VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

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

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Č.j.VSB/25/116933
vsboee25037FA8

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
[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

17. listopadu 2172/15
708 00 Ostrava-Poruba
Česká republika

spojovatelka: +420 597 321 111
epodatelna: epodatelna@vsb.cz
ID datové schránky: d3kj88v

IČ: 61989100
DIČ: CZ61989100

email: studijni.fei@vsb.cz
www.fei.vsb.cz

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: 30.04.2026

Garant studijního programu: Ing. Tomáš Fabián, Ph.D.

V IS EDISON zadáno: 18.08.2025 09:42:23

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), *problém neprázdnosti bezkontextové gramatiky* a *kombinatorické hry na grafech*. 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 zkratek a symbolů

MCVP	–	Monotone Circuit Value Problem
P	–	Třída problémů řešitelných v polynomiálním čase
NC	–	Nick's Class – třída efektivně paralelizovatelných problémů
BFS	–	Breadth First Search – Algoritmus průchodu do šířky
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 značkovací jazyk
ID	–	Identifier – Identifikátor
JSON	–	JavaScript Object Notation – Objektový zápis JavaScriptu
LS	–	Levá strana
PS	–	Pravá strana
SPA	–	Single Page Application – Jednostránková webová aplikace
TI	–	Teoretická informatika
UI	–	User Interface – Uživatelské rozhraní

Seznam obrázků

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 na deterministickém Turingov stroji [12] – tedy takové, jejichž časovou složitost lze vyjádřit jako $O(n^k)$ [13] pro nějakou konstantu k, kde n je velikost vstupu – na deterministickém Turingově stroji [12]. Přestože jsou problémy v této třídě obvykle pokládány za „efektivně řešitelné“, projevují se mezi nimi významné odlišnosti, zejména pokud 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 paměťovou složitostí převést jakýkoliv jiný problém z třídy P [9, 1]. Hlavní problém u této podmnožiny spočívá v předpokladu, že P-úplné problémy pravděpodobně nelze efektivně paralelizovat. To znamená, že tyto úlohy nespadají do třídy NC, která obsahuje problémy řešitelné v polylogaritmickém čase $O(\log^k n)$ pomocí paralelního výpočtu s polynomiálně mnoha procesory [10]. 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 ukázku 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 tvořený pouze hradly typu AND a OR [10]. 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 [10].

- **Problém neprázdnosti jazyka bezkontextové gramatiky:** Rozhoduje, zda daná gramatika generuje neprázdný jazyk [10].

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 [10].

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 neprázdnosti jazyka bezkontextové gramatiky. 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* [7]. 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* [14]. 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 samostatných 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 jednotné 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* [2], která je postavena na knihovně *D3.js* [4]. 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* [3]. 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:

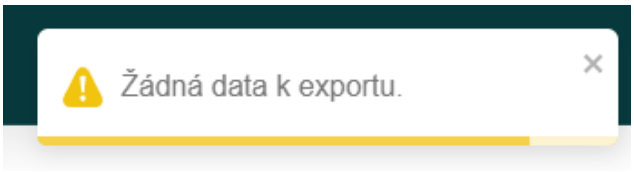
1. **Manuální zadání** – uživatel zadává vstup pomocí textového pole nebo grafického editoru
2. **Náhodné generování** – aplikace vygeneruje náhodný příklad podle zadaných parametrů
3. **Připravené příklady** – výběr z předpřipravených ukázkových příkladů

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* [6]. 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 (viz obrázek 2.1).

