

# Komponenta výukového serveru TI - P-úplné problémy

Component of Teaching Server for Theoretical Computer Science - Pcomplete problems

Tomáš Kirnig

Bakalářská práce

Vedoucí práce: Ing. Martin Kot, Ph.D.

Ostrava, 2025

# Zadání bakalářské práce

Student:

**Tomáš Kirnig**

Studijní program:

B0613A140014 Informatika

Téma:

Komponenta výukového serveru TI - P-úplné problémy  
Component of Teaching Server for Theoretical Computer Science - P-  
complete problems

Jazyk vypracování:

čeština

Zásady pro vypracování:

V rámci diplomových a bakalářských prací vzniká výukový server pro předměty teoretické informatiky. Jedná se o sadu dynamických webových stránek umožňujících studentům pochopení různých typů úloh a problémů. Na rozdíl od běžných výukových textů s pevně daným počtem ukázkových příkladů umí tyto stránky generovat libovolně mnoho ukázek na základě vstupů od uživatele. Cílem této konkrétní bakalářské práce je vytvořit komponentu pro pomoc s výukou tzv. P-úplných problémů.

Vytvořte dynamické webové stránky umožňující uživateli následující:

1. Simulovat výpočet řešení problému Monotone Circuit Value Problem (MCVP) a alespoň 2 dalších P-úplných problémů.
2. Vstupy těchto algoritmů bude moci uživatel zadávat třemi způsoby:
  - a) Vhodným, uživatelsky přívětivým, způsobem ručně.
  - b) Nechat si vstup vygenerovat zcela náhodně podle nastavených parametrů.
  - c) Vybrat z předpřipravené sady vhodně zvolených vstupů.
3. Bude možné si zobrazit převod instance problému MCVP na ty dva zvolené P-úplné problémy. Přitom:
  - a) Instanci MCVP pro převod bude možné zadat kterýmkoliv z výše uvedených způsobů.
  - b) Převod si bude moci uživatel krokovat se zobrazením slovního vysvětlení jednotlivých kroků.
  - c) Na převodem vytvořenou instanci bude opět možné použít výše požadovanou simulaci výpočtu řešení.

Seznam doporučené odborné literatury:

- [1] Miyano, S., Shiraishi, S., Shoudai, T.: "A List of P-Complete Problems", Kyushu University, RIFIS-TR-CS-17, December 1990, dostupné z URL: [https://catalog.lib.kyushu-u.ac.jp/opac\\_download\\_md/3123/rifis-tr-17.pdf](https://catalog.lib.kyushu-u.ac.jp/opac_download_md/3123/rifis-tr-17.pdf)
- [2] Sawa, Z.: "Teoretická informatika", podklady pro přednášky, VŠB - Technická univerzita Ostrava, dostupné z URL: <https://www.cs.vsb.cz/sawa/ti/slides/ti-slides-03.pdf>
- [3] Papadimitriou, C.: Computational Complexity, Addison Wesley, 1993
- [4] Arora, S., Barak, B.: Computational Complexity: A Modern Approach, Cambridge University Press, 2009

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Martin Kot, Ph.D.**

Datum zadání: 01.09.2024

Datum odevzdání: 30.04.2025

Garant studijního programu: doc. Mgr. Miloš Kudělka, Ph.D.

V IS EDISON zadáno: 30.10.2024 09:50:55

## Abstrakt

Tato bakalářská práce se zabývá vývojem výukové webové aplikace pro demonstraci P-úplných problémů. Cílem je usnadnit studentům pochopení a procvičování těchto problémů. Aplikace se zaměřuje na tři P-úplné problémy: *Monotone Circuit Value Problem* (MCVP), *prázdnou gramatiku* a *kombinatorickou hru*. Práce zahrnuje implementaci modulů pro interaktivní zadávání vstupů (ručně, náhodnou generací nebo výběrem z připravené sady) a simulaci jejich řešení. Uživatel může také sledovat krokový převod instance MCVP na další zmíněné problémy a následně jejich řešení.

## Klíčová slova

Teoretická informatika, P-úplné problémy, Monotone Circuit Value Problem, webová aplikace, simulace, převod instancí

## Abstract

This bachelor's thesis focuses on the development of a teaching-oriented web application for illustrating P-complete problems. The main goal is to facilitate students' understanding and practice of such tasks. The application focuses on the *Monotone Circuit Value Problem* (MCVP), *Empty Grammar*, and the *Combinatorial Game*. The project implements modules for interactive input of problem instances (manually, via random generation, or by selecting from a pre-defined set) and provides a simulation of their solutions. Users can also observe a step-by-step reduction from an MCVP instance to the other mentioned problems and subsequently explore how those are solved. The resulting application demonstrates key concepts of theoretical computer science, including the notion of P-completeness, and provides a flexible basis for educational use.

## Keywords

Theoretical computer science, P-complete problems, Monotone Circuit Value Problem, web application, simulation, instance reduction

# Obsah

Seznam použitých symbolů a zkratk	7
Seznam obrázků	8
Seznam tabulek	9
<b>1 Úvod</b>	<b>10</b>
<b>2 Použité technologie a architektura</b>	<b>13</b>
2.1 Webové technologie . . . . .	13
2.2 React a Vite . . . . .	13
2.3 Architektura a struktura kódu . . . . .	14
2.4 Vizualizace grafů . . . . .	14
2.5 Stylování a responzivita . . . . .	14
2.6 Správa vstupů a výstupů . . . . .	15
2.7 Notifikace a zpětná vazba . . . . .	15
2.8 Vývojové nástroje . . . . .	15
2.9 Nasazení aplikace . . . . .	15
<b>3 Monotone Circuit Value Problem</b>	<b>16</b>
3.1 Teoretický základ . . . . .	16
3.2 Formát vstupu a parsování výrazů . . . . .	16
3.3 Vyhodnocení obvodu . . . . .	17
3.4 Interaktivní editace obvodu . . . . .	18
3.5 Generování náhodných obvodů . . . . .	19
3.6 Vizualizace obvodu . . . . .	19
3.7 Převod na kombinatorickou hru . . . . .	20
3.8 Převod na bezkontextovou gramatiku . . . . .	21
3.9 Předpřipravené sady . . . . .	22

<b>4</b>	<b>Kombinatorická hra</b>	<b>23</b>
4.1	Teoretický základ . . . . .	23
4.2	Formát vstupu . . . . .	24
4.3	Interaktivní editace grafu . . . . .	24
4.4	Generování náhodných her . . . . .	25
4.5	Předpřipravené sady . . . . .	25
4.6	Vizualizace grafu . . . . .	26
4.7	Algoritmus analýzy . . . . .	26
4.8	Krokové vyhodnocení . . . . .	27
<b>5</b>	<b>Prázdnost bezkontextových gramatik</b>	<b>28</b>
5.1	Teoretický základ . . . . .	28
5.2	Formát vstupu . . . . .	29
5.3	Manuální zadání gramatiky . . . . .	29
5.4	Generování náhodných gramatik . . . . .	30
5.5	Předpřipravené sady . . . . .	32
5.6	Algoritmus vyhodnocení . . . . .	32
5.7	Vizualizace derivačního stromu . . . . .	34
5.8	Krokové vyhodnocení . . . . .	35
<b>6</b>	<b>Závěr</b>	<b>36</b>
	<b>Literatura</b>	<b>37</b>
	<b>Přílohy</b>	<b>37</b>
<b>A</b>	<b>Dlouhý zdrojový kód</b>	<b>38</b>

# Seznam použitých zkratek a symbolů

MCVP	– Monotone Circuit Value Problem
P	– Třída problémů řešitelných v polynomiálním čase
NC	– Nick's Class – třída efektivně paralelizovatelných problémů
API	– Application Programming Interface – Aplikační programové rozhraní
CFG	– Context-Free Grammar – Bezkontextová gramatika
DAG	– Directed Acyclic Graph – Orientovaný acyklický graf
DFS	– Depth First Search – Algoritmus průchodu do hloubky
DOM	– Document Object Model – Objektový model dokumentu
HTML	– Hypertext Markup Language – Hypertextový značkový jazyk
JSON	– JavaScript Object Notation – Objektový zápis JavaScriptu
LS	– Levá strana
PS	– Pravá strana
SPA	– Single Page Application – Jednostránková webová aplikace
UI	– User Interface – Uživatelské rozhraní

