FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

| | |
|---|---|
| **Název:** | Vylepšení GUI ke knihovně algoritmů ALIB |
| **Student:** | Martin Hanzík |
| **Vedoucí:** | Ing. Jan Trávníček |
| **Studijní program:** | Informatika |
| **Studijní obor:** | Teoretická informatika |
| **Katedra:** | Katedra teoretické informatiky |
| **Platnost zadání:** | Do konce zimního semestru 2019/20 |

## Pokyny pro vypracování

Nastudujte současnou implementaci grafického rozhraní pro knihovnu algoritmů [1] a vlastní knihovnu algoritmů [2] vyvíjenou na katedře teoretické informatiky.
Proveďte rešeršní rozbor aplikací vizualizujících koncept "pipes and filters".
Na základě rešerše analyzujte nedostatky současného návrhu grafického rozhraní a navrhněte jejich vylepšení. Analyzujte možnosti knihovny poskytnout informace o dostupných algoritmech.
Rozšiřte implementaci grafického rozhraní aby při spuštění reagovalo na knihovnou algoritmů poskytované informace o dostupných algoritmech.
Pomocí nalezení topologického uspořádání grafu propojů mezi algoritmy implementujte efektivní plánování vyhodnocení schématu propojení algoritmů.
Odstraňte nedostatky současné implementace zjištěné v analýze.
Hotové řešení vhodnými prostředky otestujte.

## Seznam odborné literatury

[1] MAREŠ, Václav. GUI k automatové knihovně ALIB. 2017. Bachelor's Thesis. České vysoké učení technické v Praze. Vypočetní a informační centrum.
[2] Martin Žák: Automatová knihovna – vnitřní a komunikační formát. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

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

V Praze dne 22. února 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

# Vylepšení GUI ke knihovně algoritmů ALIB

***Martin Hanzík***

Katedra Teoretické Informatiky

Vedoucí práce: Ing. Jan Trávníček

14. května 2018

# Poděkování

Děkuji Ing. Janu Trávníčkovi za jeho připomínky při psaní této práce. Dále děkuji své rodině a přátelům za podporu při studiích a za pomoc při testování.

# Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické 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. § 46 odst. 6 tohoto zákona 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 14. května 2018 .....

České vysoké učení technické v Praze

Fakulta informačních technologií



### Odkaz na tuto práci

Hanzík, Martin. *Vylepšení GUI ke knihovně algoritmů ALIB.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

# Abstrakt

Předmětem této práce je analyzovat nedostatky grafického uživatelského rozhraní existující aplikace pro práci s algoritmovou knihovnu ALT, navrhnout jejich vylepšení a implementovat je. Text se věnuje návrhovému vzoru "pipes and filters" a prozkoumání existujících aplikací využívajících tento vzor. Dále analyzuje způsob, kterým knihovna poskytuje informace o existujících algoritmech a prozkoumá možnosti plánování paralelního provádění pomocí kombinace synchronizačních primitiv a plánovacích algoritmů. Nakonec je provedeno jednoduché uživatelské testování.

**Klíčová slova** ALIB, ALT, C++, Qt, grafické uživatelské rozhraní, paralelismus

# Abstract

The subject of this bachelor's thesis is to analyse the flaws of an existing graphical user interface for the algorithm library ALT, suggest improvements and implement them. The text contains research of the design pattern "pipes and filters" and applications that use it. Furthermore, it focuses on analysing the means of algorithm discovery provided by the library, exploring the possibilities of parallel job scheduling using synchronisation primitives and scheduling algorithms. Finally, simple user testing is performed.

**Keywords** ALIB, ALT, C++, Qt, graphical user inferace, parallelism

# Obsah

# Seznam obrázků

# Úvod

Algoritmová knihovna ALT (původně ALIB) [1], která poslední čtyři roky vzniká na Katedře teoretické informatiky FIT ČVUT, pod vedením Ing. Jana Trávníčka, poskytuje referenční implementace algoritmů, se zaměřením na automaty a gramatiky, a práci s nimi.

V současné době je práce s touto knihovnou možná API v jazyce C++, rozhraní v příkazové řádce a také pomocí GUI aplikace, která vznikla jako bakalářská práce Václava Mareše [2].

## Cíl práce

Prvním cílem této práce je nastudovat knihovnu ALT a současnou implementaci GUI pro tuto knihovnu, analyzovat nedostatky tohoto rozhraní a navrhnout jejich vylepšení. Tato vylepšení by měla být realizována podle rešerše grafického konceptu "pipes and filters".

Druhým cílem je úprava existujícího GUI tak, aby reagovalo na dostupné algoritmy, a umožnovalo práci s větší podmnožinou funkcí knihovny ALT.

Třetím cílem je navržení a implementace paralelního spouštění a vyhodnocení schématu propojení algoritmů, v případech kdy je to možné.

## Struktura práce

Tato práce je rozdělena do pěti kapitol. První kapitola popisuje využití návrhového vzoru "pipes and filters" v operačním systému UNIX a v GUI aplikacích, konkrétně v herním enginu Unreal Engine 4 a 3D modelovacím programu Blender.

Druhá kapitola se věnuje analýze. Zjistíme jak můžeme z knihovny ALT získat informace o jednotlivých algoritmech a jak je spouštět. Také analyzujeme nedostatky současné implementace GUI a navrhujeme možnosti jejich řešení.

Třetí kapitola popisuje různé způsoby paralelizace výpočetního plánu. Kromě metody používající pouze synchronizační primitiva také prozkoumá topologické uspořádání a Coffmanův-Grahamův algoritmus.

Čtvrtá kapitola se věnuje popisu implementace nových funkcí a porovnání s původní verzí.

Poslední kapitola popisuje implementované automatické jednotkové testy, provedené uživatelské testování a možná budoucí vylepšení aplikace.

Kapitola 1

# Pipes and filters

Návrhový vzor pipes and filters, česky také “roury a filtry”, popisuje jednosměrný tok informací mezi uzly, které provádí jejich zpracování. Dalším možným názvem je anglické slovo “pipeline”, které poukazuje na analogii tohoto vzoru v chemickém průmyslu. Je totiž velmi podobný struktuře ropných rafinerií, či podobných továren, kde různé tekutiny putují trubkami mezi jednotlivými fázemi zpracování, většinou jednosměrně.

Tato architektura se skládá ze dvou hlavních částí. První částí je filtr. Ten reprezentuje funkci, či algoritmus, který zpracovává vstupní data, a produkuje výstupy. Pojmenování filtr je lehce zavádějící, jeho činnost nemusí nutně být filtrování - odebírání určitých prvků ze vstupu, podle předem daných pravidel, je to libovolná funkce která vytváří, či zpracovává data. Jednotlivé filtry mohou mít více vstupů, ale také nemusí mít vstupy žádné, jako například generování náhodných dat. Výstup je většinou jen jeden, ale může jich také být více.

Druhou součástí je roura, která jednosměrně propojuje vstupy a výstupy filtrů. Ke každému vstupu filtru většinou můžeme připojit jen jednu rouru, ale může nastat situace, kde možné připojit rour více a přicházející data se nějakým způsobem kombinují. Obecně nelze říci, že z jednoho výstupu může vést více rour, tato data mohou být určena pouze k jednomu použití. V případě naší GUI aplikace ale data můžeme bez problému kopírovat, takže připojení více rour k jednomu výstupu je teoreticky možné.

Roury často obsahují frontu, ve které jednotlivé informace čekají na zpracování následujícím filtrem. Toto je nutné pro případy, kdy se rychlost zpracování dat u jednotlivých filtrů liší a filtr na konci roury nestíhá zpracovávat data rychlostí větší, či rovnou, než je filtr na druhém konci produkuje.

Tento návrhový vzor je využíván například v operačním systému UNIX,

konkrétně pro UNIXové roury, které jsou používány pro práci na příkazové řádce. Zde jde o tok binárních, často textových, dat mezi jednotlivými programy. Standardní výstup jednoho programu je přesměrován na standardní vstup dalšího programu. Tyto roury obsahují již zmíněnou frontu, v současné době většinou o velikosti 64 KiB.

```
$ find . -type f | perl -ne 'print $1 if m/\.([^.\/]+)$/' \
  | sort | uniq -c | sort -r
```

Ukázka kódu 11: Použití UNIXových rour v shellu Bash

Pro tuto práci zásadním využitím této architektury je její použití v GUI aplikacích. Hlavními příklady, které v této části popíšeme, jsou návrh "blueprintů" pro Unreal Engine 4 a tvorba materiálů v modelovacím programu Blender.