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* [5], 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 [10]. Jeho definice je následující:

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

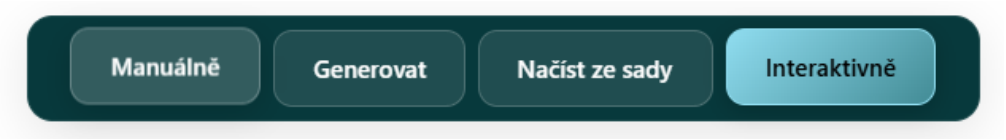
Obvod lze reprezentovat jako orientovaný acyklický graf (DAG), kde uzly představují buď vstupní proměnné (listy) nebo logická hradla (vnitřní uzly). Hrany reprezentují tok logických hodnot – přenášejí výsledky vyhodnocení z jednoho uzlu jako vstupy do uzlů následujících. V monotónním obvodu chybí hradla NOT, což zajišťuje, že funkce reprezentovaná obvodem je monotónní – zvýšení hodnoty libovolného vstupu nikdy nesníží hodnotu výstupu [10].

3.1.1 P-úplnost MCVP

MCVP je P-úplný problém [10, 8]. 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 (viz obrázek 3.1). 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 `O` (OR) a `A` (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] A x2[0]) O (x3[1] A 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 (A, O) 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.

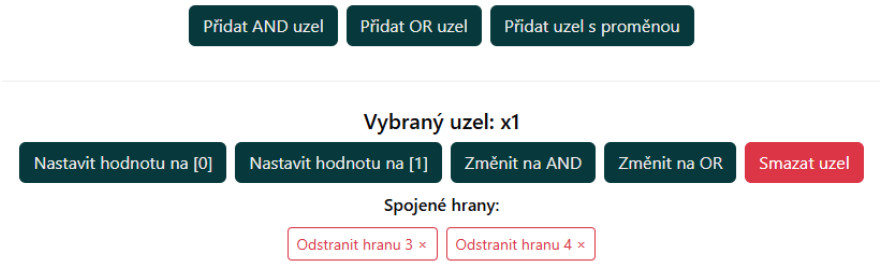
Komponenta `StepByStepTree` zobrazuje každý krok graficky – aktuálně vyhodnocovaný uzel je zvýrazněn a vedle grafu je zobrazena informace o hodnotách vstupů a výsledku operace.

3.4 Interaktivní editace obvodu

Kromě textového zadání umožňuje aplikace vytvářet a upravovat obvody pomocí grafického editoru implementovaného v komponentě `InteractiveMCVPGraph` (viz obrázek 3.2). 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

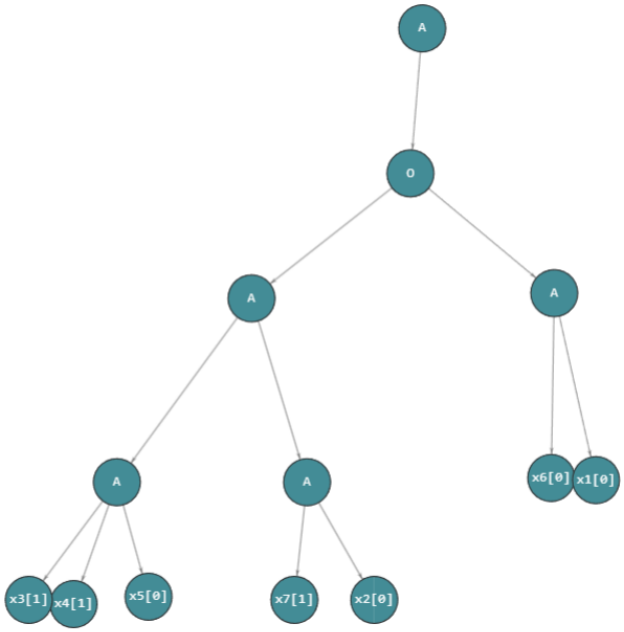
Aplikace obsahuje generátor náhodných obvodů v modulu `Generator.js`. Uživatel může upravit dva parametry:

- **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 zbývající 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, hodnoty se tedy šíří směrem zdola nahoru (viz obrázek 3.3).



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.j` a `MCVPtoCombinatorialGameConverter.jsx`.

3.7.1 Princip převodu

V tomto převodu se mapují uzly obvodu MCVP na pozice ve hře dvou hráčů [10]. 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 krokovatelně. 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 [10]:

- **Kořen obvodu** se mapuje na počáteční symbol gramatiky S.
- **Hradlo AND s potomky** A **a** B vytvoří pravidlo $X \to AB$, 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: $X \to A$ a $X \to B$ ($X \to A \mid B$). Stačí, když lze řetězec odvodit alespoň z jednoho potomka.