# Seznam obrázků

1.1	Porovnání růstu funkcí v Big O notaci [2]	11
2.1	Příklad notifikační zprávy v aplikaci	15
3.1	Výběr způsobu zadání vstupu pro MCVP	17
3.2	Ovládací prvky pro interaktivní editor obvodů MCVP	19
3.3	Vizualizace MCVP obvodu pomocí TreeRenderCanvas	20
3.4	Ovládací prvky pro import a export MCVP obvodů	22



# Seznam tabulek

# Kapitola 1

## Úvod

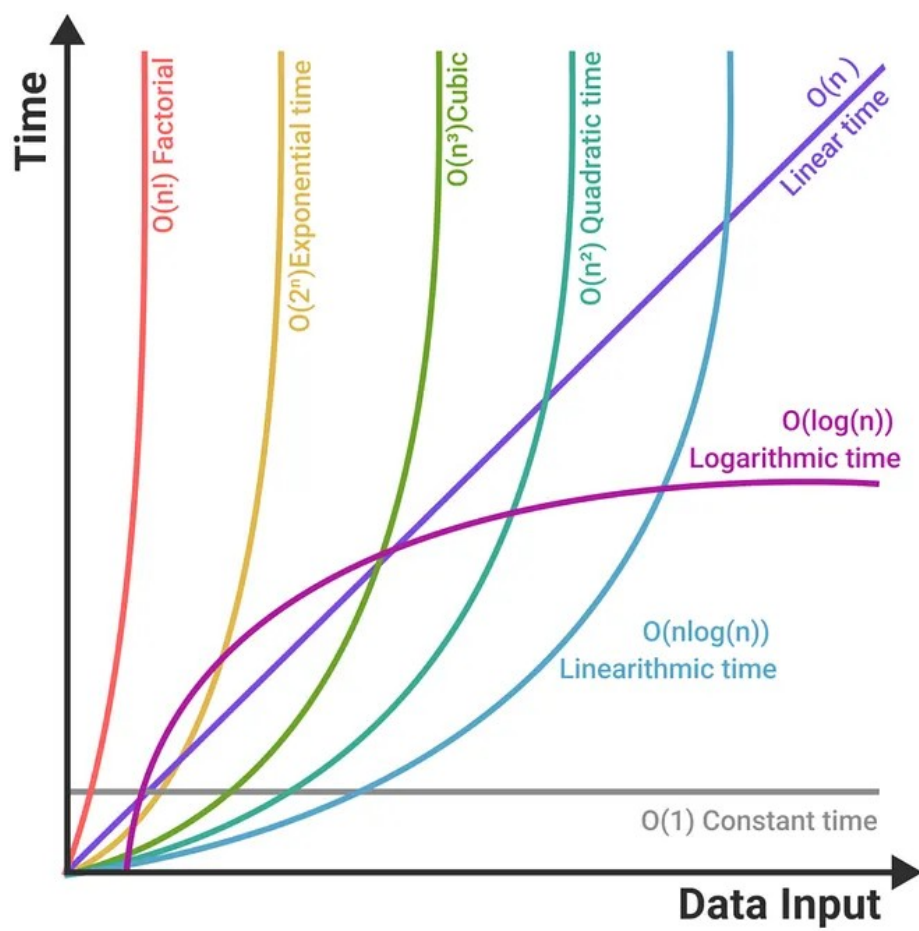
Teoretická informatika poskytuje způsob jak zkoumat, porozumět a optimalizovat možnosti a limity výpočetních systémů. Hlavním bodem tohoto zkoumání je teorie složitosti, která klasifikuje problémy na základě zdrojů potřebných pro jejich vyřešení, jako je čas a paměť. Jednou z nejvýznamnějších složitostních tříd je třída  $P$ , zahrnující problémy řešitelné v polynomiálním čase – tedy takové, jejichž časová složitost lze vyjádřit jako  $O(n^k)$  pro nějakou konstantu  $k$  (například  $k = 2$  pro kvadratickou složitost,  $k = 3$  pro kubickou), kde  $n$  je velikost vstupu – na deterministickém Turingově stroji [1]. Přestože jsou problémy v této třídě obvykle pokládány za „efektivně řešitelné“, projevují se mezi nimi významné odlišnosti, obzvláště když začneme řešit jejich paralelizovatelnost.

V tomto kontextu hraje klíčovou roli podmnožina  $P$ -úplných problémů ( $P$ -complete problems).  $P$ -úplné problémy představují výpočetně nejnáročnější úlohy v rámci třídy  $P$ . Jsou to problémy, na které lze s logaritmickou pamětovou složitostí převést jakýkoliv jiný problém z třídy  $P$ . Hlavní problém u této podmnožiny problémů spočívá v předpokladu, že  $P$ -úplné problémy pravděpodobně nelze efektivně paralelizovat. To znamená, že tyto úlohy nespádají do třídy  $NC$ , která obsahuje problémy řešitelné v polylogaritmickém čase  $O(\log^k n)$  pomocí paralelního výpočtu s polynomiálně mnoha procesory [3]. Z toho vyplývá, že na rozdíl od  $NC$  problémů řešení  $P$ -úplných problémů vyžaduje sekvenční zpracování. Studium  $P$ -úplnosti nám tak pomáhá vymezit hranici mezi paralelizovatelným a čistě sekvenčním výpočtem.

Tento projekt představuje interaktivní ukázkou konceptu  $P$ -úplnosti prostřednictvím webové aplikace. Cílem je vytvořit nástroj, který umožní uživatelům vizualizovat a pochopit výpočet a převod mezi zvolenými  $P$ -úplnými problémy.

Jako hlavní problém byl zvolen *Monotone Circuit Value Problem* (MCVP), ve kterém se vyhodnocuje logický obvod složený z pouze dvou logických hradel [4] (tedy logický součin a součet). Pro demonstraci univerzálnosti konceptu  $P$ -úplnosti aplikace implementuje také simulace dvou dalších problémů:

- **Kombinatorické hry na grafu:** Úloha analyzující existenci vítězné strategie v deterministické hře dvou hráčů [4].



Obrázek 1.1: Porovnání růstu funkcí v Big O notaci [2]

- **Prázdnost bezkontextových gramatik:** Problém rozhodující, zda daná gramatika generuje neprázdný jazyk [4].

Hlavním přínosem vytvořené aplikace je možnost vizualizace nejen samotného řešení těchto úloh, ale především *převodů* (redukcí) mezi nimi. Uživatel může sledovat krokovou transformaci instance MCVP na instanci hry nebo gramatiky, což názorně ilustruje princip polynomiálních redukcí a vzájemnou převoditelnost P-úplných problémů [3].

Výsledkem práce je webová aplikace navržená jako flexibilní nástroj pro výuku. Umožňuje uživatelům pracovat s vlastními vstupy, generovat náhodné instance pro testování a využívat předpřipravené sady úloh.

Text práce je členěn do několika částí. Po úvodu následuje kapitola věnovaná použitým technologiím a architektuře aplikace. Jádro práce tvoří tři kapitoly, z nichž každá se detailněji věnuje jednomu z implementovaných problémů: nejprve Monotone Circuit Value Problem (MCVP), následně kombinatorické hry na grafu a nakonec problém prázdnosti bezkontextových gramatik. Závěr práce shrnuje dosažené výsledky a navrhuje možnosti dalšího rozšíření.

## Kapitola 2

# Použité technologie a architektura

Tato kapitola popisuje technologický základ vytvořené aplikace a její architekturu.

### 2.1 Webové technologie

Pro nejjednodušší distribuci výukové aplikace byly zvoleny webové technologie, které nabízejí řadu výhod oproti desktopovým nebo mobilním aplikacím. Webová aplikace nevyžaduje instalaci a běží v libovolném moderním webovém prohlížeči, což zajišťuje maximální dostupnost pro uživatele napříč různými platformami a operačními systémy. Dalším přínosem je snadná údržba – aktualizace aplikace se projeví u všech uživatelů bez nutnosti instalace nových verzí koncovým uživatelem.

