označením prvků



Obrázek 8: Vyplněné GUI v provozu

1 – „Input alphabet“

Vstupní textové pole určené pro zadávání vstupní abecedy skákajícího konečného automatu. Uživatel do tohoto pole zadává symboly, které bude skákající konečný automat číst.

2 – „Input string“

Vstupní textové pole určené pro zadávání vstupního řetězce, který bude skákající konečný automat zpracovávat. Uživatel do tohoto pole zadává řetězec jako posloupnost symbolů. Uživatel není v GUI omezován, jaké symboly může do textového pole napsat. Pokud budou v textovém poli při načítání předpisu automatu nevalidní symboly (tj. symboly, které nejsou obsažené ve vstupní abecedě) nebo jich zadá příliš moc, aplikace načítání předpisu zruší a uživatele upozorní chybovou hláškou a informacemi o chybě v textovém poli 10 - „Current machine status“.

3 – „Formatted input string“

Výstupní textové pole, do kterého se při načtení předpisu a provedení kroku simulace skákajícího konečného automatu přepíše zbývající nepřečtený vstupní řetězec převedený na zápis znázorňující počet výskytů symbolů. Převod je pouze ilustrační pro obousměrný skákající automat, jehož průběh není omezený pořadím symbolů v řetězci. Toto pole je nastaveno jako „read-only“ a uživatel do něj nemůže zapisovat (Obsah pole se automaticky aktualizuje během simulace automatu).

4 – „Machine states“

Vstupní textové pole určené pro zadávání stavů skákajícího konečného automatu. Uživatel do tohoto pole zadá stavy, mezi kterými bude skákající konečný automat přecházet.

5 – „Initial state“

Vstupní combobox, který slouží pro výběr počátečního stavu. Obsah tohoto elementu se dynamicky aktualizuje podle zápisu stavů automatu v poli 4 – „Machine states“ a vždycky obsahuje zástupnou položku s názvem „No Selection“. Tento zástupný symbol slouží k zachytávání prázdného výběru v back-end logice.

6 – „End states“

Vstupní layout element, do kterého se dynamicky přidávají a odebírají checkbox elementy reprezentující stavy v textovém poli 4 – „Machine states“. Tento layout a jeho obsah slouží pro výběr koncových stavů zaškrtnutím checkboxů těch stavů, které by měla aplikace považovat za koncové.

7 – „Jump modes“

Vstupní textové pole určené pro zadávání možných přechodů mezi stavy skákajícího konečného automatu. Zápis přechodů je závislý na zadaných stavech a zadané vstupní abecedě v polích 1 a 4. V případě, že se aplikace pokusí načíst uživatelem zadaný popis automatu a zjistí, že některý ze zadaných přechodů je nevalidní (např. přechod mezi stavy, které nebyly zadány nebo symbol, který nebyl zadán), načítání se přeruší a aplikace uživatele upozorní chybovou hláškou a informacemi o chybě v textovém poli 10 - „Current machine status“.

8 – „Evaluation selection“

Vstupní přepínače, které slouží pro výběr modelu skákajícího konečného automatu. Možnost „One-Way“ označuje model jednosměrného konečného automatu. Možnost „Both-Ways“ označuje model obousměrného konečného automatu. Výchozí stav tlačítek při zapnutí aplikace je takový, že je vybrána možnost „Both-Ways“. V případě že uživatel vybere možnost „One-Way“, přepínač NJFA se v páru přepínačů 9 – „Determinism selection“ vypne a může být vybráno pouze DJFA.

9 – „Determinism selection“

Vstupní přepínače, které slouží pro výběr chování skákajícího konečného automatu. Možnost „DJFA“ (**D**eterministic **J**umping **F**inite **A**utomata) označuje deterministické chování skákajícího konečného automatu, zatímco možnost „NJFA“ (**N**ondeterministic **JFA**) označuje nedeterministické chování skákajícího konečného automatu.

10 – „Current machine status“

Výstupní textové pole, do kterého se zapisují informace typu chybové hlášky z důvodu nevalidního předpisu skákajícího konečného automatu (symbol není v abecedě, není vybraný počáteční stav atd.) nebo informace ohledně simulace skákajícího konečného automatu (automat provedl přechod ze stavu *p* do stavu *q* atd.).

11 – „Start button“

Ovládací tlačítko, které reprezentuje načítání předpisu skákajícího konečného automatu ze vstupních elementů. Na tlačítko je napojené volání back-end logiky pro získání vstupních informací, jejich validaci a převod a vygenerování instance skákajícího konečného automatu.

12 – „Step button“

Ovládací tlačítko, které reprezentuje provedení kroku (jeden přechod automatu) v simulaci výpočtu skákajícího konečného automatu. Provedení kroku obsahuje zkontrolování možných přechodů současně načteného automatu, zkontrolování vstupního řetězce pro validní symbol pro provedení přechodu(ů), aktualizace informací v kontextu back-end logiky a aktualizace informací v GUI.

13 – „Exit button“

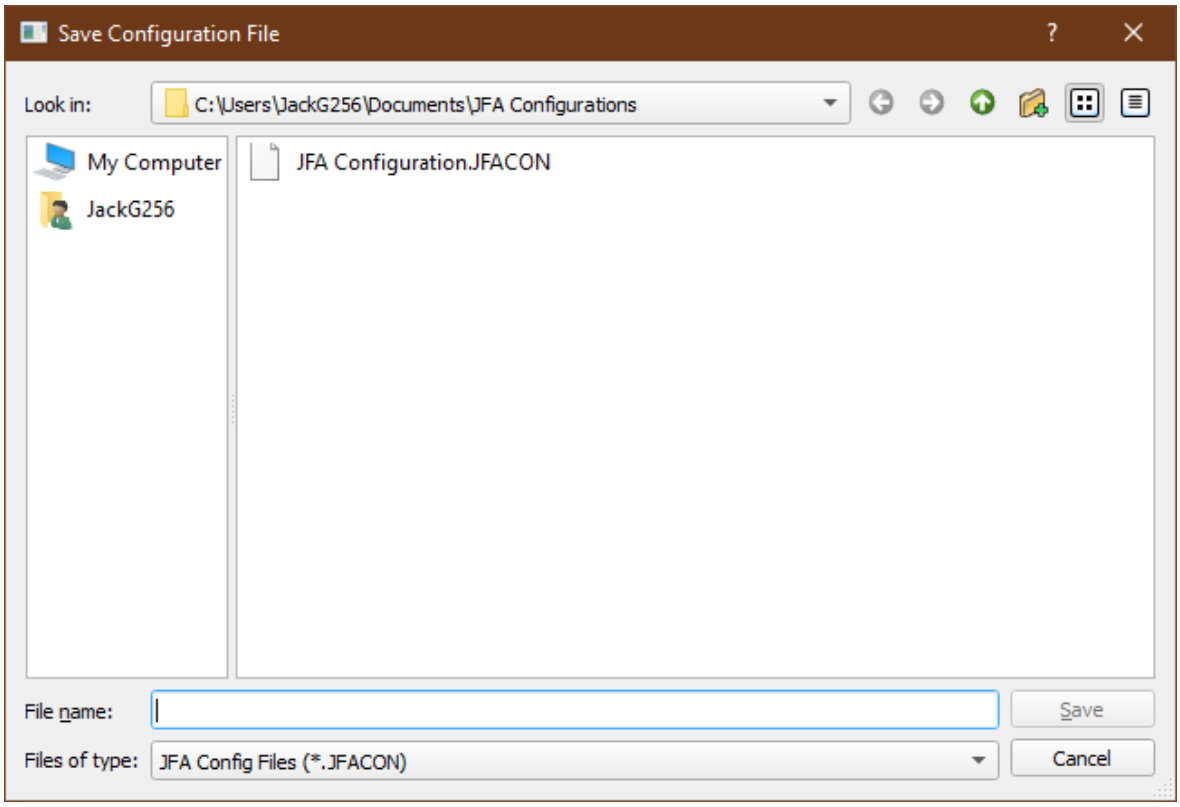
Ovládací tlačítko, které zavře aplikaci a ukončí její proces.

14 – „Run to end button“

Ovládací tlačítko, které opakovaně provádí stejné výpočty jako tlačítko „Step“ dokud se nedokončí výpočet skákajícího konečného automatu.

15 – „Configuration save button“

Ovládací tlačítko, které slouží k ukládání předpisu skákajícího konečného automatu z aplikace do souboru. Po zmáčknutí se objeví ukládací dialogové okno s přednastavenou „*.JFACON“ souborovou příponou. Dialogové okno se automaticky otevře ve vytvořené složce v dokumentech uživatele.



Obrázek 9: Dialogové okno pro ukládání

16 – „Configuration load button“

Ovládací tlačítko, které slouží k načtení předpisu skákajícího konečného automatu ze souboru do aplikace. Po zmáčknutí se objeví dialogové okno podobné ukládacímu dialogovému oknu. Dialogové okno se automaticky otevře ve vytvořené složce v dokumentech uživatele.

17 – „Automata instance layout“

Výstupní layout, do kterého se zapisují informace ohledně instance skákajícího konečného automatu, jako jsou zpracovávaný řetězec, současný stav, možné přechody z tohoto stavu atd. Informace se do layoutu poprvé zapíšou úspěšným načtením předpisu ze vstupních elementů a aktualizují se po každém kroku simulace.

3.3 Logika simulace a výpočtů

3.3.1 Hlavička projektu

```python
import ctypes
import sys
import os

from PyQt5 import uic
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QIcon, QFont
from PyQt5.QtWidgets import (
    QApplication,
    QMainWindow,
    QCheckBox,
    QLabel,
    QVBoxLayout,
    QWidget,
    QFileDialog,
)

from customExceptions import *
import preRun
import runLogicDET
import runLogicNDET
```

Úryvek kódu 1: Import knihoven

Zde provádím import knihoven pro práci s aplikací. První 3 řádky importují zabudované knihovny v pythonu, které používám pro manipulaci procesu aplikace, vlastnosti procesu v rámci. Co která knihovna dělá a k čemu je použita je popsáno v kapitole 3.1.

Na dalších řádcích se vybírají moduly knihovny PyQt5, což je knihovna vazeb pro Qt framework. Použité moduly jsou především:

- *uic*
    - Načítání a dynamická tvorba rozhraní aplikace.
- *from QtCore import Qt*
    - Zpracovávání událostí, signálů a mezi-objektová komunikace.
- *from QtGui import QIcon, QFont*
    - Třídy pro tvorbu a manipulaci s ikonkami a textovými styly.
- *QApplication*
    - Třída pro řízení řídícího toku aplikace. Inicializace aplikace a zpracovávání událostí.
- *QMainWindow*
    - Třída pro hlavní okno aplikace. Kontejner elementů aplikace a aplikačního rozhraní.
- *QCheckBox, QLabel*
    - Třídy pro checkbox a textový label element rozhraní.
- *QVBoxLayout*
    - Třída pro Vertical Layout element rozhraní. Tento element slouží primárně jako organizovaný kontejner pro jiné elementy.
- *QWidget*
    - Základní třída pro objekty grafického rozhraní.
- *QFileDialog*
    - Třída pro dialogové okno určené pro výběr cest a souborů v souborovém systému operačního systému.

Na posledních 4 řádcích úryvku jsou importy knihoven, které obsahují logiku aplikace za běhu.

- *customExceptions*
    - Třídy vlastní výjimek navržené pro vyvolávání a odchytávání problémových eventů za běhu aplikace (nevalidní vstup, automat nemá v současném stavu odchozí přechod atd.)
- *preRun*
    - Knihovna metod sloužící k filtraci a kontrole uživatelského vstupu ze vstupních polí v aplikačním okně. (převod textu na list hodnot, integrity checks atd.)
- *runLogicDET*
    - Knihovna metod výpočetní logiky a simulace kroků automatu v režimu deterministického chodu.
- *runLogicNDET*
    - Knihovna metod výpočetní logiky a simulace kroků automatu v režimu nedeterministického chodu.
- *runLogicBoth*
    - Pomocná knihovna metod pro modularizaci kódu, který nemusí být prováděn v kontextu hlavní aplikační třídy. (Převod textu na jiný formát, generace obsahu labelu atd.)

```
23 qtCreatorFile = "baseUI.ui"
24 helpUI = "helpWindow.ui"
25 Ui_MainWindow, QtBaseClass =
   uic.loadUiType(qtCreatorFile)
26
27 # Get options for file dialog
28 dialogOptions = QFileDialog.Options()
29 dialogOptions |= QFileDialog.DontUseNativeDialog
30
31 # Preset a path to the configs folder in documents
32 dialogDefaultDir =
   os.path.join(os.path.expanduser("~"), "Documents")
33 dialogDefaultDir = os.path.join(dialogDefaultDir,
   "JFA Configurations")
```

Úryvek kódu 2: Deklarace proměnných pro dialogová okna

V prvních řádcích této části definuji cestu k UI souboru, který aplikace bude načítat při spuštění[4]. V dalším řádku se cesta předává metodě z PyQt5 frameworku pro generaci řídící *UI_MainWindow* třídy, která reprezentuje hlavní okno grafického rozhraní a prostředí aplikace. Na dalších řádcích se volá metoda *QFileDialog.Options()*, která generuje sadu nastavení souborového dialogu předávaná třídě *QFileDialog* pro přizpůsobení chování. Mezi možná nastavení může patřit možnost omezení zobrazení jen na soubory pouze pro čtení, zobrazení pouze složek atd.