- **Proměnná s hodnotou 1** generuje epsilon pravidlo $X \to \varepsilon$, což umožňuje odvození prázdného řetězce a gramatika tak může generovat neprázdný jazyk.
- **Proměnná s hodnotou 0** vytvoří pravidlo $X \to t$, 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.

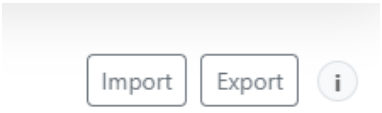
Stejně jako u prvního převodu, komponenta `MCVPtoGrammarConverter` nabízí krokovou vizualizaci celého procesu. Uživatel tak může sledovat, jak obvod postupně přechází na gramatiku, a vidět, která pravidla vznikají z konkrétních uzlů.

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 formatu 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 (viz obrázek 3.4). 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 [10]. 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 graf (DG), 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.

Pravidla hry: Hra končí, když se dostane do pozice, kde hráč na tahu nemá žádný možný tah – tento hráč pak prohrává. Hráči se střídají v tazích podle toho, jaký hráč je přiřazen aktuální pozici: je-li uzel přiřazen Hráči I, táhne Hráč I; je-li přiřazen Hráči II, táhne Hráč II.

Výherní strategie: Výherní strategie pro Hráče I je taková posloupnost tahů, která garantuje výhru Hráče I bez ohledu na to, jak hraje Hráč II. Jinými slovy, Hráč I má výherní strategii, pokud existuje způsob, jak vždy volit tahy tak, aby se hra dostala do pozice, kde je Hráč II na tahu a nemá žádný možný tah [10].

Hráč I tedy vyhrává v těchto případech:

- Může vynutit situaci, kdy se hra dostane do pozice, kde je Hráč II na tahu a nemá žádný možný tah (terminální pozice pro Hráče II).
- Hra začíná v pozici přiřazené Hráči II, která nemá žádné odchozí hrany – Hráč II okamžitě prohrává a Hráč I vyhrává.

4.1.1 P-úplnost problému

Tento problém je P-úplný [10, 8]. Určení výherní strategie lze provést v polynomiálním čase pomocí tzv. retrográdní analýzy [10], která zpětně vyhodnocuje pozice od koncových uzlů. Algoritmus pracuje iterativně a dokáže zpracovat i grafy s cykly – v případě cyklů bez jednoznačného výsledku přiřadí pozicím status remízy.

Logika rozhodování funguje následovně: Pro každou pozici určujeme, zda je výherní pro hráče, který je v ní na tahu. Pokud má hráč v dané pozici alespoň jeden tah do pozice, která je výherní pro něj, pak je i aktuální pozice výherní pro něj. Naopak, pokud všechny možné tahy vedou do pozic výherních pro protihráče, pak je aktuální pozice výherní pro protihráče (tedy prohrávající pro hráče na tahu). V případě, že některé tahy vedou do pozic výherních pro protihráče a některé do pozic, jejichž výsledek ještě nebyl určen, zůstává aktuální pozice také nerozhodnuta (remíza).

Implementace využívá frontu (queue) pro postupné zpracování pozic – začínáme koncovými pozicemi a postupně šíříme výsledky směrem k počáteční pozici. Liší se od algoritmu vyhodnocování MCVP (viz kapitola 3.3) tím, že místo kombinace logických hodnot zde pracujeme s výherními stavy 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. Uživatel je upozorněn, 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 (s indexem menším než i) do uzlu i. Tím zajistíme, že uzel 0 (počáteční pozice) může dosáhnout všechny ostatní uzly, a současně vytvoříme acyklickou kostru grafu.
2. **Přidání dalších hran:** Pro každou dvojici uzlů i, j (kde $i \neq j$), podle dané pravděpodobnosti přidáme hranu $i \to j$, pokud ještě neexistuje. Hrany mohou být přidány v libovolném směru, což umožňuje vznik cyklů v grafu.

