FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Knihovna algoritmů ALT - webové rozhraní
Student:	Michael Vrána
Vedoucí:	Ing. Tomáš Pecka
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

Seznamte se se současnou implementací knihovny algoritmů ALT (Algorithms Library Toolkit) [1] vyvíjenou na katedře teoretické informatiky.
Proveďte rešeršní rozbor aplikací, které vizualizují koncept “pipes and filters” (např. GNU Radio Companion [2]).
Analyzujte možnosti implementace grafického rozhraní pro knihovnu ALT, které bude fungovat jako webová služba.
Analyzujte možnosti knihovny poskytnout informace o dostupných algoritmech, datových typech a jejich dokumentaci.
Navrhněte a implementujte interaktivní webové grafické rozhraní “pipes and filters” pro knihovnu algoritmů [1].
Implementujte efektivní plánování vyhodnocení schématu propojení algoritmů.
Implementujte možnost tvorby vstupního automatu pomocí nástroje Statemaker [3].
Hotové řešení vhodnými prostředky otestujte.

Seznam odborné literatury

[1] Martin Žák. Automatová knihovna – vnitřní a komunikační formát. Bakalářská práce, České vysoké učení technickév Praze, Fakulta informačních technologií, Praha, 2014.
[2] GNURadioCompanion [online]. [cit. 2019-12-20]. Dostupné z: https://wiki.gnuradio.org/index.php/GNURadioCompanion
[3] Svoboda, Petr. Webový editor konečných automatů. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 2. února 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Knihovna algoritmů ALT - webové rozhraní

Michael Vrána

Katedra softwarového inženýrství
Vedoucí práce: Ing. Tomáš Pecka

4. června 2020

Poděkování

Chtěl bych především poděkovat vedoucímu práce Ing. Tomáši Peckovi za jeho trpělivost a velice cenné rady. Dále také děkuji mé přítelkyni Veronice za celkovou podporu při studiu a také za kontrolu pravopisu. Na závěr chci poděkovat celé mé rodině za podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 4. června 2020 . . . . . .

Odkaz na tuto práci

Vrána, Michael. *Knihovna algoritmů ALT - webové rozhraní.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato práce se zabývá návrhem a implementací webového GUI pro knihovnu Algorithms Library Toolkit. Toto GUI v podobě webové aplikace znázorňuje jednotlivé algoritmy jako boxy, jejichž výstupy lze dál řetězit do dalších algoritmů. Těmto algoritmům uživatel může specifikovat vstupy a následně je může vyhodnotit. Tato webová aplikace zároveň integruje již existující React aplikaci Statemaker, která zjednodušuje vytváření a zobrazování konečných automatů. Výsledná webová aplikace je implementována jakožto fork Statemakeru. Pro zpřístupnění ALT webové aplikaci bylo vytvořeno webové API.

Klíčová slova webová aplikace, Algorithms Library Toolkit, Pipes-And-Filters, fronta zpráv, React, TypeScript, Kotlin, C++

Abstract

This thesis deals with the design and implementation of web GUI for the Algorithms Library Toolkit. This GUI in the form of a web application shows individual algorithms as boxes, the outputs of which can be further piped to other algorithms. User can specify inputs of algorithms and then also evaluate them. This web application integrates an already existing React application Statemaker which simplifies creation and visualization of finite automata. For ALT to be accessible to the web application a web API was created.

Keywords web application, Algorithms Library Toolkit, Pipes-And-Filters, message queue, React, TypeScript, Kotlin, C++

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódů

Seznam algoritmů

Úvod

Algorithms Library Toolkit (ALT) [1] je knihovna algoritmů a datových struktur z oblasti teorie automatů, arbologie a teorie grafů, vyvíjena na Katedře teoretické informatiky Fakulty informačních technologií ČVUT. Cílem ALT je implementovat formální algoritmy a datové struktury totožně, jako jsou popsány v odborné literatuře. Mimo standardní použití, jakožto C++ knihovna, ALT nabízí také terminálové rozhraní v podobě shellu aql2 [2].

Aql2 je založeno na architektuře Pipes-And-Filters, která byla zpopularizována operačním systémem UNIX [3]. Pipes-And-Filters architektura v aql2 umožňuje jednotlivé algoritmy skládat do větších funkčních celků. Ovšem aql2 s sebou přináší specifický jazyk pojmenovaný Algorithm Query Language, který může být pro nové uživatele značně komplexní.

Z tohoto důvodu vzniklo grafické rozhraní pro ALT agui2, které je ale teprve ve fázi raného prototypu. Agui2 graficky znázorňuje jednotlivé algoritmy a umožňuje uživateli přes grafické rozhraní jim zadat vstup a následně je za sebou řetězit. Toto rozhraní je pro nové uživatele lépe pochopitelné než aql2, ale protože je to pouze desktopová aplikace, vyžaduje po uživateli nainstalování ALT. ALT je momentálně dostupné jen na operačních systémech GNU/Linux, což může sloužit jako překážka pro uživatele, kteří používají jiné operační systémy

Z toho důvodu vznikla myšlenka zpřístupnit ALT jako webovou aplikaci. Tato aplikace má za cíl graficky znázornit algoritmy a jejich řetězení jako agui2 a zároveň umožní uživateli zadat vstupy a zobrazit výstupy. Aplikace by byla dostupná z jakéhokoliv webového prohlížeče, to umožní snížit přístupovou bariéru novým uživatelům.

V roce 2019 vznikla v rámci bakalářské práce Bc. Petra Svobody [4] webová aplikace Statemaker [5], která umožňuje graficky znázornit a kreslit konečné stavové automaty. Jelikož ALT nabízí mnoho algoritmů pro různé druhy konečných stavových automatů, nabízí se zde možnost tuto aplikaci využít pro vytváření a zobrazování automatů ve webovém grafickém rozhraní pro ALT.

To by umožňovalo tuto aplikaci využít například pro výuku.

Tato práce se proto zabývá návrhem a implementací webového grafického rozhraní pro ALT a zpřístupněním ALT pro web. Zároveň práce cílí do této aplikace zaintegrovat aplikaci Statemaker.

Kapitola 1

Cíl práce

Cílem práce je navrhnout a implementovat webové grafické rozhraní pro ALT, které bude sloužit jako alternativa k ALT agui2.

Tato aplikace umožní uživateli rychle a přehledně řetězit jednotlivé výstupy algoritmů za sebou dle principu Pipes-And-Filters. Aplikace by měla umožnit uživateli pomocí aplikace Statemaker vytvořit konečný stavový automat, který nadále může použít jako vstup algoritmů. Výstupy bude možné zobrazit jako textový výstup anebo jako automat ve Statemakeru.[5]

Pro vyhodnocení algoritmů bude nutné vyřešit zpřístupnění ALT pro webovou aplikaci. Toho je možné dosáhnout více způsoby. Tato práce se zabývá implementováním webového API pro ALT, se kterým bude webová aplikace komunikovat.

Kapitola 2

Základní pojmy a definice

V této kapitole jsou popsány a vysvětleny základní pojmy a definice, které jsou použity v této práci.

2.1 Teorie grafů

Definice týkající se teorie grafů jsou převzaty z Průvodce labyrintem algoritmů [6].

2.1.1 Orientovaný graf

Orientovaný graf G je uspořádaná dvojice (V, E), kde:

- V je množina vrcholů G
- $E \subseteq V \times V$ je množina hran G
- uv označuje hranu z vrcholu u do vrcholu v

Tato práce využívá orientované grafy k reprezentaci algoritmů a jejich řetězení, tyto grafy jsou v této práci označovány jako *algoritmové grafy*. Vstupní či výstupní hodnoty a algoritmy jsou reprezentovány jako vrcholy grafu. Jednotlivé předávání hodnot mezi vrcholy následně reprezentují hrany, ovšem v tomto případě musí hrana ještě nést informaci, jaký argument v cílovém algoritmu, kam je hodnota předávána, představuje.

2.1.2 Následníci a předchůdci

Následníci vrcholu v budou takové vrcholy, do kterých vede hrana z vrcholu v. Analogicky z *předchůdců* vede hrana do v. Množina všech následníků v je v textu značena jako $suc(v)$.

2.1.3 Vstupní stupeň

Vstupní stupeň $\deg^{\text{in}}(v)$ vrcholu v udává počet předchůdců v. *Výstupní stupeň* $\deg^{\text{out}}(v)$ vrcholu v udává počet následníků v.

2.1.4 Acyklický orientovaný graf

Acyklický orientovaný graf, neboli *Directed Acyclic Graph (DAG)*, je orientovaný graf G, který neobsahuje cyklus. Jinými slovy v G neexistuje cesta $uv_1, v_1v_2 \dots v_nu$.

Každý algoritmový graf je v našem případě DAG. Cykly v grafu by totiž způsobily nekonečnou smyčku ve vyhodnocování. To je také dáno tím, že předpokládáme, že algoritmy jsou čisté funkce a proto nemají žádné vedlejší efekty, které by mohly měnit chování algoritmu i když by pokaždé obdržel stejný vstup.

2.1.5 Topologické uspořádání

Lineární uspořádání $\prec$ na vrcholech grafu nazveme *topologickým uspořádáním vrcholů*, pokud pro každou hranu uv platí, že $u \prec v$. Neformálněji řečeno, pro každou hranu uv platí, že u je v topologickém uspořádání před v.

Topologické uspořádání je nutné najít pro vyhodnocení algoritmového grafu. Určuje totiž v jakém pořadí vyhodnotit jednotlivé algoritmy. Aby se mohl algoritmus vyhodnotit, musí se nejdříve vyhodnotit všechny předešlé algoritmy a vstupy. Topologické uspořádání seřadí tyto jednotlivé vrcholy tak, že při postupném vyhodnocování topologicky uspořádaných vrcholů algoritmového grafu se vždy vyhodnotí předchůdci algoritmového vrcholu. To zajistí, že každý algoritmus bude mít k dispozici všechny výsledky předchůdců, které mu jsou předávány jako parametry.

2.2 Konečné automaty

Definice různých konečných automatů jsou nutné z důvodu rozšíření modulu Statemakeru pro FIT textový formát konečných automatů. Všechny následující definice v této podkapitole mimo definici nedeterministického konečného automatu s více počátečními stavy a ϵ-přechody, jsou převzaty ze sbírky řešených příkladů Automaty a gramatiky [7].

2.2.1 Konečný automat

Konečný automat je uspořádaná pětice $(Q, \Sigma, \delta, q_0, F)$, kde:

- Q je konečná neprázdná množina stavů
- Σ je konečná vstupní abeceda

- δ je přechodová funkce
- $q_0 \in Q$ je počáteční stav
- $F \subseteq Q$ je množina koncových stavů

Konečný automat je výpočetní model. Čte ve směru vpravo neměnnou pásku, kde každá buňka na pásce obsahuje symbol vstupní abecedy. Automat začíná výpočet v počátečním stavu a s každým přečteným symbolem mění svůj stav. Pokud po přečtení celé pásky automat skončí v koncovém stavu, pak vstup přijímá, v opačném případě vstup nepřijímá.

2.2.2 Deterministický konečný automat

Deterministický konečný automat je konečný automat, jehož přechodová funkce δ je zobrazení z množiny $Q \times \Sigma$ do množiny stavů Q. Neformálněji řečeno, deterministický konečný automat vždy jednoznačně ví, jak ve výpočtu pokračovat.

2.2.3 Nedeterministický konečný automat

Nedeterministický konečný automat je konečný automat, jehož přechodová funkce přechodová funkce δ je zobrazení z množiny $Q \times \Sigma$ do množiny všech podmnožin Q. Jinými slovy řečeno, pokud nedeterministický automat si může vybrat při přechodu mezi více stavy, vyzkouší všechny možnosti.

2.2.4 Nedeterministický konečný automat s ϵ-přechody

Přechod, při kterém automat nečte žádný symbol ze vstupu, se nazývá ϵ-přechod.

Nedeterministický konečný automat s ϵ-přechody je nedeterministický konečný automat, jehož přechodová funkce δ je zobrazení z množiny $Q \times (\Sigma \cup \{\epsilon\})$ do množiny všech podmnožin Q.

2.2.5 Nedeterministický konečný automat s více počátečními stavy

Nedeterministický konečný automat s více počátečními stavy je nedeterministický konečný automat, který místo počáteční stavu q_0 obsahuje množinu počátečních stavů I. Automat v tomto případě vyzkouší výpočet započít jednotlivě v každém počátečním stavu. Automat vstup přijme, pokud alespoň jeden z těchto možných výpočtů vstup přijímá.

2.2.6 Nedeterministický konečný automat s více počátečními stavy a ϵ-přechody

Nedeterministický konečný automat s více počátečními stavy a ϵ-přechody je automat, který je zároveň nedeterministický konečný automat s ϵ-přechody a nedeterministický konečný automat s více počátečními stavy.

2.3 Pipes-And-Filters

Pipes-And-Filters je architektonický styl umožňující rozdělit složité výpočetní úkony do sekvence menších nezávislých výpočetních kroků (filtry) [8]. Každý filtr přijímá vstup pomocí roury (pipe). Po vyhodnocení filtru lze jeho výstup řetězit dál pomocí výstupní roury. To umožňuje vytvářet sekvence jednotlivých filtrů, které občas bývají označovány jako pipeline. Výhodou tohoto přístupu je nezávislost filtrů na sobě, což umožňuje filtry přepoužívat v jiných sekvencích a zároveň tento přístup zvyšuje modulárnost.

Kapitola 3

Analýza

Cílem této kapitoly je analýza podobných aplikací a následná analýza Statemakeru a ALT, které výsledná aplikace bude využívat. Na základě těchto analýz jsou pak sestaveny funkční a nefunkční požadavky.