### 2.2 React a Vite

Jako hlavní framework (aplikační rámec) pro vývoj uživatelského rozhraní byl zvolen *React* [5]. React je moderní JavaScriptová knihovna vyvinutá společností Meta (dříve Facebook), která umožňuje vytvářet interaktivní uživatelská rozhraní na bázi komponent. Hlavní výhodou Reactu je koncept *reaktivity* – uživatelské rozhraní se automaticky aktualizuje při změně dat bez nutnosti manuální manipulace s DOM (Document Object Model).

React využívá deklarativní přístup k tvorbě UI. Na rozdíl od tradičního imperativního programování, kde vývojář musí krok za krokem instruovat prohlížeč, jak upravit rozhraní, v Reactu stačí popsat, jak má výsledné rozhraní vypadat, a React se postará o potřebné změny v DOM. Tento přístup výrazně zjednodušuje vývoj komplexnějších aplikací a minimalizuje chyby spojené s nekonzistentním stavem.

Pro sestavení aplikace byl použit nástroj (bundler) *Vite* [6]. Vite zajišťuje přípravu všech souborů aplikace (kódu, stylů, obrázků) pro běh v prohlížeči. Během vývoje Vite umožňuje vidět změny v kódu okamžitě po uložení souboru bez nutnosti obnovovat stránku. Pro finální verzi aplikace Vite veškeré soubory optimalizuje a zmenší pro rychlejší načítání.

## 2.3 Architektura a struktura kódu

Aplikace je strukturována jako *Single Page Application* (SPA), kde celá aplikace běží v rámci jedné HTML stránky a navigace mezi jednotlivými moduly probíhá bez opětovného načítání stránky. Hlavní komponenta `App.jsx` funguje jako kořen aplikační struktury a spravuje globální stav aplikace pomocí React Hooks, především `useState` pro udržení aktuální stránky a předávaných dat.

Zdrojový kód je rozdělen do tří hlavních modulů podle řešených problémů – MCVP, kombinatorické hry a bezkontextové gramatiky (viz kapitoly 3, 4 a 5). Každý modul obsahuje vlastní logiku a vizualizační komponenty a je organizován do separátních složek podle funkcionality:

- **Hlavní komponentu** – kontejnerovou komponentu řídící celý modul
- **Utils** – pomocné funkce obsahující algoritmy (parsery, generátory, evaluátory)
- **InputSelectionComponents** – komponenty pro různé způsoby zadání vstupu
- **Vizualizační komponenty** – komponenty pro grafické zobrazení problémů a jejich řešení

Společné prvky jako tlačítka nebo modální okna jsou sdíleny mezi všemi moduly. Pro jednoduché nastavení grafů byly vytvořeny pomocné moduly v adresáři `src/Hooks` – modul `useGraphColors` spravuje barvy použité v grafických vizualizacích a modul `useGraphSettings` obsahuje parametry pro vzhled grafů (velikost uzlů, vzdálenosti apod.).

Tato struktura umožňuje snadné úpravy na jednom místě.

## 2.4 Vizualizace grafů

Pro vizualizaci grafů používá aplikace knihovnu *react-force-graph* [7], která je postavena na knihovně *D3.js*. Grafy jsou automaticky rozmístěny pomocí force-directed layoutu – uzly se vzájemně odpuzují a hrany je přitahují k sobě, což vytváří vizuálně přehledné uspořádání. Pro stromové struktury v MCVP modulu (viz kapitola 3) jsou uzly uspořádány do úrovní shora dolů, s kořenem nahoře a listy dole.

Během vyhodnocování se uzly a hrany v grafu vybarvují podle aktuálního stavu, což uživateli umožňuje sledovat průběh výpočtu.

## 2.5 Stylování a responzivita

Pro vzhled uživatelského rozhraní aplikace využívá framework *Bootstrap 5* [8]. Bootstrap poskytuje předpřipravené styly pro tlačítka, formuláře a další prvky rozhraní. Použití Bootstrapu urychlilo vývoj a zajistilo jednotný vzhled.

## 2.6 Správa vstupů a výstupů

Každý modul aplikace nabízí tři způsoby zadání vstupu:

1. **Manuální zadání** – uživatel zadává vstup pomocí textového pole nebo grafického editoru
2. **Náhodné generování** – aplikace vygeneruje náhodný příklad podle zadaných parametrů
3. **Připravené příklady** – výběr z předpřipravených ukázkových příkladů

Aplikace umožňuje ukládání a načítání dat pomocí souborů ve formátu JSON. Soubory lze načíst přetažením do aplikace nebo výběrem ze složky. Formát nahraného souboru je validován, aby nedošlo k chybám při zpracování neplatných dat.

## 2.7 Notifikace a zpětná vazba

Pro zobrazení upozornění a chybových hlášek aplikace používá knihovnu *react-toastify* [9]. Notifikace se objevují v rohu obrazovky a automaticky mizí po chvíli. Aplikace zobrazuje notifikace zejména při chybách ve vstupu, úspěšném uložení dat nebo varováních při neplatných operacích.

Notifikace poskytují okamžitou zpětnou vazbu pro uživatele.



Obrázek 2.1: Příklad notifikační zprávy v aplikaci

## 2.8 Vývojové nástroje

Pro zajištění kvality kódu aplikace používá nástroj *ESLint* [10], který kontroluje dodržování standardů a detekuje potenciální chyby v kódu.

## 2.9 Nasazení aplikace

Pro zveřejnění aplikace nástroj Vite připraví a optimalizuje všechny soubory. Protože veškerá logika aplikace běží přímo v prohlížeči uživatele, není potřeba žádný speciální server.

Aplikace je nasazena pomocí služby *Vercel*, která je přímo propojena s GitHub repozitářem. Při každé změně v hlavní větvi repozitáře dojde automaticky k novému nasazení aplikace. Toto řešení výrazně zjednodušuje proces aktualizace a zajišťuje, že živá verze aplikace je vždy aktuální.

## Kapitola 3

# Monotone Circuit Value Problem

### 3.1 Teoretický základ

Monotone Circuit Value Problem je základní problém v teorii složitosti, který patří mezi P-úplné problémy [4]. Jeho definice je následující:

- **Vstup:** Booleovský obvod bez negací, tvořený pouze hradly AND ( $\wedge$ ) a OR ( $\vee$ ), společně se vstupními proměnnými (pouze hodnoty 0 nebo 1 - false nebo true).
- **Výstup:** Hodnota výstupního uzlu obvodu (kořene stromu).

Obvod lze reprezentovat jako orientovaný acyklický graf (DAG), kde uzly představují buď vstupní proměnné (listy) nebo logická hradla (vnitřní uzly). Hrany reprezentují tok logických hodnot – přenášejí výsledky vyhodnocení z jednoho uzlu jako vstupy do uzlů následujících. V monotónním obvodu chybí hradla NOT, což zajišťuje, že funkce reprezentovaná obvodem je monotónní – zvýšení hodnoty libovolného vstupu nikdy nesníží hodnotu výstupu [1].

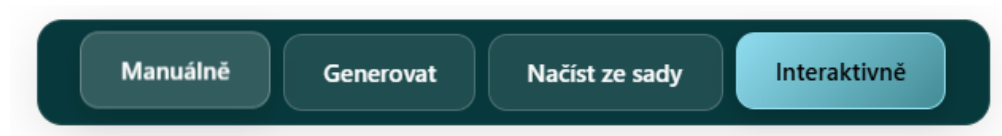
#### 3.1.1 P-úplnost MCVP

MCVP je P-úplný problém [3]. Vyhodnocení monotónního obvodu lze provést v polynomiálním čase postupným vyhodnocováním uzlů od vstupů směrem k výstupu. Přestože je problém v třídě P, jeho P-úplnost naznačuje, že pravděpodobně neexistuje efektivní paralelní algoritmus pro jeho řešení – problém vyžaduje sekvenční zpracování.