Výsledný graf je vždy souvislý s uzlem 0 jako počáteční pozicí. Graf může obsahovat cykly, což odpovídá reálným herním situacím, kde se hra může dostat do opakujících se pozic. Algoritmus analýzy (viz sekce 4.7) je navržen tak, aby správně zpracoval i grafy s cykly a přiřadil těmto pozicím status remízy, pokud nelze jednoznačně určit výherce.

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 myší 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 směrované 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 využívá iterativní přístup (retrográdní analýzu) pro postupné označování pozic od koncových uzlů směrem k počáteční pozici.

4.7.1 Vyhodnocení výherních pozic

Algoritmus určuje, zda je daná pozice výherní pro hráče, který je v ní na tahu, nebo zda skončí remízou (v případě cyklů bez vynuceného výsledku). Vyhodnocení probíhá následovně:

1. **Koncové pozice:** Pozice bez dalších tahů jsou prohrávající pro hráče, který je v nich na tahu.

2. **Zpětné šíření:** Od koncových pozic se postupně propagují výsledky:
    - Pokud existuje tah do prohrávající pozice soupeře, aktuální pozice je vyhrávající.
    - Pokud tentýž hráč pokračuje v tahu (bez změny hráče), tah do vyhrávající pozice znamená, že i původní pozice je vyhrávající.
    - Pokud všechny tahy vedou do vyhrávajících pozic soupeře, aktuální pozice je prohrávající.
3. **Neurčené pozice:** Pozice, jejichž status nelze jednoznačně určit (například kvůli cyklům), zůstávají označeny jako remíza.

Časová složitost je $O(V + E)$, kde V je počet uzlů a E počet hran.

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. Algoritmus zaznamenává každý krok aktualizace stavu grafu:

- **Inicializace:** Všechny pozice jsou na začátku ve stavu REMÍZA.
- **Terminální stavy:** Identifikace pozic bez tahů (PROHRA).
- **Aktualizace:** Postupné šíření výherních a prohrávajících stavů grafem.
- **Vysvětlení:** U každého kroku je zobrazeno vysvětlení, proč došlo ke změně stavu (např. „Z pozice X lze táhnout do prohrávající pozice Y").

Uživatel může procházet kroky analýzy pomocí navigačních tlačítek. Aktuálně aktualizovaný uzel je v grafu zvýrazněn. Tato funkce pomáhá pochopit, jak algoritmus postupně řeší hru a jak se vypořádává s cykly.

Kapitola 5

Problém neprázdnosti jazyka bezkontextové gramatiky

5.1 Teoretický základ

Problém neprázdnosti jazyka bezkontextové gramatiky (CFG Non-emptiness Problem) je dalším příkladem P-úplného problému [10]. 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 $G = (N, \Sigma, P, S)$ [11], kde:

- N je konečná množina neterminálních symbolů (neterminálů).
- Σ je konečná množina terminálních symbolů (terminálů), disjunktní s N.
- P je konečná množina přepisovacích pravidel tvaru $A \to \alpha$, 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 [11]. 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 neprázdnosti jazyka bezkontextové gramatiky je P-úplný [10, 8]. 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 [10].

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 (ϵ) 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 se mohou zadat hodnoty v rozmezí 1 až 5, přičemž délka každého pravidla je náhodně zvolena v intervalu od 1 do hodnoty 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 $A \to A\alpha$
  - Pravá rekurze: Povoluje pravidla tvaru $A \to \alpha A$
  - Pokud jsou obě vypnuté, generátor vytváří pouze pravidla bez rekurze
  - Pravděpodobnost aplikace rekurze je 30% při povoleném režimu
- **Generování epsilon (ϵ):** Tři režimy:
  - **Ne:** Negenerují se žádná epsilon pravidla
  - **Náhodně:** Epsilon pravidla se generují s pravděpodobností 8%
  - **Vždy:** Epsilon pravidla se generují s pravděpodobností 15%, přičemž je garantován alespoň jeden výskyt v gramatice