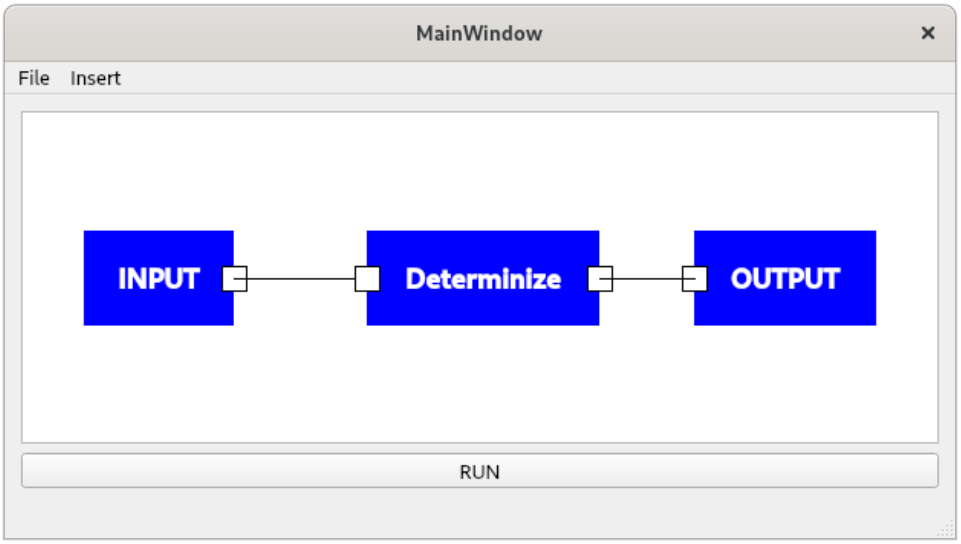
3.1 Podobné aplikace

Tato podkapitola se věnuje analýze podobných aplikací, z nichž nejpodobnější je ALT agui2, ke které bude výsledná aplikace alternativou. Agui2 je také jedinou aplikací, která je podobná funkcionalitou. GNURadio Companion je analyzován více se zaměřím na GUI.

3.1.1 ALT agui2

Agui2 je GUI pro ALT napsané v jazyce C++ využívající Qt framework [9], které je přímo součástí ALT. Je teprve v rané fázi vývoje a proto nabízí jen velmi omezenou funkcionalitu. Umožňuje uživateli kreslit acyklický orientovaný graf, který znázorňuje pořadí vyhodnocování algoritmů a řetězení jejich výsledků. Vrcholy v tomto grafu mohou reprezentovat buď vstup, algoritmus a nebo výstup. Hrany pak reprezentují předávání hodnoty dalším algoritmům. Aplikace zatím podporuje pouze algoritmy nad konečnými automaty, gramatikami a regulárními výrazy. Všechny algoritmy v agui2 přijímají pouze jeden vstup.

Hlavním prvkem GUI je plátno na které uživatel kreslí graf, jehož podoba je vidět na obrázku 3.1. V něm může uživatel tažením myši buď vrcholy přemisťovat, nebo propojovat. Toto ovládání je značně intuitivní, nicméně uživatel takto nemůže hýbat s plátnem samotným. K tomuto slouží scrollbary, které se zobrazí na kraji plátna, pokud je plátno dostatečně velké. To může uživateli znepříjemnit ovládání aplikace, pokud bude pracovat s grafem s velkým počtem vrcholů. Hrany se na plátně automaticky zalamují do pravoúhlých ohybů, aby byly vizuálně přehlednější.



Obrázek 3.1: Ukázka agui2

Vrcholy se přidávají pomocí nástrojové lišty. Zde jsou v kontextovém menu kategorizovány algoritmy dle jejich jednotlivých jmenných prostorů. Bohužel už neexistuje možnost, jak přidat výstupní vrchol a ani neexistuje možnost vrcholy či hrany odstraňovat.

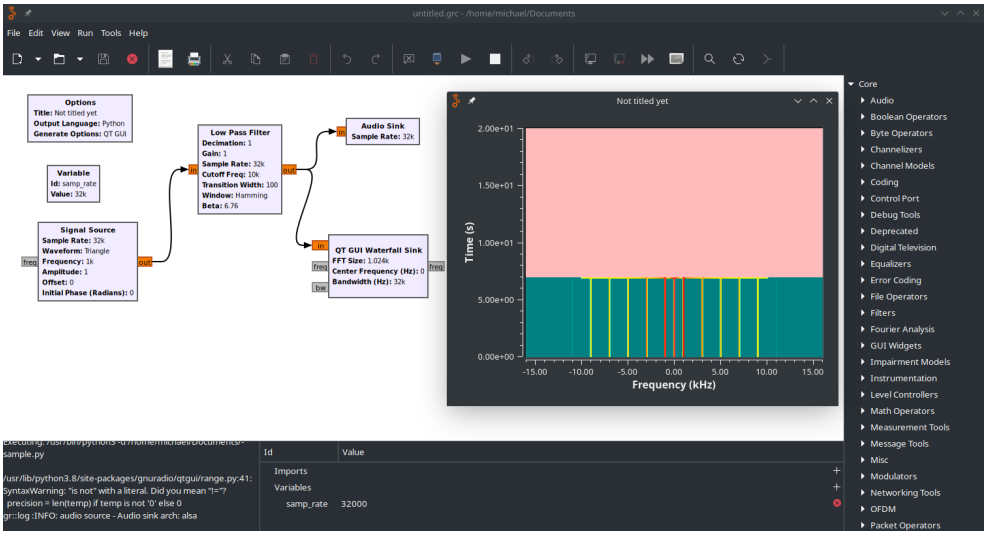
Zadávání vstupů je realizováno přes dialogové okno, které se otevírá přes kontextové menu vstupního vrcholu. V tomto okně uživatel může ručně zadat vstup, nebo je zde možnost načíst vstup ze souboru. Aplikace umí přijímat vstupy ve specifických textových formátech ALT [10] pro konečné stavové automaty, gramatiky a regulární výrazy. Zároveň umí ještě přijímat vstupy ve formátu XML [11] a nabízí možnost vygenerování náhodného konečného automatu nebo gramatiky pomocí algoritmů, které jsou zabudovány v ALT.

Výstupy se zobrazují ve stejném dialogovém okně jako vstupy, tudíž jsou buď textové, nebo ve formátu XML. Aplikace nadále umí graficky vykreslit konečné automaty pomocí nástroje Graphviz [12], nicméně vykreslený výstup nelze nijak kopírovat či uložit.

Celý graf lze uložit do JSON [13] souboru pro pozdější načtení.

3.1.2 GNURadio Companion

GNURadio Companion [14] je nástroj, který umožňuje pomocí stavebních bloků pro zpracovávání signálů vyvíjet softwarová rádia. Cílem aplikace je nahrazovat hardwarové obvody pro zpracovávání různých signálů softwarem. Tento nástroj není co se týče funkcionality nijak podobný, ale jeho GUI, které znázorňuje architekturu Pipes-And-Filters, stojí za zmínění.



Obrázek 3.2: Ukázka GNURadio Companion

Opět jako spousta dalších aplikací znázorňující Pipes-And-Filters architekturu, hlavním prvkem aplikace je plátno, na které uživatel kreslí DAG. Zde jednotlivé vrcholy představují nějakou transformaci signálu. Vrcholy jsou vykreslovány jako boxy a prezentují uživateli rovnou podstatné informace, bez toho, aniž by s nimi musel jakkoliv interagovat. Hrany aplikace vykresluje jako křivky.

GNURadio Companion oproti agui2 vykresluje komplexnější grafy, nějaké bloky zde mohou přijímat více vstupů a zároveň mohou produkovat více výstupů. Toto je z hlediska GUI problém, jelikož musí uživateli dostatečně srozumitelně rozlišit jednotlivé vstupy/výstupy od sebe. GNURadio proto zobrazuje jméno vstupu/výstupu přímo v pojném bodě. Toto řešení ovšem omezuje délku jména daného vstupu/výstupu, to může nutit vývojáře bloků ke zkracování jmen, což může mást uživatele.

Každému jednotlivému bloku lze nastavit jeho vlastnosti. Toho je docíleno přes dialogové okno, které se liší pro každý druh bloku. V tomto okně je možné nastavit různé parametry, které ovlivňují chování bloku. Dále aplikace zde poskytuje možnost napsat komentář ke konkrétnímu bloku a také je zde i záložka s dokumentací. Nastavení celého grafu je obsaženo v samotném bloku na plátně, které lze výše popsaným způsobem upravovat. To samé platí například i pro proměnné.

3.1.3 Souhrn

Podobné aplikace umožňují stavět pomocí stavebních bloků vyhodnocovací graf na plátně, které je hlavním ovládacím prvkem obou aplikací. Sekundární

ovladácí prvky zobrazují pomocí postranních lišt. Ke komunikaci dalších informací využívají dialogová okna, která mohou být i specializovaná pro zobrazení hodnoty konkrétních datových typů.

3.2 Analýza ALT

ALT je knihovna poskytující algoritmy a datové struktury z oblasti teorie automatů, stringologie, arbologie a teorie grafů. Celá knihovna se skládá z několika menších C++ knihoven, které tvoří jednotlivé komponenty ALT.

Součástí komponenty `alib2abstraction` je registr, který zpřístupňuje informace o samotné knihovně. Skrze něj je možné získat informace o jednotlivých algoritmech a jejich přetíženích, či pokud ji mají, tak i jejich dokumentaci. Dále nabízí možnost získat seznam jednotlivých implicitních a explicitních konverzí všech datových typů, které ALT využívá.

ALT umožňuje pomocí statické metody `evalAlgorithm` třídy `EvalHelper` vyhodnocovat algoritmy dynamicky pouze na základě jejich jména. To je v kombinaci s registrem velice užitečná funkcionalita, protože v případě dynamického vyhodnocování algoritmů odpadá nutnost vytvářet rozhodovací konstrukce, které by určovaly, jaký algoritmus vyhodnotit. Tato třída je primárně využívána k vyhodnocování příkazů v aql2, ovšem nic nebrání ji využít i jinde. Parametry algoritmů a jednotlivé výsledky jsou reprezentovány pomocí abstraktní třídy `abstraction::Value`. Pro vytváření konkrétních hodnot, které lze předat dál algoritmům existuje třída `abstraction::ValueHolder`, jež je potomkem `abstraction::Value`.

ALT dále nabízi i systém pro vypisování hodnot. Vypsání hodnoty je zrostředkováno třídou `abstraction::ValuePrinterAbstraction`. Tato třída je potomkem `abstraction::OperationAbstraction`, protože z hlediska ALT je vypsání hodnoty operace mezi hodnotou a streamem. Vyhodnocení této operace způsobí vložení stringové reprezentace hodnoty do streamu, to umožňuje hodnoty převádět do textové podoby, ovšem ne všechny datové typy ALT to podporují.

3.2.1 Shrnutí

ALT umožňuje pomocí registru získat informace o všech algoritmech, podle kterých lze libovolné algoritmy volat metodou `evalAlgorithm`. K reprezentaci hodnot využívá ALT abstraktní třídu `abstraction::Value`, jejichž hodnotu lze vypsat pomocí `abstraction::ValuePrinterAbstraction`. Všechny tyto třídy jsou součástí komponenty `alib2abstraction`.

3.3 Analýza Statemakeru

Statemaker [5] je webová aplikace sloužící k vytváření konečných stavových automatů vytvořena v rámci bakalářské práce Bc. Petra Svobody [4]. Aplikace je napsána v jazyce TypeScript [15] a postavena pomocí JavaScriptové knihovny React [16]. UI Statemakeru vychází z designového systému Material Design [17], k tomu také hodně vužívá knihovnu React komponent Material-UI [18]. Aplikace využívá state management knihovnu Redux [19], která umožňuje mít jeden centralizovaný stav v celé aplikaci. Ten je dobře testovatelný a debugovatelný díky vývojářským nástrojům, poskytovanými přímo vývojáři knihovny.

Hlavním UI prvkem aplikace je SVG [20] plátno na které se kreslí orientovaný graf reprezentující konečný stavový automat. Vytváření automatu je zde velice intuitivní, při vytváření stavu aplikace vykresluje náznak nového stavu pod kurzorem, to uživateli dává dobře najevo, že zrovna vytváří stav a zároveň mu tento stavěcí mód umožňuje rovnou nový stav napozicovat. Plátno dále umí základní operace, jako jsou přiblížení, oddálení a posun celého plátna. Pro přehlednější pozicování poskytuje mód `grid`, ve kterém se přes plátno vykreslí mřížka, podle které se lépe pozicují stavy.

Aplikace umí pracovat s několika formáty souborů, pro které podporuje importování a exportování konečného automatu. Prvním z nich je JSON formát specifický pro Statemaker. Dalším formátem je JSON formát aplikace XState [21], což je JavaScriptový vizualizační nástroj pro stavové automaty a stavové diagramy. Poslední dva formáty jsou čistě textové, jedním z nich je dle slov autora „Formát zvolený pro jeho jednoduchost a snadnou možnost úpravy“, a druhým z nich je textový formát ALT pro popis konečných stavových automatů, který je v aplikaci nazýván jako FIT textový formát [22].

FIT textový formát je z hlediska aplikace nejužitečnější, protože jej umí přijímat ALT. Ovšem Statemaker tento formát podporuje jen z malé části a pro naše potřeby je nepoužitelný. Statemaker neumí rozlišovat mezi typy jednotlivých automatů. Soubory FIT textového formátu automatu začínají identifikátorem typu automatu. Zde nastává problém, protože Statemaker všechny automaty klasifikuje identifikátorem `FA`, jež není podporován ALT. Identifikátory pro typy konečných automatů jsou popsány v tabulce 3.1. Mimo identifikátor Statemaker správně exportuje nedeterministické automaty. Problém ale nastává při importu nedeterministických automatů. S tím autor aplikace nepočítal a importování přechodů automatu selže. Díky modulárnímu systému pro import a export není problém moduly pro FIT textový formát upravit, aby fungovaly korektně.

Aplikace dále umí automaticky pozicovat stavy automatu. K tomu nabízí dva algoritmy: *force-directed* a *hierarchical*. Tyto algoritmy se snaží co nejkompaktněji a nejpřehledněji pozicovat automat. Tato funkcionalita je obzvlášť užitečná při importování automatu z formátů souborů, které neposkytují žádnou informaci o vizuální podobě automatu.

Typ konečného automatu	Identifikátor typu
Deterministický konečný automat	DFA
Nedeterministický konečný automat	NFA
Nedeterministický konečný automat s ϵ-přechody	ENFA
Nedeterministický konečný automat s více počátečními stavy	MISNFA
Nedeterministický konečný automat s více počátečními stavy a ϵ-přechody	MISENFA

Tabulka 3.1: Tabulka indentifikátorů typů konečných automatů ve FIT textovém formátu

3.4 Funkční a nefunkční požadavky

Po diskuzi s vedoucím práce a autory ALT se dospělo k následujícím funkčním a nefunkčním požadavkům.