V kódu je akorát vynecháno nastavení *QFileDialog.DontUseNativeConfig*, které normálně vynucuje použití systémového dialogového okna. Aplikace v tomto stavu bude používat zabudované Qt dialogové okno. Toto rozhodnutí bylo čistě pro unifikaci designu aplikace s designem dialogového okna a nemá žádný dopad na funkcionalitu.

[4] „helpWindow.ui“ bylo experimentální provedení pomocných nápověd aplikace. Tento způsob jsme nevyužili a řádek nám v kódu zůstal.

Následující řádky definují cestu do složky určené pro ukládání a načítání souborů s automaty, které si může uživatel uložit. Tato složka se nachází ve složce dokumentech uživatele ve složce „JFA Configurations“

3.3.2 Základní struktura kódu

```
class MainAppWindow(QMainWindow, UI_MainWindow):
    """
    Runtime logic
    """

    def __init__(self):
      """
      Class constructor
      """


if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainAppWindow()
    window.show()
    sys.exit(app.exec_())
```

Úryvek kódu 3: Příklad nutného obalení aplikační logiky pro PyQt5 framework

Veškerá aplikační logika spojená s aplikací nebo grafickým rozhraním musí být prováděna v kontext třídy *MainAppWindow*, což omezuje modularizaci kódu. Z tohoto důvodu je valná většina logiky (která je vázaná na globální proměnné a vlajky) ponechána v souboru *main.py* a výpočetní logika nevázaná na tuto třídu modulovaná do specifických knihoven. Konstruktor této třídy obsahuje globální proměnné a vlajky použité napříč celou aplikací.

Na konci *main.py* je if větev, která pod podmínkou, že je skript spuštěn pouze jako hlavní program namísto importu jako knihovna, vytvoří *QApplication* objekt pro spuštění aplikace. Dále vytvoří instanci hlavního okna aplikace, zobrazí ho a spustí event smyčku, dokud aplikace běží. Tato smyčka čeká na event signál (zmáčknutí tlačítka, změna obsahu elementu atd.) a provádí kód na bázi tohoto eventu.

3.3.3 Inicializace aplikace a globální proměnné

```
561 def __init__(self):
562 
563     """
564     MainApp class constructor.
565     Sets all global variables and also edits process information.
566     """
567 
568     # Generators for application class
569     QMainWindow.__init__(self)
570     Ui_MainWindow.__init__(self)
571     self.setupUi(self)
572 
573     # Global flags and variables to run correct code
574     self.machineStarted = False
575     self.oneWay = True
576     self.deterministic = True
577 
578     # Global flags and variables purely for non-deterministic runtime
579     self.nonDetPathFound = False
580     self.nonDetPath = []
581     self.nonDetSymbols = []
582     self.nonDethIter = -1
583 
584     # Global variables regarding automaton
585     self.startState = ""
586     self.endStates = []
587     self.alphabet = ""
588     self.inputString = ""
589     self.inputStringFull = ""
590     self.machineStates = ""
591     self.jTransitions = ""
592     self.lastPos = 0
593     self.currentState = ""
594     self.prevInfo = []
595     self.formattedInputDict = []
596 
597     # Global list to keep track of other values
598     # List of read symbols to prevent multiple green symbols at one step
599     self.readSymbols = []
600     # List of end state checkboxes, used for correct deletion and generation
601     self.checkBoxList = []
602 
603         """
604         . . .
605 
606         """
```

Úryvek kódu 4: Konstruktor aplikace, část 1

Konstruktor aplikace je určen hlavně pro tvorbu globálních proměnných a vlajek a konstruktory tříd pro vytvoření uživatelského rozhraní (první 3 řádky).

Následující 3 řádky definují boolean proměnné, které slouží jako vlajky pro řízení automatu. Např. vlajka *oneWay* značí, jestli je vybrán režim jednosměrný nebo obousměrný. Hod-

noty těchto vlajek (kromě *machineStarted*) jsou zde pouze zástupné, jelikož jsou aktualizované podle uživatelského vstupu za běhu aplikace.

Další čtyři proměnné jsou dodatečné informace potřebné pro řízení chodu aplikace v nedeterministickém režimu.

- *nonDetPathFound*
    - Globální vlajka značící, jestli z předpisu nedeterministického automatu byla nalezena nějaká validní cesta z počátečního do koncového stavu (pod podmínkou přečtení správných symbolů ze vstupního řetězce).
- *nonDetPath*
    - List hodnot fungující jako kontext k nalezené přijatelné cestě v nedeterministickém režimu. Pokud aplikace v nedeterministickém předpise najde možnou validní cestu, uloží informace ohledně prováděných přechodů a jejich pořadí.
- *nonDetSymbols*
    - List hodnot podobný *nonDetPath*. Do tohoto listu se vkládá pořadí čtených symbolů pro možnou validní cestu nedeterministického automatu.
- *nonDetIter*
    - Pomocné počítadlo pro řízení správného výpisu informací o automatu v nedeterministickém režimu[5].

Dalších blok jsou proměnné, které se používají v kontextu všech režimů chodu (směry pohybu, oba typy determinismus atd). Některé proměnné jsou zde definované jako jeden datový typ ale později převedené do jiného, což je možné díky dynamické typové konverzi jazyka Python. Tato změna většinou proběhne při filtrování vstupních parametrů a jejich převodů, např vstupní abeceda zadána jako string, převedena na list hodnot pro jednodušší manipulaci. Pokud budu v dokumentaci mluvit o datovém typu proměnné, budu mluvit o tom, ve kterém se v kódu vyskytuje nejvíce často. Zde uvedu informace pouze o několika z nich, které by nemusely být na první pohled zřejmé.

[5] Nedeterministický běh automatu se v aplikace liší od deterministického způsobem výpočtu. Deterministický automat je počítán a vizualizován v reálném čase krok po kroku, zatímco nedeterministický automat může být mnohem komplexnější, a proto je vypočítáván pro jednu validní cestu hned jakmile aplikace získá platný zápis automatu. Jelikož zde nemůžeme vypočítávat informace přechod po přechodu, potřebujeme počítadlo, které nám bude udržovat správný chod.

- *inputString / inputStringFull*
    - *inputString* je string proměnná, do které se dynamicky ukládají změny za běhu automatu (odebírání přečtených znaků), *inputStringFull* je string proměnná sloužící jako reference na celý vstupní řetězec beze změn.
- *jTransitions*
    - Kompletní list všech přechodů, které má automat nastavené
- *lastPos*
    - Numerická integer proměnná, ve které se udržuje index naposledy přečteného symbolu ve vstupním řetězci. Udržovaná v kódu pro správný chod jednosměrného automatu a správný zápis znaků ve výstupním labelu reprezentující čtený vstupní řetězec.
- *prevInfo*
    - List hodnot, ve kterém se udržuje info o předchozím stavu, ve kterém se automat nacházel a symbolu, který přečetl během přechodu.

Další 2 řádky obsahují deklaraci pomocných listů pro udržování informací o kontextu aplikace a automatu

- *readSymbols*
    - Globální list, ve které se udržují informace o posledně čteném symbolu a jeho pozici. Hlavní použití tohoto listu je při vypisování vstupního řetězce, kde je pro správné formátování textu potřeba vědět, které znaky již byly přečteny.
- *checkBoxList*
    - List pro udržování instancí objektů *checkBox* ve výběru koncových stavů.

```
598     """
599     . . .
600
601     """
602
603     # Global variables for labels for instance layout
604     self.labelString = None
605     self.labelState = None
606     self.labelJumps = None
607     self.labelJumpsText = None
608
609     # Connect control action to corresponding control buttons
610     self.exitButton.clicked.connect(self.exitAction)
611     self.startButton.clicked.connect(self.startAction)
612     self.stepButton.clicked.connect(self.stepAction)
613     self.runToEndButton.clicked.connect(self.runToEndAction)
614
615     # Connect file management action to corresponding file buttons
616     self.SaveButton.clicked.connect(self.saveConfigAction)
617     self.LoadButton.clicked.connect(self.loadConfigAction)
618
619     # Connect radio buttons update action to corresponding radio buttons
620     self.BothWayRadioButton.clicked.connect(self.updateRadioButtons)
621     self.OneWayRadioButton.clicked.connect(self.updateRadioButtons)
622
623     # Connect load states action on text changed flag
624     self.machineStatesText.textChanged.connect(self.loadStatesAction)
625
626     # Default value in selection box on run
627     self.statesCombobox.addItem("No Selection")
628
629     # Set alternative style
630     app.setStyle("fusion")
631
632     # Disable default exit button
633     self.setWindowFlags(Qt.WindowTitleHint)
634
635     # Set custom title to title bar
636     self.setWindowTitle("JFA Simulator")
637
638     # Set Custom icon to app title bar
639     self.setWindowIcon(QIcon("Image_Content/JFA_icon.png"))
640
641     # Set custom icon to app in taskbar
642     # This is a workaround I found on stackoverflow
643     # Works via AppUserModelsIDs. Apart from that, I have no idea how it works
644     myappid = "JFA.Sim"
645     ctypes.windll.shell32.SetCurrentProcessExplicitAppUserModelID(myappid)
```

Úryvek kódu 5: Konstruktor aplikace, část 2

Následující úryvek konstruktoru už obsahuje pouze zástupné proměnné, propojení front-end elementů s aplikační logikou a nastavení aplikace.

Zástupné proměnné pro label elementy jsou deklarovány jako typ *None*, jelikož se během běhu aplikace budou přetypovávat na label objekty.

Propojení řídících tlačítek (Start, Step, Exit atd.) je provedeno napojením na jejich příslušnou metodu. Jakmile uživatel stiskne tlačítko, aplikace vytvoří event, který metodu zavolá. Provedení je aplikováno i pro tlačítka pro ukládání a načítání do/ze souboru.

Jiná metoda ale stejný princip je proveden i pro přepínače směru chodu automatu a determinismu, akorát napojené na metodu, která podle změny přepínačů vypne a vybere jiné prvky v aplikaci. Dále je textové pole pro zápis stavů automatu napojené na metodu, která generuje checkboxy a combobox hodnoty pro výběr počátečního a koncových stavů.

Další řádek obsahuje počáteční přidání hodnoty do comboboxu pro výběr počátečního stavu.

Zbytek konstruktoru obsahuje nastavení aplikace, aplikačního okna nebo procesu aplikace: změna stylu okna a elementů, vypnutí zavíracího tlačítka v aplikaci, nastavení názvu aplikace a ikonky a nastavení ikonky v taskbaru.

3.4 Aplikační logika

Výpočetní logika aplikace je podle vazeb k hlavní třídě aplikace většinou obsáhlá v souboru *main.py* a roztříděna do funkcí napojených na kontrolní tlačítka aplikace. Kvůli tomuto začnu nejdříve popisovat dodatečné knihovny, které jsme vytvořil.

3.4.1 customExceptions.py

Tato knihovna slouží jako úložiště pro vlastní vytvořené třídy výjimek. Výjimky jsou v kódu volány a chytány za účelem signalizace nevalidních vstupních dat nebo nemožnosti pokračovat ve výpočtu (např. neexistuje přechod v současném stavu). Jelikož je v metodách výpočet posloupný a závislý na předchozích výpočtech (např. získat abecedu → získat vstupní řetězec . . .), výjimky slouží jako způsob přerušení výpočtu a vytvoření chybové hlášky.

```python
class InvalidDeterministicFormat(Exception):
    """Provided jump transitions are invalid for deterministic approach"""

    def __init__(self, state, *args):
        super().__init__(args)
        self.state = state

    def __str__(self):
        return f"Provided configuration of jump transitions doesn't behave deterministically:" \
               f"<br>The state {self.state} has too many outward transitions of same symbol"

    pass
```

Úryvek kódu 6: Příklad vlastní výjimky

Všechny výjimky byly psány podobným způsobem, jelikož jejich hlavní účel je přerušení výpočtů a generace chybové hlášky. Ne každá výjimka obsahuje dodatečný parametr, jelikož v některých případech není potřeba. Kromě konstruktoru obsahují výjimky ještě metodu *__str__(self)* která je volána automaticky při vyvolání výjimky a je určena pro textové vyjádření chyby.