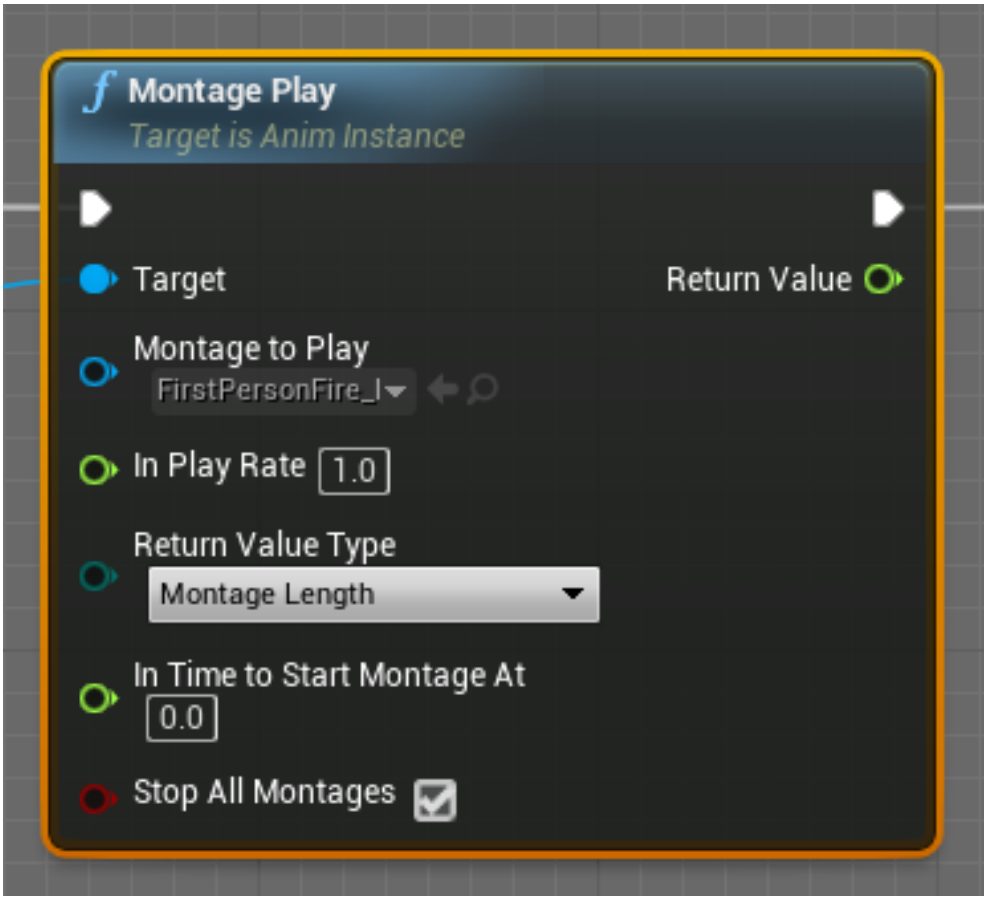
## 1.1 Unreal Engine 4

Unreal Engine je vývojové prostředí pro vytváření počítačových her. Tento engine obsahuje velké množství součástí, které jsou pro hry nezbytně nutné, například pokročilý systém vykreslování, práce se zvukem, ovládání hry, či umělou inteligenci pro ovládání postav ve hrách.

Celý engine je napsán v C++ a také je možné jeho API tímto způsobem používat, ale také obsahuje "blueprinty", systém pro vizuální skriptování. Tento systém zpřístupňuje programování i méně technologicky zaměřeným členům vývojového týmu, jako třeba návrhářům grafiky, či jednotlivých úrovní, a zásadně urychluje vytváření prototypů nových funkcí hry.

Tvorba blueprintů spočívá v umisťování funkcí na pracovní plochu a jejich následné propojování. Můžeme například získat orientaci hráče, z ní vytvořit vektor ukazující vpřed, poté vzít pozici ovládacího prvku na stejné ose, tyto dvě hodnoty zkombinovat a podle nich upravit pozici hráče v herním světě. Provádíme stejné kroky, které bychom museli udělat, pokud bychom implementovali hru v C++, jen místo ukládání hodnot do proměnných a následného předávání do funkcí stačí propojit několik uzlů.

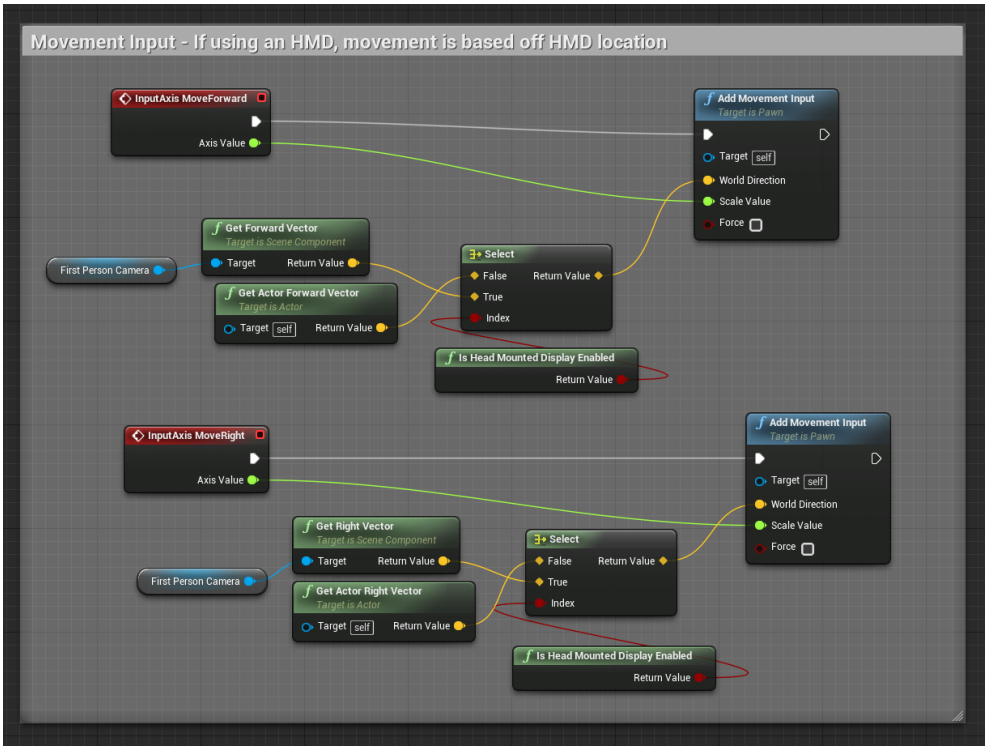
Každý uzel může mít několik vstupů, které jsou zobrazeny pomocí konektorů na levé straně uzlu, a také několik výstupů, které se nacházejí na straně pravé. Jednotlivá připojovací místa jsou odlišena tvary a barvami. Barvy určují možné datové typy pro dané místo, například červená barva znamená typ boolean, zelené je desetinné číslo ve formátu float, oranžové jsou vektory, a modré jsou různé pokročilejší objekty. Jelikož propojení nekompatibilních míst není možné, barevné odlišení ukazuje uživateli všechna možná spojení.




Obrázek 11: Ukázkový uzel blueprintu pro spuštění animace

Speciálním typem propojovacích míst jsou místa bílá, která určují kontrolní tok. Některé uzly musí do tohoto kontrolního toku být zapojeny, aby bylo pořadí jejich provádění přesně určeno. Uzly, které operují jen nad jejich vstupy, a nemají vedlejší účinky, nemusí mít toto spojení. V případě podmínek se tento datový tok větví, a uzly které jsou zapojeny na nesplněnné straně nejsou vůbec provedeny.

V enginu Unreal Development Kit, který předcházel Unreal Engine 4, byl kontrolní tok určen čistě datovým propojením a bylo možné uzly propojit tak, že jejich pořadí provádění nebylo deterministické, a mohlo vést k chybám při zpracování.