3.4.1 Funkční požadavky

1. Aplikace umožní zobrazit seznam všech algoritmů ALT s jejich dokumentací.
2. Aplikace umožní pokládat jednotlivé vrcholy představující vstupy, algoritmy a výstupy na plátno.
3. Aplikace umožní řetězit vrcholy na plátně.
4. Vstupy budou podporovat datové typy `int, double, boolean, string` a datové typy ALT pro konečné stavové automaty.
5. Aplikace umožní algoritmy vyhodnotit a zobrazit jejich výsledky.
6. Aplikace umožní pomocí grafického rozhraní vytvořit a zobrazit konečný automat.
7. Graf algoritmů půjde importovat/exportovat z/do souboru.

3.4.2 Nefunkční požadavky

1. K vytváření a zobrazování konečných automatů bude využita aplikace Statemaker
2. Čas na vyhodnocení grafu půjde omezit

Kapitola 4

Návrh

4.1 Zpřístupnění ALT pro webové aplikace

Jedním z hlavních problémů, kterým se tato práce zabývá, je zpřístupnění ALT pro web. Toto je nutné, aby webová aplikace mohla provést vyhodnocení algoritmů. Naskytují se zde dvě rozdílné varianty řešení tohoto problému.

4.1.1 WebAssembly

První variantou, jak řešit zpřístupnění ALT pro webové aplikace je zkompilovat jej do WebAssembly. WebAssembly [23] je binární instrukční formát, který lze použít jako kompilační cíl pro programovací jazyky vyšších úrovní. Jedním z hlavních cílů WebAssembly je zpřístupnění nativních programů a knihoven přímo ve webovém prohlížeči. Nadále si za cíl klade tento kód spouštět stejně rychle, jako kdyby běžel nativně. Tomuto výkonnostnímu cíli se zvládá přiblížit, ve většině případů lze očekávat 50–90% rychlost vůči nativnímu kódu. Co se týče paměťové velikosti kódu, WebAssembly kód je ve většině případů menší než nativní kód. Kupříkladu velikost WebAssembly kódu je průměrně 85,3% velikosti nativního x86-64 kódu [24].

Pro kompilaci ALT do WebAssembly by šel použít toolchain Emscripten [25]. Emscripten umožňuje kompilovat C/C++ kód nebo LLVM [26] bitcode do JavaScriptu a WebAssembly. Je navržen tak, aby sloužil jako substituce gcc [27]. Emscripten by měl zvládnout zkompilovat veškerý přenosný kód, který zvládne zkompilovat gcc, ovšem autoři uvádí, že často musí být kód upraven, aby byl kompilovatelný i Emscriptenem. Kód, který nelze zkompilovat je přímo autory popsán [28] a limitace by se ALT neměly nijak dotýkat.

Kompilace ALT do WebAssembly se jeví jako dobrý způsob zpřístupnění knihovny pro web, ovšem hlavní překážkou tohoto přístupu je velikost ALT, která se pohybuje okolo 60 MiB. Odhadovaná velikost výsledného WebAssembly souboru je tedy asi 51 MiB, to je pro webovou aplikaci nepřijatelné. Zároveň by bylo potřeba zkompilovat všechny závislosti ALT pomocí Em-

scriptenu, což by bylo také naročné. Z těchto důvodů se toto řešení jeví jako špatně realizovatelné, proto se dál tato práce zabývá návrhem zpřístupnění ALT pomocí webového API.

4.1.2 Webové API pro ALT

Druhou variantou řešení zpřístupnění ALT pro webové aplikace je vytvořit webové API pro ALT. Toto řešení oproti WebAssembly vyžaduje infrastrukturu na které bude spouštěno ALT. Od API webová aplikace pouze vyžaduje pouze jen funkcionalitu vyhodnotit algoritmový graf. Webové prohlížeče mohou s webovými službami komunikovat především pomocí HTTP [29] požadavků. Proto se nabízí vytvořit HTTP webové rozhraní s POST endpointem pro vyhodnocení grafu.

Jelikož v tomto přístupu bude komunikovat více klientů v podobě webové aplikace s jedním serverem, bude nutné vyřešit problém vyvážení zátěže jednotlivých požadavků. Vyhodnocení některých algoritmů i při jednoduchých vstupech může trvat hodně času. Pokud by vyhodnocovací server byl implementován naivně a nevyvažoval požadavky jednotlivých klientů, mohl by jeden klient požádat o vyhodnocení náročného algoritmu a tím zdržet vyhodnocení ostatních požadavků od jiných klientů. Tudíž bude nutné časově omezit vyhodnocení každého požadavku.

V ideálním případě by se vyhodnocení spustilo v jiném vlákně, než které vyřizuje požadavek, a při uplynutí času pro vyřízení požadavku by se výpočetní vlákno ukončilo, aby dále nezatěžovalo server. To POSIXový standard jistým způsobem umožňuje, ačkoliv s omezeními [30]. Vláknům lze nastavit dva typy způsobu zrušení, těmi jsou *asynchronous* a *deferred*. Vlákno s *asynchronous* způsobem zrušení může být v jakýkoliv moment zrušeno jiným vláknem. Rušené vlákno ovšem neuvolní jemu přidělené prostředky, tudíž například neuvolní svou alokovanou paměť na haldě. *Deferred* způsob zrušení uchovává všechny požadavky pro zrušení, dokud vlákno nenarazí na nějaký bod zrušení, kde tyto požadavky vyřídí. Tento bod zrušení nastává při určitých systémových voláních, popřípadě při volání IO funkcí. Při vyhodnocování algoritmů ALT zpravidla na tyto body zrušení nenaráží, tudíž tento problém nelze řešit vlákny.

Namísto vláken by ale šly použít child procesy, které by už násilně ukončit šly. Takové řešení by tedy využívalo skupinu pracovních procesů pro zpracování požadavků. Čas na zpracování každého požadavku by byl měřen hlavním procesem, který by je při překročení této časové lhůty ukončil. Škálování tohoto řešení by mohlo být provedeno kontejnerovým orchestračním nástrojem, jako je například Kubernetes [31].

Ještě je také možné využít nástroj pro frontu zpráv. Fronta zpráv by byla zprostředkována brokerem, který by umisťoval požadavky do fronty, ze které by byly zpracovávány workery. Workeři by pak mohli běžet odděleně od serveru a vyhodnocovali by požadavky z fronty. Pokud by worker požadavek

vyhodnocoval moc dlouho, ukončil by se. Po ukončení by prostředí, ve kterém jsou workeři spouštěni, mohlo reagovat buď restartováním workera, nebo nasazením rychlejšího workera. Toto je řešení je také dobře škálovatelné, protože pokud by bylo požadavků příliš, stačí spustit nové workery. Po konzultaci s vedoucím byl zvolen tento přístup.

4.2 Webová aplikace

Protože webová aplikace do sebe cílí integrovat Statemaker, jeví se jako logický krok webovou aplikaci postavit na jeho základu. Statemaker primárně řeší kreslení grafu reprezentující automat na plátno. To je velice podobný problém, jakým se zabývá tato práce, proto je dobrý nápad pokusit se toto řešení, pokud možno přepoužít, ačkoliv se kód Statemakeru bude muset zobecnit. To také znamená, že využité technologie se odvíjí ze Statemakeru. Webová aplikace proto bude psána v jazyce TypeScript [15], bude využívat knihovnu React [16] a knihovnu pro state management Redux [19]. Pro vizuální konzistenci se Statemakerem bude využívat knihovnu komponentů Material-UI.

4.2.1 JavaScript

Hlavním jazykem pro vývoj webových aplikací je JavaScript [32]. JavaScript je dynamicky typovaný jazyk, který byl primárně určen pro vytváření interaktivních webových stránek.

V roce 2009 byl nadále zpopularizován projektem Node.js [33], jež vytvořil C++ wrapper kolem V8 JavaScript enginu [34], který umožnil využít JavaScript i jinde než v prohlížeči. Node.js nově umožnilo z JavaScriptu přistupovat k filesystému, vytvářet sockety pro komunikaci přes síť a spoustu dalších, do té doby pro JavaScriptové aplikace neuskutečnitelných věcí. Nejen díky tomu je dnes JavaScript nejpopulárnějším programovacím jazykem [35].

To ale neznamená, že JavaScript nemá podstatné nevýhody a limitace. Podstatnou z nich je to, že JavaScript beží pouze v jednom vlákně. To v kombinaci s tím, že JavaScript je interpretovaný jazyk, zásadně omezuje jeho rychlost. Tento problém se snaží řešit pomocí Web Workerů [36], nicméně ty nejsou plnohodnou náhradou za spouštění kódu ve více vláknech. Další nevýhodou, která pro jednoduché aplikace je spíše výhodou, je dynamické typování. To s větší komplexností aplikace způsobuje problémy s jeho údržbou a rozšiřováním. Tento problém se snaží řešit nástroj na statickou analýzu Flow [37] a jazyk TypeScript.

4.2.2 TypeScript

TypeScript [15] je staticky typovaný jazyk od firmy Microsoft. Je navržen jakožto nadmnožina JavaScriptu, díky tomu pro JavaScript vývojáře je jednoduché se ho naučit a zároveň lze JavaScript kód lehce přemigrovat do Ty-

peScriptu. Jak už jeho název napovídá, jeho hlavním cílem je zavést statickou kontrolu typů do JavaScriptu. TypeScript se většinou kompiluje do JavaScriptu, nicméně ho lze intepretovat pomocí nástrojů deno [38] či ts-node [39], nebo ho také lze i kompilovat do WebAssembly [40]. Sám TypeScript nepřináší žádné runtime nároky, slouží spíše jako nástroj pro statickou analýzu. Co se týče funkcionality, TypeScript má podporu pro generika, dekorátory, smartcasting, nebo například podporu pro sjednocení typů a pro další složitější datové typy.

4.2.3 React

React [16] je dnes nejpopulárnější JavaScriptová knihovna pro vytváření uživatelských rozhraní [41] od firmy Facebook. Hlavní ideou Reactu je skládání UI do jednotlivých komponent, které se překreslují pokaždé se změnou stavu aplikace.

Komponenty umožňují rozdělit UI do nezávislých oddělených celků, které se mohou přepoužívat na různých místech v aplikaci. Každá komponenta se může skládat z dalších komponent nebo z HTML elementů. HTML elementy ovšem React nevykresluje hned, nejdříve je vykreslí do Virtual DOM [42] pro zjištění, které prvky má překreslit. Tato optimalizace existuje, protože manipulace s DOM elementy je pomalá, proto se vyplatí nejdříve vykreslit komponenty do Virtual DOM, a poté zjistit které komponenty potřebují překreslit, porovnáním s předchozím Virtual DOM z posledního překreslení. Rodičovské komponenty mohou předávat svým potomkovským komponentům stav, callbacky, či jiné hodnoty pomocí props. Props jsou předávány při každém překreslení. React podle nich určuje, zda a jaké komponenty je nutné překreslit.

Komponenty lze vykreslit pomocí funkce `React.createElement`, ovšem to při složitějším skládání komponent může být značně nepřehledné. Proto React přišel s deklarativním jazykem pro vytváření komponent nazvaným JSX [43]. JSX umožňuje vytvářet komponenty pomocí syntaxe podobné HTML. Zároveň ale k tomu umožňuje psát i JavaScriptový kód. To umožňuje velice čitelně skládat komponenty do sebe.

React nabízí dva styly psaní komponent. Komponenty mohou být třídy dědící z `React.Component`, anebo to mohou být funkcionální komponenty, které jsou funkce přijímající props a vracející vykreslené komponenty. Třídní komponenty byly hlavním stylem psaní komponent do verze Reactu 15.8, ve které React vydal hooks. Hooks umožňují funkcionálním komponentám vyrovnat se funkcionalitě třídních komponent. Pomocí hooks mohou funkcionální komponenty mít stav, reagovat na překreslení komponenty, konzumovat React context a mnohé další. Hooky lze zároveň skládat dohromady a tím vytvářet nové hooky.

4.2.4 Redux

Redux [19] je knihovna pro state management JavaScriptových aplikací, inspirovaná architekturou Flux [44]. Hlavním cílem Reduxu je být lehce testovatelný a předvídatelný stavový kontejner. Redux přichází se třemi základními koncepty: store, reducer a akce. Stav aplikace je immutable objekt zprostředkováván pomocí store. Tento stav je znovu sestaven s každou akcí. Akce jsou vytvářeny aplikací, většinou například pomocí uživatelem vyvolanými událostmi. Slouží primárně k vyvolání změny stavu a následného předání hodnot do reduceru. Reducer je čistá funkce, která přijímá akci a stav, vytvářející na základě těchto dvou parametrů nový stav. Reducer také vytváří původní stav při startu aplikace. Stav aplikace je tudíž definován reducerem a protože reducer je čistá funkce, lze jednoduše předvídat a testovat jeho chování. Celý tok dat je v Reduxu jednosměrný, toto je vidět na obrázku 4.1. Za antipattern je například považováno předávat callback do Reduceru, protože tím se obrací tok dat a aplikace se tím stává méně předvídatelná.

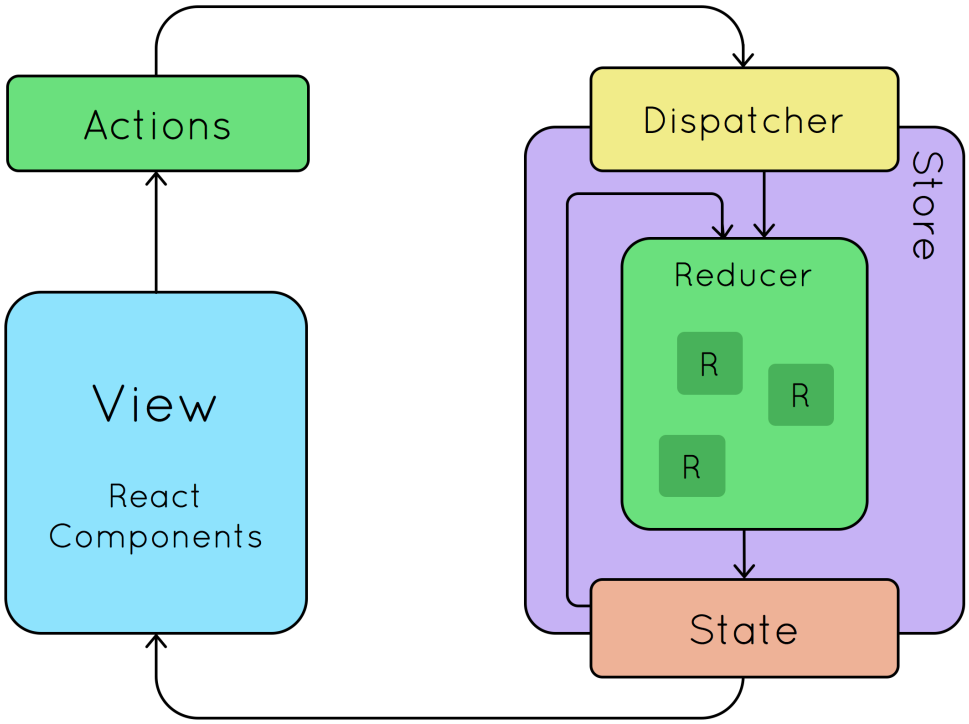
Redux je velmi často využíván v React aplikacích. Pro jednodušší aplikace může být spíše přítěží, ovšem pro složitější aplikace může být velkým přínosem co se týče architektury, čistoty kódu, či testovatelnosti. Pro React aplikace existuje knihovna react-redux [45], která se stará o propojení Reactu s Reduxem. Jedním z problémů komplexních React aplikací může být sdílení stavu mezi vzdálenými komponenty. V Reactu se tento problém řeší předáváním callbacků pomocí props, to ale při komplexní hierarchii komponent je velice neudržitelné. Tento problém se dá vyřešit pomocí React contextu, ovšem toto řešení by postrádalo výkonnostní optimalizace, které přináší react-redux, a testovatelnost Reduxu.