Výjimky, které mohou být vyvolány a odchyceny během běhu, včetně jejich účelu:

- *InvalidAlphabetFormatError*
  - Vstupní abeceda je napsána nevalidním způsobem
- *EmptyFieldError*
  - Jeden z povinných vstupních polí je prázdný.
  - Dává informaci, o které pole se jedná
- *InvalidDeterministicFormatError*
  - Definované přechody automatu nejsou deterministické
  - Dává informaci, který stav má příliš přechodů do jiných stavů
- *InputSymbolNotInAlphabetError*
  - Jeden symbol v zadaném vstupním řetězci není definován v abecedě
  - Dává informaci, o který symbol se jedná
- *StartStateNotFoundError*
  - Počáteční stav nebyl definovaný
- *EndStateNotFoundError*
  - Nebyl definovaný koncový stav/stavy
- *StateDoesNotExistError*
  - Stav v jednom z přechodů nebyl definovaný
  - Dává informaci, o který stav se jedná

- *SymbolDoesNotExistError*
    - Symbol v jednom z přechodů nebyl definovaný
    - Dává informaci, o který stav se jedná
- *NoJumpToPerformError*
    - Běžící automat nemá v současném stavu žádný přechod, který by mohl být proveden
    - Dá informaci, o který stav se jedná
- *NoAcceptPathFoundError*
    - Současný nedeterministický automat nemá žádné validní cesty z počátečního do koncového stavu
    - Informuje uživatele, že řetězec je automaticky odmítnut
- *InputStringTooLongError*
    - Uživatel zadal příliš dlouhý vstupní řetězec (více než deset symbolů)
    - Informuje uživatele o délce vstupního řetězce
- *PathNoTransitionProvided*
    - Stav důležitý pro nedeterministický chod nemá specifikovaný přechod
    - Informuje uživatele o automatickém odmítnutí řetězce

3.4.2 preRun.py

Tato knihovna obsahuje logiku pro filtrování a převádění uživatelského vstupu. V této knihovně jsou nejvíce využívány výjimky pro přerušení výpočtu.

```python
from customExceptions import *


def filterMachineAlphabet(unfiltered):

    # Split string by division symbol
    unfiltered = unfiltered.split(";")

    # Check if list is not empty
    if len(unfiltered) == 1 and unfiltered[0] == "":
        raise EmptyFieldError("Input alphabet field")

    # Failsafe to remove last empty field if user ended string with ';'
    for entry in unfiltered:
        if len(entry) == 0:
            unfiltered.remove(entry)

    # Check if alphabet is inputted one symbol at a time
    # Check if all symbols are valid letters
    for symbol in unfiltered:
        if len(symbol) != 1:
            raise InvalidAlphabetFormatError(symbol)

        if not symbol.isalpha() and not symbol.isnumeric():
            raise InvalidSymbolInAlphabetError(symbol)

    return unfiltered
```

Úryvek kódu 7: Funkce na filtrování vstupní abecedy automatu

Funkce *filterMachineAlphabet()* slouží k filtrování uživatelského vstupu vstupní abecedy. Funkce vrací list symbolů, které automat bude rozeznávat. Funkce zároveň kontroluje, jestli není vstup prázdný a jestli je validní, tzn. jestli je každý symbol oddělený středníkem, jestli mezi oddělovacími středníky je pouze jeden symbol a jestli znaky skutečně jsou písmena abecedy.

```python
def filterInputString(unfiltered, alphabet):

    # Limit string size for UI and complexity reasons
    if len(unfiltered) > 10:
        raise InputStringTooLongError(len(unfiltered))

    # Split string into char list
    unfiltered = [*unfiltered]

    # Check if list is not empty
    # Different from before, conversion not via split leaves 0 entries,
    # split() conversion leaves at least 1 empty entry
    if len(unfiltered) == 0:
        raise EmptyFieldError("Input string field")

    # Check if all symbols contained in alphabet
    for symbol in unfiltered:
        if symbol not in alphabet:
            raise InputSymbolNotInAlphabetError(symbol, alphabet)

    # Create a dictionary to store number of occurences in string
    formattedDict = {}

    # Create a dict entry for each unique symbol with
    # the value of number of occurences
    for unique in sorted(list(dict.fromkeys(unfiltered))):
        formattedDict[unique] = unfiltered.count(unique)

    return unfiltered, formattedDict
```

Úryvek kódu 8: Funkce na filtrování vstupního řetězce automatu

Funkce *filterInputString()* bere kromě vstupní hodnoty z textového pole v GUI také již filtrovaný list abecedy. Funkce provádí podobné kontroly jako *filterMachineAlphabet()* ale kontroluje i délku vstupu a jestli je symbol definovaný v abecedě. Omezení délky zde bylo implementováno z důvodu snížení komplexity a doby výpočtu možné přijaté cesty v případě nedeterministického chodu[6]. Funkce později vytvoří dictionary s počtem výskytů každého symbolu.

[6] Při pěti definovaných přechodech a patnácti zadaných symbolech vstupního řetězce se doba výpočtu validní cesty nedeterministického automatu zvýší až na 1,5 vteřiny s exponenciálním růstem při zvýšení délky řetězce a množství přechodů. Deset symbolů je z výpočetního hlediska dostatečný rozsah pro uživatele, aniž by se zhoršil výkon aplikace.

```python
def filterMachineStates(unfiltered):

    # Split string by division symbol
    unfiltered = unfiltered.split(";")

    # Check if list is not empty
    if len(unfiltered) == 1 and unfiltered[0] == "":
        raise EmptyFieldError("Machine states field")

    # Failsafe to remove all empty fields'
    for entry in unfiltered:
        if len(entry) == 0:
            unfiltered.remove(entry)

    return unfiltered
```

Úryvek kódu 9: Funkce na filtrování stavů automatu

Funkce *filterMachineStates()* pouze převádí zadaný uživatelský vstup na list hodnot, které reprezentují stavy. Kromě kontroly prázdného textového pole zde nejsou implementované žádné kontroly a výjimky. Tímto způsobem si může uživatel deklarovat názvy stavů v libovolném formátu a délce.

```python
def filterJumpTransitions(unfiltered, alphabet, machineStates, deterministic):

    # Remove redundant whitespaces, split by
    # new line and split each separate line by separator characters
    # after striping potential trailing newline symbol
    filteredJumpEntriesList = []
    for entry in unfiltered.replace(" ", "").rstrip().split("\n"):
        filteredJumpEntriesList.append(entry.split("-"))

    # Test checks if information in jump transition is valid
    for entry in filteredJumpEntriesList:
        if entry[0] not in machineStates:
            raise StateDoesNotExistError(entry[0])
        if entry[2] not in machineStates:
            raise StateDoesNotExistError(entry[2])
        if entry[1] not in alphabet:
            raise SymbolDoesNotExistError(entry[1])

    # If the automaton runs nondeterministically, return
    if not deterministic:
        return filteredJumpEntriesList

    # Looks through all filtered transitions
    jumpsByOriginSymbol = []
    for entry in filteredJumpEntriesList:
        # if transition in format initial state - symbol already
        # exists in list of checked transitions, raise error
        if entry[0] in jumpsByOriginSymbol:
            raise InvalidDeterministicFormatError(entry[0])
        # Add entry to list of checked transitions
        jumpsByOriginSymbol.append(entry[0])

    return filteredJumpEntriesList
```

Úryvek kódu 10: Funkce na filtrování přechodů automatu

Funkce *filterJumpTransitions()* kromě obsahu textového pole přijímá ještě abecedu, seznam stavů a vlajku, která signalizuje jestli automat je nebo není nastaven na deterministický chod. Po rozdělení vstupního textu na zápisy přechodů, které jsou ve formátu “<stav> - <symbol> - <stav>“, se zkontroluje, jestli byl každý stav a symbol v přechodech definován v automatu. Pokud je nastavený deterministický chod, funkce projde všechny přechody a zkontroluje, že každý stav má pouze jeden odchozí přechod do jiného stavu. Pokud se zjistí více odchozích přechodů z jednoho stavu, vyvolá se výjimka.

Pokud je ale nastavený nedeterministický chod, tato kontrola se neprovádí.

3.4.3 RunLogicBoth

Tato knihovna obsahuje logiku, která nemusí být vykonávána v kontextu hlavní třídy aplikace. Jsou zde obsaženy hlavně metody pro převod textu do jiného formátu nebo nalezení informací, které se později budou vypisovat uživateli (Výpis počtu výskytů symbolů, nalezení možných přechodů ze současného stavu a převedení do text atd.)

```
1  def generateFormattedInputDictionary(inputDict):
2
3      # For each key in formatted input string dictionary,
4      # put key in output string, and get occurence
5      # value based on index of same key.
6
7      formattedInputStr = ""
8      for key in inputDict:
9          formattedInputStr += f" {key} ^ {inputDict[key]} "
10         formattedInputStr += f"//"
11
12     # Cut last 2 dividing symbols
13
14     formattedInputStr = formattedInputStr[:-2]
15     return formattedInputStr
```

Úryvek kódu 11: Funkce pro převod počtu výskytu symbolů na string

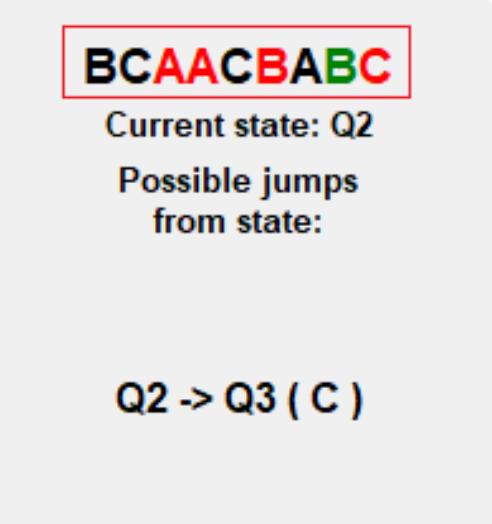
Funkce *generateFormattedInputDictionary()* převede slovník výskytů symbolů na string s hodnotami reprezentující počet daného symbolu v řetězci. Funkce je volána při načítání zápisu automatu a při každém kroku automatu.

```python
def createFormattedStringLabel(inputStringFull, readSymbols, lastPos):

    # Initialize/reinitialize the output string to put in label
    labelString = ""

    # Helping flag to prevent multiple green symbols
    markedGreen = False

    # Iterate over each symbol in the input string
    for i, symbol in enumerate(inputStringFull):
        # Check if the current symbol was just read by the JFA
        if [symbol, i] in readSymbols and i == lastPos and not markedGreen:
            # If the current symbol was just read, format it with green color,
            # then temporarily save the writen symbol and it's possition. (now obsolete)
            labelString += f"<span style='color:green'>{symbol}</span>"
            # symbolToUpdate = [symbol, i]

            # Flip the bool value to prevent multiple green symbols
            markedGreen = True

            # After that, skip this iteration
            continue

        # Check if the symbol has been read before by the JFA before
        if [symbol, i] in readSymbols:
            # If the symbol has been read before, format it with red color
            labelString += f"<span style='color:red'>{symbol}</span>"
        else:
            # If the symbol has not been read before, format it with black color
            labelString += f"<span style='color:black'>{symbol}</span>"

    return labelString
```

Úryvek kódu 12: Funkce pro generování obsahu labelu řetězce

Funkce *createFormattedStringLabel()* generuje HTML formátovaný obsah pro label řetězce, který se zobrazuje v instanci automatu. Při načtení zápisu automatu je celý label vybarven černou barvou a každý krok automatu aktualizován touto funkcí. Barvy jsou aplikovány skrze HTML span elementy s přednastavenými styly.