5.4.3 Algoritmus generování

Algoritmus generování probíhá v těchto krocích:

1. **Vytvoření symbolů:** Generují se pole neterminálů a terminálů podle zadaných počtů. První neterminál je automaticky nastaven jako počáteční symbol S gramatiky.
2. **Generování pravidel pro každý neterminál:**
   (a) Náhodně se určí počet pravidel pro tento neterminál (v rozmezí vstupních parametrů - min-max)
   (b) Pro každé pravidlo:
   - Kontrola epsilon režimu – jestli je povolená generace epsilon terminálu, je zde šance na jeho vytvoření (podle zvoleného režimu)
   - Jinak určení náhodné délky pravé strany (1 až vstupní parametr max délka pravé strany)
   - 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.

5.5 Předpřipravené sady

Ve složce `Sady/Grammar` jsou uloženy předpřipravené příklady gramatik. Tyto sady slouží ke kontrolované 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

Modul `GrammarEvaluator.js` obsahuje implementaci algoritmu pro vyhodnocení neprázdnosti gramatiky. Algoritmus zjišťuje, zda je počáteční symbol S produktivní – tedy zda z něj lze odvodit slovo složené pouze z terminálů nebo prázdný řetězec ϵ.

5.6.1 Identifikace produktivních neterminálů

Algoritmus pracuje iterativně a postupně označuje produktivní neterminály [10]:

1. **Inicializace:** Vytvoříme prázdnou množinu produktivních symbolů $\mathcal{P}$.
2. **Základní krok:** Do $\mathcal{P}$ přidáme všechny neterminály A, které mají pravidlo $A \to w$, kde $w \in \Sigma^*$ (pravá strana obsahuje pouze terminály nebo je prázdná).
3. **Iterativní rozšiřování:** Opakovaně procházíme všechna pravidla. Pokud existuje pravidlo $A \to X_1 X_2 \dots X_k$, 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), což odpovídá principu algoritmu průchodu do šířky (BFS). 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 $O(|P| \cdot l)$ [11], kde $|P|$ značí celkový počet pravidel a l je maximální délka pravé strany pravidla. Každé pravidlo je totiž kontrolováno nejvýše jednou pro každý symbol na pravé straně.

5.6.2 Rekonstrukce derivace

Algoritmus během svého běhu zaznamenává pro každý produktivní neterminál všechna jeho produktivní pravidla. Na rozdíl od základní verze algoritmu, která ukládá pouze první nalezené produktivní pravidlo, implementace uchovává všechna produktivní pravidla pro každý neterminál. Tato data jsou ukládána do mapy `allWitnesses`, která pro každý produktivní neterminál obsahuje pole všech jeho produktivních pravých stran.

Po dokončení analýzy produktivních neterminálů je možné provést sestavení derivačního stromu, pokud je počáteční symbol S produktivní. Sestavení probíhá rekurzivně:

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:** Rekurzivní konstrukce stromu je omezena na maximální hloubku 30 úrovní. Pokud je tato hloubka překročena, vrátí se chyba a místo vykreslení stromu se zobrazí informační zpráva.

3. **Stavba uzlů:** Strom se buduje rekurzivně funkcí `buildNode()`:

   - Pro každý neterminál vytvoříme uzel s unikátním identifikátorem
   - Náhodně vybereme jedno z jeho produktivních pravidel
   - Pro každý symbol na pravé straně pravidla rekurzivně vytvoříme potomky (zvýšíme hloubku o 1)
   - Terminály a ϵ vytvoří listy stromu bez dalších potomků

4. **Extrakce slova:** Funkce `extractTerminals()` prochází strom zleva doprava a čte terminální listy, čímž získá řetězec vygenerovaný danou derivací.

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 ukáže důkaz vykreslením derivačního stromu pro jedno z možných slov.

5.7.1 Komponenta pro vykreslení

Pro vykreslení derivačního stromu je využit algoritmus z komponenty `DerivationTreeVisual`, která využívá knihovnu *react-force-graph-2d*. Strom je zobrazen ve stromovém rozložení pomocí režimu `dagMode="td"` (top-down).