4.3 API server

API server bude sloužit jako webové rozhraní pro webovou aplikaci k vyhodnocování algoritmů. Bude sloužit jako vrstva mezi brokerem a webovou aplikací. Jednotlivé požadavky bude předávat brokerovi do fronty zpráv k vyhodnocení. Z fronty zpráv dále budou požadavky zpracovávány odděleně bežícími workery.

4.3.1 Fronta zpráv

Fronta zpráv zprostředkovává komunikaci mezi různými aplikacemi. Pro tuto službu existuje spoustu různých řešení, ovšem ne všechny, jako například populární RabbitMQ [47], podporují oficiální C++ klienty. Z toho důvodu bylo zvažováno mezi řešeními, které C++ oficiálně podporují. Těmi jsou například ZeroMQ [48], Apache ActiveMQ [49] a Apache Qpid [50]. Většina brokerů fronty zpráv podporuje více messaging protokolů, jako jsou například

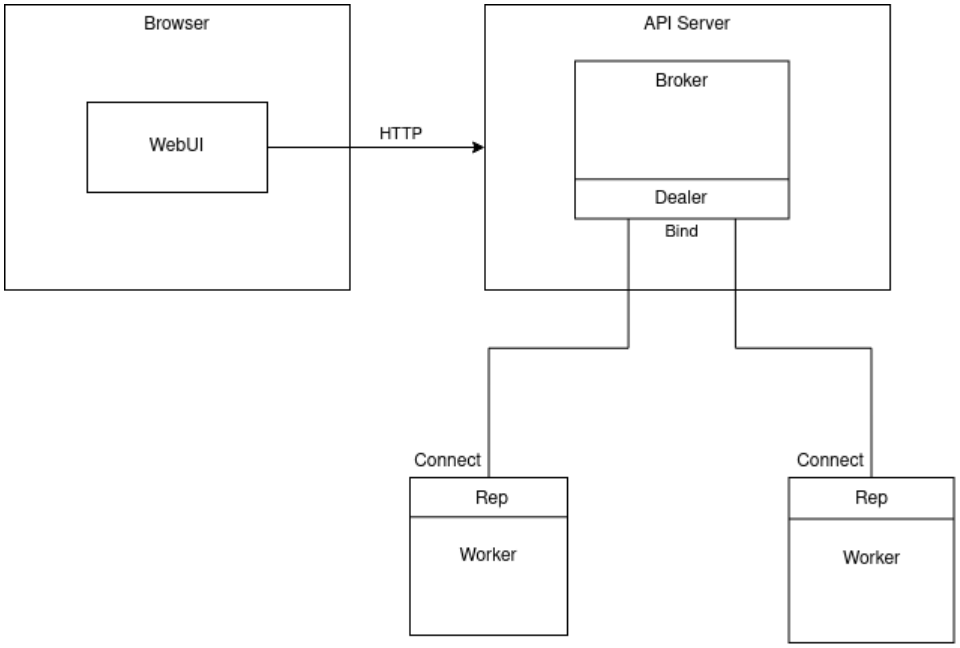


Obrázek 4.1: Diagram znázorňující tok dat v Reduxu [46]

AMQP [51] či STOMP [52], nicméně i přesto byl při výběru kladen důraz na dostupnost oficiální klientské knihovny pro C++.

První návrh využíval ZeroMQ. ZeroMQ je knihovna napsaná v jazyce C, zprostředkovávající frontu zpráv bez dedikovaného brokera. Základním stavebním blokem ZeroMQ jsou sockety, které se dělí na několik různých typů. Nejjednoduššími sockety jsou Req a Rep sockety, které zprostředkovávají jednoduchý Request-Reply pattern. Req a Rep nemohou přijímat zprávy asynchronně a musí střídavě přijímat a poté odesílat zprávy. Pro asynchronní rozesílání zpráv slouží Dealer socket. Ten umožňuje rozeslat libovolný počet zpráv a může přijímat libovolný počet odpovědí. ZeroMQ nabízí ještě další sockety zprostředkovávající Publish-Subscribe patern, Pipeline (Push-Pull) patern a mnohé další. Všechny tyto typy socketů se mohou kombinovat mezi sebou pro vytváření komplexních komunikačních schémat.

Každý typ socketu se může buď připojit na adresu jiného socketu anebo se může vázat na adresu hosta, což umožňuje jiným socketů se k němu připojit. V případě našeho návrhu by server využíval vázaný Dealer socket, pomocí kterého by distribuoval požadavky workerům. Workeři by se k server připojovali pomocí Rep socketů. Tento návrh je vidět na obrázku 4.2.

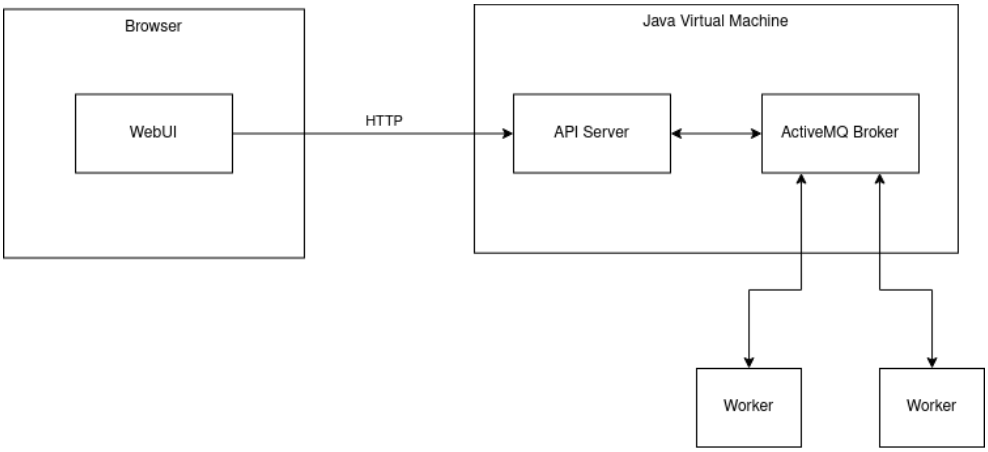


Obrázek 4.2: Návrh vužívající knihovnu ZeroMQ

Problém v tomto návrhu je v implementaci ZeroMQ, která nenabízí žádný způsob, jak se vypořádat s pomalým konzument zpráv, což by v případě navrhované aplikace byl worker zpracovávající náročný požadavek. To je způsobeno tím, že v případě vázaného socketu ZeroMQ vytváří oddělené fronty pro všechny připojené sockety, kterým jsou zprávy přidělovány rovnoměrně. To vytváří problém, pokud by jeden worker vyhodnocoval náročný požadavek, protože by mu server přiděloval nové požadavky do fronty, místo toho, aby je přidělil workerům, kteří by v ten moment nic nevyhodnocovali. Tato knihovna je díky tomu pro tento případ užití neefektivní.

Místo ZeroMQ bylo pro implementaci fronty zpráv zvoleno Apache ActiveMQ. ActiveMQ je message broker pro distribuci zpráv napsaný v jazyce Java [53]. Má také oficiální klientskou knihovnu pro C++ [54]. ActiveMQ umožňuje vytvářet nejen fronty zpráv a zároveň podporuje širokou škálu komunikačních protokolů.

Oproti ZeroMQ umožňuje řešit problém pomalého konzumenta. ActiveMQ umožňuje sdílet jednu frontu zpráv s více konzumenty a zároveň umožňuje nastavit takzvaný prefetch, který určuje kolik zpráv může konzument mít u sebe v jeden moment. Toto je nutné nastavit na hodnotu pouze jedné zprávy. V opačném případě by worker mohl obdržet z fronty více požadavků, což by bylo nežádoucí v momentě, kdy se worker ukončí z důvodu vypršení časové lhůty na vyhodnocení požadavku, protože by to způsobilo ztrátu ještě nevyhodnocených požadavků. Zároveň je ještě nutné vytvořit druhou frontu



Obrázek 4.3: Návrh využívající Apache ActiveMQ

pro odpovědi. Ta bude konzumována serverem, který bude výsledky odesílat konkrétním klientům.

ActiveMQ lze také spustit vestavěně v Java Virtual Machine (JVM), skrz kterou může komunikovat pomocí intra-vm protokolu [55]. Proto je vhodné API server napsat v jazyce běžící na JVM, tím se odstraní síťová komunikace mezi API serverem a brokerem. Komunikace jednotlivých komponent je zobrazena na obrázku 4.3.

4.3.2 Kotlin

Kotlin [56] je moderní programovací jazyk vyvinutý firmou Jetbrains. Původně byl vyvinut jako moderní alternativa k jazyku Java. Díky tomu byl navrhnut tak, aby byl 100% interoperabilní s Javou. Jinými slovy lze všechen Kotlin kód používat v Javě a naopak. To mu umožňuje využívat bohatý ekosystém Java knihoven s novou funkcionalitou, kterou oproti Javě přináší. Nejpodstatnějšími výhodami, které má Kotlin oproti Javě jsou například null safety, lambda funkce, extension funkce, typová inference a mnohé další. Kotlin lze mimo JVM bytecodu také kompilovat do JavaScriptu nebo do nativního kódu.

Kotlin byl kvůli jeho výhodám oproti Javě zvolen jako jazyk, ve kterém bude implementován API server.

4.3.3 Ktor

Ktor [57] je Kotlin framework pro vytváření asynchronních serverů a klientů. Umožňuje jednoduché a rychlé vytvoření serveru a zároveň je dobře škálovatelný, diky své modulárnosti. Je zároveň vyvíjen firmou JetBrains, tudíž má

velice dobrou podporu. Díky své jednoduchosti a rozšiřitelnosti je použit pro implementaci HTTP rozhraní.

4.4 Worker

Worker bude konzumovat požadavky z fronty požadavků a následně je bude vyhodnocovat. Kvůli vyhodnocení bude muset mít přístup k ALT, proto se pro jeho implementaci musí zvolit jazyk, ve kterém lze ALT použít, nebo do kterého ho lze přeportovat. Nejjednodušší způsob je workera implementovat ve stejném jazyce jako využívá ALT, jímž je C++.

Tvůrce C++ Bjarne Stroustrup ve své přednášce *The Essence of C++* [58] uvádí, že C++ primárně cílí na systémové programování, vestavěné systémy, a jiné systémy s omezenými prostředky. Z tohoto důvodů spíše než, aby usnadnilo programátorovi práci, cílí na výkonnost. Proto taky není nejvhodnějším jazykem pro implementaci workera, kde výkonnost není až tak důležitá. Jediná část workera, kde je výkonnost důležitá je vyhodnocování konkrétních algoritmů, které ale už jsou implementovány v C++.

C++ kód lze volat i z jiných vyšších programovacích jazyků. Například Node.js [33] umožňuje vytvářet C++ moduly volatelné z JavaScriptu. K tomuto slouží N-API [59], umožňující vytvářet nativní moduly pro Node.js. Také například Python umožňuje několik způsobů, jak volat C++ kód [60, 61, 62]. Toto přeportování by ale mohlo být do budoucna náročné na údržbu. Nakonec tedy bylo rozhodnuto ve prospěch C++.

4.4.1 CMake

Pro sestavování C++ projektů bude využit CMake [63]. CMake zjednodušuje sestavování C++ projektů generováním Makefilů, nebo také slouží také jako konfigurace pro build tool Ninja. Je vhodný pro komplexní projekty, protože umožňuje najít nainstalované knihovny a správně je zkompilovat a slinkovat. ALT nabízí CMake modul[1], který usnadňuje integraci se CMake. To je hlavní důvod, proč byl zvolen CMake před ostatními nástroji.

4.5 Docker

Docker je nástroj pro kontejnerovou virtualizaci [64]. Virtualizace je využívána na serverech ke spuštění více aplikací na jednom stroji. Umožňuje každou aplikaci spustit ve vlastním image, tím je oddělena od ostatních aplikací běžících na serveru. Díky tomu, že každá aplikace má vlastní image se všemi potřebnými knihovnami, je tak umožněno aplikace spustit pokaždé ve stejném prostředí. Dále virtualizace také slouží k abstrakci hardwaru.

[1] CMake modul se v ALT nachází na cestě `extra/FindALT.cmake`

Ovšem hypervisorová virtualizace, která byla od začátku tisíciletí standardem, nese s sebou spoustu nevýhod. Protože emuluje hardware, každý image spouští novou instanci operačního systému se všemi potřebnými knihovnami, což může být výkonnostně i paměťově náročné. Díky tomu jsou také image velké, typicky několik GiB.

Kontejnerová virtualizace, nebo také OS-level virtualizace, emuluje namísto hardwaru operační systém. To umožňuje přepoužít hostovské jádro operačního systému pro všechny běžící image. To oproti hypervisorové virtualizaci snižuje výkonnostní a paměťové nároky, a protože image nemusí obsahovat celý operační systém, jsou také menší. O běh kontejnerů se stará container engine.

Docker není jediný nástroj pro kontejnerovou virtualizaci, nicméně díky své popularitě nabízí široké spektrum imagů různých operačních systémů. Také ho už zároveň využívá ALT pro nasazování různých projektů. Z těchto důvodů bude Docker použit pro vytvoření imagů pro jednotlivé komponenty, které budou poté nasazovány.