- Symbol, který byl právě přečtený je zapsán se zelenou barvou
    - Symbol je zapsán v listu přečtených symbolů, shoduje se pozice symbolu v řetězci a uložená pozice posledního čteného symbolu, a ještě nebyl zapsaný jiný symbol zelenou barvou
- Symbol, který byl již někdy přečtený je zapsán červenou barvou
    - Symbol je zapsán v listu přečtených symbolů
- Ostatní symboly jsou zapsány s černou barvou



Obrázek 10: Příklad labelu řetězce

```
48 def returnRedInputString(inputStringFull):
49     labelString = ""
50     for symbol in inputStringFull:
51         labelString += f"<span style='color:red'>{symbol}</span>"
52 
53     return labelString
```

Úryvek kódu 13: Funkce pro generování obsahu labelu řetězce při ukončení výpočtu

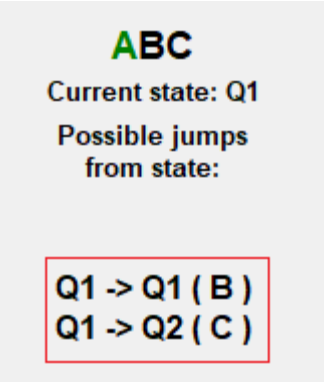
Funkce *returnRedInputString()* vrací vstupní řetězec v podobném formátu jako funkce *createFormattedStringLabel()* ale nastavuje všechny symboly na červenou barvu. Tato funkce je využitá hlavně na konci výpočtu, když už byly přečtené všechny symboly řetězce[7].

[7] Bez této funkce by na konci výpočtu po přečtení všech symbolů zůstal zobrazený jeden symbol zelenou barvou. Toto by mohlo uživatele potenciálně zmást, a proto je zde přidána tato funkce

```python
def findNextJumps(jTransitions, currentState, inputString):

    output = ""
    maxText = 5

    for entry in jTransitions:
        if entry[0] == currentState and entry[1] in inputString:
            output += f"{entry[0]} -> {entry[2]} ( {entry[1]} )\n"
            maxText -= 1
        if not maxText:
            break

    return output
```

Úryvek kódu 14: Funkce pro generování obsahu labelu možných skoků

Funkce *findNextJumps()* generuje informace ohledně dalších možných skoků ze současného stavu automatu. Text se zapíše do instance automatu.



Obrázek 11: Příklad labelu dalších možných skoků

3.4.4 RunLogicDET

Tato knihovna obsahuje výpočetní logiku pro simulaci v deterministickém chodu. Funkce jsou dělané na výpočet přechodů automatu v reálném čase pokaždé, když uživatel zmáčkne tlačítko „Step“. Tato knihovna importuje jednu výjimku z knihovny *customExceptions*, a to *NoJumpToPerform.*

```python
def findAndRunJumpOneSide(jTransitions, currentState, inputDict, inputString, lastPos):

    # Variable declaration
    listOfEndpoints = []

    # Go through all transitions
    # Save all relative to intial and target state
    for entry in jTransitions:
        if entry[0] == currentState and entry[1] in inputString:
            listOfEndpoints.append(entry)

    # If there are no transitions, throw custom error
    # NOTE: length of listOfEndpoints should never
    # go above 1, prevented by sanity checks
    if len(listOfEndpoints) != 1:
        raise NoJumpToPerformError(currentState)

    # Get currently read symbol from transition
    currentReadSymbol = listOfEndpoints[0][1]

    # Update formatted input dictionary
    if inputDict[currentReadSymbol] > 1:
        inputDict[currentReadSymbol] -= 1
    else:
        inputDict[currentReadSymbol] = 0

    # Start searching throgh input string for symbol
    outputString1 = ""
    outputString2 = ""
    symbolPosition = lastPos
    readSymbolIndex = -1
    symbolReached = False

    # First loop, searches from last read symbol to end
    for symbol in inputString[lastPos:]:
        # if empty, write and continue
        if symbol == "_":
            outputString2 += "_"
            symbolPosition += 1

        # if first found, save index, flip flag, write empty and continue
        elif symbol == currentReadSymbol and not symbolReached:
            symbolReached = True
            outputString2 += "_"
            readSymbolIndex = symbolPosition

        # otherwise write and continue
        else:
            outputString2 += symbol
            symbolPosition += 1

    # Second loop, searches from first to last read symbol - 1
    symbolPosition = 0
    for symbol in inputString[:lastPos]:
        # if empty, write and continue
        if symbol == "_":
            outputString1 += "_"
            symbolPosition += 1
            continue

        # if first found, save index, flip flag, write empty and continue
        if symbol == currentReadSymbol and not symbolReached:
            symbolReached = True
            outputString1 += "_"
            readSymbolIndex = symbolPosition
            continue

        # otherwise write and continue
        else:
            outputString1 += symbol
            symbolPosition += 1

    # Join strings together
    # Save information to a tuple and return
    outputString = outputString1+outputString2
    previousInfo = [listOfEndpoints[0][0], listOfEndpoints[0][1]]
    currentState = listOfEndpoints[0][2]

    return inputDict, outputString, currentState, previousInfo, readSymbolIndex
```

Úryvek kódu 15: Funkce pro výpočet jednosměrného kroku

Funkce *findAndRunJumpOneSide()* hledá na základě vstupních parametrů možný přechod ze současného stavu do jiného. V případě, že přechod nenajde nebo přechod najde ale není pro něj dostupný symbol, vyvolá *NoJumpToPerform* výjimku. Pokud přechod pro dostupný symbol najde, začne procházet řetězec od posledního čteného symbolu doprava, dokud symbol nenajde. Jakmile symbol najde, nahradí ho za zástupný prázdný symbol[8] a zkopíruje zbytek. Zároveň aktualizuje slovník výskytů symbolů a nastaví návratové hodnoty (previous info je tuple hodnot reprezentující předchozí stav a předchozí symbol).

[8] Jelikož v aplikaci pracujeme s pozicí symbolů v řetězci, nemůžeme z řetězce symboly pouze odebírat. Tento zástupný prázdný symbol je v aplikaci považován za prázdné pole, které by se mělo automaticky přeskočit.

```python
def findAndRunJumpBothSides(jTransitions, currentState, inputDict, inputString):

    # Variable declaration
    listOfEndpoints = []

    # Go through all transitions
    # Save all relative to intial and target state
    for entry in jTransitions:
        if entry[0] == currentState and entry[1] in inputDict:
            listOfEndpoints.append(entry)

    # If there are no transitions, throw custom error
    # NOTE: length of listOfEndpoints should never go above 1, prevented by sanity checks
    if len(listOfEndpoints) != 1:
        raise NoJumpToPerformError(currentState)

    # Save information to a tuple
    previousInfo = [listOfEndpoints[0][0], listOfEndpoints[0][1]]
    currentState = listOfEndpoints[0][2]
    # Get currently read symbol
    currentReadSymbol = listOfEndpoints[0][1]

    # Update formatted input dictionary
    if inputDict[currentReadSymbol] > 1:
        inputDict[currentReadSymbol] -= 1
    else:
        inputDict[currentReadSymbol] = 0

    # Start searching throught input string for symbol
    # Looks always left to right
    outputString = ""
    symbolPosition = 0
    readSymbolIndex = -1
    symbolReached = False
    # Go through all symbols in input string
    for symbol in inputString:
        # if empty, write and continue
        if symbol == "_":
            outputString += "_"
            symbolPosition += 1

        # if first found, save index, flip flag, write empty and continue
        elif symbol == currentReadSymbol and not symbolReached:
            symbolReached = True
            outputString += "_"
            readSymbolIndex = symbolPosition

        # otherwise write and continue
        else:
            outputString += symbol
            symbolPosition += 1

    return inputDict, outputString, currentState, previousInfo, readSymbolIndex
```

Úryvek kódu 16: Funkce pro výpočet obousměrného kroku

Funkce *findAndJumpBothSides()* funguje podobně jako funkce *findAndJumpOneSide()* ale liší se ve vyhledávání znaků v řetězci. Provádí stejné kontroly existence přechodů pro stav

a existence symbolů v řetězci pro přechod ale hledání symbolu v řetězci provádí pokaždé z nejlevějšího symbolu doprava[9].

3.4.5 RunLogicNDET

Tato knihovna obsahuje výpočetní logiku pro simulaci v nedeterministickém chodu. Oproti funkcím pro deterministický chod, funkce v této knihovně slouží primárně k nalezení a výpočtu jedné validní cesty z počátečního stavu do koncového stavu pro zápis nedeterministického automatu a jeho přechodů. Výpočty validních cesty se dělají skrze výpočet matice sousednosti (zjištění existence cesty z počátečního stavu do koncového stavu) a nalezení cesty pomocí přechodů automatu jako orientovaný graf (posloupnost přechodů ze stavů do stavů a posloupnost čtených symbolů).

Jelikož zde není potřeba každý krok iterovat, funkce jsou volány pouze při načtení zápisu automatu. Ze zjištěných informací (cesta a symboly) jsou poté iteračně aktualizovány labely instance automatu.

[9] Obousměrný skákající automat není limitovaný čtením pouze jedním směrem, proto se dá vstupní řetězec vyjádřit čistě jako množina výskytu symbolů. Procházení je zde kvůli vizualizaci v instanci automatu.

```python
import numpy as np

def createInitialMatrix(statePaths):

    # Helping variable to store list of active states
    loadedStates = []

    # Find all active states based on Jump Transitions
    for entry in statePaths:
        if entry[0] not in loadedStates:
            loadedStates.append(entry[0])
        if entry[2] not in loadedStates:
            loadedStates.append(entry[2])

    # Sort list alphabetically for easier evaluation
    loadedStates = sorted(loadedStates)

    # Get number of items in list
    xySize = len(loadedStates)

    # Create a 2D matrix with zeroes to store data into
    initialMatrix = [[0 for _ in range(xySize)] for _ in range(xySize)]

    # Iterate through active states and fill adjacency matrix if paths exist
    for i, stateY in enumerate(loadedStates):
        for j, stateX in enumerate(loadedStates):
            # Compare every pair of states with transition entries
            for entry in statePaths:
                # If checks, write a value to adjacency
                # matrix and save transition for later use
                if stateY == entry[0] and stateX == entry[2]:
                    initialMatrix[i][j] += 1
    return initialMatrix, loadedStates
```

Úryvek kódu 17: Funkce pro vytvoření matice sousednosti automatu

Funkce *createIntialMatrix()* slouží k vytvoření matice sousednosti stavů automatu podle zadaných přechodů. Velikost matice se určuje podle stavů v přechodech automatu. Dále se v cyklu prochází každá možná kombinace stavů a kontroluje se, jestli je tato kombinace obsažena v přechodech. Pokud ano, značí to existenci cesty ze stavu do stavu a do matice na pozici určenou stavy se zvedne číslo o jedna.

Funkce dodatečně ukládá, které stavy jsou aktivně používány a list vrací. Tento list se později využívá pro kontrolu mrtvých větví a nedosažitelných stavů automatu.

```python
def findPath(transitions, startState, endState, moves, path=None, symbols=None):
    # Initialize the path and symbols lists if they are not provided
    if path is None:
        path = [startState]
    if symbols is None:
        symbols = []

    # If no moves left, check if we have arrived at the end state
    if moves == 0:
        if startState == endState:
            # If the end state has been reached,
            # return the path and symbols as a single-element list
            return [[path, symbols]]
        else:
            # If the end state has not been reached, return an empty list
            return []

    # If there is only one move left
    elif moves == 1:
        paths = []
        for stateX, symbol, stateY in transitions:
            if stateX == startState and stateY == endState:
                # If the edge connects the start and end states,
                # add the new state and symbol to the path
                new_path = path + [endState]
                new_symbols = symbols + [symbol]

                # Add the path to the list of valid paths
                paths.append([new_path, new_symbols])

        # Return the list of valid paths
        return paths

    # If there are more than one moves left
    else:
        paths = []
        for stateX, symbol, stateY in transitions:
            if stateX == startState:
                # Generate a new path from state to state via transition
                new_path = path + [stateY]

                # Add symbol read of transition
                new_symbols = symbols + [symbol]

                # Recursively call findPath on the neighbor
                # of the start state with moves decremented by 1
                sub_paths = findPath(transitions, stateY, endState,
                                     moves - 1, new_path, new_symbols)

                # Add each valid sub-path to the list of valid paths
                for sub_path in sub_paths:
                paths.append(sub_path)

        # Return the list of valid paths
        return paths
```

Úryvek kódu 18: Funkce pro nalezení cesty ze stavu do stavu v n krocích

Funkce *findPath()* slouží pro nalezení cesty v přechodech automatu vyjádřenými jako orientovaný graf s jednosměrnými hranami. Funkce rekurzivně prochází všechny možné pře-

chody od počátečního stavu za cílem nalezení možné cesty do stavu koncového ve specifickém počtu kroků[10]. Cesty jsou postupně ukládány jako dvou buňkový list, kde první buňka listu obsahuje posloupnost stavů, ve kterém se automat nacházel a druhá buňka obsahuje posloupnost symbolů, které automat četl a zpracovával.