### 3.2 Formát vstupu a parsování výrazů

Aplikace umožňuje zadat monotónní obvod několika způsoby. Jako první je textový zápis ve formě logického výrazu, který je následně převeden na stromovou strukturu.





Obrázek 3.1: Výběr způsobu zadání vstupu pro MCVP

### 3.2.1 Gramatika vstupního jazyka

Vstupní výrazy odpovídají jednoduché gramatice:

- Operátory  $\cup$  (OR) a  $\wedge$  (AND) reprezentují logický součet a součin.
- Proměnné jsou zapsány ve tvaru  $x1[0]$  nebo  $x2[1]$ , kde číslo v hranatých závorkách udává hodnotu proměnné (0 nebo 1).
- Výrazy lze libovolně uzávorkovat pomocí kulatých závorek pro určení priority vyhodnocení.

Příklad platného výrazu:  $((x1[1] \wedge x2[0]) \cup (x3[1] \wedge x4[1]))$

### 3.2.2 Lexikální a syntaktická analýza

Převod textového výrazu na stromovou strukturu probíhá ve dvou fázích, které jsou implementovány v modulu `Parser.js`:

1. **Tokenizace (lexikální analýza):** Funkce `tokenize()` rozdělí vstupní řetězec na posloupnost tokenů. Každý token je dvojice (typ, hodnota), například `['LPAREN', '(']` nebo `['VARIABLE', 'x1[1]']`. Tokenizér rozpoznává závorky, operátory ( $\wedge$ ,  $\cup$ ) a proměnné pomocí regulárních výrazů.
2. **Parsování (syntaktická analýza):** Třída `Parser` implementuje rekurzivní sestupný parser. Parser je strukturován podle priority operátorů: `parseOrExpr()` zpracovává OR výrazy, `parseAndExpr()` zpracovává AND výrazy a `parseFactor()` zpracovává závorky a proměnné. Parser vytváří uzly typu `Node`, které tvoří stromovou strukturu reprezentující zadaný obvod.

Výsledkem parsování je stromová struktura, kde listy jsou proměnné s přiřazenými hodnotami a vnitřní uzly reprezentují logické operace.

## 3.3 Vyhodnocení obvodu

Vyhodnocení monotónního obvodu probíhá rekurzivním průchodem stromové struktury, který je implementován v modulu `EvaluateTree.js`.

### 3.3.1 Algoritmus vyhodnocení

Funkce `evaluateTree()` používá algoritmus průchodu do hloubky (DFS) s memoizací:

1. **Listy (proměnné):** Pro uzly typu `variable` funkce vrací přímo přiřazenou hodnotu (0 nebo 1).
2. **Vnitřní uzly (operace):** Pro uzly reprezentující logické operace funkce nejdříve vyhodnotí všechny potomky. Pokud je uzel typu AND, výsledek je 1 právě tehdy, když všichni potomci mají hodnotu 1. Pokud je uzel typu OR, výsledek je 1 pouze tehdy, když alespoň jeden potomek má hodnotu 1.
3. **Memoizace:** Vyhodnocené hodnoty uzlů jsou uloženy do cache (slovníku), aby nedocházelo k opakovanému výpočtu stejných podstromů.
4. **Detekce cyklů:** Algoritmus používá množinu `visiting` pro detekci případných cyklů v grafu, které by způsobily nekonečnou rekurzi.

Časová složitost algoritmu je  $O(n)$ , kde  $n$  je počet uzlů v obvodu, protože každý uzel je vyhodnocen právě jednou díky memoizaci.

### 3.3.2 Krokové vyhodnocení

Aplikace také nabízí krokovatelné vyhodnocení pomocí funkce `evaluateTreeWithSteps()`. Tato funkce provádí stejný výpočet jako `evaluateTree()`, ale navíc zaznamenává při každém výpočtu i stav zbytku stromové struktury. Uživatel tak může sledovat, jak se hodnoty šíří od listů ke kořeni stromové struktury a lépe pochopit princip vyhodnocování.

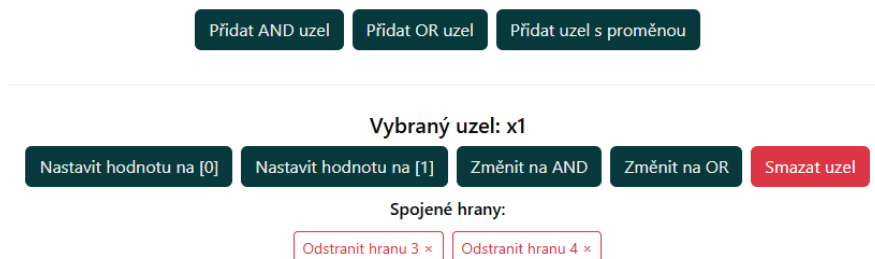
Komponenta `StepByStepTree` zobrazuje každý krok graficky – aktuálně vyhodnocovaný uzel je zvýrazněn a vedle grafu je zobrazena informace o hodnotách vstupů a výsledku operace.

## 3.4 Interaktivní editace obvodu

Kromě textového zadání umožňuje aplikace vytvářet a upravovat obvody pomocí grafického editoru implementovaného v komponentě `InteractiveMCVPGraph`. Uživatel může:

- **Přidávat uzly:** Pomocí tlačítek pod grafem lze vložit nové uzly typu AND, OR nebo vstupní proměnnou s hodnotou 0 nebo 1. Nový uzel se umístí do grafu, ale není propojen s ostatními uzly.
- **Vytvářet hrany:** Označit uzel, ten následně může:
  - Nastavit na AND nebo OR operaci.

- Nastavit na proměnnou s hodnotou 0 nebo 1.
  - Odstranit uzel z grafu.
  - Vytvořit nebo odstranit hranu s dalším uzlem.
- **Reorganizovat graf:** Uzly lze přesouvat myší pro lepší vizuální uspořádání obvodu.



Obrázek 3.2: Ovládací prvky pro interaktivní editor obvodů MCVP

Grafický editor využívá knihovnu *react-force-graph* pro vykreslování a interakci (viz kapitola 2.4). Aplikace průběžně aktualizuje interní stromovou strukturu odpovídající aktuálnímu stavu grafu.

### 3.5 Generování náhodných obvodů

Aplikace obsahuje generátor náhodných obvodů v modulu **Generator.js**. Uživatel může upravit dva parametry:

- **Počet hradel:** Kolik uzlů s operacemi (hradel AND a OR) bude obvod obsahovat.
- **Počet proměnných:** Kolik uzlů s proměnnými (listů) bude obvod obsahovat.

Generování probíhá v těchto fázích: Nejprve algoritmus vytvoří zadaný počet proměnných, s náhodnou hodnotou 0 nebo 1. Pak postupně přidává hradla. Pro každé hradlo vypočítáme cílový počet potomků podle vzorce  $\lceil n/k \rceil$ , kde  $n$  je počet dosud dostupných uzlů a  $k$  počet zbývajících hradel. K této hodnotě přidáme náhodnou odchylku  $\pm 20\%$  – tak dosáhneme vyváženosti mezi úplnou náhodností a vyrovnanou distribucí. Hradlo pak náhodně vybere určený počet uzlů jako potomky a náhodně zvolí operaci (AND nebo OR) a přidá se do množiny použitelných uzlů pro potomky. Poslední hradlo spojí všechny zbývajících uzly a stane se kořenem. Výsledný graf je platný DAG.

### 3.6 Vizualizace obvodu

Vizualizace monotónního obvodu je implementována v komponentě **TreeRenderCanvas**. Obvod je zobrazen jako stromový graf s kořenem nahoře a listy dole, takže hodnoty se šíří zdola nahoru.



Obrázek 3.3: Vizualizace MCVP obvodu pomocí TreeRenderCanvas

### 3.6.1 Hierarchické rozložení

Pro vizualizaci stromu používá aplikace knihovnu *react-force-graph-2d*. Uzly jsou automaticky umístěny do vrstev podle jejich vzdálenosti od kořene díky režimu `dagMode="td"` (top-down neboli shora dolů). Rozložení kombinuje hierarchickou strukturu s fyzikálními silami pro optimální vizuální uspořádání grafu.