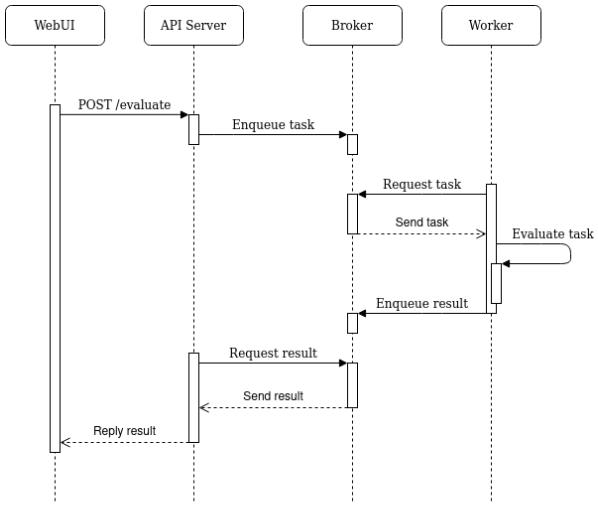
Kapitola 5

Implementace

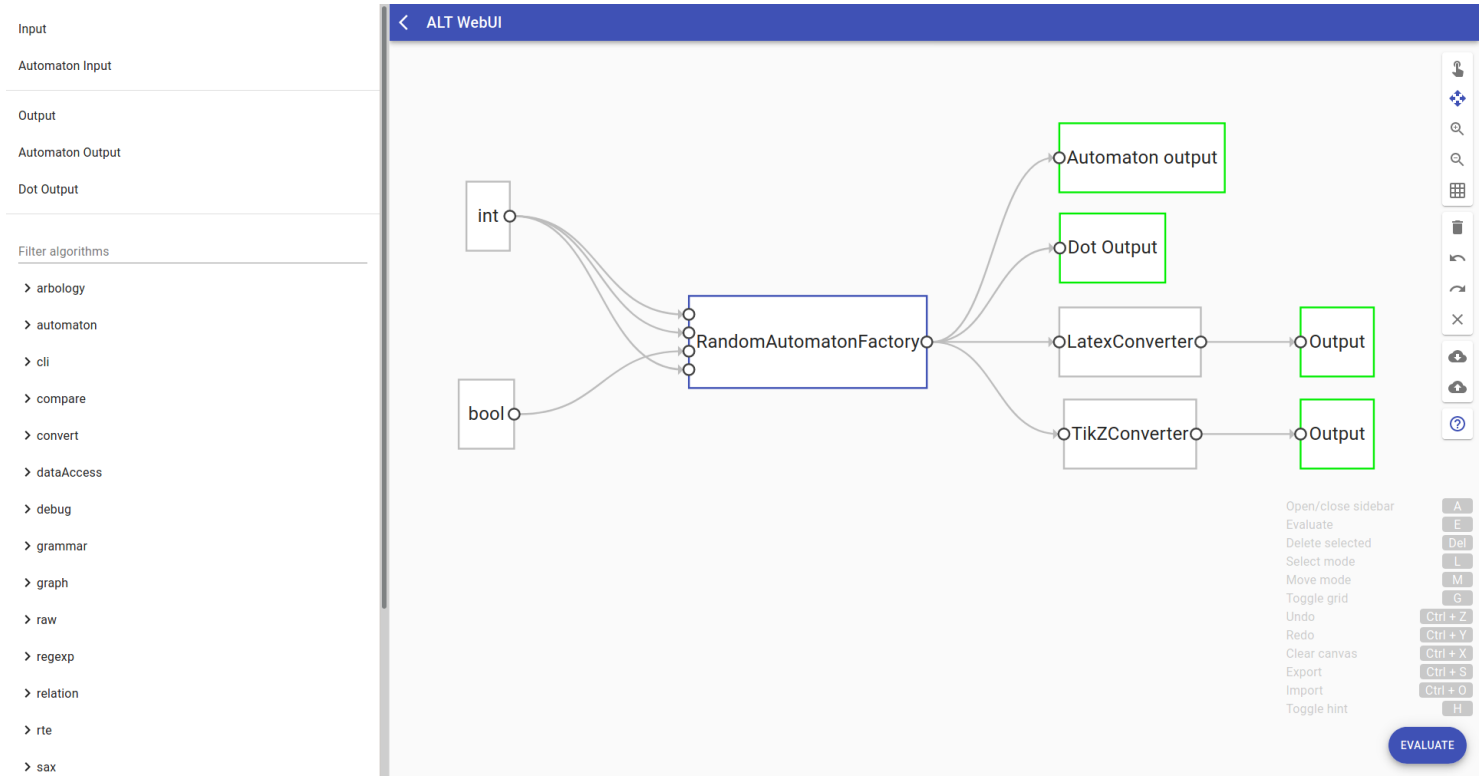
Celá aplikace se skládá ze tří podprojektů: webová aplikace nazvaná WebUI, API server a worker. Komunikace mezi jednotlivými komponenty při vyhodnocování je popsána na obrázku 5.1.

5.1 Webová aplikace

Webová aplikace umožňuje uživateli pokládat vrcholy představující ALT algoritmy, vstupy a výstupy a následně je řetězit. Hlavní obrazovka aplikace je vidět na obrázku 5.2.



Obrázek 5.1: UML sekvenční diagram popisující komunikaci mezi jednotlivými komponenty při vyhodnocovacím požadavku.



Obrázek 5.2: Screenshot hlavní obrazovky WebUI

5.1.1 Plátno

Hlavním prvkem webové aplikace je plátno, na které uživatel kreslí algoritmový graf. Plátno sdílí funkcionalitu Statemakeru, proto umožňuje uživateli plátno oddálit, přiblížit, či pohnout s celým plátnem. Zároveň plátno podporuje také grid mód. Statemaker využívá knihovnu redux-undo [65] pro implementaci akcí zpět/vpřed na plátně. Tato knihovna byla použita i pro algoritmové plátno.

Aplikace kontroluje, zda se v grafu nenachází cykly. Pokud je v grafu cyklus nalezen, všechny hrany jsou zvýrazněny červeně. Aplikace zároveň kontroluje, zda jsou všechny typy řetězených hodnot navzájem kompatibilní. K zobrazování chybových a informačních hlášek byla využita knihovna notistack [66], jež umožňuje vytvářet hlášky z jakékoliv komponenty.

Plátno je perzistentní, to znamená že po každé modifikaci se stav uloží do lokálního úložiště prohlížeče. Tato funkcionalita umožní uživateli po zavření aplikace navázat na rozpracované plátno, bez toho, aniž by musel jakkoliv přemýšlet nad ukládáním. Pro vývoj této funkcionality byl velice užitečný Redux, díky němuž stačilo pouze vytvořit reducer zaobalující `algorithmDataReducer`, který načítá a ukládá stav plátna z lokálního úložiště. Lokální úložiště prohlížeče ovšem není nijak určeno pro ukládání většího množství dat, proto aplikace umožňuje import a export plátna pomocí JSON souboru.

5.1.2 Side drawer

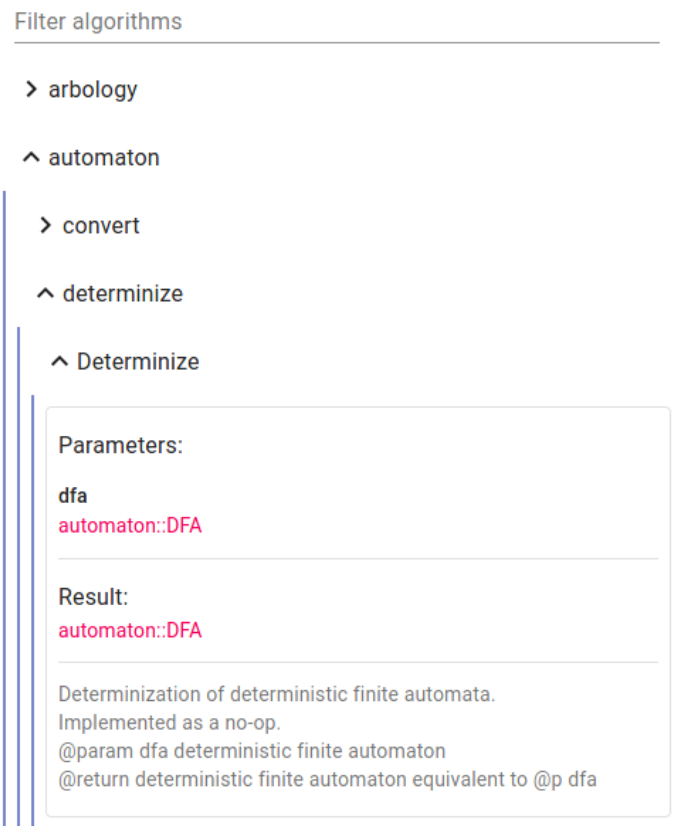
Na plátno se pokládají jednotlivé boxy pomocí side draweru. Side drawer byl zvolen z důvodu feedbacku aplikace. Při vytváření vrcholu se totiž drawer může skrýt a uživatel tak rovnou vidí náznak vrcholu na plátně. Pokud by tomu takto nebylo, nezkušený uživatel by nemusel vědět, že může vytvořit nový vrchol.

Side drawer obsahuje seznam algoritmů, jež je vyobrazen ve figuře 5.3. Zde jsou algoritmy rekurzivně kategorizovány dle jejich jmenných prostorů, podobně jako jsou kategorizovány filtry v GNURadio Companion. Aplikace umožňuje filtrovat jméno kategorie nebo algoritmu. Při filtrování aplikace vyhledává v řetězcích každé kategorie filtrovaný podřetězec. Pokud je podřetězec v kategorii nalezen, zobrazí se všechny podkategorie. Pokud podřetězec není v kategorii nalezen, ale je nalezen v nějaké podkategorii, jsou zobrazeny pouze podkategorie obsahující podřetězec.

Celý seznam je generován pomocí pomocného C++ programu. Ten čte registr ALT a na jeho základě vypíše JSON soubor obsahující seznam algoritmů včetně jejich dokumentace a implicitních konverzí všech datových typů.

5.1.3 Vstupní vrcholy

WebUI dále podporuje vstupy typů int, double, bool a string. Pro editování těchto hodnot slouží dialogové okno. U číselných hodnot aplikace automaticky

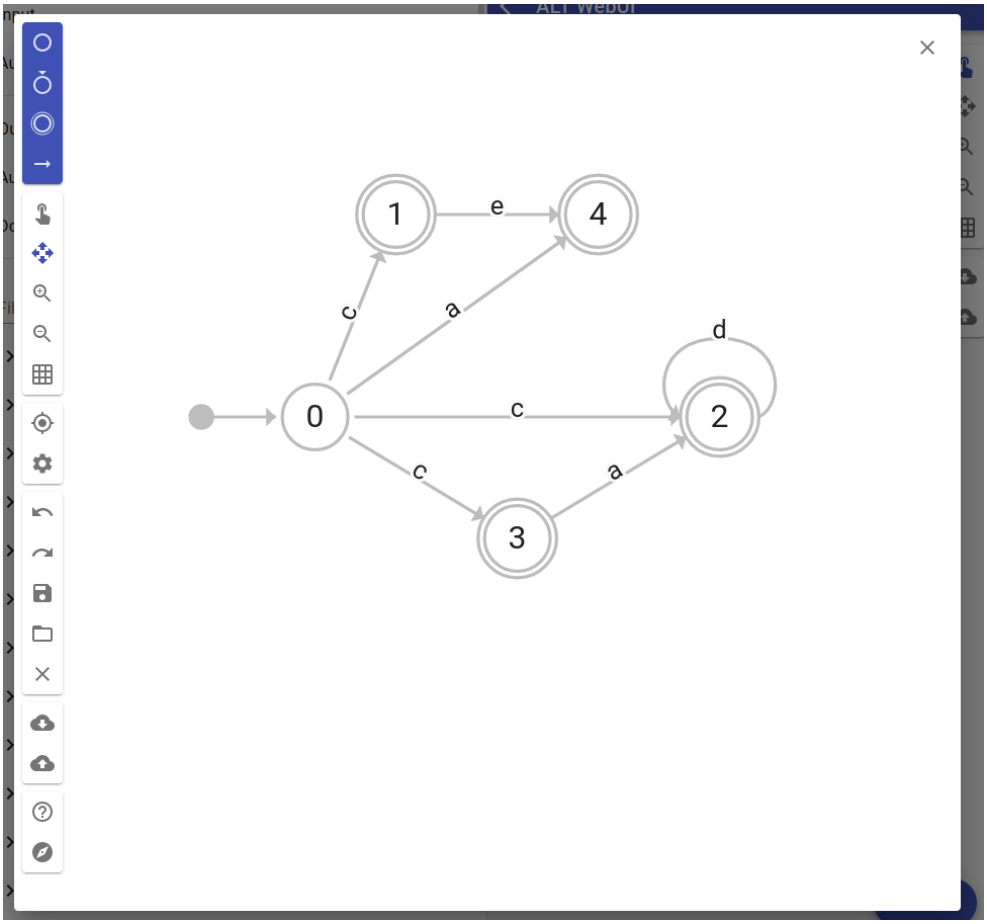


Obrázek 5.3: Detailní screenshot seznamu algoritmů ve WebUI

rozpozná, jestli se jedná o celé číslo, nebo o číslo s desetinou čárkou, dle čeho určí typ vstupu.

Vstupy dále ještě mohou být konečné automaty. Pro ty existuje speciální vstupní vrchol `Automaton Input`, který umožní uživateli pomocí aplikace Statemaker automat vytvořit. Při editování hodnoty se Statemaker otevře v dialogovém okně, které je vidět ve figuře 5.4. Po dokončení vytváření automatu, aplikace na základě něj určí výstupní typ vrcholu.

Pro implementaci tohoto, a i dalších vrcholů bylo využitu principu Pipes-And-Filters, jež umožňuje skládat jednoduché vrcholy do komplexních celků. Při vytváření vyhodnocovacího požadavku se na pozadí vytvoří nový string vstupní vrchol obsahující exportovaný automat ve FIT textovém formátu. Výstup tohoto vstupního vrcholu je předán do také nově vytvořeného vrcholu algoritmu `string::Parse`.