V případě, že se funkce v některém zanoření nedostane do koncového stavu v povoleném počtu kroků, funkce vrátí prázdný list. Po ukončení chodu vrací funkce list možných cest, kde každá buňka listu reprezentuje jednu nalezenou cestu a obsahuje 2 vnitřní buňky, tj. posloupnost stavů a posloupnost symbolů.

[10] Počtem kroků se zde rozumí počet symbolů, které může automat zpracovat, než celý řetězec vyčerpá

```python
def generateAdjMatrixAndPath(jTransitions, iterationMax, inputString, startState, endStates):

    # Create an adjacency matrix of first rank
    # and sorted list of active states (used as coordinates)
    initialMatrix, loadedStates = createInitialMatrix(jTransitions)

    # Transform nested list into numpy matrix for easier calculation
    nextMatrix = np.array(initialMatrix)

    # Raise adjacency matrix to the power of
    # number of symbols in input string
    nextMatrix = np.linalg.matrix_power(nextMatrix, iterationMax)

    # Check if start state wasn't ommited in matrix generation
    if startState not in loadedStates:
        raise PathNoTransitionProvided(startState)

    # Get coordinate of initial state
    startStateCoord = loadedStates.index(startState)

    # Get all end states considered by the adjacency matrix
    activeEndStates = []
    for state in endStates:
        if state in loadedStates:
            activeEndStates.append(state)

    # Get coordinates of end states
    endStatesCoords = [loadedStates.index(endstate) for endstate in activeEndStates]

    returnPath = None
    validPathFound = False
    # Iteratively check over all coordinates if value of cell is greater than 1
    # (a path in n moves exist from start state to end state exists)
    for xCoord in endStatesCoords:
        if validPathFound:
            break

        if nextMatrix[startStateCoord, xCoord] >= 1:
            # Try to find first accepting path
            path = findPath(jTransitions, startState, loadedStates[xCoord], iterationMax)

            # if path exists, check if symbols read
            # along the way match up symbols in input string
            if path is not None:
                for i, entry in enumerate(path):

                    readSymbols = ''.join(sorted(entry[1]))
                    sortedString = ''.join(sorted(inputString))
                    # If yes, break and return
                    if readSymbols == sortedString:
                        returnPath = [entry[0], entry[1]]
                        validPathFound = True
                        break

    return returnPath
```

Úryvek kódu 19: Funkce pro plný výpočet nedeterministického chodu automatu

Funkce *generateAdjMatrixAndPath()* slouží pro ucelení výpočtů matice sousednosti a nalezení cesty nedeterministického automatu do jednoho celku, nalezení jedné validní cesty z nalezených cest a vrácení potřebných informací do kontextu třídy aplikace.

Funkce zavolá generaci matice sousednosti na základě přechodů. Tuto matici poté převede na numpy matici a matici umocní na n-tou mocninu, kde n představuje počet symbolů

v řetězci[11]. Funkce následně zkontroluje, jestli byl počáteční stav vypočítáván v matici sousednosti. Pokud ne, vyvolá výjimku, jelikož automaticky neexistuje žádná validní cesta, jinak zjistí index (pozici v matici) stavu. Podobná kontrola se provádí i pro koncové stavy ale bez volání výjimky a ukládání stavů, které byly použité ve výpočtu matice sousednosti. Po zjištění těchto stavů se zjistí a uloží jejich souřadnice v setříděném listu z předchozí funkce. Funkce následovně zkontroluje buňky v matici sousednosti podle souřadnic počátečního a koncových stavů. Pokud buňka na této pozici má hodnotu větší než nula, existuje cesta z počátečního stavu do tohoto koncového stavu.

Funkce zavolá *findPath()* funkci pro nalezení všech možných cest z počátečního stavu do koncového stavu v n krocích a začne procházet všechny možné cesty, tj. funkce seřadí vstupní řetězec a posloupnost symbolů z dané cesty. Pokud jsou řetězce rozdílné, současně kontrolovaná cesta četla symboly, které nejsou dostupné v řetězci (například cesta přečetla jeden symbol B navíc namísto symbolu A atd.).

Pokud se řetězce rovnají, cesta je validní, funkce uloží pár informací (posloupnost stavů a posloupnost symbolů), vyskočí z cyklu procházející buňky matice a vrátí uložené informace.

[11] Matice na takovém exponentu představuje matici sousednosti pro n kroků. Tímto zjišťujeme, jestli existuje cesta ze stavu a do stavu b po přečtení všech symbolů v řetězci, tj. provedení přechodů pro každý symbol v řetězci.

```python
def findNextSymbolPosition(currentReadSymbol, inputString, inputDict):

    # Variables declaration
    outputString = ""
    symbolPosition = 0
    readSymbolIndex = -1
    symbolReached = False

    # Go through all symbols in input string
    for symbol in inputString:
        # If symbol is empty, write to output and continue
        if symbol == "_":
            outputString += "_"
            symbolPosition += 1

        # If symbol matches current symbol for first time,
        # replace by empty, save index, flip flag and continue
        elif symbol == currentReadSymbol and not symbolReached:
            symbolReached = True
            outputString += "_"
            readSymbolIndex = symbolPosition
            continue

        # Otherwise write to output and continue
        else:
            outputString += symbol
            symbolPosition += 1

    # Check if value in dictionary and update
    if inputDict[currentReadSymbol] > 1:
        inputDict[currentReadSymbol] -= 1
    else:
        inputDict[currentReadSymbol] = 0

    return outputString, readSymbolIndex, inputDict

```

Úryvek kódu 20: Pomocná funkce pro nalezení pozice symbolu v řetězci

Funkce *findNextSymbolPosition()* je pomocná funkce pro nalezení indexu specifického ještě nepřečteného symbolu v řetězci. Výpočet probíhá úplně stejně jako funkce pro provedení kroku v obousměrném deterministickém chodu.

3.4.6 main.py

Hlavní soubor, ve kterém probíhá většina aplikační logiky (komunikace s GUI, volání funkcí pro simulaci kroků automatu atd.) Kód je rozdělen do funkcí, které jsou napojené a volané při stisknutí kontrolních tlačítek v GUI. Tyto funkce jsou zahrnuty v třídě *MainAppWindow*, jak bylo popsáno v kapitole 3.3.2 této práce.

```python
class MainAppWindow(QMainWindow, Ui_MainWindow):

    def saveConfigAction(self):

        """
        Method called for saving JFA context to a file.
        Saves important global variables to a custom .JFACON file.
        """

    def loadConfigAction(self):

        """
        Method called for loading JFA context from a file.
        Loads and updates important global variables based on content
        from a custom .JFACON file
        """

    def startAction(self):

        """
        Method called when loading the context of a JFA configuration from frontend.
        Takes user inputs, runs filters and integrity checks, updates global variables.
        Checks other input fields and updates global flags.
        Generates the instance of JFA in the instances' layout.
        """

    @staticmethod
    def exitAction():

        """
        Called when the exit button is pressed.
        Closes the window and terminates the process.
        """

    def loadStatesAction(self):

        """
        Method called for loading and generating checkbox instances.
        Called everytime the "Machine States" textbox is updated
        Generated based on user input in the 'Machine States' field.
        """

    def updateRadioButtons(self):

        """
        Method called to enable / disable correct radio buttons based on evaluation selection
        Selecting One-Way disables NJFA
        Selecting Two-ways reenables it
        """

    def stepAction(self):

        """
        Method called for simulating a logical step/jump in the JFA
        Calls specific method based on user selection and updates specific variable
        to provide user feedback
        """

    def runToEndAction(self):

        """
        Method to loop steps until the machine is finished evaluating
        """

    def __init__(self):

        """
        MainApp class constructor.
        Sets all global variables and also edits process information.
        """
```

Úryvek kódu 21: Funkce v hlavním souboru

3.4.6.1 saveConfigAction()

```python
def saveConfigAction(self):

    # Check if default_dir exists, and create it if it doesn't
    if not os.path.exists(dialogDefaultDir):
        os.makedirs(dialogDefaultDir)

    # Open fileDialog
    filename, _ = QFileDialog.getSaveFileName(
        None,
        "Save Configuration File",
        dialogDefaultDir,
        "JFA Config Files (*.JFACON)",
        options=dialogOptions,
    )

    # Add extension to filename if it's not already there
    if not filename.endswith(".JFACON"):
        filename += ".JFACON"

    # Check of only extension to prevent saving in current directury
    if filename != ".JFACON":
        # Save data to the selected file
        with open(filename, "w") as f:
            f.write(str(self.inputAlphabetText.toPlainText()) + "\n")
            f.write(str(self.inputStringText.toPlainText()) + "\n")
            f.write(str(self.machineStatesText.toPlainText()) + "\n")
            f.write(str(self.jumpsDeclareText.toPlainText()) + "\n")

            print("Data saved to file:", filename)

```

Úryvek kódu 22: saveConfigAction() funkce

Funkce, napojená na „Configuration Save Button“, slouží pro generování dialogového okna a ukládání zápisu automatu z aplikace do souboru. Funkce se nejdříve podívá, jestli existuje složka pro ukládání souborů na adrese dokumentů současného uživatele (používané proměnné deklarovány v hlavičce souboru, kapitola 3.3.1). Pokud složku nenajde, vytvoří ji. Dále se zavolá metoda pro generování dialogového okna s těmito parametry:

- *None*
    - Rodičovský element tohoto dialogového okna
- *„Save Configuration File“*
    - Název dialogového okna, který se zobrazuje v horní části dialogového okna.
- *dialogDefaultDir*
    - Proměnná s cestou do složky, kam se bude soubor ukládat
    - Předdefinováno v hlavičce souboru, kapitola 3.3.1
- *„JFA Config Files (*.JFACON)“*
    - Souborový filtr pro dialogové okno. Zobrazí pouze soubory s touto koncovkou

- *options=dialogOptions*
    - Dodatečné nastavení dialogového okna.
    - Proměnná byla definována v hlavičce souboru

Funkce si následovně zkontroluje, že cesta k souboru získaná z dialogového okna obsahuje i správnou koncovku souboru. Pokud cesta koncovku neobsahuje, doplní ji tam. Další podmínka kontroluje, jestli cesta k souboru neobsahuje pouze dodanou koncovku souboru[12]. Pokud je cesta platná, vytvoří se soubor na cestě a uloží se do něj všechny potřebné informace z textových polí přes *open()* metodu v režimu zápisu. Funkce dále vypíše do konzole informace, že zápis proběhl úspěšně s cestou k souboru.

3.4.6.2 loadConfigAction()

```
1    def loadConfigAction(self):
2
3        # Check if default_dir exists, and create it if it doesn't
4        if not os.path.exists(dialogDefaultDir):
5            os.makedirs(dialogDefaultDir)
6
7        # Open fileDialog
8        filename, _ = QFileDialog.getOpenFileName(
9            None,
10           "Load Configuration File",
11           dialogDefaultDir,
12           "JFA Config Files (*.JFACON)",
13           options=dialogOptions,
14       )
15
16       if filename:
17           # Load data from the selected file
18           with open(filename, "r") as f:
19               # Update variables based on file content
20               self.alphabet = f.readline().strip()
21               self.inputString = f.readline().strip()
22               self.machineStates = f.readline().strip()
23               self.jTransitions = f.read().strip()
24
25           print("Data loaded from file:", filename)
26
27           # Set text fields to new variables
28           self.inputAlphabetText.setText(self.alphabet)
29           self.inputStringText.setText(self.inputString)
30           self.machineStatesText.setText(self.machineStates)
31           self.jumpsDeclareText.setText(self.jTransitions)
32
```

Úryvek kódu 23: loadConfigAction() funkce

[12] Cesta k souboru, která obsahuje pouze koncovku značí, že si uživatel nevybral cestu k uložení (zrušil/zavřel dialogové okno). Poté se nemůže provádět žádné ukládání.

Funkce, napojená „Configuration Load Button“, je velice podobná *saveConfigAction()* ale slouží pro načítání automatu ze souboru do aplikace. Dialogovým oknem získání cesty k souboru, následně načtení soubor na cestě přes *open()* metodu v režimu čtení, následné aktualizace globálních proměnných a výpisu těchto proměnných do textových polí grafického rozhraní.