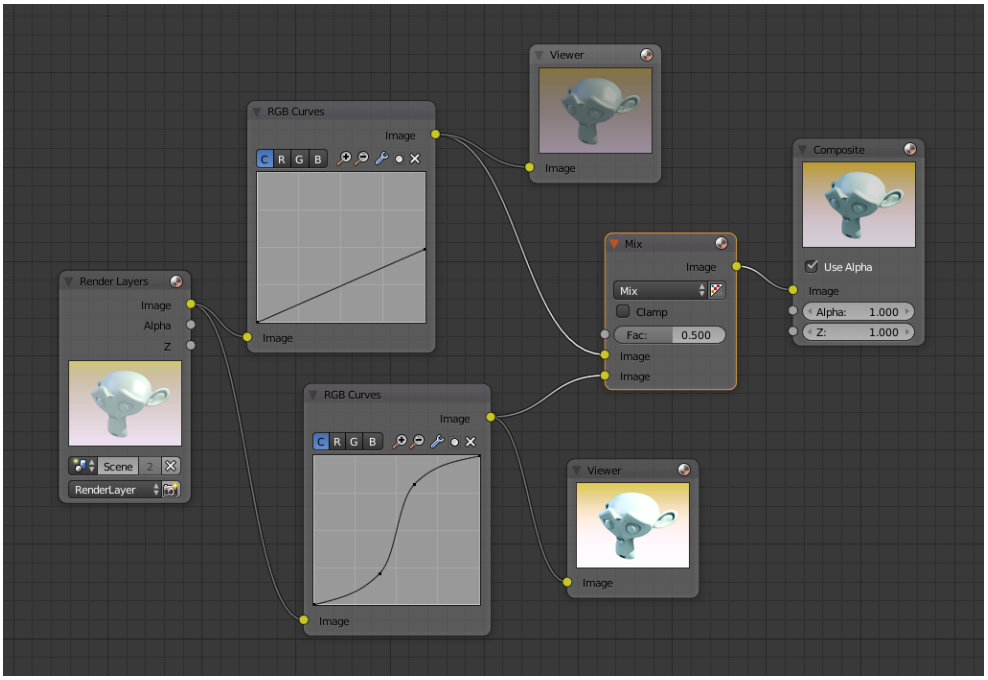

Obrázek 12: Ukázka blueprintu z Unreal Engine 4

## 1.2 Blender

Dalším programem, který ve velkém využívá vizuální skriptování a návrhový vzor roury a filtry, je Blender. Blender je program pro tvorbu a vykreslování 3D modelů a animací. Je velmi rozšířený a oblíbený, také díky tomu, že je zdarma dostupný na všech hlavních platformách a jeho zdrojový kód je volně přístupný.

Tento návrhový vzor je v Blenderu použit pro návrh materiálů. Ty společně s texturami určují jak bude vypadat povrch modelů, například průhlednost, lesk, povrchové vzory, či barvu objektu při různém nasvícení. Díky tomu je možné imitovat veliké množství různých skutečných povrchů a dosáhnout tak vysoké míry realismu.

Podobně jako v Unreal Engine 4, Blender používá uzly se vstupy na levé, výstupy na pravé straně, které reprezentují jednotlivé operace. Tyto jednoduché operace se různými způsoby kombinují a nabalují a na konci může vzniknout velmi pokročilý materiál.




Obrázek 13: Ukázka shaderového programu v Blenderu, z [3]. Creative Commons Attribution-ShareAlike 4.0 Int. Licence

Grafická karta ale neumí pracovat s touto reprezentací materiálů, je třeba ji převést na shader. Shader je malý program, určen pro akcelerovaný běh na grafické kartě počítače. Těchto programů je několik typů, ten nejčastější je pixelový (fragmentový) shader, který je grafickou kartou spouštěn na každém pixelua určuje jeho výslednou barvu. Podle toho jaká grafická knihovna je použita se mění i jazyk, ve kterém jsou shadery psány. Například, pro knihovnu OpenGL se používá jazyk GLSL, pro DirectX je to HLSL. Tyto shadery se pak překládají do společného formátu pro grafickou kartu.

Blender umožňuje přeskočit tuto manuální tvorbu a kompilaci shaderu, a je schopen daný graf uzlů minimalizovat, optimalizovat a převést na shader, který pak přímo spouští grafická karta. Tímto zpřístupňuje práci na materiálech i těm, kteří neumí programovat a nejsou schopni pracovat s shaderovými programy tradičním způsobem.

Kapitola 2

# Analýza

## 2.1 Knihovna ALT

Knihovna ALT, která vzniká na katedře teoretické informatiky FIT ČVUT, se zaměřuje na práci s automaty, gramatikami, stromy či řetězci, poskytuje implementace těchto datových struktur a algoritmů, které s nimi pracují.

Současná verze knihovny ALT umožňuje několik způsobů spouštění algoritmů. První způsob je využívání jednotlivých spustitelných programů na příkazové řádce a jejich propojování pomocí souborů, nebo rour.

Mezi takové programy patří například `aminimize2` pro minimalizaci automatu, `adeterminize2` pro determinizaci, nebo `aconvert2` pro převádění jednotlivých datových struktur mezi různými textovými reprezentacemi.

```
$ ./aminimize2 -i automaton.xml \
| ./adeterminize2 | ./aconvert2 --automaton_to_string
```

Ukázka kódu 21: Použití knihovny z příkazové řádky

Tento přístup je značně nepohodlný pro manuální použití, ale může být užitečný pro využití knihovny uvnitř jiných projektů, zvláště pokud nejsou napsány v jazyce, který by mohl přímo využívat její API. Další nevýhodou je také neefektivnost, při spouštění několika algoritmů po sobě je třeba neustále data převádět z vnitřní reprezentace na textovou a zpět.

Druhým způsobem využití funkcí knihovny, je nástroj `aql2`. Tento spustitelný program je součástí knihovny a poskytuje příkazovou řádku s jednoduchou syntaxí, která umožňuje spouštění a propojování jednotlivých algoritmů, ale oproti prvnímu způsobu může data zanechat ve vnitřní reprezentaci a poskytuje tedy větší efektivitu.

### 2.1.1 API

Třetí možnost použití knihovny je její API v jazyce C++. Pomocí tohoto API je možné přímo spouštět algoritmy, používat implementace datových struktur, nebo je převádět do různých dalších reprezentací. Díky tomu je možné těsně zakomponovat knihovnu do vlastního kódu a dosáhnout tak nejmenších ztrát na výkonu.

Knihovna poskytuje API k získání informací o existenci jednotlivých algoritmů v komponentě `alib2abstraction`. Jednotlivé algoritmy se při spuštění programu zaregistrují do třídy `registry::AlgorithmRegistry`. Registrace spočívá ve vytvoření instance třídy `registration::AbstractRegister`, které se jako parametry předá ukazatel na funkci s implementací daného algoritmu. Tato třída pomocí C++ šablon získá typy parametrů algoritmu a také návratový typ a uloží tento algoritmus do registru.

Následně můžeme použít metodu `AlgorithmRegistry::list`, která vrátí všechny algoritmy, které jsou v danou chvíli zaregistrovány. Tento seznam algoritmů můžeme redukovat pomocí funkce `listGroup`, která pomocí parametru vybere jen algoritmy, které patří do dané skupiny, například pouze algoritmy, které se týkají automatů.

Algoritmy mohou být použitelné pro větší množství typů vstupních dat a podle těchto typů se také mohou lišit typy výstupů. Proto každý algoritmus může existovat ve více verzích s rozdílnými implementacemi. Před spuštěním takového algoritmu se rozhodne, kterou variantu je třeba použít, podle typu instancí na jeho vstupech. Z registru je možné získat seznam těchto různých variant pomocí metody `AlgorithmRegistry::listOverloads`, která pro daný název algoritmu vrátí množinu názvů typů parametrů a návratového typu.

Pokud známe název algoritmu a datové typy, které budou na jeho vstupech, můžeme použít metodu `AlgorithmRegistry::getAbstraction`, která nám vrátí instanci `abstraction::OperationAbstraction`, pokud hledaná kombinace algoritmu a vstupních datových typů existuje.