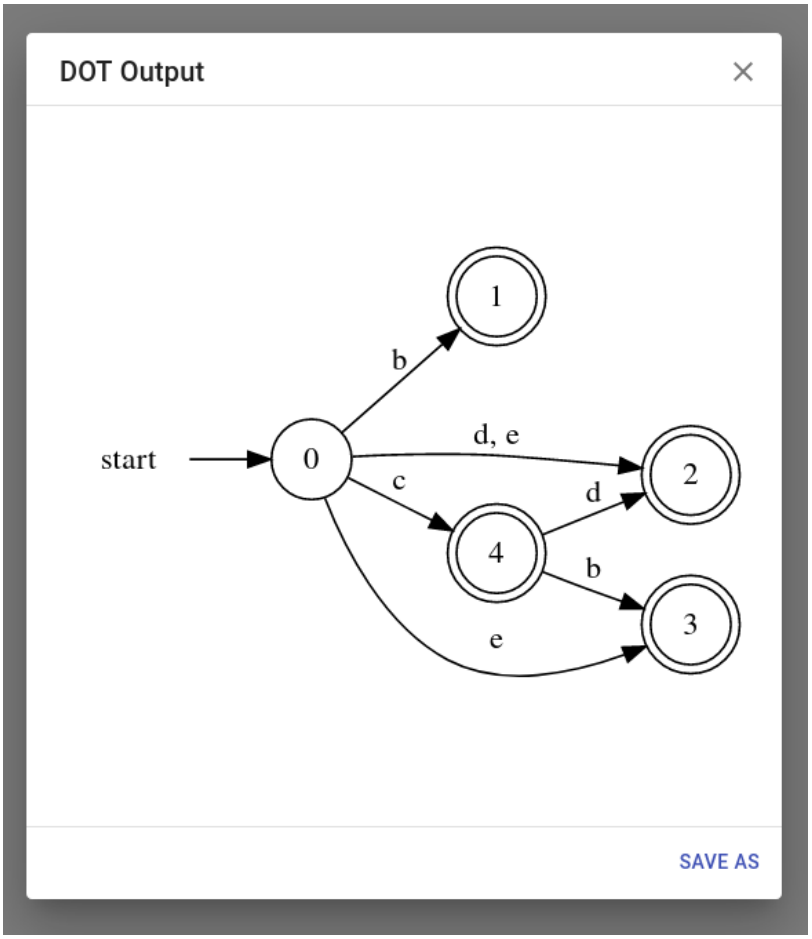


Obrázek 5.4: Aplikace Statemaker v dialogovém okně ve WebUI

5.1.4 Výstupní vrcholy

Výstupní vrchol slouží k vypsání hodnoty uživateli. Výstupní vrcholy mohou být buď obyčejné, v tom případě se aplikace pokusí vypsat hodnotu v textové podobě, nebo také mohou být jedním ze dvou specializovaných výstupních vrcholů.

Prvním specializovaným výstupním vrcholem je `Automaton Output`, který slouží k zobrazování automatu v aplikaci Statemaker. Tento vrchol se při vytváření požadavku v pozadí přetvoří ve dva vrcholy, stejně jako u `Automaton Input` vrcholu. Původní vrchol je nahrazen výstupním vrcholem, před který je připojen vrchol algoritmu `string::Compose`, jež výsledný automat vypíše ve FIT textovém formátu [10]. Všechny hrany, které vedly do původního vrcholu, jsou přesměrovány do algoritmového vrcholu. Při zobrazování automatu se textová reprezentace algoritmu importuje do Statemakeru a napozicuje se



Obrázek 5.5: DOT Output dialog

pomocí pozicovacího algoritmu *hierarchical*. Důvod, proč byl vybrán *hierarchical* před *force-directed*, je ten, že *force-directed* nepozicuje automat vždy stejným způsobem. To by bylo pro uživatele matoucí, pokud by několikrát za sebou zobrazoval totožný výstupový automat.

Druhým specializovaným výstupním vrcholem je `DOT Output`, který slouží k vykreslování a stahování DOT souborů [67]. Tento vrchol se přetváří v pozadí stejným způsobem, jako `Automaton Output`, jediný rozdíl je, že využívá ALT algoritmus `convert::DotConverter`. Výstup tohoto vrcholu je zobrazen v dialogovém okně, kde je tento výstup vykreslen pomocí knihovny `graphviz-react` [68]. V dialogu má uživatel možnost stáhnout výstup v podobě DOT souboru či SVG souboru. DOT dialog je vidět na obrázku 5.5.

5.1.5 Refaktorování Statemakeru

Základ webové aplikace byl vytvořen jako fork Statemakeru. Z důvodu přepoužití kódu ke kreslení grafu bylo nutné udělat refaktoring hooků používaných ke kreslení automatu na plátno, aby se dali přepoužít i pro kreslení algoritmového grafu.

Největší překážkou k přepoužití kódu byla vázanost událostí na celý dokument, což znemožňovalo Statemaker použít například v dialogovém okně. Tento problém byl vyřešen refaktorováním komponenty `Canvas` a hooků závislých na této komponentě. Těmi jsou `useCanvas`, `usePosition` a `useCursor`, které vytváří event listenery pro `Canvas` a jednotlivé pohyblivé komponenty na plátně.

`Canvas` komponenta se stará o renderování `svg` tagu, který představuje plátno. Tímto tagem zaobaluje komponenty reprezentující stavy a přechody, kterými jsou `StateContainer` a `TransitionContainer`. Komponenta byla refaktorována, aby šla přepoužít i pro jiná plátna. Toho bylo docíleno odstraněním závislosti na vykreslované prvky, které se nyní předávají přes children props [69].

Výše zmíněné hooky vytvářely listenery, které reagovaly na eventy celého dokumentu. To bránilo přemístění Statemakeru do dialogového okna, protože například scrollování v seznamu algoritmů způsobovalo zoom ve Statemakeru. Tento problém byl lehce řešitelný předáním reference na plátno, pro které hooky vytváří listenery.

Hook `useCursor`, který se stará pozici kurzoru na plátně, počítal pouze s tím, že plátno bude zabírat celou stránku. Jakmile plátno Statemakeru nezačínalo od levého horního rohu stránky, pohyb se stavy a přechody přestal korektně fungovat. Bylo to dáno tím, že tento hook počítal s pozicí kurzoru na stránce, nikoliv na plátně. Naštěstí existuje jednoduchý způsob, jak převést DOM souřadnice na SVG souřadnice plátna [70]. Tímto přepočítáním byl tento problém vyřešen.

Pro přepoužití funkcionality plátna byl `canvasReducer` refaktorován na `canvasReducerFactory`. To je funkce přijímající identifikátor plátna v podobě řetězce vracející reducer. Tento reducer je stejný jako původní `canvasReducer`, jen typy akcí, na které reaguje mají předpony `<identifikátor>_<typ>`, tudíž například akce zoom na plátně Statemakeru, jenž má identifikátor `state`, má typ `state_zoom`. To umožňuje přepoužít reducer jak pro plátno Statemakeru, tak pro plátno na kreslení algoritmového grafu.

Modul pro FIT textový formát byl rozšířen o klasifikaci typu automatů a o podporu nedeterministických automatů, aby byl kompatibilní s ALT.

5.2 API Server

Implementace API serveru je velice jednoduchá. Aplikace využívá Ktor framework a zároveň spouští vestavěného ActiveMQ brokera. Celá aplikace v době

psaní této kapitoly má pouze 112 řádků kódu. Aplikace vystavuje jediný HTTP POST endpoint na adrese `/evaluate`. Tento endpoint přijímá požadavky s tělem v podobě JSON objektu, který je nadále předáván do fronty požadavků. Struktura tohoto JSON objektu je zdokumentována pomocí jazyku TypeScript ve výpisu kódu 5.1. Každému požadavku, který je zařazen do fronty, je přiřazeno task ID, to se následně používá k určování, kterému klientu odpovědět. Toto ID se vkládá přímo do předávaného JSON objektu do pole `taskId`. Pro manipulaci s JSON objekty je využívána knihovna Klaxon [71].

5.3 Worker

Worker přijímá požadavky z fronty požadavků a následně je vyhodnocuje. Výsledky vyhodnocení poté předává do fronty odpovědí. Komunikace s brokerem probíhá pomocí OpenWire [72] protokolu. Knihovna activemq-cpp ještě umožňuje komunikaci pomocí STOMP protokolu, ale ten je pomalejší a má omezenou funkcionalitu. Je to dáno tím, že OpenWire je nativní komunikační protokol ActiveMQ [72]. Z tohoto důvodu se využívá oproti STOMP.

Požadavky, které worker přijímá, jsou v JSON formátu. Pro práci s JSON objekty byla využita knihovna jsoncpp [73]. Odpovědi jsou také v JSON formátu, jejich struktura je zdokumentována pomocí jazyka TypeScript ve výpisu kódu 5.2.

Algoritmový graf je reprezentovaný třídami dědících z třídy `AbstractNode`. Z té dědí třídy `InputNode`, `AlgorithmNode` a `OutputNode`, které reprezentují vstupní, algoritmové a výstupní vrcholy respektive. `AbstractNode` definuje abstraktní metodu `evaluate` sloužící k vyhodnocení vrcholu a předání hodnoty následníkům. Při vyhodnocování si vrcholy neuchovávají výsledek, namísto toho ho po vyhodnocení předávají jako parametr dalším vrcholům. Toto opatření slouží ke snížení paměťové náročnosti, to by při vyhodnocování velikých algoritmových grafů s náročnými vstupy mohl být problém. Jediná třída která, si uchovává výsledek po vyhodnocení je `OutputNode`.

Graf je sestavován třídou `NodesFromJSONBuilder` z JSON objektu. Pro jeho vyhodnocení algoritmového grafu slouží třída `AlgorithmEvaluator`. Ta pomocí metody `evaluate` vyhodnocuje vrcholy. Prochází vrcholy dle topologického uspořádání a pokud je vrchol výstupní, uloží jeho výsledek.

Ta využívá algoritmus vycházející z algoritmu DFS 5.1 k nalezení topologického uspořádání. Tento algoritmus zároveň odfiltrovává takzvané zbytečné vrcholy. Zbytečné vrcholy jsou ty, které netvoří žádnou cestu mezi vstupním a výstupním vrcholem, tudíž jejich vyhodnocení nevede k výsledku.

`NodesFromJSONBuilder` a `AlgorithmEvaluator` jsou využívány třídou `Worker`. Ta má na starost hlavně komunikaci s Brokerem a vyhodnocování požadavků. Hierarchie tříd je vidět na obrázku 5.6.

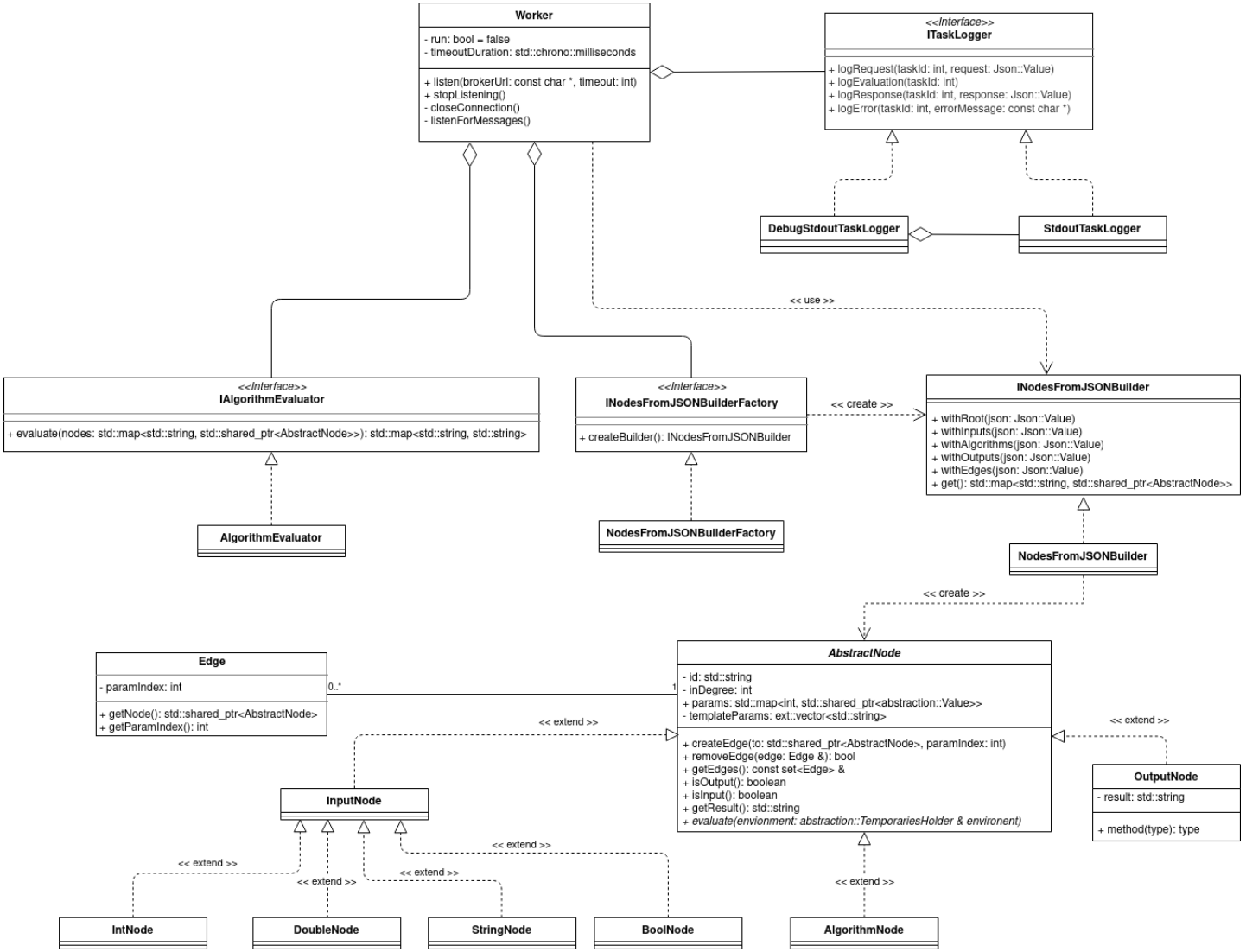
Omezení doby pro vyhodnocení požadavku bylo realizováno jednoduše pomocí C++11 `std::future`. Vyhodnocení se spouští v nově vytvořeném vlákně pomocí `std::packaged_task`. Ten umožňuje přístup k `std::future` reprezentující výsledek asynchroní operace. `std::future` nabízí užitečnou metodu `wait_for`. Ta umožňuje čekat pouze omezenou dobu na vyhodnocení. Pokud se vyhodnocení neprovede včas, ukončí celý program. Při spouštění je tedy zapotřebí mít mechanismus, který ho restartuje. V nasazené aplikaci restartovaní řeší Docker. Ukončení celého programu je nutné, protože jak už bylo popisováno dříve, výpočetní vlákno nelze bez následků zrušit při výpočtu. Zjednodušenou ukázku lze vidět ve výpisu kódu 5.3.

```
interface EvaluateRequest {
  algorithms: {
    [id: string]: {
      name: string
      templateParams: string[]
    }
  }
  inputs: {
    strings?: {
      [id: string]: {
        value: string
      }
    }
    ints?: {
      [id: string]: {
        value: number // must be integer
      }
    }
    doubles?: {
      [id: string]: {
        value: number // must be float
      }
    }
    bools?: {
      [id: string]: {
        value: boolean
      }
    }
  }
  outputs: string[] // array of output node IDs
  pipes: {
    from: string // ID of the source node
    to: string // ID of the destination node
    paramIndex: number // index of the param in the destination
  }[]
}
```