## 3.7 Převod na kombinatorickou hru

Pro ilustraci vzájemné převoditelnosti P-úplných problémů implementujeme převod z MCVP na kombinatorickou hru (popsaný v kapitole 4). Tento převod se nachází v modulech `ConversionCombinatorialGame.` a `MCVPtoCombinatorialGameConverter.jsx`.

### 3.7.1 Princip převodu

V tomto převodu se mapují uzly obvodu MCVP na pozice ve hře dvou hráčů [4]. Každý typ uzlu odpovídá specifické herní situaci:

- **Hradlo OR** vytváří pozici pro Hráče 1 (hráč I je na tahu). Ten si může vybrat libovolnou následující pozici odpovídající potomkům hradla. Stačí, aby jedna z možností vedla k jeho výhře.

- **Hradlo AND** vytváří pozici pro Hráče 2 (hráč II je na tahu). Ten vybírá následující pozici a snaží se zabránit výhře Hráče 1. Hráč 1 vyhraje pouze pokud vyhrává ve všech možných pokračováních.
- **Proměnná s hodnotou 1** odpovídá konečné pozici, ve které Hráč 2 nemá žádné tahy – Hráč 1 tedy vyhrává.
- **Proměnná s hodnotou 0** odpovídá konečné pozici, ve které Hráč 1 nemá žádné tahy a prohrává.

Převod zachovává výsledek řešení: Hráč I v kombinatorické hře vyhrává tehdy, když se MCVP obvod vyhodnotí na 1.

### 3.7.2 Implementace převodu

Třída `MCVPToGameStepGenerator` prochází MCVP strom rekurzivně a pro každý uzel vytváří odpovídající pozici ve hře včetně hran k pozicím potomků. Pro efektivitu používáme memoizaci – každý uzel zpracujeme právě jednou, i kdyby se v obvodu vyskytoval vícekrát (například při sdílených podstromech).

Vizualizace probíhá pomocí komponenty `MCVPtoCombinatorialGameConverter`, která zobrazuje převod krokodatelně. Na levé straně vidí uživatel původní obvod se zvýrazněným aktuálně zpracovávaným uzlem, vpravo vznikající graf kombinatorické hry. Navigační tlačítka umožňují procházet kroky převodu k dalšímu i předešlému kroku.

## 3.8 Převod na bezkontextovou gramatiku

Druhá implementovaný převod transformuje MCVP obvod na bezkontextovou gramatiku (viz kapitola 5). Hodnota výsledku je zde taktéž zachována – gramatika generuje neprázdný jazyk právě tehdy, když obvod vyhodnotí na 1.

### 3.8.1 Pravidla převodu

Převod mapuje uzly MCVP obvodu na symboly gramatiky [4]:

- **Kořen obvodu** se mapuje na počáteční symbol gramatiky  $S$ .
- **Hradlo AND s potomky  $A$  a  $B$**  vytvoří pravidlo  $X \rightarrow AB$ , kde  $X$  reprezentuje hradlo. Řetězec lze z  $X$  odvodit právě tehdy, když lze odvodit řetězce z obou potomků  $A$  i  $B$ .
- **Hradlo OR s potomky  $A$  a  $B$**  vytvoří dvě pravidla:  $X \rightarrow A$  a  $X \rightarrow B$  ( $X \rightarrow A \mid B$ ). Stačí, když lze řetězec odvodit alespoň z jednoho potomka.

- **Proměnná s hodnotou 1** generuje epsilon pravidlo  $X \rightarrow \varepsilon$ , což umožňuje odvození prázdného řetězce a gramatika tak může generovat neprázdný jazyk.
- **Proměnná s hodnotou 0** vytvoří pravidlo  $X \rightarrow t$ , kde  $t$  je terminál bez dalších odvozovacích pravidel. Z tohoto neterminálu tak nelze odvodit žádný řetězec složený pouze z terminálů.

Výsledná gramatika generuje neprázdný jazyk právě tehdy, když obvod vyhodnotí na 1.

### 3.8.2 Implementace převodu

Převod implementuje třída `MCVPToGrammarConverter` (modul `MCVPtoGrammarConverter.jsx`). Nejprve `NonTerminalGenerator` přiřadí každému uzlu obvodu unikátní symbol – kořen dostane počáteční symbol  $S$ , ostatní uzly postupně symboly  $A, B, C$ , atd. Poté funkce `createProductionsRecursively()` rekurzivně prochází strom a generuje gramatická pravidla podle typu jednotlivých uzlů.

Pro konstrukci gramatiky používáme pomocnou třídu `ConversionGrammar` (soubor `ConversionGrammar.js`), která rozšiřuje standardní třídu `Grammar` o metody pro přidávání symbolů a pravidel.

Stejně jako u prvního převodu, komponenta `MCVPtoGrammarConverter` nabízí krokovou vizualizaci celého procesu. Uživatel tak může sledovat, jak obvod postupně přechází na gramatiku, a vidět, která pravidla vznikají z konkrétních uzlů.

## 3.9 Předpřipravené sady

Aplikace nabízí několik předpřipravených sad MCVP obvodů ve složce **Sady/MCVP**. Tyto sady slouží jako základní ukázkové příklady. Tyto uložené obvody jsou taktéž ve formátu JSON.

### 3.9.1 Ukládání a načítání MCVP obvodů

Aby si mohl uživatel uložit specifický obvod, podporuje aplikace export a import obvodů ve formátu JSON. Modul `Serialization.js` obsahuje potřebné funkce. Funkce `treeToFlatGraph()` převede stromovou strukturu na JSON objekt s poli uzlů a hran, kde každý uzel má unikátní ID, typ, hodnotu a případně přiřazenou binární hodnotu (export). Opačný převod zajišťuje `flatGraphToTree()` funkce, která z JSON formátu načte stromovou strukturu včetně hran mezi uzly.



Obrázek 3.4: Ovládací prvky pro import a export MCVP obvodů

## Kapitola 4

# Kombinatorická hra

### 4.1 Teoretický základ

Kombinatorická hra dvou hráčů na orientovaném grafu je další příklad P-úplného problému [4]. Hra má tyto vlastnosti:

- **Dva hráči:** Každé pole grafu má specifikováno, který hráč je na tahu – Hráč I (první hráč) nebo Hráč II (druhý hráč).
- **Konečná pozice:** Hra končí, když je hráč na tahu v pozici bez možných dalších tahů.

Problém spočívá v rozhodnutí, zda Hráč I má výherní strategii ze zadané počáteční pozice:

- **Vstup:** Orientovaný acyklický graf (DAG), kde každý uzel reprezentuje herní pozici přiřazenou některému z hráčů, hrany reprezentují možné tahy a jeden uzel je označen jako počáteční pozice.
- **Výstup:** Rozhodnutí, zda Hráč I má výherní strategii ze startovní pozice.

Hráč I vyhrává, pokud se dostane do pozice, kde Hráč II nemá žádný možný tah. Naopak, Hráč I prohrává, když se sám dostane do pozice bez možných tahů [1].

#### 4.1.1 P-úplnost problému

Tento problém je P-úplný [3]. Určení výherní strategie lze provést v polynomiálním čase pomocí tzv. retrográdní analýzy [4], která zpětně vyhodnocuje pozice od koncových uzlů. Implementace je podobná algoritmu vyhodnocování MCVP (viz kapitola 3.3) – opět používáme průchod do hloubky s memoizací. Liší se pouze logika rozhodování: místo kombinace logických hodnot zde určujeme, která pozice je výherní na základě možností volby jednotlivých hráčů. I když je problém řešitelný v třídě P, jeho P-úplnost naznačuje, že pravděpodobně neexistuje efektivní paralelní algoritmus – řešení vyžaduje sekvenční zpracování pozic.

## 4.2 Formát vstupu

Aplikace nabízí tři způsoby zadávání herního grafu:

- **Interaktivní editace:** Uživatel vytváří a upravuje graf pomocí grafického editoru (viz sekce 4.3).
- **Generování náhodných her:** Automatické vytvoření náhodného grafu podle zadaných parametrů (viz sekce 4.4).
- **Předpřipravené sady:** Načtení ukázkových připravených příkladů (viz sekce 4.5).