Třída `OperationAbstraction` je předkem mnoha dalších tříd v jmenném prostoru `abstraction`. Tyto třídy reprezentují zaregistrované algoritmy, či přímé hodnoty a umožňují jejich propojování a spouštění. Například třída `ValueOperationAbstraction` reprezentuje operaci s možným výsledkem a její instanci je možné převést na instanci třídy `ValueProvider`, kterou lze využít k získání daného výsledku.

Další potomci těchto tříd reprezentují operace s různou aritou - nulární, unární, binární či n-ární. Od těchto tříd dále dědí specializované třídy pro jednotlivé kategorie abstrakcí, například pro algoritmy, převody mezi vnitřními

typy, nebo pro převody mezi různými textovými reprezentacemi.

Tyto kategorie abstrakcí také mají své registry, jako například:

**CastRegistry** Registr funkcí pro převod datových struktur, například pro převod deterministického konečného automatu na nedeterministický.

**StringWriterRegistry/StringReaderRegistry** Tyto dva registry poskytují funkce pro převod mezi různými datovými typy a jejich textovými reprezentacemi.

**XmlRegistry** Tento registr poskytuje podobnou funkcionalitu jako StringWriterRegistry/StringReaderRegistry, ale pro formát XML, s použitím knihovny `libxml2`.

## 2.2 Nedostatky GUI

Posledním způsobem použití knihovny ALT je GUI aplikace, která byla vytvořena jako bakalářská práce Bc. Václava Mareše [2]. Výsledkem jeho práce je použitelný prototyp, který má několik zásadních nedostatků. V této části některé z nich analyzujeme a navrhneme možná řešení.

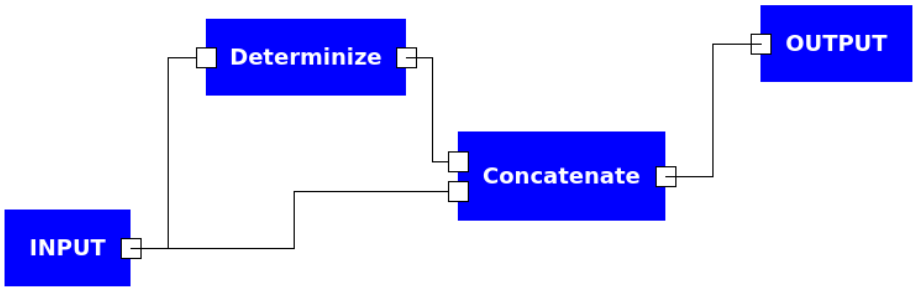
### 2.2.1 Vstupy a výstupy algoritmů

Algoritmy, které byly dostupné v první verzi GUI aplikace měly maximálně 2 vstupní parametry. Mezi uzly algoritmů s jedním a dvěma vstupními parametry ale nebyl žádný viditelný rozdíl. Uživatel musel vědět, které algoritmy to jsou a musel být opatrný v jejich zapojování. Pro takové uzly bylo velmi důležité pořadí připojování. Nejdříve byl zapojen první vstup, až poté druhý vstup.

K předání informace o počtu vstupů a výstupů by pomohlo vytvoření připojovacích míst na grafické reprezentaci uzlů. Na levé straně by se nacházely konektory pro vstupy, a na pravé straně pro výstupy. Na první pohled je pak jasně vidět, kolik vstupů a výstupů daný uzel má a které z nich jsou, či nejsou zapojeny. Je možné poté využít tvaru, či barvy těchto konektorů pro předání dalších informací, jako například povolené datové typy pro daná spojení.

### 2.2.2 Propojování algoritmů

V původní verzi aplikace bylo propojování jednotlivých uzlů poněkud komplikované. Bylo třeba pravým tlačítkem kliknout na jeden z uzlů, v kontextovém menu vybrat položku “Connect” a poté kliknout na druhý uzel. V průběhu propojování neexistovala žádná indikace tohoto procesu, jako například změna tvaru kurzoru, či dočasná čára mezi kurzorem a počátečním uzlem.




Obrázek 21: Ukázka nového propojování algoritmů

Pro změnu propojení bylo nutné pravým tlačítkem kliknout na uzel a zvolit položku "Reconnect". Hned poté původní spojení zmizelo a bylo třeba vybrat jiný cílový uzel, nebo kliknout na prázdné místo pro rozpojení.

Samotný způsob zahájení procesu propojování není až tak problematický, ale častěji bývá řešen pomocí přetahování. Uživatel myší uchopí konektor jednoho uzlu a přetáhne ho na jiný konektor. Když přestane držet tlačítko myši, je vytvořeno spojení, pokud to zvolené konektory umožňují - například spojení dvou vstupních konektorů nebude umožněno.

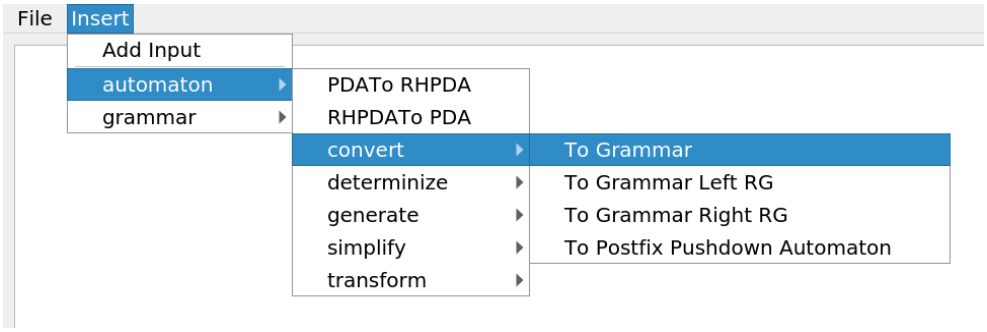
Důležitější je indikace v průběhu tvorby nového propojení. Je vhodné v tuto chvíli zobrazovat dočasné propojení mezi počátečním uzlem a kurzorem myši. Pomocí různých barev je možné uživateli předat informaci o tom, zda je spojení, které chce vytvořit, vůbec možné.

### 2.2.3 Přidávání nových uzlů

K přidání nového uzlu algoritmu na pracovní plochu bylo třeba stisknout jedno z tlačítek v panelu na levé straně. Následně se uzel algoritmu objevil na určitém místě, které bylo pro každý algoritmus specifikováno v kódu. Až poté je možné přesunout uzel na žádané místo.

Jednou alternativou řady tlačítek je použití rozbalovacího seznamu se stromovou strukturou. Tento způsob umožňuje zobrazit více algoritmů při zachování výšky okna. Navíc je možné využít "drag'n drop" pro umisťování algorimů na pracovní plochu. Uživatel uchopí vybraný algoritmus ze seznamu a rovnou ho přetáhne na cílové místo.

Další možností je využití lišty na horním okraji okna. Zde je možné vytvořit menu s rozbalovací stromovou strukturou. Uživatel si zvolí algoritmus, klikne na příslušné místo na pracovní ploše a tím vytvoří nový uzel.




Obrázek 22: Ukázka menu pro přidávání nových uzlů

### 2.2.4 Zadávání vstupních dat

Na pracovní ploše se při spuštění nachází vstupní uzel s jedním výstupem. Při kliknutí pravým tlačítkem na tento uzel máme možnost otevřít okno s nastavením vstupu. Toto okno obsahuje UI prvky pro načtení a uložení souboru, textové pole a přepínač formátu a tlačítko pro vytvoření náhodného automatu.