Výpis kódu 5.1: Struktura JSON objektu požadavku pro vyhodnocení popsaná v jazyce TypeScript

```
interface EvaluateResponse {
  error?: string
  outputs?: {
    [outputId: string]: string
  }
}
```

Výpis kódu 5.2: Struktura JSON objektu odpovědi vyhodnocení popsaná v jazyce TypeScript



Obrázek 5.6: UML diagram tříd workera

```
std::packaged_task<
    std::shared_ptr<std::map<std::string, std::string>>()
    > task(
    [ & json,
        evaluator = evaluator,
        nodesFromJSONBuilderFactory = nodesFromJSONBuilderFactory
    ] {
        auto builder
            = nodesFromJSONBuilderFactory->createBuilder();
        auto nodes = builder->withRoot(json).get();
        return evaluator->evaluate(*nodes);
    });

    auto future = task.get_future();
    std::thread evalThread(std::move(task));

    if (future.wait_for(timeoutDuration)
            == std::future_status::timeout) {
        // timeout handling ...
    }

    evalThread.join();
    std::shared_ptr<std::map<std::string, std::string>> outputMap;

    try {
        outputMap = future.get();
    } catch (std::exception & e) {
        // exception handling ...
    }
}
```

Výpis kódu 5.3: Zjednodušený úryvek kódu implementace omezení doby vyhodnocení požadavku

Algoritmus 5.1 TopSort s filtrováním zbytečných vrcholů

Vstup: Orientovaný graf $G = (V, E)$
Výstup: Seznam vrcholů řazený dle topologického uspořádání vrcholů bez zbytečných vrcholů

```
1:  visited ← ∅                    ▷ Množina navštívených vrcholů v aktuálním zanoření
2:  valid ← ∅                      ▷ Množina vrcholů, které nejsou zbytečné
3:  topSort ← List                 ▷ Vrcholy seřazené dle topologického uspořádání
4:  for v ∈ V ∧ deg^in(v) == 0 do
5:      DFS(v, visited, valid, topSort)
6:  end for
7:  return topSort
8:  procedure DFS(v)
9:      if v ∈ visited then
10:         throw Cycle detected
11:     end if
12:     if v ∈ valid then
13:         return
14:     end if
15:     visited ← visited ∪ {v}
16:     isValid ← deg^out(v) == 0
17:     for u ∈ suc(v) do
18:         DFS(u)
19:         if u ∈ valid then
20:             isValid ← true
21:         end if
22:     end for
23:     if isValid then
24:         valid ← valid ∪ {v}
25:         topSort.push_front(v)
26:     end if
27:     visited ← visited \ {v}
28: end procedure
```

Kapitola 6

Testování

Aplikace je testována unit testy a manuálními systémovými testy.

6.1 Unit testy

Při vývoji byla aplikace primárně testována unit testy. Unit testy slouží k oddělenému testování jednotlivých částí systému. Hlavní výhodou unit testů je jejich jednoduchost a malé nároky na spuštění, oproti například systémovým testům.

Protože WebUI je fork Statemakeru, používá také stejný testovací framework, jímž je Jest [74]. Worker k testování využívá knihovnu Catch2 [75]. Ta byla zvolena primárně kvůli své jednoduchosti, celá knihovna je totiž obsažena v jednom hlavičkovém souboru, to umožňuje jednoduchou instalaci. API Server kvůli své jednoduchosti není nijak testován unit testy.

6.2 Systémové testy

Pro otestování splnění funkčních požadavků byla aplikace otestována systémovými testy. Ty jsou popsány v následujících scénářích.

6.2.1 Testování vyhodnocení

Vyhodnocení algoritmů je hlavní funkcionalita aplikace, proto tato funkcionalita musí být korektní a stabilní. Tento scénář se zaměřuje na vyhodnocení složitějších algoritmových grafů tvořených více komponenty.

Postup

Scénář předpokládá, že aplikace je otevřena v desktopovém prohlížeči a je ve výchozím stavu.

1. Postavte korektní algoritmový graf s více komponenty.
2. Zadejte vstupy.
3. Spusťte vyhodnocení.
4. Počkejte na dokončení vyhodnocení.
5. Ověřte, zda jsou výsledky korektní.

Očekávaný výsledek

Aplikace umožní uživateli sestavit korektní algoritmový graf a po vyhodnocení mu umožní zobrazit korektní výsledky. To vše bez jakýchkoliv chybových hlášek či technických problémů.

Výsledek

Aplikace umožnila uživateli sestavit korektní algoritmový graf s více komponenty a následně jej korektně vyhodnotila. Test tedy uspěl.

6.2.2 Testování korektního předávání požadavků workerům

Tento scénář testuje, zda broker umí korektně distribuovat zprávy workerům. Náročný požadavek, který jeden worker může vyhodnocovat i několik minut, nesmí blokovat distribuci požadavků jiným workerům. Zároveň také testuje, zda se worker po uplynutí lhůty na vyhodnocení požadavku korektně restartuje.

Pro testování je užitečná chyba v algoritmu `RandomAutomatonFactory`. Ten přijímá parametr *density*, kterým podle dokumentace může být jakékoliv číslo s plovoucí čárkou z intervalu 0–100. V případě zadání hodnoty vyšší než 100 vyhodnocování algoritmu neskončí. Tato chyba je pro tento scénář užitečná, protože umožňuje jednoduše simulovat náročný požadavek.

Postup

Scénář předpokládá, že aplikace je nasazena a běží se dvěma workery, kteří mají nastavený timeout na 20 sekund.

1. Otevřete dvě různé instance WebUI.
2. V první instanci vytvořte algoritmový vrchol `RandomAutomatonFactory`.
3. Tomuto algoritmovému vrcholu předejte korektní vstupy mimo paramater *density*, jemuž předáte hodnotu 101.
4. Dotvořte zbytek grafu, aby byl korektní.

5. V druhé instanci vytvořte ten samý graf, ovšem hodnota argumentu *density* zde bude 1.
6. Spusťte vyhodnocení v první instanci.
7. Spusťte třikrát za sebou vyhodnocení ve druhé instanci dříve, než se dokončí vyhodnocení v první instanci.
8. Zkontrolujte, zda se dokončili všechny tři vyhodnocení ve druhé instanci dříve, než se ukončilo vyhodnocení v první instanci.
9. Zkontrolujte, zda se korektně ukončilo vyhodnocení v první instanci s korektní chybovou hláškou.

Výsledek

V první instanci vyhodnocení trvalo 20 sekund, to odpovídá maximální časové lhůtě pro vyhodnocení. Ve druhé instanci mezi tím korektně proběhla tři vyhodnocení. V první instanci se ovšem zobrazila špatná chybová hláška začínající časovým údajem. Test tedy pomohl odhalit jednu mírnou chybu, nicméně svůj primární účel otestovat vyhodnocení při zatíženém workerovi splnil.

6.2.3 Testování stahování výstupu DOT dialogu

Cílem tohoto scénáře je otestovat, zda korektně funguje stahování výstupu DOT dialogu ve formátech DOT a SVG. Scénář se zabývá vytvoření jednoduché grafu, který obsahuje pouze jen automatový vstupní vrchol a DOT výstupní vrchol.

Postup

Scénář předpokládá otevřenou aplikaci v desktopovém prohlížeči, která je ve výchozím stavu.

1. Vytvořte na plátně vstupní vrchol `Automaton Input`.
2. Vytvořte na plátně výstupní vrchol `DOT Output`.
3. Vytvořené vrcholy propojte.
4. Výsledný algoritmový graf vyhodnoťte.
5. Zobrazte výstup `DOT Output` vrcholu.
6. Uložte výstup ve formátu DOT.
7. Uložte výstup ve formátu SVG.
8. Zkontrolujte korektnost výstupů.

Výsledek

Aplikace uložila korektní výstupy ve formátech DOT a SVG. Ovšem při manipulaci s UI se přišlo na chybu při odkliknutí z kontextového menu pro výběr ukládaného formátu, to totiž způsobilo zavření dialogu. Primární cíl testu byl úspěšný.

6.2.4 Testovaní integrace Statemakeru

Integrace Statemakeru pro vytváření a zobrazování vstupů a výstupů je klíčovou funkcionalitou aplikace. Tento scénář proto slouží k otestování této funkcionality. Scénář popisuje vytváření jednoduchého algoritmového grafu s algoritmovým vstupem, který je řetězen do algoritmu `Determinize`, jehož výstup je zobrazen opět ve Statemakeru.

Postup

Scénář předpokládá otevřenou aplikaci v desktopovém prohlížeči, která je ve výchozím stavu.

1. Položte na plátno vstupní vrchol `Automaton Input`.
2. Položte na plátno vrchol algoritmu `Determinize` přijímající nedeterministický konečný automat.
3. Položte na plátno výstupní vrchol `Automaton Output`.
4. Výsledný algoritmový graf vyhodnoťte.
5. Zobrazte výstup `Automaton Output` vrcholu.
6. Ověřte, zda zobrazený automat v aplikaci Statemaker je korektní.

Výsledek

Algoritmový graf se podařilo vyhodnotit a výsledný zobrazený automat je korektní.

6.2.5 Testování importu a exportu

Aplikace umožňuje ukládat algoritmový graf do JSON souboru, ze kterého ho lze později načíst. Otestování této funkcionality je přímočaré. Uživatel nejdříve sestaví graf, uloží jej do JSON souboru a následně jej znovu načte.

Postup

Scénář předpokládá otevřenou aplikaci v desktopovém prohlížeči, která je ve výchozím stavu.

1. Vytvořte libovolný algoritmový graf.
2. Graf exportujte do JSON souboru.
3. Výsledný soubor uložte.
4. Vyčistěte plátno pomocí akce *Clear canvas*.
5. Importujte stažený soubor.
6. Zkontrolujte, zda je načtený graf stejný jako dříve uložený.

Výsledek

Exportovaný soubor se podařilo uložit a následně se jej i povedlo načíst. Načtený algoritmový graf je totožný s uloženým. Test je tedy úspěšný.

6.2.6 Shrnutí

Aplikace byla otestována 5 scénáři. Vykonané testy pomohly nalézt 2 menší chyby, ovšem ve výsledku byly všechny testy úspěšné.

Závěr

Cílem této práce bylo vytvořit webové grafické rozhraní pro ALT využívající Statemaker k vytváření a zobrazování stavových automatů. Toho bylo docíleno pomocí React webové aplikace, která byla postavena jako fork Statemakeru. Aplikace umožňuje pomocí principu Pipes-And-Filters znázornit řetězení algoritmů a tyto algoritmy umí vyhodnotit. Pro vyhodnocení bylo vytvořeno webové HTTP API zpřístupňující ALT.

Tato práce zároveň řešila problém vyhodnocení náročných algoritmů, jejichž vyhodnocení může trvat dlouhou dobu, pomocí odděleně bežících workerů. Workeři komunikují se serverem pomocí fronty zpráv a zpracovávají jednotlivé požadavky na vyhodnocení algoritmů.

Jako vstup algoritmů aplikace umožňuje vytvořit hodnoty typů `string`, `int`, `double`, `bool`. Dále umožňuje vytvořit konečný stavový automat pomocí Statemakeru, který lze použít jako vstup algoritmů. Výstupy je možné zobrazit v textovém formátu, ve vykresleném DOT formátu, nebo jako automat ve Statemakeru. U DOT formátů aplikace umožňuje výsledek stáhnout jak v originální podobě, tak ve vykresleném SVG. Celý algoritmový graf lze exportovat do JSON souboru pro pozdější načtení.

Aplikaci dál lze rozšířit následujícími způsoby:

- Přidat moduly podobné Statemakeru pro podporu dalších komplexních datových typů ALT, jako jsou například gramatiky, regulární výrazy, neorientované a orientované grafy, zásobníkové automaty nebo Turingovy stroje.

- Přidat další typy výstupních vrcholů. Mezi nimi může být například vrchol pro Latex výstup.

- Optimalizovat velikost ALT, aby šla zkompilovat do WebAssembly. To by snížilo nároky na infrastrukturu, protože by nebylo potřeba vyhodnocovat algoritmy vzdáleně.

Výsledná aplikace je nasazena na adrese `https://alt.fit.cvut.cz/webui`, kam je automaticky nasazována pomocí Gitlab CI. Je plánováno, že bude využita k výuce na FIT ČVUT v předmětu BI-AAG.

Bibliografie

1. TRÁVNÍČEK, Jan; PECKA, Tomáš; PLACHÝ, Štěpán et al. Algorithms Library Toolkit [online] [cit. 2020-04-19]. Dostupné z: `https://alt.fit.cvut.cz/`.
2. Algorithms Query Language. In: *Algorithms Library Toolkit Documentation* [online] [cit. 2020-04-19]. Dostupné z: `https://alt.fit.cvut.cz/docs/querylanguage/`.
3. BERGEN, Patrick van. Pipe-And-Filter. In: *Garfixia Software Architectures* [online] [cit. 2020-04-19]. Dostupné z: `http://www.dossier-andreas.net/software_architecture/pipe_and_filter.html`.
4. SVOBODA, Petr. *Webový editor konečných automatů*. Praha, 2019. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.
5. SVOBODA, Petr. Statemaker. In: *Gitlab* [online] [cit. 2020-04-19]. Dostupné z: `https://gitlab.com/petrdsvoboda/statemaker`.
6. MAREŠ, Martin; VALLA, Tomáš. *Průvodce labyrintem algoritmů*. Praha: CZ.NIC, z. s. p. o., 2017. ISBN 978-80-88168-22-5.
7. ŠESTÁKOVÁ, Eliška. *Automaty a gramatiky: Sbírka řešených příkladů*. Praha: České vysoké učení technické v Praze, 2017. ISBN 978-80-01-06306-4.
8. HOHPE, Gregor; WOOLF, Bobby. Pipes and Filters. In: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003. ISBN 978-0321200686.
9. THE QT COMPANY. Qt framework [online] [cit. 2020-05-02]. Dostupné z: `https://www.qt.io/product/framework`.
10. The language of string::Parse algorithm. In: *Algorithms Library Toolkit Documentation* [online] [cit. 2020-05-24]. Dostupné z: `https://alt.fit.cvut.cz/docs/parse/`.