3.4.6.3 startAction()

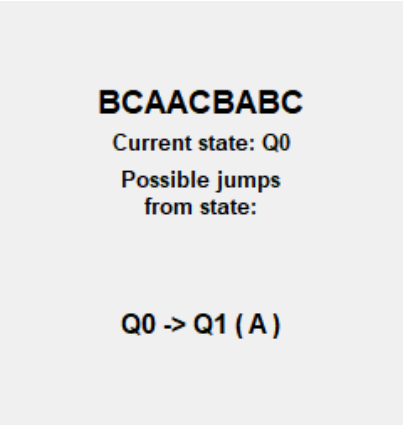
```
1    def startAction(self):
2
3        try:
4            # Check evaluation characteristic radio buttons and update flag
5            if self.OneWayRadioButton.isChecked():
6                self.oneWay = True
7            elif self.BothWayRadioButton.isChecked():
8                self.oneWay = False
9
10           # Check determinism characteristic radio buttons and update flag
11           if self.DJFARadioButton.isChecked():
12               self.deterministic = True
13           elif self.NJFARadioButton.isChecked():
14               self.deterministic = False
15
16           # Preemptively clear some variables
17           self.readSymbols.clear()
18           self.machineStarted = False
19           self.lastPos = 0
20
21           # Loop to clear and remove all sublayouts in the instancesGrid layout
22           while self.instancesGrid.count():
23               # Get and check if item is a widget, if yes, delete it
24               item = self.instancesGrid.takeAt(0)
25               widget = item.widget()
26               if widget is not None:
27                   widget.deleteLater()
28               else:
29                   # Get and check if item is a layout, if yes, remove all items from it
30                   sublayout = item.layout()
31                   if sublayout is not None:
32                       while sublayout.count():
33                           subitem = sublayout.takeAt(0)
34                           subwidget = subitem.widget()
35                           if subwidget is not None:
36                               subwidget.deleteLater()
```

Úryvek kódu 24: startAction() funkce, část 1

Funkce, napojená na „Start button“ je obalená v *try / except* bloku, jelikož volá filtrační podfunkce, které mohou vyvolávat výjimky za cílem zastavení výpočtu.

Funkce si zkontroluje uživatelský výběr přepínacích tlačítek a nastaví podle nich globální vlajky pro pozdější výpočet. Funkce resetuje specifické globální proměnné a vlajky, které by mohly způsobovat problémy v případě, že se vyvolá výjimka během filtrace a nenačtou se důležité proměnné nebo v případě, že se automatu v půlce výpočtu načte znova.

Další while cyklus slouží k odstranění výstupních elementů, které zobrazují informaci o načtené instanci automatu.



Obrázek 12: Příklad instance automatu

Cyklus prochází každý element v hlavním layoutu a zkontroluje, jestli je to widget. Pokud ano, odstraní jej. Pokud je element další layout, prochází tento layout a odstraňuje všechny elementy z něj.

```
37
38              # Get and filter inputs
39              # Get Input alphabet
40              self.alphabet = \
41                  preRun.filterMachineAlphabet(
42                          self.inputAlphabetText.toPlainText()
43                  )
44
45              # Get Input string and formatted
46              self.inputString, self.formattedInputDict = \
47                  preRun.filterInputString(
48                      self.inputStringText.toPlainText(),
49                      self.alphabet
50              )
51
52              # Save the original input string for reference
53              # This prevents incorrect info when input string gets updated in runtime
54              self.inputStringFull = self.inputString
55
56              # Get currently selected start and end states
57              tmp = self.statesCombobox.currentText()
58              # Prevent empty selection
59              if tmp != "No Selection":
60                  self.startState = tmp
61                  self.currentState = self.startState
62              else:
63                  raise StartStateNotFoundError()
64
65              # Go through all checkboxes and check if they are selected
66              # If they are, add them to list as str
67              # NOTE: This doesn't have to be preemptively cleared due to
68              # implicit assignment (overwrites values from previous runs)
69              self.endStates = [checkbox.text() for checkbox in
70                                self.checkBoxList if checkbox.isChecked()]
71
72              if len(self.endStates) < 1:
73                  raise EndStateNotFoundError
```

Úryvek kódu 25: startAction() funkce, část 2

Funkce následovně začne z textových polí filtrovat vstupní data[13]: filtrování abecedy, filtrování vstupního řetězce. Funkce do dodatečné globální proměnné uloží získaný už filtrovaný vstupní řetězec.

Funkce dále získá aktuálně vybranou položku z *Initial State* comboboxu a nastaví podle této položky proměnnou počátečního stavu automatu. Pokud uživatel nechal vybranou výchozí položku („No Selection“), vyvolá se výjimka, jelikož nebyl definován počáteční stav.

Do listu koncových stavů se uloží každý stav, jehož checkbox byl označen. V případě, že v listu nebude ani jeden stav, vyvolá se výjimka.

[13] Funkce *.toPlainText()* vrací obsah elementu textového pole jako string

```
74
75              # Get and filter inputs
76              # Get list of machine states
77              self.machineStates = \
78                  preRun.filterMachineStates(
79                self.machineStatesText.toPlainText()
80              )
81
82              # Get list of provided transitions
83              self.jTransitions = \
84                  preRun.filterJumpTransitions(
85                      self.jumpsDeclareText.toPlainText(),
86                      self.alphabet,
87                      self.machineStates,
88                      self.deterministic
89              )
90
91              # Get formmated dictionary of occurences
92              formattedInputStrToPrint = \
93                  runLogicBoth.generateFormattedInputDictionary(
94                      self.formattedInputDict
95              )
96
97              # Update the formatted input string text field
98              self.outputStringTextFormatted.setText(formattedInputStrToPrint)
99
100             # Debug prints
101             print(
102                 f"\nSpecified alphabet: {self.alphabet}\n"
103                 f"Specified input str: {self.inputString}\n"
104                 f"Specified states:{self.machineStates}\n"
105                 f"Specified starting state: {self.startState}\n"
106                 f"Specified end states: {self.endStates}\n"
107                 f"Specified jumps: {self.jTransitions}"
108             )
109
110             # If non-deterministic behaviour, try and find an accepting path
111             if not self.deterministic:
112                 # Get the earliest possible path
113                 # for one instance of non-deterministic automaton
114                 path = runLogicNDET.generateAdjMatrixAndPath(
115                         self.jTransitions,
116                         len(self.inputStringFull),
117                         self.inputStringFull,
118                         self.startState,
119                         self.endStates
120                 )
121
122                 # Reset global counter for non-deterministic runtime
123                 self.nonDethIter = 0
124
125                 # Check if method didn't return empty string (no path found)
126                 if path is not None:
127                     # If path was found, save values and flip global flags
128                     self.nonDetPathFound = True
129                     self.nonDetPath = path[0]
130                     self.nonDetSymbols = path[1]
131                     print(f"Found acceptable path in non-deterministic evaluation:"
132                           f"\n{self.nonDetPath}\n\n{self.nonDetSymbols}")
133
134                 else:
135                     # If path was not found, raise exception to generate user feedback
136                     self.nonDetPathFound = False
137                     print("Didn't manage to find an acceptable path"
138                           "in non-deterministic evaluation."
139                           " Throwing exception")
140
141                     raise NoAcceptPathFoundError()
```

Úryvek kódu 26: startAction() funkce, část 3

Funkce dále pokračuje ve filtrování vstupních hodnot: filtrování stavů automatu, filtrování přechodů automatu, převod vstupního řetězce na formátovaný slovník výskytů symbolů a následné zapsání do výstupního textového pole.

V tuto chvíli jsou všechny vstupní informace ověřené a převedené a program do konzole vypíše informace on načteném automatu.

Další if větev je volána v případě, že automat byl nastaven na nedeterministický chod. V tomto případě se už v této funkci musí řešit hledání validní cesty pro načtení nedeterministický automat. Zavolá se funkce pro hledání validní cesty a resetuje se globální counter pro průchod listem cesty. Pokud se validní cesta najde, aktualizují se globální proměnné a vlajky pro tyto cesty a do konzole se vypíší bližší informace. Pokud se cesta nenajde, aktualizuje se globální vlajka, vypíší se informace do konzole a vyvolá se výjimka.

```
142
143             # Generate labels for an instance of JFA with their content
144             # Label containing input string
145             self.labelString = QLabel("".join(self.inputStringFull))
146
147             # Label containing current state (starting state)
148             self.labelState = QLabel(f"Current state: <b>{self.startState}</b>")
149
150             # Label containing all possible next jumps
151             self.labelJumps = QLabel(
152                 str(runLogicBoth.findNextJumps
153                     (self.jTransitions,
154                      self.currentState,
155                      self.inputString)
156                     )
157             )
158
159             # Main block for generating instances
160             # Format labels correctly
161             self.labelString.setFont(QFont("Arial", 14, QFont.Bold))
162             self.labelString.setAlignment(Qt.AlignCenter)
163             self.labelString.setFixedSize(158, 20)
164
165             self.labelState.setFont(QFont("Arial", 10, QFont.Bold))
166             self.labelState.setAlignment(Qt.AlignCenter)
167             self.labelState.setFixedSize(158, 17)
168
169             self.labelJumps.setFont(QFont("Arial", 12, QFont.Bold))
170             self.labelJumps.setAlignment(Qt.AlignCenter)
171             self.labelJumps.setFixedSize(158, 134)
172
173             self.labelJumpsText = QLabel("Possible jumps\nfrom state:\n")
174
175             # Manual formatting of a label
176             self.labelJumpsText.setFont(QFont("Arial", 10, QFont.Bold))
177             self.labelJumpsText.setAlignment(Qt.AlignCenter)
178             self.labelJumpsText.setFixedSize(158, 30)
```

Úryvek kódu 27: startAction() funkce, část 4

Funkce začne generovat label objekty pro vytvoření výstupních elementů instance a nastavovat jejich formátování.

```
179
180             # Create a sub-layout object containing labels
181             layout = QVBoxLayout()
182             # Put all labels into the sub-layout
183             layout.addWidget(self.labelString)
184             layout.addWidget(self.labelState)
185             layout.addWidget(self.labelJumpsText)
186             layout.addWidget(self.labelJumps)
187
188             # Assign the sub-layout to a layout widget
189             layoutWidget = QWidget()
190             layoutWidget.setFixedSize(158, 201)
191             layoutWidget.setLayout(layout)
192
193             # Add the widget to a list of widgets
194             # Used to reset instances in layout if new machine loaded
195             self.instancesGrid.addWidget(layoutWidget)
196
197             # If all fetches and checks passed, inform the user and flig global flag
198             self.statusText.setText("Passed!\n" "Machine has been loaded!")
199
200             # Global flag that loading the machine was succesful
201             self.machineStarted = True
202
203         # Except branch to catch all custom exceptions and print them to status field
204         # Functions as feedback to user about incorrect input
205         except (
206                 EmptyFieldError,
207                 InvalidAlphabetFormatError,
208                 InvalidSymbolInAlphabetError,
209                 InputSymbolNotInAlphabetError,
210                 StartStateNotFoundError,
211                 EndStateNotFoundError,
212                 StateDoesNotExistError,
213                 SymbolDoesNotExistError,
214                 InvalidDeterministicFormatError,
215                 InputStringTooLongError,
216                 PathNoTransitionProvided
217         ) as exc:
218             self.statusText.setText(f"<b>ERROR</b><br><br>{exc}")
219         except NoAcceptPathFoundError as exc:
220             self.statusText.setText(f"<b>STRING REFUSED</b><br><br>{exc}")
```

Úryvek kódu 28: startAction() funkce, část 5

Následovně se vytvoří vertikální layout, do kterého se dají vytvořené labely. Tento layout se naformátuje a přidá do layoutu instancí automatu.

Vy výstupního textového pole se napíše informace, že automat se správně načetl a otočí se globální vlajka signalizující, že automat je aktivní.

Na konci funkce je *except* blok zachycující výjimky, které můžou být vyvolány. V případě, že jsou vyvolány, do výstupního textového pole se vypíší informace ohledně této chyby[14].

3.4.6.4 exitAction()

```
1    @staticmethod
2    def exitAction():
3
4        sys.exit(1)
```

Úryvek kódu 29: exitAction() funkce

Funkce, napojená na „Exit Button“, slouží k ukončení aplikace skrze ukončení procesu aplikace s výstupním kódem 1.

[14] *NoAcceptPathFound* je výjimka vyvolávaná, pokud se pro nedeterministický automat nenajde cesta, ovšem není chyba při načítání automatu. Proto je ve vedlejší *except* větvi.