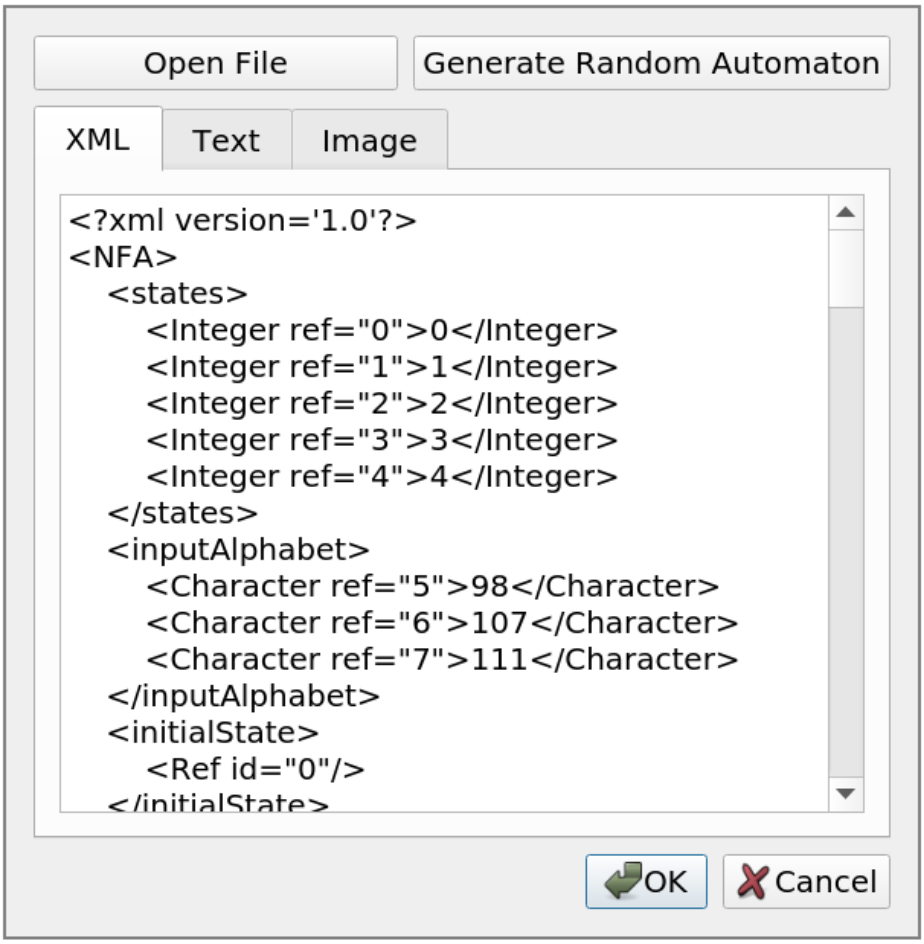
Způsob práce se soubory je poměrně nešťastný, je třeba nejprve zvolit cestu k souboru a poté stisknout tlačítko pro načtení či uložení. Možnost uložení automatu v tomto okně je celkem zbytečná, tudíž by bylo lepší těchto několik prvků nahradit pouze jedním tlačítkem pro otevření souboru.

Při výběru souboru pro otevření bylo možné zvolit textové i XML soubory, ale uživatel musel následně zvolit správný formát pomocí přepínače. Z principu fungování těchto formátů lze jednoduše odvodit, že neexistují data, která by byla validní pro oba dva. Můžeme tedy zkusit přečíst soubor v obou formátech a zvolit ten, pro který se to podaří, nebo se pokusit uhádnout správný formát podle koncovky načítaného souboru.

### 2.2.5 Zobrazení výsledků

Na pracovní ploše se vždy nachází pouze jeden výstupní uzel. V okně s nastavením tohoto uzlu si můžeme vybrat, zda chceme výstup uložit do souboru či rovnou zobrazit, a také v jakém formátu. Na výběr jsou dva textové formáty - XML a vlastní formát podobný formátu DOT, dále také obrázek formátu PNG a vektorový formát SVG.

Pokud uživatel zvolí přímé zobrazení, po skončení běhu všech algoritmů se zobrazí dialogové okno s výsledkem, které neumožňuje žádnou další akci. Tento výsledek není možné v tomto okně přímo uložit, nebo zobrazit v jiném formátu.

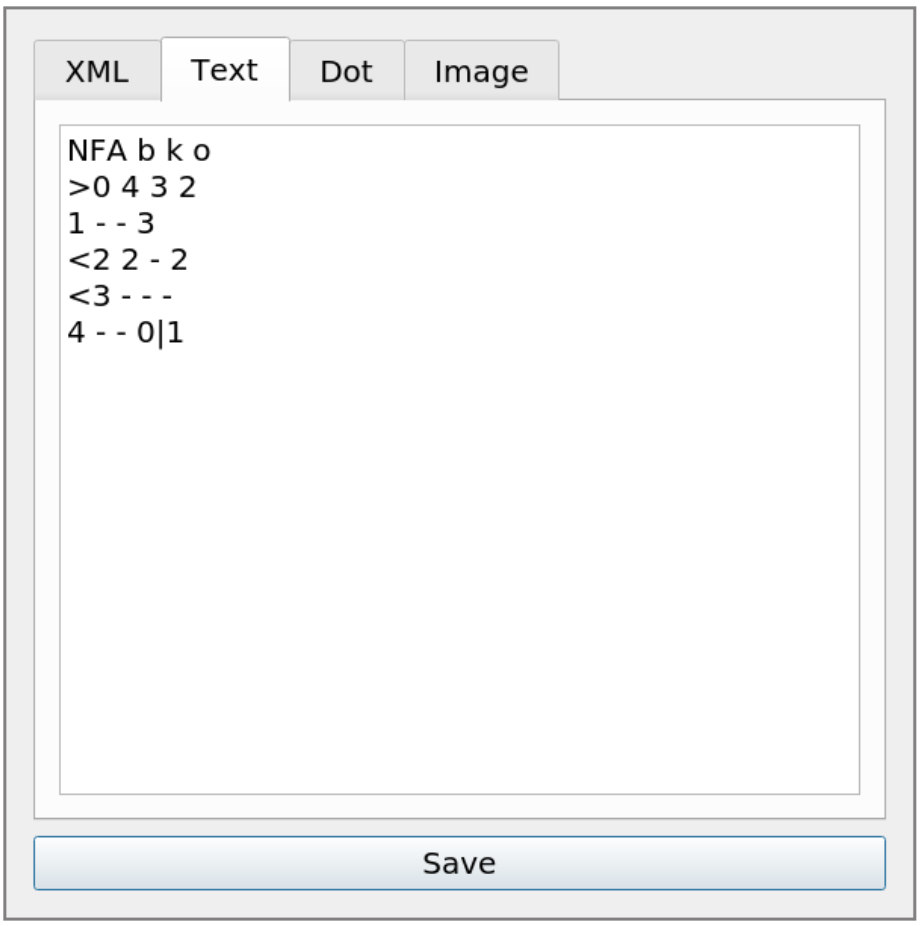



Obrázek 23: Ukázka nového okna pro zadávání vstupu

Alternativou k tomuto přístupu by bylo sloučení těchto dvou oken. Po skončení provádění plánu se zobrazí okno, které umožní uživateli zobrazit výsledek ve všech dostupných formátech a poté taky tato data uložit do souboru.

### 2.2.6 Ukládání a načítání pracovní plochy

Při každém spuštění se uživateli zobrazí téměř prázdná pracovní plocha, obsahující pouze jeden vstupní a jeden výstupní uzel. Je tedy třeba pokaždé znovu nakonfigurovat požadovaný plán. Možnost uložit stav pracovní plochy a při dalším spuštění jej znovu načíst by tedy mohla ušetřit spoustu času.




Obrázek 24: Ukázka okna s výsledkem v textové podobě

### 2.2.7 Přidání dalších datových typů

Aplikace byla navržena tak, aby umožňovala práci pouze s automaty. Knihovna ALT ale poskytuje implementaci dalších datových struktur, jmenovitě třeba gramatiky, stromy, či regulární výrazy. Jelikož implementace GUI aplikace neklade vnitřně žádná omezení na data, která jsou algoritmy používána, tak je pro povolení využívání dalších datových typů pouze rozšíření vstupu a výstupu dat.

Okno pro nastavení dat vstupního uzlu musí být rozšířeno tak, aby bylo schopné zpracovat textové reprezentace ostatních typů a obdobně je třeba do okna s výsledkem přidat možnost zobrazování těchto dat ve všech dostupných formátech.

### 2.2.8 Dostupné algoritmy

Seznam algoritmů dostupných v GUI aplikaci byl omezen pouze na 18 algoritmů operující nad automaty. Pro přidání nového algoritmu bylo třeba změnit seznam v zdrojovém kódu.

Vzhledem k předchozímu bodu by tento seznam měl být rozšířen o algoritmy pracující například nad gramatikami. Je tedy vhodné nahradit neměnný seznam algoritmů systémem, který při spuštění aplikace získá všechny podporované algoritmy z knihovny ALT. Tento systém může využít API popsané v 2.1.1.