Všechny metody využívají komponentu `DisplayGraph` pro vizualizaci herního grafu (viz sekce 4.6).

## 4.3 Interaktivní editace grafu

Komponenta `ManualInput` umožňuje vytvářet a upravovat herní grafy pomocí interaktivního grafického editoru. Uživatel může:

- **Přidávat uzly:** Vytvořit novou pozici.
- **Upravovat uzly:** Kliknutím na uzel ho označíme. Označený uzel můžeme:
  - Změnit hráče který je na tahu (Hráč I nebo Hráč II)
  - Odstranit pozici z grafu
  - Nastavit jako počáteční pozici
  - Použít jako zdroj nebo cíl pro vytvoření hrany
  - Vytvořit hranu z nebo do tohoto uzlu
  - Odstranit hrany k nebo od tohoto uzlu
- **Reorganizovat graf:** Uzly lze přesouvat myší pro lepší vizuální uspořádání.

Aplikace průběžně validuje graf a zabráňuje vytvoření neplatných struktur. Pokud uživatel zkusí přidat hranu, která by vytvořila cyklus, aplikace tuto akci zablokuje a zobrazí chybovou zprávu. Dále upozorní uživatele, pokud není nastavena počáteční pozice, což je nutné pro spuštění analýzy.



## 4.4 Generování náhodných her

Modul `Generator.js` obsahuje funkci `generateGraph()` pro vytváření náhodných herních grafů. Uživatel nastavuje dva parametry:

- **Počet pozic:** Kolik uzlů (herních pozic) bude graf obsahovat.
- **Pravděpodobnost hrany:** Hodnota 0% – 100% určující, jak pravděpodobné je vytvoření hrany mezi dvěma uzly.

Algoritmus generování probíhá ve dvou fázích:

1. **Vytvoření kostry:** Vytvoříme uzly očíslované 0 až  $n - 1$  a každému náhodně přiřadíme hráče. Pro každý uzel  $i > 0$  pak vytvoříme hranu z náhodného předchozího uzlu do uzlu  $i$ . Tím zajistíme souvislý acyklický graf – hrany vedou vždy od nižších čísel k vyšším, takže cyklus vzniknout nemůže.
2. **Přidání dalších hran:** Pro každou dvojici uzlů  $i, j$ , kde  $i < j$ , podle dané pravděpodobnosti přidáme hranu  $i \rightarrow j$ , pokud ještě neexistuje.

Výsledný graf je vždy platný DAG s uzlem 0 jako počáteční pozicí.

## 4.5 Předpřipravené sady

Ve složce `Sady/CombinatorialGame` najdeme předpřipravené herní grafy různé velikosti a složitosti.

### 4.5.1 Ukládání a načítání her

Herní grafy lze exportovat a importovat ve formátu JSON pomocí komponenty `FileTransferControls`. Formát obsahuje:

- Pole `nodes` s uzly – každý má ID a přiřazeného hráče
- Pole `edges` s hranami ve formátu `source-target`
- `startingPosition` určující ID počáteční pozice

Tento formát umožňuje sdílení her mezi uživateli a vytváření knihovny předpřipravených příkladů.

## 4.6 Vizualizace grafu

Vizualizace herního grafu využívá komponentu `DisplayGraph`, která využívá knihovnu *react-force-graph-2d*. Graf zobrazuje:

- **Uzly:** Každý uzel reprezentuje herní pozici. Pod uzlem je zobrazen hráč, který je na tahu (I nebo II). Při najetí myši na libovolný uzel se v centrech všech uzlů zobrazí jejich ID pro snadnější orientaci.
- **Počáteční pozice:** Označena oranžovou barvou.
- **Hrany:** Možné tahy jsou zobrazeny jako směřované hrany mezi uzly. Hrany patřící do výherní strategie jsou zvýrazněny tlustší čarou a žlutou barvou.

Graf používá fyzikální simulaci pro automatické rozmístění uzlů. Uživatel může uzly přesouvat myší a graf přibližovat nebo oddalovat kolečkem myši.

## 4.7 Algoritmus analýzy

Řešení problému kombinatorických her je implementováno v modulu `ComputeWinner.js`. Algoritmus je technicky velmi podobný vyhodnocování MCVP stromu – používá průchod do hloubky (DFS) s memoizací. Rozdíl spočívá v logice, kterou určujeme výsledek každé pozice.

### 4.7.1 Vyhodnocení výherních pozic

Algoritmus rozhoduje, zda je daná pozice výherní pro Hráče I. Koncové pozice (listové uzly) bez dalších tahů jsou výherní pro Hráče I, když je na tahu Hráč II (nemá možnost pokračovat). U ostatních pozic záleží na tom, kdo je na tahu: pokud je to Hráč I, stačí mu jedna výherní možnost pokračování – může si ji vybrat. Pokud je na tahu Hráč II, musí všechny možné tahy vést do výherních pozic Hráče I, protože Hráč II si vybere nejlepší tah pro sebe.

Časová složitost je  $O(V + E)$ , kde  $V$  je počet uzlů a  $E$  počet hran, protože každá pozice a hrana je zpracována právě jednou.

### 4.7.2 Optimální tahy

Funkce `getOptimalMoves()` identifikuje hrany, které jsou součástí výherní strategie. Hrana z pozice  $u$  do pozice  $v$  je optimální, pokud obě pozice jsou výherní pro Hráče I. Tyto hrany jsou zvýrazněny ve vizualizaci a ukazují uživateli cestu k výhře.

## 4.8 Krokové vyhodnocení

Pro detailnější průchod grafem implementuje komponenta `StepByStepGame` krokovatelnou analýzu. Funkce `computeWinningStrategySteps()` zaznamenává každý krok v celkovém vyhodnocení:

- Hráč na tahu v dané pozici
- Výsledky vyhodnocení všech potomků
- Finální rozhodnutí, zda je pozice výherní pro Hráče I
- Textové vysvětlení rozhodnutí

Uživatel může procházet kroky analýzy pomocí navigačních tlačítek – kromě standardních tlačítek „Předchozí“ a „Další“ jsou k dispozici také tlačítka „Začátek“ a „Konec“ pro rychlý přesun na první nebo poslední krok analýzy. Aktuálně analyzovaný uzel je ve zvýrazněn světle modrou barvou (stejnou jako při najetí myši) a pod grafem je slovní popis rozhodnutí. Tato funkce pomáhá pochopit, jak algoritmus postupně buduje cesty z výherních pozic od listů ke kořeni.

## Kapitola 5

# Prázdnost bezkontextových gramatik

### 5.1 Teoretický základ

Problém prázdnosti bezkontextových gramatik (CFG Emptiness Problem) je dalším příkladem P-úplného problému [4]. Bezkontextová gramatika představuje formální systém sloužící k generování jazyka, který hraje klíčovou roli v definování syntaxe programovacích jazyků i v analýze neprogramovacích jazyků.

Formálně je gramatika definována jako čtveřice  $G = (N, \Sigma, P, S)$ , kde:

- $N$  je konečná množina neterminálních symbolů (neterminálů).
- $\Sigma$  je konečná množina terminálních symbolů (terminálů), disjunktní s  $N$ .
- $P$  je konečná množina přepisovacích pravidel tvaru  $A \rightarrow \alpha$ , kde  $A \in N$  a  $\alpha \in (N \cup \Sigma)^*$ .
- $S \in N$  je počáteční symbol (start symbol).

Problém spočívá v tom, zda daná gramatika generuje alespoň jedno slovo složené pouze z terminálních symbolů:

- **Vstup:** Bezkontextová gramatika  $G = (N, \Sigma, P, S)$ .
- **Výstup:** Rozhodnutí, jestli je jazyk  $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$  neprázdný.

Symbol  $\Rightarrow^*$  označuje derivaci v nula nebo více krocích [4]. Gramatika generuje neprázdný jazyk právě tehdy, když z počátečního symbolu  $S$  lze odvodit alespoň jedno slovo složené pouze z terminálních symbolů.