3.4.6.5 loadStatesAction()

```python
def loadStatesAction(self):

    # Pull inputed states from text box a
    # and split them into a list via a splitting symbol
    states = self.machineStatesText.toPlainText().replace("\n", "").split(";")

    # Clear the selection combobox before assigning new values
    self.statesCombobox.clear()

    # Clear the list of checkboxes
    # NOTE: list is used to check selected checkboxes when loading automaton
    self.checkBoxList.clear()

    # Loop to remove all checkbox widgets
    for i in reversed(range(self.endStatesGrid.count())):
        # Get widget by index
        widget = self.endStatesGrid.itemAt(i).widget()
        if widget is not None:
            # Remove from layout
            self.endStatesGrid.removeWidget(widget)
            # Remove widget itself
            widget.deleteLater()

    # Default value assignment
    self.statesCombobox.addItem("No Selection")

    # Helping sizing variables for generating checkboxes
    num_cols = 4
    row = 0
    col = 0

    # For each state in list
    for i, state in enumerate(states):
        # Failsafe for empty entry
        if str(state) == "":
            continue

        # Add item to combobox
        self.statesCombobox.addItem(state)

        # Create a checkbox widget and assign it to the layout of checkboxes
        checkbox = QCheckBox(state)
        self.endStatesGrid.addWidget(checkbox, row, col)

        # Add instance of checkbox to global list
        self.checkBoxList.append(checkbox)
        # Update sizing variables
        col += 1
        if col == num_cols:
            col = 0
            row += 1
```

Úryvek kódu 30: loadStatesAction() funkce

Funkce, volaná pokaždé, když se aktualizuje obsah textového pole „Machine States“, slouží k aktualizaci „Initial State“ comboboxu a generování nových checkboxů pro výběr koncového automatu.

Funkce si naformátuje obsah „Machine State“ textového pole stejným způsobem jako *filterMachineStates()* funkce, vyprázdní globální listy pro ukládání záznamů v komboboxu a ukládání instancí checkboxů.

Dále se odebere každý checkbox z „End States“ layoutu aby se mohli vygenerovat nové s aktuálními informacemi.

Do „Initial state“ komboboxu se uloží výchozí hodnota pro prázdný výběr.

Definují se pomocní proměnné a začnou se generovat checkboxy pro každý zapsaný stav. Checkboxy se generují maximálně do 4 sloupců, poté se začnou generovat na nový řádek.

3.4.6.6 updateRadioButtons()

```
1     def updateRadioButtons(self):
2
3         if self.OneWayRadioButton.isChecked():
4             self.NJFARadioButton.setEnabled(False)
5
6             self.NJFARadioButton.setChecked(False)
7             self.DJFARadioButton.setChecked(True)
8
9         elif self.BothWayRadioButton.isChecked():
10            self.NJFARadioButton.setEnabled(True)
```

Úryvek kódu 31: updateRadioButtons() funkce

Funkce, volána pokaždé, když se aktualizuje výběr přepínačů v „Evaluation Selection“, slouží k vypínání nebo zapínání přepínače *NJFA* v „Determinism Selection“.

Aplikace se zapne s přepínačem na obousměrný chod. Pokud uživatel přepne na jednosměrný chod, vypne se nedeterministický přepínač a vybere se deterministický, jelikož jednosměrný skákající konečný automat je pokaždé deterministický.

V případě, že později opět vybere přepínač na obousměrný chod, nedeterministický přepínač se opět zapne.

3.4.6.7 stepAction()

```python
def stepAction(self):

    # Check to see if JFA was loaded
    if not self.machineStarted:
        return

    # Try/Catch for custom exceptions
    try:
        # Call the main method based on radio button selection
        # Deterministic approach
        if self.deterministic:
            if self.oneWay:
                (
                    self.formattedInputDict,
                    self.inputString,
                    self.currentState,
                    self.prevInfo,
                    self.lastPos
                ) = runLogicDET.findAndRunJumpOneSide(
                    self.jTransitions,
                    self.currentState,
                    self.formattedInputDict,
                    self.inputString,
                    self.lastPos
                )
            # Two-ways logic
            else:
                (
                    self.formattedInputDict,
                    self.inputString,
                    self.currentState,
                    self.prevInfo,
                    self.lastPos
                ) = runLogicDET.findAndRunJumpBothSides(
                    self.jTransitions,
                    self.currentState,
                    self.formattedInputDict,
                    self.inputString
                )
```

Úryvek kódu 32: stepAction() funkce, část 1

Funkce, napojená na „Step Button“, slouží k vykonání jednoho kroku automatu.

Funkce si nejdříve zkontroluje, že automat byl správně načtený.

Deterministické krokové funkce mohou vyvolat výjimku, proto máme volání funkcí obalené v *try / except* bloku.

```
40
41              # Non-deterministic approach
42              else:
43                  # Increment global index counter
44                  self.nonDethIter += 1
45
46                  # Get index values for previous and current information
47                  previousIndex = self.nonDethIter - 1
48                  currentIndex = self.nonDethIter
49
50                  # Manually fill the previous info variable
51                  # NOTE: This was normally done in a method, since we don't need to
52                  # search for a step to be done, we can just add it manually
53                  self.prevInfo = [self.nonDetPath[previousIndex],
54                                   self.nonDetSymbols[previousIndex]]
55
56                  # Call a function to get index of
57                  # currently read symbol in input string
58                  # NOTE: This was also normally done
59                  # in a method, but we cannot call it here because it modifies
60                  # the input string in deterministic way
61                  (   self.inputString,
62                      self.lastPos,
63                      self.formattedInputDict
64                  ) = runLogicNDET.findNextSymbolPosition(
65                      self.nonDetSymbols[previousIndex],
66                      self.inputString,
67                      self.formattedInputDict
68                  )
```

Úryvek kódu 33: stepAction() funkce, část 2

Funkce později zavolá výpočetní funkce pro aktualizaci globálních proměnných podle daného chodu a směru (deterministický automat) nebo začne aktualizovat proměnné podle globálního počítadla z nalezené cesty (nedeterministický automat).

Funkce si dále zavolá pomocnou funkci pro nalezení pozice současně čteného symbolu[15].

[15] Jelikož už nový stav a čtený symbol známe, není třeba provádět výpočet přes funkce v deterministickém chodu, které na to ani nejsou stavěné.

```
69
70                  # update current state variable
71                  self.currentState = self.nonDetPath[currentIndex]
72
73              # Update the list of read symbols (positions)
74              self.readSymbols.append([self.prevInfo[1], self.lastPos])
75
76              # Update status text with user feedback
77              self.statusText.setText(
78                  f"A jump has been performed!\n{self.prevInfo[0]} ->"
79                  f"{self.currentState}"
80                  f" via reading {self.prevInfo[1]}")
81
82          # Custom exception handling
83          except NoJumpToPerformError as exc:
84              self.statusText.setText(f"<b>String REFUSED</b><br><br>{exc}")
85              self.machineStarted = False
86
87          # Debug prints
88          # Print info about jump and automata
89          print(f"\nA jump has been performed\nNew formatted string:")
90
91          # Print info about new formatted string
92          for key in self.formattedInputDict:
93              print(f"Key: '{key}': {self.formattedInputDict[key]}")
94
95          # Update textbox with occurence values
96          self.outputStringTextFormatted.setText(
97            runLogicBoth.generateFormattedInputDictionary(
98              self.formattedInputDict
99            )
100         )
101
102         # Print info about new input string and current state
103         print(
104             f"\nNew input string: {self.inputString}"
105             f"\nNew current state: {self.currentState}")
```

Úryvek kódu 34: stepAction() funkce, část 3

Po nalezení pozice dalšího čteného symbolu se ještě nastaví z cesty současný stav. Zde končí *else* větev pro nedeterministický chod.

Do globálního listu se přidá čtený symbol a jeho pozice. Do výstupního textového pole se vypíše informace ohledně provedeného přechodu. Funkce má zde *except* blok pro zachycení možné výjimky z deterministického výpočtu. Pokud se tato výjimka vyvolá a zachytí, otočí se vlajka signalizující aktivní automat.

Do konzole se vypíše informace ohledně změny vstupního řetězce a slovníku výskytů symbolů. Zároveň se aktualizuje výstupní pole „Formatted Input String“.

```
105         # Get new content for instance string label
106         labelString = runLogicBoth.createFormattedStringLabel(
107             self.inputStringFull,
108             self.readSymbols,
109             self.lastPos
110         )
111 
112         # Update the content of instance string label
113         self.labelString.setText(labelString)
114 
115         # Update the state label with new current state
116         self.labelState.setText(f"Current state: <b>{self.currentState}</b>")
117 
118         # Update the jumps label with new text
119         self.labelJumps.setText(
120             str(
121                 runLogicBoth.findNextJumps(
122                     self.jTransitions,
123                     self.currentState,
124                     self.inputString
125                 )
126             )
127         )
128 
129         # Check if every symbol in input string was read
130         stringHasSymbols = False
131         for symbol in self.inputString:
132             if symbol != "_":
133                 stringHasSymbols = True
134                 break
135 
136         # Once the input string is empty, check if JFA is accepted or not,
137         # then print information to status textbox
138         if not stringHasSymbols:
139             if self.currentState in self.endStates:
140                 self.statusText.setText(
141                     f"<b>String ACCEPTED</b><br><br>"
142                     f"A jump has been performed!<br>{self.prevInfo[0]}"
143                     f"-> {self.currentState}"
144                     f" via reading {self.prevInfo[1]}"
145                 )
146             else:
147                 self.statusText.setText(
148                     f"<b>String REFUSED</b><br><br>"
149                     f"A jump has been performed!<br>{self.prevInfo[0]}"
150                     f"-> {self.currentState}"
151                     f" via reading {self.prevInfo[1]}"
152                 )
153 
154                 # Update the content of instance string label
155                 # All symbols read -> entire label is red
156                 labelString = runLogicBoth.returnRedInputString(self.inputStringFull)
157                 self.labelString.setText(labelString)
158 
159                 # Flip the flag to prevent running logic on empty data
160                 self.machineStarted = False
```

Úryvek kódu 35: stepAction() funkce, část 4

Funkce začne aktualizovat labely v instanci automatu podle zjištěných parametrů: label obsahující řetězec naformátovaný barvami podle stavu přečtení, label obsahující současný stav, label obsahující možné přechody ze současného stavu.

Funkce poté zkontroluje, jestli byl každý symbol v řetězci přečtený (všechny symboly nahrazeny za zástupný symbol). Pokud ano, aktualizuje lokální vlajku.

Pokud byly všechny symboly přečteny, funkce zkontroluje, jestli je automat v koncovém stavu. Podle výsledku porovnání zapíše informaci o přijetí / odmítnutí řetězce automatem do „Current Machine Status“ textového pole. Následně nastaví label řetězce na červenou barvu zavoláním funkce *returnRedInputString()* a nastaví vlajku signalizující aktivní automat na False.

3.4.6.8 runToEndAction()

```
1     def runToEndAction(self):
2
3         while self.machineStarted:
4             self.stepAction()
```

Úryvek kódu 36: runToEnd() funkce

Funkce, napojená na „RunToEnd Button“, slouží k urychlení provedení kroků automatu.

Ve *while* cyklu opakovaně volá provedení následujícího kroku, dokud aplikace nenastaví globální vlajku pro signalizující aktivní automat na False.

4 NÁVOD K APLIKACI

V této kapitole bude pospán návod pro spuštění a ovládaní aplikace uživatelem. Kapitola se bude zabývat požadavky na spuštění, zadáváním zápisu automatu, spuštěním výpočtu automatu, orientací ve výstupních informacích a popřípadě asistencí opravy nesprávného zápisu automatu.

Pokud si uživatel během zadávání zápisu automatu nebude jistý, jaké informace nebo v jakém formátu zadávat, může položit kurzor na element, u kterého si není jistý. Na místě kurzoru se zobrazí nápovědné okno.