Kapitola 3

# Paralelizace

Jelikož spouštění algoritmů poskytované knihovnou ALT může být na určitých vstupních datech časově náročné, bylo by vhodné co nejvíce využít výpočetní výkon, který nám současné procesory poskytují. Čistě sekvenční zpracování může použít pouze jedno procesorové jádro. Dnes jich máme ale k dispozici většinou více, v běžných počítačích můžeme vidět dvě až čtyři jádra, ve specializovaných počítačích a serverech mnohem více.

Sekvenční verze využívá rekurze. Vyhodnocovat začneme koncový uzel, který je pouze jeden. Ten nejdříve projde všechny uzly připojené na jeho vstupy a spustí je. Tyto uzly následně provedou to samé s jejich vstupními parametry. Rekurze se zastaví u počátečních uzlů, které při vyhodnocení vrátí hodnotu, kterou jim nastavil uživatel, například přečtením ze souboru. Následně postupujeme zpět stromem rekurze a spouštíme algoritmy, které v tuto chvíli mají vstupní data připravena.

Při paralelním zpracování musíme brát v potaz datové závislosti mezi uzly. Jednotlivé algoritmy jsou v knihovně ALT implementovány jako referenčně transparentní funkce - jejich výsledek závisí pouze na jejich vstupech. Jelikož vstupy algoritmů jsou často připojeny na výstupy jiných algoritmů, není tedy možné dvojice uzlů, které jsou na sobě závislé, zpracovávat paralelně. V tuto chvíli narážíme na problém plánování.

Plánovací problém můžeme definovat jako nalezení přiřazení úloh k jádrům procesoru tak, aby byly splněny určité podmínky a aby celkový čas provádění byl co nejmenší. Většinou může jedno jádro provádět pouze jednu úlohu najednou a jedna úloha může běžet jen na jednom jádře.

Možnost přerušení zpracování úlohy dělí tyto problémy na dvě kategorie, preemptivní a nepreemptivní plánování. V preemptivním plánování můžeme kdykoli průběh zpracování zastavit a úlohu přesunout na jiné jádro, případně

pokračovat její zpracování později. Nepreemptivní plánování vyžaduje aby úloha doběhla až do konce, či do bodu, kdy explicitně počítači povolí své pozastavení či přesunutí.

V následujícím seznamu popíšeme časté podmínky které se k plánování vztahují:

- Požadavek spuštění na specifických jádrech
- Nutnost "přestávky" mezi úlohami
- Fixní časy či intervaly spuštění
- Nutnost rozdělení úlohy na menší části
- Závislost mezi úlohami

Je možné ukázat, že problém plánování úloh bez závislostí s konstantním časem běhu je NP-úplný [4]. Některé varianty těchto problému je možné převést na problém obchodního cestujícího a tím pádem je zařadit do NP-těžkých problémů. Díky tomu budeme používat heuristická řešení.

### 3.0.1 Graf závislostí

Zapojení uzlů na pracovní ploše můžeme převést na graf závislostí. Ten definujeme jako graf $G = (V, E)$, kde $V$ je množina všech úloh a $E$ je množina hran, tedy uspořádaných dvojic úloh $a$ a $b$ takových, že úloha $a$ musí být splněna před úlohou $b$. Uzly odpovídají jednotlivým úlohám v grafu, počáteční a koncové uzly sice k sobě žádný výpočet vázaný nemají, každopádně je v grafu závislostí budeme také reprezentovat. Každé spojení mezi dvěmi uzly $x$ a $y$ převedeme na orientovanou hranu mezi jejich odpovídajícími úlohami $a$ a $b$ tak, že hrana povede z $a$ do $b$, když výstup uzlu $x$ je zapojen na vstup uzlu $y$.

Pokud takový graf obsahuje cyklus, není možné nalézt pořadí provádění. Algoritmy pro řešení plánovacích problémů jsou většinou schopny tyto cyklické závislosti detekovat, ale často bývá vhodné hledání cyklů provést odděleně a dříve, abychom mohli tuto informaci lépe zobrazit uživateli. Budeme tedy dále předpokládat, že graf závislostí je bez cyklů.

## 3.1 Naivní řešení

Naivní řešení nevyužívá žádného algoritmu na řazení uzlů, místo toho se spoléhá na dvě synchronizační primitiva, konkrétně zámek a podmíněnou proměnnou. Problémem je ale nutnost častého procházení seznamu všech uzlů.

Podmíněná proměnná umožňuje vláknům pozastavit svůj běh a čekat na probuzení jiným vláknem. V jazyce C++ je implementována pomocí třídy `std::condition_variable`. Pokud se chce vlákno uspat, musí nejdříve zamknout zámek (instanci `std::mutex`), který následně předá metodě `wait`. Na jedné podmíněné proměnné může v jednu chvíli být uspáno více vláken. Pro probuzení můžeme použít metodu `notify_one`, která probudí jedno ze spících vláken, či metodu `notify_all`, která probudí všechna.

Jedním průchodem můžeme identifikovat uzly, které žádné vstupy nemají, neboť všechny uzly připojené k jejich vstupům již byly zpracovány. Všechny tyto uzly můžeme začít paralelně spouštět. Ve chvíli, kdy nemáme žádné další uzly ke zpracování, vlákno uspíme na podmíněné proměnné. Tuto proměnnou následně probudí uzly při jejich dokončení a tím nám dávají signál, že je třeba opět všechny uzly zkontrolovat a případně některé z nich začít provádět.

Toto řešení je velmi jednoduché na implementaci, potřebuje pouze jednu instanci zámku a podmíněné proměnné. Vzhledem k tomu, že počet uzlů bude většinou velmi nízký, tak bude bude malá i režie a tento způsob paralelizace je překvapivě použitelný.

## 3.2 Topologické setřídění

Další možnost řešení využívá algoritmu k topologickému seřazení uzlů, v kombinaci s použitím synchronizačních primitiv "promise/future".

Topologické setřídění orientovaného grafu je takové lineární uspořádání jeho vrcholů, kde pokud existuje hrana z vrcholu `u` do vrcholu `v` tak je v tomto uspořádání vrchol `u` před vrcholem `v`. Pokud provedeme na našem grafu toto setřídění a poté budeme v tomto pořadí jednotlivé uzly provádět, nemůžeme narazit na situaci, kdy data na vstupu nějakého uzlu nebudou ještě dostupná.

Samotné topologické setřídění není dostačující pro úspěšnou paralelizaci. Využijeme tedy synchronizačních primitiv promise a future k vytvoření asynchroního komunikačního kanálu. Promise, která je v C++ reprezentována třídou `std::promise`, je vstupním bodem tohoto kanálu. Při vytvoření je prázdná, můžeme do ní ale později uložit buď hodnotu, nebo výjimku indikující chybu při výpočtu.

Pomocí metody `get_future` můžeme z promise získat future, instanci třídy `std::future`, která tvoří konec komunikačního kanálu. Tyto dvě instance jsou propojeny sdíleným vnitřním stavem a hodnotu, či výjimku, uloženou do promise, lze poté získat z future. Ve chvíli, kdy v promise ještě nic uloženo není, můžeme na future čekat a uspat vlákno dokud není hodnota dostupná.

**Algoritmus 1** Topologické setřídění pomocí prohledávání do hloubky

```
function TopoSortDFS(V, E)                    ▷ graf závislostí G = (V, E)
    L ← []                                    ▷ výsledný seznam

    function Visit(u)
        if u je označen then
            return
        end if
        Označ u
        for all v ∈ V, (u, v) ∈ E do
            Visit(v)
        end for
        vlož u na začátek L
    end function

    for all u ∈ V do
        if u není označen then
            Visit(u)
        end if
    end for
end function
```

Topologicky setřídíme uzly a začneme je procházet. K danému uzlu vytvoříme nové vlákno a instanci `std::promise`, najdeme jeho vstupní uzly a z jejich `std::promise` získáme instance `std::future`. Na těchto instancích poté zavoláme metodu `wait`. Ve chvíli kdy všechny tyto metody vrátí, víme, že všechna vstupní data jsou připravena. Hodnoty z `std::future` předáme na vstupy algoritmu, spustíme ho a výstup uložíme do naší `std::promise`. To způsobí probuzení všech propojených `std::future` a pokračování výpočtu.

Tento způsob zpracování sice musí vytvořit vlákno pro každý uzel, ale tato vlákna budou uspána, dokud nebudou vstupní data pro jejich uzel dostupná.