### 5.1.1 P-úplnost problému

Problém prázdnosti bezkontextových gramatik je P-úplný [4]. Tento problém lze vyřešit v polynomiálním čase pomocí algoritmu iterativního označování produktivních neterminálů. Produktivní neterminál je takový, ze kterého lze derivovat řetězec složený pouze z terminálů [4].

I přes řešitelnost v polynomiálním čase je problém P-úplný, což naznačuje obtížnou paralelizovatelnost. Produktivita jednoho neterminálu často závisí na produktivitě jiných neterminálů. Tato vzájemná závislost vyžaduje sekvenční zpracování podobné vyhodnocování MCVP obvodu.

## 5.2 Formát vstupu

Aplikace nabízí tři způsoby zadávání bezkontextové gramatiky:

- **Manuální zadání:** Uživatel definuje gramatiku pomocí textového pole, kde se zadá celá gramatika najednou (viz sekce 5.3).
- **Generování náhodných gramatik:** Automatické vytvoření náhodné gramatiky podle zadaných parametrů (viz sekce 5.4).
- **Předpřipravené sady:** Načtení ukázkových příkladů gramatik (viz sekce 5.5).

Všechny formy vstupu využívají stejnou komponentu pro zobrazení derivačního stromu gramatiky implementovanou ve třídě `Grammar` (modul `Grammar.js`).

## 5.3 Manuální zadání gramatiky

Komponenta `ManualInput` poskytuje rozhraní ve formě textového okna pro přímé zadávání pravidel gramatiky.

- **Syntaktický formát:** Každé pravidlo se zapisuje na nový řádek ve tvaru `LS -> PS`. Pro oddělení více alternativ na pravé straně se používá symbol `|`. Symboly v rámci jedné alternativy musí být odděleny mezerami.
- **Kategorizace symbolů:** Algoritmus v modulu `GrammarParser.js` automaticky rozpoznává typy symbolů. Symboly složené čistě z velkých písmen (včetně českých znaků s diakritikou) jsou považovány za neterminály. Jakékoliv jiné řetězce jsou identifikovány jako terminály.
- **Epsilon pravidla:** Pro vyjádření prázdného řetězce ( $\epsilon$ ) lze použít přímo slovo `epsilon` nebo řecké písmeno.
- **Počáteční symbol:** První neterminál uvedený na levé straně prvního pravidla je automaticky nastaven jako počáteční symbol `S`.

Příklad vstupu:

`S -> a S b | epsilon`

Aplikace během zpracování textu provádí validaci a kontroluje správnost formátu pravidel. Validace upozorňuje na tyto typy chyb:

- **Chybějící levá nebo pravá strana:** Pravidlo musí obsahovat jak levou stranu (neterminál), tak pravou stranu. Parser zobrazí chybu, pokud v pravidle chybí šipka `->` nebo pokud je některá strana prázdná.
- **Neplatný neterminál:** Levá strana pravidla musí být složena pouze z velkých písmen (včetně českých znaků s diakritikou).
- **Zapomenutá mezera:** Parser detekuje pravděpodobné chyby, jako je zapomenutá mezera mezi neterminálem a terminálem (např. `aS`), což by bylo chybně identifikováno jako jeden terminální symbol. V takovém případě parser navrhne správné oddělení symbolů mezerami.

## 5.4 Generování náhodných gramatik

Modul `GrammarGenerator.js` obsahuje funkci `generateGrammar()` pro vytváření náhodných bezkontextových gramatik na základě vstupních parametrů. Komponenta `GenerateInput` poskytuje uživatelské rozhraní s možností nastavení základních i pokročilých parametrů.

### 5.4.1 Základní parametry

Základní parametry umožňují rychlé vygenerování gramatiky a určují její velikost a strukturu. Prvním parametrem je počet neterminálů v rozmezí 1 až 10. Druhým parametrem je počet terminálů, taktéž v rozmezí 1 až 10. Třetí parametr určuje maximální délku pravé strany, zde jsou hodnoty v rozmezí 1 až 5, přičemž délka každého pravidla je náhodně zvolena mezi hodnotou 1 a hodnotou tohoto parametru.

### 5.4.2 Pokročilé parametry

Pokročilé parametry umožňují větší kontrolu nad vlastnostmi generované gramatiky:

- **Počet pravých stran na neterminál:**
  - Min (1-10): Minimální počet pravidel pro každý neterminál
  - Max: Maximální počet pravidel, automaticky omezen na rozmezí od hodnoty parametru Min do 10
  - Pro každý neterminál se náhodně zvolí počet pravidel v tomto rozmezí

- **Rekurze:** Kontrola nad rekurzivními pravidly:
  - Levá rekurze: Povoluje pravidla tvaru  $A \rightarrow A\alpha$
  - Pravá rekurze: Povoluje pravidla tvaru  $A \rightarrow \alpha A$
  - Pokud jsou obě vypnuté, generátor vytváří pouze pravidla s terminály
  - Pravděpodobnost aplikace rekurze je 30% při povoleném režimu
- **Generování epsilon ( $\epsilon$ ):** Tři režimy:
  - **Ne:** Negenerují se žádná epsilon pravidla
  - **Náhodně:** Epsilon pravidla se generují s pravděpodobností 8%
  - **Vždy:** Epsilon pravidla se generují s pravděpodobností 15%

### 5.4.3 Algoritmus generování

Algoritmus generování probíhá následovně:

1. **Vytvoření symbolů:** Generují se pole neterminálů a terminálů podle zadaných počtů. První neterminál ( $A$ ) je implicitně počáteční symbol.
2. **Generování pravidel pro každý neterminál:**
  - (a) Náhodně se určí počet pravidel pro tento neterminál (v rozmezí min-max)
  - (b) Pro každé pravidlo:
    - Kontrola epsilon režimu – případné vytvoření  $\epsilon$  pravidla
    - Jinak určení náhodné délky pravé strany (1 až maxRuleLength)
    - Náhodný výběr symbolů – terminály s pravděpodobností 50%, neterminály s 50%
    - Aplikace rekurze – přidání neterminálu na začátek/konec dle nastavení
3. **Sestavení gramatiky:** Vytvoření instance třídy **Grammar** s vygenerovanými symboly a pravidly.

### 5.4.4 Vlastnosti generovaných gramatik

Výsledná gramatika nemusí nutně generovat neprázdný jazyk – to závisí na náhodnosti generování a zvolených parametrech:

- Gramatiky s vypnutou rekurzí a bez epsilon pravidel mají vyšší šanci generovat neprázdný jazyk
- Gramatiky s povolenou rekurzí mohou obsahovat nedosažitelné nebo neproduktivní neterminály

- Epsilon režim "Vždy" zvyšuje pravděpodobnost neprázdného jazyka

Vygenerované gramatiky slouží jako testovací případy pro algoritmus vyhodnocení prázdnoti jazyka.

## 5.5 Předpřipravené sady

Ve složce `Sady/Grammar` jsou uloženy předpřipravené příklady gramatik. Tyto sady slouží ke kontrované demonstraci specifických příkladů.

### 5.5.1 Ukládání a načítání gramatik

Gramatiky lze exportovat a importovat ve formátu JSON pomocí komponenty `FileTransferControls`. Formát obsahuje:

- Pole `terminals` se seznamem terminálních symbolů
- Pole `nonTerminals` se seznamem neterminálních symbolů
- Pole `productions` s pravidly, kde každé pravidlo má `ls` (levou stranu) a `ps` (pravou stranu jako pole symbolů)
- `startSymbol` určující počáteční neterminál

Tento formát umožňuje sdílení gramatik mezi uživateli a vytváření předpřipravených příkladů.

## 5.6 Algoritmus vyhodnocení

Řešení problému prázdnoti gramatiky je implementováno v modulu `GrammarEvaluator.js`. Algoritmus zjišťuje, zda je počáteční symbol  $S$  produktivní – tedy zda z něj lze odvodit slovo složené pouze z terminálů nebo prázdňý řetězec  $\epsilon$ .