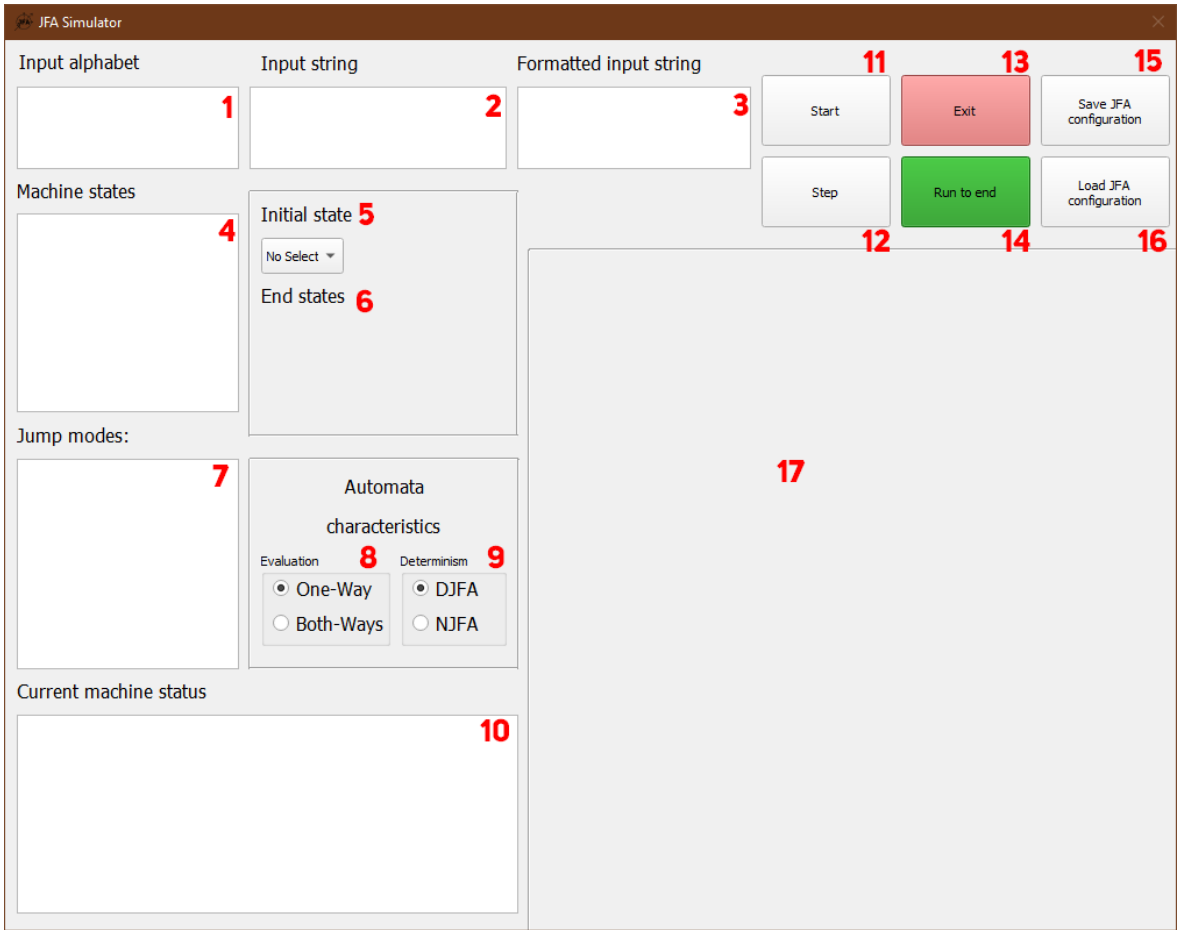
4.1 Požadavky aplikace

Aplikace vyžaduje pro spuštění interpreter programovacího jazyka Python verze 3.10 a víš.

Zároveň musí mít tento interpreter nainstalované následující knihovny:

- ctypes
- sys
- os
- PyQt5
- Numpy

Pro jednoduchou orientaci se použije GUI na obrázku 7 z předchozí kapitoly.



4.2 Zadávání vstupních hodnot

Uživatel zadá vstupní abecedu automatu do pole 1 - „Input Alphabet“. Abeceda by měla být zadána symbol po symbolu rozdělená středníkem. Např.:

$$A; B; C; D$$

Uživatel zadá vstupní řetězec automatu do pole 2 – „Input String“. Symboly vstupního řetězce musí být složeny pouze ze symbolů definovaných ve vstupní abecedě. Např.:

$$ABACBD$$

Uživatel zadá stavy automatu do pole 4 – „Machine States“. Uživatel si může definovat jakýkoliv název jakékoliv délky. Stavy musí být oddělené středníkem. Např.:

$$Stav1; Stav2; Stav3$$

Po zadání stavů automatu si uživatel vybere počáteční stav v „Initial State“ komboboxu (element 5) a koncové stavy z vygenerovaných zaškrtávacích polí (element 6)

Pokud uživatel změní nějaký definovaný stav v „Machine States“ poli, „Initial state“ kombobox a volitelné zaškrtávající pole se resetují podle nového zápisu.

Uživatel zadá přechody mezi stavy do pole 7 – „Jump Modes“. Přechody musí být napsané ve formátu $[stav1] - [symbol] - [stav2]$.

- *stav1* je stav, ze kterého může přechodová funkce vycházet.
- *symbol* je symbol, který bude automat číst při provádění přechodu.
- *stav2* je stav, do kterého automat přejde po vykonání přechodu.

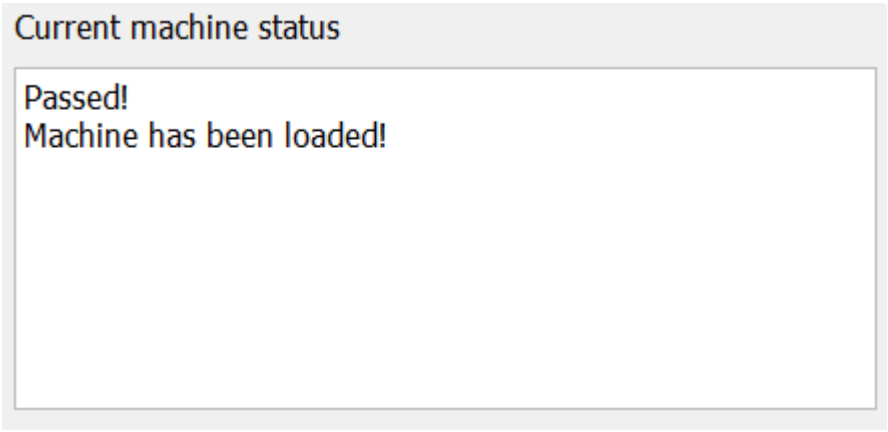
Každý přechod musí být napsaný na samostatný řádek, tj. přechody jsou oddělené klávesou *Enter*.

Uživatel si dále vybere, jestli má být automat jednosměrný nebo obousměrný a deterministický nebo nedeterministický v přepínačích 8 – „Evaluation Selection“ a 9 – „Determinism Selection“.

Pokud si uživatel vybere v přepínačích jednosměrný chod, nedeterministický výpočet se vypne a nemůže být vybraný.

4.3 Ovládání automatu

Jakmile uživatel zadá všechny vstupní hodnoty, zmáčkne ovládací tlačítko 11 – „Start Button“ pro načtení automatu. Pokud jsou zadané informace ve správném tvaru, aplikace informuje uživatele textem v poli 10 – „Current Machine Status“.



Obrázek 13: Vypsaný text při správném načtení automatu

Do tohoto pole se zapisují bližší informace ohledně stavu automatu (provedený přechod, přijetí / odmítnutí řetězce na konci výpočtu atd.).

Pokud byly zadané informace v nesprávném tvaru, aplikace do tohoto pole vypíše bližší informace ohledně chyby. Informace o některých chybách navádí uživatele, kde se chyba vyskytuje, aby jí uživatel mohl opravit.

Jakmile byl automatu úspěšně načten, uživatel může začít posouvat výpočet automatu za pomocí tlačítek 12 – „Step Button“ a 14 – „Run To End Button“.

Tlačítko „Step Button“ provede „jeden krok“ automatu. Automat přečte jeden ze symbolů řetězce a provede podle něj přechod.

Tlačítko „Run To End“ skočí až na konec výpočtu, kdy je uživatel informován, jestli je vstupní řetězec přijat nebo odmítnut.

Aplikace vytvoří grafické vyjádření hlavních informací načteného automatu a postupně vypisuje aktuální informace v elementu 17 – „Automata Instance Layout“.

- Vstupní řetězec včetně informace o tom, které symboly byly přečteny:
  - **Černě vybarvený** symbol nebyl doposud automatem přečtený.
  - **Červeně vybarvený** symbol již byl automatem přečtený.
  - **Zeleně vybarvený** symbol byl právě nyní přečtený automatem.

Jakmile uživatel bude chtít ukončit aplikaci, zmáčkne tlačítko 13 – „End Button“.

4.4 Ukládání zápisu do souboru

Pro uložení zápisu do souboru uživatel zmáčkne tlačítko 15 – „Configuration Save Button“. Objeví se okno, kde si uživatel vybere, kam chce soubor uložit. Aplikace následovně do této cesty uloží soubor zápisu automatu.

4.5 Načítání zápisu ze souboru

Pro načtení zápisu ze souboru uživatel zmáčkne tlačítko 16 – „Configuration Load Button“. Objeví se dialogové okno, kde si uživatel vybere soubor, který chce načíst. Aplikace následovně z tohoto souboru načte informace o zápisu a tyto informace zapíše do správných textových polí.

ZÁVĚR

V rámci této bakalářské práce jsme se zaměřili na skákající konečné automaty jako rozšíření tradičních konečných automatů. Naším cílem bylo poskytnout ucelený pohled na tento koncept a jeho aplikace.

V teoretické části práce jsme provedli formální popis skákajících konečných automatů. Zaměřili jsme se na popis konceptu skákajících konečných automatů, popis hlavních vlastností a pojmů a popis modelů skákajícího konečného automatu.

Pro lepší pochopení a vizualizaci chování skákajících konečných automatů jsme vytvořili demonstrační aplikaci v jazyce Python nazvanou *JFA Simulator*. Tato aplikace umožňuje uživatelům interaktivně simulovat a vizualizovat výpočty skákajících konečných automatů. Díky zobrazení informací jako je řetězec, současný stav automatu a další, uživatelé mohou lépe porozumět výpočetním schopnostem těchto automatů. Navíc jsme do aplikace implementovali logiku pro ukládání zápisu automatu do souboru a jeho načítání ze souboru, což umožňuje uživatelům uchovávat a sdílet své automaty.

Tato práce přináší nejen samotný demonstrační program, ale také rozšiřuje dostupnost teorie skákajících konečných automatů v českém jazyce. Věříme, že naše práce poslouží studentům, výzkumníkům a zájemcům o oblast formálních jazyků a automatů jako cenný zdroj informací a nástroj pro experimentování s těmito zajímavými modely výpočtu.

SEZNAM POUŽITÉ LITERATURY

1. **Martinek, Pavel.** *Základy Teoretické Informatiky.* Katedra Informatiky, Přírodovědecká fakulta, Univerzita Palackého. Olomouc : Univerzita Palackého, 2006. Učební text.

2. **Moore, Karleigh a Gupta, Dishant.** Finite State Machines. *Brilliant.* [Online] [Citace: 4. Duben 2023.] https://brilliant.org/wiki/finite-state-machines/.

3. **Waltz, Frederick M.** *Image Processing Using Finite-State Machines.* Department of Electrical and Computer Engineering, University of Minnesota. Minneapolis, USA : University of Minnesota, 2012. 978-1-84996-169-1.

4. **Lecture 22: Finite automata.** ***Cornell Bowers College of Computing and Information Science.*** **[Online] Cornell Bowers C-IS. [Citace: 8. Duben 2023.] https://www.cs.cornell.edu/courses/cs2800/2017sp/lectures/lec22-dfa.html.**

5. Anderson, James A. ***Automata theory with modern applications.*** **Cambridge, England : Cambridge University Press, 2006. 9780521613248.**

6. Meduna, Alexander a Zemek, Petr. ***Jumping Finite Automata.*** **místo neznámé : International Journal of Foundations of Computer Science, 2012.**

7. Holzer, Markus a Beier, Simon. ***Nondeterministic right one-way jumping finite automata.*** **Giessen : Institut für Informatik, Universität Giessen, 2020.**

8. Szilárd, Fazekas Zsolt, Kaito, Hoshi a Akihiro, Yamamura. ***Two-Way Jumping Automata, in proceedings of 14th International Workshop, FAW 2020,.*** **China : FAW 2020, 2020.**

9. Shallit, J. Two-Way Automata. [autor knihy] Jean-Éric Pin. ***Handbook of Automata Theory.*** **Paris : Université de Paris and CNRS, 2010.**

10. Hiroyuki, Chigahara, Szilárd, Fazekas Zsolt a Akihiro, Yamamura. ***One-Way Jumping Finite Automata.*** **místo neznámé : World Scientific Pub Co Pte, 2016.**

11. Fernau, Henning, a další. ***Characterization and complexity results on jumping finite automata.*** **2015. 0304-3975.**

12. ***Multiset Automata.*** **Csuhaj-Varjú, Erszébet, Martín-Vide, Carlos a Mitrana, Victor. Berlin : Springer, Heidelberg, 2001.**

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

GUI Uživatelské rozhraní.

DJFA Deterministický skákající konečný automat.

NJFA Nedeterministický skákající konečný automat

SEZNAM OBRÁZKŮ

SEZNAM ÚRYVKŮ KÓDU

SEZNAM PŘÍLOH

PŘÍLOHA P I: REPOZITÁŘ ZDROJOVÉHO KÓDU

https://github.com/JackG256/JFA-Sim

PŘÍLOHA P II: ZDROJOVÝ KÓD VE FORMÁTU .ZIP

Soubor s názvem „SourceCode.zip“ obsahuje repositář se zdrojovým kódem