## 3.3 Coffmanův-Grahamův algoritmus

Třetí způsob řešení použije Coffmanův-Grahamův algoritmus pro naplánování celého průběhu výpočtu předem a nevyžaduje žádná synchronizační primitiva, jako předchozí algoritmy.

Tento algoritmus rozdělí vrcholy našeho grafu do několika úrovní tak, aby každá úroveň měla velikost menší nebo rovnou parametru $W$ a zároveň aby se každý uzel nacházel v nižší úrovni, než všechny uzly na kterých závisí. Každou úroveň je poté možné zpracovat paralelně.

Nevýhodou tohoto algoritmu je předpoklad toho, že zpracování každého uzlu bude trvat stejnou dobu. V tuto chvíli ale nemáme jednoduchý způsob jak odhadnout dobu trvání každé úlohy a budeme tedy tento předpoklad požadovat za splněný.

Pro $W = 2$ tento algoritmus nalezne optimální řešení s počtem úrovní $O$ [5]. Pokud $W > 2$, výsledný počet úrovní je maximálně $(2 - 2/W) * O$ [6].

Prvním krokem algoritmu je sestrojení tranzitivní redukce - graf se stejnou množinou vrcholů a co nejmenším počtem hran tak, aby tranzitivní uzávěr obou grafů byl stejný. Pokud z uzlu `u` do uzlu `v` existuje cesta v původním grafu, musí také existovat v redukovaném grafu.

**Algoritmus 2** Tranzitivní redukce

```
M ← matice sousednosti grafu G = (V, E)

for k ← 1, |V| do                                   ▷ tranzitivní uzávěr
    for i ← 1, |V| do
        for j ← 1, |V| do
            if M_ik = 1 ∧ M_kj = 1 then
                M_ij ← 1
            end if
        end for
    end for
end for

for j ← 1, |V| do                                   ▷ tranzitivní redukce
    for i ← 1, |V| do
        if M_ij = 1 then
            for k ← 1, |V| do
                if M_jk = 1 then
                    M_ik ← 0
                end if
            end for
        end if
    end for
end for
```

Tranzitivní redukci sestrojíme následujícím způsobem. Nejprve vytvoříme matici sousednosti grafu. Následně na tuto matici aplikujeme variantu Floyd-Warshallova algoritmu, kterým spočteme tranzitivní uzávěr matice. Následně se podíváme na každou trojici uzlů `i`, `j`, `k`. Pokud existuje hrana z uzlu `i` do uzlu `j` a z uzlu `j` do uzlu `k`, odebereme z grafu hranu z uzlu `i` do uzlu `k`, pokud existuje.

Druhým krokem bude provedení topologického setřídění. Definice pro tento kroku určuje použití Kahnova algoritmu. Tento algoritmus je také lehce modifikován, takže v případě výběru z více uzlů je bere v lexikografickém pořadí. Tato modifikace je důležitá hlavně pro ostatní aplikace tohoto algoritmu, například pro vrstvené vykreslování grafů.

**Algoritmus 3** Kahnův algoritmus pro topologické setřídění

```
function TopoSortKahn(V, E)                    ▷ graf závislostí G = (V, E)
    L ← []                                                ▷ výsledný seznam
    S ← množina všech uzlů bez příchozích hran
    while |S| > 0 do
        odeber uzel u z S                          ▷ v lexikografickém pořadí
        vlož u na konec L
        for ∀v ∈ V, (u, v) ∈ E do
            E ← E \ (u, v)
            if v nemá příchozí hrany then
                S ← S ∪ v
            end if
        end for
    end while
end function
```

Posledním krokem je samotné přiřazování uzlů do vrstev. Projdeme topologické uspořádání v opačném pořadí. Každý vrchol `v` umístíme do úrovně s co nejmenším číslem tak, že všechny uzly které na uzlu `v` závisí budou ve vyšší úrovni a žádná úroveň nebude obsahovat více než `W` vrcholů.

Nyní můžeme projít úrovně sestupně a všechny úlohy paralelně zpracovat. Posledním krokem algoritmu máme zajištěno, že všechny závislosti každé úlohy již byly provedeny v některé z předchozích úrovní.

Díky svému předpokladu doby běhu úloh není tento algoritmus ideální, avšak pro naše účely je dostačující.

Kapitola 4

# Implementace

Původní návrh aplikace obsahoval celkem 3 vrstvy. Modelovou, která reprezentovala logickou část, tvořenou instancemi třídy `ModelBox`, respektive jejích potomků. Dále vrstvou grafickou, která se skládala z instancí `GraphicsBox`. Poslední vrstvou byly instance třídy `WrapperBox`. Tyto instance asociovaly jednotlivé uzly v grafické a modelové vrstvě. Třídy `GraphicsBox` a `ModelBox` navzájem na sebe neměly ukazatele, musely k sobě přistupovat přes svůj `WrapperBox`.

Existence třetí vrstvy byla zbytečná a přinášela do kódu nejasnosti. Vztah mezi modelovou a grafickou vrstvou je navíc jednosměrný, modelová vrstva nepotřebuje přistupovat k vrstvě grafické. Tato vrstva byla tedy odebrána a třída `GraphicsBox` nyní vlastní instanci třídy `ModelBox`. Třída `GraphicsBox` je potomkem `QGraphicsObject`, takže její životní cyklus je interně spravován knihovnou Qt.

## 4.1 Grafická vrstva

V grafické vrstvě původně existovaly celkem čtyři třídy. Třídy `GraphicsBox` a `InputGraphicsBox` byly zachovány. Třída `DoubleGraphicsBox` původně zobrazovala uzly se dvěma vstupy. Tato funkcionalita byla implementována jinde a tato třída byla odstraněna.

### 4.1.1 Vstupy a výstupy

Vstupní a výstupní konektory uzlů jsou reprezentovány třídou `ConnectionBox` a jejími potomky `InputConnectionBox` a `OutputConnectionBox`. Každá instance `GraphicsBox` k sobě může mít přiřazen jeden `OutputConnectionBox` a několik instancí `InputConnectionBox`. Tyto třídy zajišťují možnost propo-

jování uzlů pomocí přetahování myší a vykreslení dočasného spojení v průběhu propojování.

Třída `GraphicsBox` ve svém konstruktoru použije virtuální metodu `ModelBox::getMaxInputCount`, kterou zjistí jaký počet vstupů může daný uzel mít. Pro `AlgorithmModelBox` je tento počet určen příslušejícím algoritmem, pro `InputModelBox` a `OutputModelBox` je počet vstupů stálý, konkrétně 0 a 1.

Obdobně je použita funkce `ModelBox::canHaveOutput`, která určuje, zda daný uzel má výstup. Tato funkce pro `OutputModelBox` vrací `false` a pro ostatní `true`. V dalším vývoji by tato funkcionalita mohla být upravena tak, aby bylo možné mít více než jeden výstup.

Po vzoru původní třídy `GraphicsConnection` je finální propojení dvou uzlů nyní reprezentováno třídou `Connection`. Ta zajišťuje propojení jednotlivých instancí `ConnectionBox` a také přímé propojení uzlů modelové vrstvy.

## 4.2 Modelová vrstva

Původní implementace obsahovala pět tříd pro reprezentaci uzlů v modelové vrstě. Třída `ModelBox` a její potomci `InputModelBox` a `OutputModelBox` se nacházejí i v nové verzi. Tyto varianty pro vstupní a výstupní uzly se velmi málo liší od jejich předka `ModelBox`, na rozdíl od jejich původních implementací, které zastávaly i některé funkce příslušející grafické vrstvě.

Třídy `SingleModelBox` a `DoubleModelBox`, které reprezentovaly algoritmové uzly s jedním, respektive dvěma vstupními parametry, byly sloučeny do třídy `AlgorithmModelBox`. Tato třída umožňuje libovolné množství vstupů, a při vytvoření získá jejich počet z instance třídy `Algorithm`, který je předán parametrem.

## 4.3 Registr algoritmů

Informace poskytované knihovnou ALT o dostupných algoritmech jsou předávány v složité formě. Návratové typy metod třídy `AlgorithmRegistry` využívají kombinace několika `std::tuple` a `std::pair` a bylo by náročné s nimi pracovat na více místech v programu. Třída `Registry` tedy pracuje s registrem algorimů v knihovně a převádí tyto informace do jednodušší podoby.