### 5.6.1 Identifikace produktivních neterminálů

Algoritmus pracuje iterativně a postupně označuje produktivní neterminály [4]:

1. **Inicializace:** Vytvoříme prázdňou množinu produktivních symbolů  $\mathcal{P}$ .
2. **Základní krok:** Do  $\mathcal{P}$  přidáme všechny neterminály  $A$ , které mají pravidlo  $A \rightarrow w$ , kde  $w \in \Sigma^*$  (pravá strana obsahuje pouze terminály nebo je prázdňá).
3. **Iterativní rozšiřování:** Opakovaně procházíme všechna pravidla. Pokud existuje pravidlo  $A \rightarrow X_1 X_2 \dots X_k$ , kde všechny symboly  $X_i$  jsou buď terminály nebo již jsou v množině  $\mathcal{P}$ , přidáme  $A$  do  $\mathcal{P}$ .



4. **Ukončení:** Opakujeme krok 3, dokud se množina  $\mathcal{P}$  mění.
5. **Výsledek:** Pokud  $S \in \mathcal{P}$ , pak jazyk  $L(G)$  není prázdný. V opačném případě je  $L(G) = \emptyset$ .

Implementace používá optimalizovanou verzi s frontou (work-list algorithm). Místo procházení všech pravidel v každém cyklu udržujeme frontu nově označených produktivních neterminálů a kontrolujeme pouze ta pravidla, která tyto neterminály obsahují na pravé straně.

Časová složitost algoritmu je  $O(|P| \cdot l)$ , kde  $|P|$  je počet pravidel a  $l$  je maximální délka pravé strany pravidla, protože každé pravidlo je kontrolováno nejvýše jednou pro každý symbol na pravé straně.

### 5.6.2 Rekonstrukce svědka

Během běhu algoritmu si pro každý produktivní neterminál ukládáme tzv. svědky (witnesses) – všechna produktivní pravidla pro daný neterminál, nikoliv pouze první. Tato informace je uložena v mapě `allWitnesses`, která pro každý neterminál obsahuje pole všech jeho produktivních pravých stran.

Po skončení analýzy, pokud je počáteční symbol  $S$  produktivní, můžeme rekurzivně rekonstruovat derivační strom:

1. **Výběr pravidla:** Pro každý neterminál náhodně vybereme jedno z jeho produktivních pravidel. Při větší hloubce derivace ( $> 20$  kroků) preferujeme nerekurzivní pravidla, aby se předešlo nekonečným derivacím.
2. **Limitace hloubky:** Rekursivní konstrukce stromu je omezena na maximální hloubku 50 úrovní. Pokud je tato hloubka překročena, vyhodí se chyba a strom se nevykreslí.
3. **Stavba uzlů:** Pro každý neterminál:
  - Vytvoříme uzel se symbolem a unikátním ID
  - Vybereme náhodné produktivní pravidlo
  - Rekursivně vytvoříme potomky pro každý symbol na pravé straně
  - Terminály a  $\epsilon$  tvoří listy stromu
4. **Extrakce slova:** Z vytvořeného stromu extrahujeme vygenerované slovo zleva doprava čtením terminálních listů.

Tato rekonstrukce slouží nejen jako důkaz neprázdnosti jazyka, ale také jako vizuální pomůcka pro pochopení struktury generovaných slov. Náhodný výběr pravidel znamená, že při každém vyhodnocení může být vygenerováno jiné slovo.

## 5.7 Vizualizace derivačního stromu

Pokud gramatika generuje neprázdný jazyk, aplikace nejen potvrdí tento fakt, ale také vizuálně demonstruje důkaz vykreslením derivačního stromu pro jedno z možných slov.

### 5.7.1 Komponenta pro vykreslení

Pro vykreslení derivačního stromu je využita komponenta `DerivationTreeVisual`, která je postavená na knihovně *react-force-graph-2d*. Strom je zobrazen v hierarchickém rozložení pomocí režimu `dagMode="td"` (top-down).

#### 5.7.1.1 Vizuální prvky

- **Neterminály:** Zobrazeny standardní barvou definovanou v konfiguraci (`useGraphColors`), reprezentují mezikroky v derivaci.
- **Terminály:** Zvýrazněny kontrastní barvou (standardně žlutá), aby bylo na první pohled zřejmé, které symboly tvoří výsledné slovo. Listy stromu čtené zleva doprava tvoří vygenerované slovo.
- **Epsilon ( $\epsilon$ ):** Zobrazeny specifickou barvou (standardně šedá) pro reprezentaci prázdného řetězce. Tyto uzly signalizují místa, kde byla aplikována epsilon pravidla.
- **Hrany:** Reprezentují aplikaci přepisovacích pravidel – hrana z uzlu *A* do uzlu *B* znamená, že symbol *B* se objevuje na pravé straně pravidla použitého pro derivaci z *A*.

#### 5.7.1.2 Interaktivita a responsivita

Komponenta využívá několik moderních přístupů pro zajištění dobrého uživatelského zážitku:

- **Responsivní rozměry:** Komponenta využívá `ResizeObserver` API pro automatické přizpůsobení velikosti grafu při změně velikosti okna nebo kontejneru.
- **Jednotné nastavení:** Barvy a další vizuální parametry jsou spravovány pomocí custom React hooks:
  - `useGraphColors`: Poskytuje konzistentní barevné schéma napříč aplikací
  - `useGraphSettings`: Poskytuje specifická nastavení pro derivační stromy gramatik
- **Interaktivní ovládání:** Uživatel může:
  - Přibližovat a oddalovat pomocí kolečka myši
  - Posouvat pohled tažením
  - Přesouvat jednotlivé uzly pro lepší přehlednost

## 5.8 Krokové vyhodnocení

Pro detailnější pochopení algoritmu implementuje aplikace krokovatelnou analýzu. Komponenta sleduje jednotlivé iterace algoritmu a zobrazuje:

- Aktuální stav množiny produktivních neterminálů  $\mathcal{P}$
- Právě kontrolované pravidlo
- Rozhodnutí, zda pravidlo přidává nový produktivní neterminál
- Textové vysvětlení každého kroku

Uživatel může procházet kroky analýzy pomocí navigačních tlačítek – k dispozici jsou tlačítka „Předchozí“, „Další“, „Začátek“ a „Konec“ pro rychlý přesun. Tato funkce pomáhá pochopit, jak algoritmus postupně buduje množinu produktivních neterminálů a jak závislosti mezi pravidly ovlivňují výsledek.

## **Kapitola 6**

### **Závěr**

# Literatura

1. PAPADIMITRIOU, Christos H. *Computational Complexity*. Addison-Wesley, 1993. ISBN 978-0201530827.
2. SAWA, Zbyněk. *Úvod do teoretické informatiky – Výpočetní složitost algoritmů* [Online]. 2014. Dostupné také z: <https://www.cs.vsb.cz/kot/download/uti2014/uti-12-cz.pdf>.
3. ARORA, Sanjeev; BARAK, Boaz. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. ISBN 978-0521424264.
4. SAWA, Zbyněk. *Teoretická informatika – Podklady pro přednášky* [Online]. 2025. Dostupné také z: <https://www.cs.vsb.cz/sawa/ti/slides/ti-slides-08.pdf>.
5. META PLATFORMS, INC. *React – A JavaScript library for building user interfaces* [Online]. 2024. Dostupné také z: <https://react.dev/>.
6. VITE TEAM. *Vite – Next Generation Frontend Tooling* [Online]. 2024. Dostupné také z: <https://vitejs.dev/>.
7. ASTURIANO, Vasco. *react-force-graph – React component for 2D, 3D, VR and AR force directed graphs* [Online]. 2024. Dostupné také z: <https://github.com/vasturiano/react-force-graph>.
8. BOOTSTRAP TEAM. *Bootstrap – The most popular HTML, CSS, and JS library in the world* [Online]. 2024. Dostupné také z: <https://getbootstrap.com/>.
9. KHADRA, Fadi. *react-toastify – React notification made easy* [Online]. 2024. Dostupné také z: <https://fkhadra.github.io/react-toastify/>.
10. ESLINT TEAM. *ESLint – Pluggable JavaScript linter* [Online]. 2024. Dostupné také z: <https://eslint.org/>.

## **Příloha A**

### **Dlouhý zdrojový kód**