11. WORLD WIDE WEB CONSORTIUM. Extensible Markup Language (XML) [online] [cit. 2020-05-03]. Dostupné z: https://www.w3.org/XML/.

12. ELLSON, John; GANSNER, Emden; HU, Yifan et al. Graphviz [online] [cit. 2020-05-03]. Dostupné z: https://www.graphviz.org/.

13. Introducing JSON [online] [cit. 2020-05-03]. Dostupné z: https://www.json.org/json-en.html.

14. GNU RADIO PROJECT. About GNU Radio [online] [cit. 2020-05-03]. Dostupné z: https://www.gnuradio.org/about/.

15. MICROSOFT INC. TypeScript [online] [cit. 2020-05-05]. Dostupné z: https://www.typescriptlang.org/.

16. FACEBOOK INC. React [online] [cit. 2020-05-05]. Dostupné z: https://reactjs.org/.

17. GOOGLE LLC. Material Design [online] [cit. 2020-05-05]. Dostupné z: https://material.io/.

18. MATERIAL-UI. Material-UI [online] [cit. 2020-05-05]. Dostupné z: https://material-ui.com/.

19. ABRAMOV, Dan et al. Redux [online] [cit. 2020-05-06]. Dostupné z: https://redux.js.org/.

20. THE WORLD WIDE WEB CONSORTIUM. Scalable Vector Graphics (SVG) 2 [online] [cit. 2020-05-05]. Dostupné z: https://www.w3.org/TR/SVG/.

21. XState [online] [cit. 2020-05-05]. Dostupné z: https://xstate.js.org/.

22. SVOBODA, Petr. *Webový editor konečných automatů: Import a export*. Praha, 2019. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.

23. WORLD WIDE WEB CONSORTIUM. WebAssembly [online] [cit. 2020-05-06]. Dostupné z: https://webassembly.org/.

24. HASS, Andreas; ROSSBERG, Andreas; SHUFF, Derek L. et al. Bringing the web up to speed with WebAssembly. In: *PLDI 2017: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* [online]. Association for Computing Machinery, 2017, s. 185–200 [cit. 2020-05-06]. ISBN 978-1-4503-4988-8. Dostupné z: https://dl.acm.org/doi/pdf/10.1145/3062341.3062363.

25. Emscripten [online] [cit. 2020-05-06]. Dostupné z: https://emscripten.org/.

26. The LLVM Compiler Infrastructure [online] [cit. 2020-05-06]. Dostupné z: https://llvm.org/.

27. FREE SOFTWARE FOUNDATION, INC. GCC, the GNU Compiler Collection [online] [cit. 2020-05-06]. Dostupné z: https://gcc.gnu.org/.

28. Portability Guidelines [online] [cit. 2020-05-06]. Dostupné z: https://emscripten.org/docs/porting/guidelines/portability_guidelines.html.

29. CORPORATION, Mozilla. HTTP [online] [cit. 2020-05-07]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP.

30. THE OPEN GROUP. The Single UNIX ® Specification, Version 2: Threads [online] [cit. 2020-05-30]. Dostupné z: https://pubs.opengroup.org/onlinepubs/007908799/xsh/threads.html.

31. Kubernetes [online] [cit. 2020-05-07]. Dostupné z: https://kubernetes.io/.

32. ECMA INTERNATIONAL. Standard ECMA-262: ECMAScript® 2019 Language Specification [online]. 10. vyd. [cit. 2020-05-17]. Dostupné z: https://www.ecma-international.org/publications/standards/Ecma-262.htm.

33. OPENJS FOUNDATION. Node.js [online] [cit. 2020-05-17]. Dostupné z: https://nodejs.org.

34. GOOGLE LLC. What is V8? [online] [cit. 2020-05-17]. Dostupné z: https://v8.dev/.

35. GITHUB INC. The State of the Octoverse [online] [cit. 2020-05-17]. Dostupné z: https://octoverse.github.com.

36. MOZILLA CORPORATION. Using Web Workers [online] [cit. 2020-05-17]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

37. FACEBOOK INC. Flow [online] [cit. 2020-05-17]. Dostupné z: https://flow.org/.

38. DAHL, Ryan et al. Deno: A secure runtime for JavaScript and TypeScript. [online] [cit. 2020-05-17]. Dostupné z: https://deno.land/.

39. ts-node [online] [cit. 2020-05-17]. Dostupné z: https://github.com/TypeStrong/ts-node.

40. WIRTZ, Daniel; GRAEY, Max. The AssemblyScript Book [online] [cit. 2020-05-17]. Dostupné z: https://docs.assemblyscript.org/.

41. KROTOFF, Tanguy. Front-end frameworks popularity (React, Vue and Angular) [online] [cit. 2020-05-12]. Dostupné z: https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190.

42. FACEBOOK INC. Virtual DOM and Internals [online] [cit. 2020-05-17]. Dostupné z: https://reactjs.org/docs/faq-internals.html.

43. FACEBOOK INC. Introducing JSX [online] [cit. 2020-05-17]. Dostupné z: https://reactjs.org/docs/introducing-jsx.html.

44. FACEBOOK INC. Flux [online] [cit. 2020-05-17]. Dostupné z: https://facebook.github.io/flux/.

45. ABRAMOV, Dan et al. React Redux [online] [cit. 2020-05-17]. Dostupné z: https://react-redux.js.org/.

46. VAN DEN BERGH, Michael. React Redux: Building Modern Web Apps with the ArcGIS JS API [online] [cit. 2020-06-01]. Dostupné z: https://www.esri.com/arcgis-blog/products/3d-gis/3d-gis/react-redux-building-modern-web-apps-with-the-arcgis-js-api/.

47. VMWARE INC. RabbitMQ [online] [cit. 2020-05-17]. Dostupné z: https://www.rabbitmq.com/.

48. ZeroMQ [online] [cit. 2020-05-17]. Dostupné z: https://zeromq.org/.

49. APACHE SOFTWARE FOUNDATION. Apache ActiveMQ [online] [cit. 2020-05-17]. Dostupné z: https://activemq.apache.org/.

50. APACHE SOFTWARE FOUNDATION. Apache Qpid™ [online] [cit. 2020-05-17]. Dostupné z: https://qpid.apache.org/.

51. OASIS. AMQP: Advanced Message Queueing Protocol [online] [cit. 2020-05-17]. Dostupné z: https://www.amqp.org/.

52. Stomp: The Simple Text Oriented Messaging Protocol [online] [cit. 2020-05-17]. Dostupné z: https://stomp.github.io/.

53. ORACLE CORPORATION. Java Documentation [online] [cit. 2020-05-17]. Dostupné z: https://docs.oracle.com/en/java/.

54. APACHE SOFTWARE FOUNDATION. ActiveMQ: CMS Client [online] [cit. 2020-05-17]. Dostupné z: https://activemq.apache.org/components/cms/.

55. APACHE SOFTWARE FOUNDATION. ActiveMQ: VM Protocol [online] [cit. 2020-05-14]. Dostupné z: https://activemq.apache.org/vm-protocol.

56. JETBRAINS S.R.O. Kotlin [online] [cit. 2020-05-17]. Dostupné z: https://kotlinlang.org/.

57. JETBRAINS S.R.O. Ktor [online] [cit. 2020-05-17]. Dostupné z: https://ktor.io/.

58. STROUSTRUP, Bjarne. The Essence of C++ [video] [cit. 2020-06-03]. Dostupné z: https://www.youtube.com/watch?v=86xWVb4XIyE.

59. OPENJS FOUNDATION. Node.js v14.4.0 Documentation: N-API [online] [cit. 2020-06-03]. Dostupné z: https://nodejs.org/api/n-api.html#n_api_n_api.

60. THE PYTHON SOFTWARE FOUNDATION. Python/C API Reference Manual [online] [cit. 2020-06-03]. Dostupné z: https://docs.python.org/3/c-api/index.html.

61. THE PYTHON SOFTWARE FOUNDATION. ctypes: A foreign function library for Python [online] [cit. 2020-06-03]. Dostupné z: https://docs.python.org/3/library/ctypes.html.

62. SWIG [online] [cit. 2020-06-03]. Dostupné z: http://www.swig.org/exec.html.

63. KITWARE INC. CMake [online] [cit. 2020-05-24]. Dostupné z: https://cmake.org/.

64. DOCKER INC. Docker [online] [cit. 2020-05-24]. Dostupné z: https://www.docker.com/.

65. BUGL, Daniel. Redux undo/redo [online] [cit. 2020-06-03]. Dostupné z: https://www.npmjs.com/package/redux-undo.

66. DEHNOKHALAJI, Hossein. Notistack [online] [cit. 2020-06-03]. Dostupné z: https://www.npmjs.com/package/notistack.

67. The DOT Language [online] [cit. 2020-05-01]. Dostupné z: https://graphviz.gitlab.io/_pages/doc/info/lang.html.

68. PARFITT, Dom. graphviz-react [online] [cit. 2020-06-03]. Dostupné z: https://www.npmjs.com/package/graphviz-react.

69. FACEBOOK INC. Composition vs Inheritance: Containment [online] [cit. 2020-05-18]. Dostupné z: https://reactjs.org/docs/composition-vs-inheritance.html.

70. BUCKLER, Craig. How to Translate from DOM to SVG Coordinates and Back Again [online] [cit. 2020-05-18]. Dostupné z: https://www.sitepoint.com/how-to-translate-from-dom-to-svg-coordinates-and-back-again/.

71. BEUST, Cedric. Klaxon [online] [cit. 2020-06-03]. Dostupné z: https://github.com/cbeust/klaxon.

72. APACHE SOFTWARE FOUNDATION. ActiveMQ: OpenWire [online] [cit. 2020-06-03]. Dostupné z: https://activemq.apache.org/openwire.

73. JsonCpp [online] [cit. 2020-06-03]. Dostupné z: https://github.com/open-source-parsers/jsoncpp.

74. FACEBOOK INC. Jest [online] [cit. 2020-06-03]. Dostupné z: https://jestjs.io/.

75. Catch2 [online] [cit. 2020-06-03]. Dostupné z: https://github.com/catchorg/Catch2.

76. NPM INC. npm [online] [cit. 2020-04-06]. Dostupné z: https://www.npmjs.com/.

77. ORACLE CORPORATION. Java SE Development Kit 8 [online] [cit. 2020-04-06]. Dostupné z: https://www.oracle.com/java/technologies/javase-jdk8-downloads.html.

Příloha A

Instalační příručka

Každou z komponent lze nasadit pomocí Dockeru. Je to sice jednodušší způsob, protože Docker image nainstalují všechny závislostí a zajistí konzistenci prostředí, ve kterém se aplikace spouští, ovšem není to nutné. Při nasazování je nutné nasadit API server před workery. Pokud se tak nestane, worker se ukončí s chybou, protože se nemůže připojit k brokerovi. Vytváření každého Docker image je nutné provést s kontextem v kořenovém adresáři projektu. Všechny image jsou vužívány pomocí Gitlab CI k automatickému nasazení. Jednotlivé image jsou také dostupné pomocí container registry v repozitáři projektu [2].

A.1 WebUI

Pro nasazení bez Dockeru je nutné mít nainstalované npm [76] a Node.js [33].

A.1.1 Postup

1. Přepněte se do adresáře `<project_root>/webui`
2. Nainstalujte závislosti pomocí `npm install`
3. Aplikace sestavte pomocí `npm run build`
4. Výsledné aplikace je v adresáři `<project_root>/webui/build` v podobě HTML stránky a přiložených CSS a JS souborů

A.1.2 Docker

Docker image lze sestavit pomocí přiloženého Dockerfile. Ten je umístěn na cestě `<project_root>/webui/Dockerfile`. Výsledný image při spuštění hostuje WebUI na portu 80.

[2] `https://gitlab.fit.cvut.cz/algorithms-library-toolkit/webui-client`

A.2 API server

Jediná prerekvizita pro sestavení bez Dockeru a spuštění je JDK alespoň verze 8 [77], nebo jiná ekvivalentní alternativa.

A.2.1 Postup

1. Přepněte se do adresáře `<project_root>/server`
2. Aplikaci sestavte pomocí příkazu `./gradlew build jar`
3. V podadresáři `./build/libs/server-1.0-SNAPSHOT.jar` se nachází výsledný jar soubor
4. Pomocí příkazu `java -jar server-1.0-SNAPSHOT.jar` aplikaci spustíte
5. Aplikace naslouchá HTTP požadavkům na portu 3001. Broker naslouchá na portu 61616

A.2.2 Docker

Docker image sestavte pomocí přiloženého Dockerfile. Docker image naslouchá na stejných portech, jako je výše zmíněno.

A.3 Worker

Pro sestavení workera bez Dockeru je nutné mít nainstalované následující závislosti:

- g++
- Make
- CMake
- pkg-config
- ALT
- activemq-cpp
- jsoncpp

A.3.1 Postup

- Přepněte se do adresáře `<project_root>/worker`
- Spusťte příkaz `cmake .`
- Spusťte příkaz `make`
- Výsledný soubor je `alib_web_worker` v aktuálním adresáři
- Spusťte s následujícími argumenty `./alib_web_worker <broker_url> <timeout>`, kde *broker_url* je adresa brokera a *timeout* značí maximální doby vyhodnocování požadavku v milisekundách

A.3.2 Docker

Dockerfile je poskytnut v adresáři `<project_root>/worker`. Výsledný image se připojuje na adresu brokera specifikovanou pomocí proměnné prostředí `ALT_WORKER_SERVER`. Timeout pro vyhodnocení lze specifikovat pomocí proměnné prostředí `ALT_WORKER_TIMEOUT`.

Příloha B

Seznam použitých zkratek

ALT Algorithms Library Toolkit

API Application program interface

GUI Graphical user interface

DAG Directed acyclic graph

UI User interface

CLI Command line interface

JVM Java virtual machine

Příloha C

Obsah přiloženého USB disku

- `src` ..... zdrojové kódy implementace
  - `webui` ..... zdrojové kódy WebUI
  - `server` ..... zdrojové kódy API serveru
  - `worker` ..... zdrojové kódy workera
- `thesis` ..... zdrojová forma práce ve formátu LaTeX se soubory šablony