Algoritmy jsou poté reprezentovány jako instance třídy `Algorithm`, které jsou sdílené pro celou aplikaci. Tato třída obsahuje jméno algoritmu, jeho zařazení do skupin a seznam jeho možných vstupních a výstupních parametrů.

Implementace algoritmů jsou totiž přetížené pro různé vstupní typy a mohou mít různý počet parametrů a to by způsobilo komplikace v grafické vrstvě. Pokud takový algoritmus detekujeme, přeskočíme ho. Množství takovýchto algoritmů je naštěstí velmi nízké, jsou to spíše pomocné funkce, a jejich vyřazením nepřicházíme o funkcionalitu.

## 4.4 Problémy implementace

V současné implementaci zůstává stále několik problémů. Většina z nich je způsobena velikou složitostí knihovny Qt a vyžadovala by podrobnější znalosti na jejich vyřešení.

Jedním z hlavních problémů je chování pracovní plochy při vytvoření nového uzlu. Ta se vždy automaticky vystředí tak, aby bylo co nejvíce uzlů viditelný. Bylo by vhodné toto chování upravit tak, aby plocha zůstala v původní poloze, ale aby bylo možné plochou pohybovat.

Dalším problémem je chování dialogového okna pro nastavení vstupu. V něm existjí dvě textová pole pro různé formáty vstupních dat. Při úpravách v jednom poli je třeba ho synchronizovat do toho druhého a to způsobuje problémy s nechtěným pohybem kurzoru.

Kapitola 5

# Testování

## 5.1 Unit testing

Unit testing je metoda testování zdrojového kódu, při které jsou testovány jednotlivé malé části programu a ověřována jejich základní funkčnost.

Testy byly vytvořeny pomocí knihovny Catch2 [7]. Tato knihovna poskytuje preprocesorová makra k vytváření jednotlivých testů a porovnávání skutečných hodnot či stavů s očekávanými. Tyto testy jsou určeny k automatickém spouštění a poskytují síto pro velké chyby, které by mohly programátorovi uniknout.

Testována je hlavně modelová vrstva a část programu, která komunikuje s knihovnou ALT. Testovány je následující funkcionalita:

- Inicializace registry algoritmů
- Správné informace o parametrech a názvech algoritmů
- Základní struktura počátečních, koncových a algoritmových uzlů
- Připojování a odpojování uzlů
- Odpojení a připojení jiného uzlu
- Nemožnost připojení více uzlů na jeden vstup
- Připojení jednoho výstupu na více vstupů
- Práce s neexsistujícími připojovacími konektory

## 5.2 Uživatelské testování

Pro úplnost bylo provedeno jednoduché uživatelské testování po vzoru [2]. Cílem bylo porovnat původní a novou verzi aplikace, se zaměřením na nové, či vylepšené funkce, a identifikace chyb a oblastí, které stále vyžadují změny.

Testování proběhlo s dvěma subjekty, studenty FIT ČVUT. Oba měli znalosti z oblasti automatů a k nim příslušným algoritmům, ale nikdy nepracovali s knihovnou ALT, ani s touto GUI aplikací. Testovací úlohy spočívaly v provedení stejných činností v obou verzích implementace, pokud to bylo možné, a porovnání náročnosti těchto úkonů.

### 5.2.1 Testovací úlohy

- Načtěte stav pracovní plochy ze souboru, obraťte pořadí provádění algoritmů a znovu stav uložte do původního souboru
- Nahraďte vstupní data náhodně generovaným automatem a výsledek uložte ve všech možných formátech
- Smažte všechny uzly z pracovní plochy
- Zapojte zřetězení dvou automatů, které načtete ze souboru
- Změňte pořadí zřetězení z předchozí úlohy

Oba uživatelé se shodli, že nová implementace je intuitivnější hlavně v oblasti propojování uzlů a existence konektorů pro rozlišení uzlů s různým počtem vstupů. Také ocenili zlepšení dialogového okna s výstupem a možnost vložení uzlu přímo na zvolené místo.

Co se nové funkcionality týče, načítání a ukládání stavu pracovní plochy se jim podařilo okamžitě, jelikož se tato funkce nachází na stejném místě, jako v ostatních programech.

Obecně byla upravená verze hodnocena pozitivně a uživatelům se s ní pracovalo lépe.

### 5.2.2 Chybějící funkce

V průběhu dalšího vývoje a integrace tohoto GUI do knihovny ALT by bylo vhodné doplnit tuto aplikaci o další funkce, například:

- Možnost návrhu vstupního automatu

- Hromadné přesouvání uzlů
- Povolení více koncových uzlů
- Validace datových typů parametrů
- Prohlížení průběžných výsledků
- Dokumentace k jednotlivým algoritmům

# Závěr

Hlavním cílem této bakalářské práce bylo vylepšení existující GUI aplikace pro práce s algoritmovou knihovnou ALT.

V první kapitole byl prozkoumán návrhový vzor "pipes and filters" a jeho použití v aplikacích Unreal Engine 4 a Blender.

Druhá kapitola se věnovala analýze struktury knihovny ALT a způsobu získání informací o jednotlivých algoritmech. Dále se tato kapitola věnovala rozboru jednotlivých nedostatků původní implementace a byly navrženy jejich vylepšení, s využitím znalostí získaných v první kapitole.

Ve třetí kapitole jsme prozkoumali problematiku paralelního plánování závislých úloh a navrhli několik možných způsobů řešení. Popsali jsme synchronizační primitiva, která mohou být využita k řešení tohoto problému. Také jsme využili Coffmanova-Grahamova algoritmu a topologického setřídění grafu závislostí.

Ve čtvrté kapitole jsme se věnovali popisu implementace nových funkcí a porovnání s předchozí verzí.

V poslední kapitole jsme se věnovali automatickým jednotkovým testům a základnímu uživatelskému testování. Také jsme navrhli další možná vylepšení.

Výsledkem této práce je vylepšená aplikace pro práci s knihovnou ALT, která neobsahuje některé nedostatky vyplývající z původního návrhu a implementace. Do budoucna by bylo vhodné doplnit další funkce, každopádně tato aplikace už začíná být použitelnou, ne pouze prototypem.

# Reference


[1] Žák, Martin: *Automatová knihovna – vnitřní a komunikační formát.* Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.

[2] Mareš, Václav: *GUI k automatové knihovně ALIB.* Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2017.

[3] Blender Foundation: Blender shader example. [Online, cit. 8.5.2018]. Dostupné z: `https://docs.blender.org/manual/en/dev/_images/compositing_types_color_mix_contrast-enhancement.png`

[4] Ullman, J.: NP-complete scheduling problems. *Journal of Computer and System Sciences*, ročník 10, č. 3, 1975: s. 384 – 393, ISSN 0022-0000, doi:https://doi.org/10.1016/S0022-0000(75)80008-0. Dostupné z: `http://www.sciencedirect.com/science/article/pii/S0022000075800080`

[5] Coffman, E. G.; Graham, R. L.: Optimal scheduling for two-processor systems. *Acta Informatica*, ročník 1, č. 3, Sep 1972: s. 200–213, ISSN 1432-0525, doi:10.1007/BF00288685. Dostupné z: `https://doi.org/10.1007/BF00288685`

[6] Lam, S.; Sethi, R.: Worst Case Analysis of Two Scheduling Algorithms. *SIAM Journal on Computing*, ročník 6, č. 3, 1977: s. 518–536, doi:10.1137/0206037, `https://doi.org/10.1137/0206037`. Dostupné z: `https://doi.org/10.1137/0206037`

[7] Catch Org: Catch 2. 2018-04-06. Dostupné z: `https://github.com/catchorg/Catch2`

Příloha A

# Seznam použitých zkratek

**GUI** Graphical user interface

**XML** Extensible markup language

**ALIB** Automata Library

**ALT** Algorithms Library Toolkit

**DOT** textový formát pro popisování grafů

Příloha **B**

# Obsah přiloženého elektronického média

- `thesis.pdf` ..... text práce ve formátu PDF
- `readme.txt` ..... stručný popis obsahu elektronického média
- `thesis` ..... zdrojový text práce ve formátu LaTeX
- `src`
  - `automata-library` ..... zdrojový kód knihovny ALT
  - `gui` ..... zdrojový kód implementace