5.8 Krokové vyhodnocení

Pro detailnější pochopení algoritmu implementuje aplikace krokovatelnou analýzu prostřednictvím komponenty `StepByStepGrammar`. Tato komponenta využívá modul `GrammarStepEvaluator.js` k simulaci algoritmu a zobrazuje průběh výpočtu v modálním okně.

5.8.1 Struktura zobrazení

Rozhraní krokového vyhodnocení je rozčleněno do několika sekcí:

- **Seznam pravidel gramatiky:** Zobrazuje všechna produktivní pravidla. Aktuálně kontrolované pravidlo je vizuálně zvýrazněno žlutým pozadím a oranžovým ohraničením. Pravidla, jejichž levá strana je již produktivní, jsou označena zeleným štítkem s textem „Produktivní".
- **Množina produktivních symbolů:** Aktuální obsah množiny $\mathcal{P}$ je zobrazen formou zelených štítků s písmenem neterminálu. Tato množina se postupně rozšiřuje během jednotlivých iterací algoritmu, za podmínky že gramatika obsahuje produktivní neterminály.
- **Vysvětlení kroku:** Textový popis aktuální operace algoritmu, například „Pravidlo $A \to a$ obsahuje pouze terminály. A je produktivní"nebo „Pravidlo se stalo produktivním díky B. A přidán mezi produktivní".

5.8.2 Navigace a ovládání

Pro pohyb mezi jednotlivými kroky analýzy jsou k dispozici čtyři navigační tlačítka:

- **Začátek:** Přesun na první krok algoritmu (inicializace prázdné množiny $\mathcal{P}$)
- **Předchozí:** Krok zpět v historii výpočtu
- **Další:** Posun na následující krok
- **Konec:** Skok na finální krok s výsledkem analýzy

Aktuální pozice je indikována čítačem ve formátu „Krok x z n", kde n představuje celkový počet kroků. Tato interaktivní vizualizace umožňuje sledovat, jak algoritmus postupně identifikuje produktivní neterminály na základě již známých produktivních symbolů a jak závislosti mezi pravidly určují pořadí jejich vyhodnocení.

Kapitola 6

Závěr

Cílem této práce bylo vytvořit interaktivní webovou aplikaci pro vizualizaci a demonstraci převoditelnosti a řešení P-úplných problémů na příkladu tří vybraných P-úplných problémů: Monotone Circuit Value Problem, kombinatorických her na grafu a problému neprázdnosti jazyka bezkontextové gramatiky.

6.1 Dosažené výsledky

Výsledná aplikace splňuje všechny stanovené cíle a poskytuje jednotné prostředí pro práci se třemi P-úplnými problémy. Pro každý problém je implementováno kompletní řešení zahrnující:

- **Flexibilní vstupní systém:** Uživatelé mohou zadávat instance problémů třemi způsoby – manuálním zadáním, generováním náhodných instancí nebo pomocí předpřipravených příkladů. Tyto možnosti umožňují jak experimentování s vlastními příklady, tak rychlé testování algoritmu na náhodných datech.
- **Vizualizaci řešení:** Každý problém je doprovázen grafickou reprezentací, která využívá knihovnu *react-force-graph-2d* pro interaktivní zobrazení grafových struktur. Uživatel může manipulovat se zobrazenými grafy, přibližovat je a přesouvat uzly pro lepší orientaci.
- **Krokové vyhodnocení:** Implementace krokovatelného průchodu algoritmů umožňuje sledovat každý jednotlivý krok výpočtu s textovým vysvětlením.
- **Export a import dat:** Možnost ukládání a načítání instancí problémů ve formátu JSON podporuje sdílení příkladů a vytváření knihoven testovacích případů.

Aplikace taktéž demonstruje dva konkrétní převody z MCVP na kombinatorickou hru a na bezkontextovou gramatiku. Oba převody jsou implementovány krokovatelně, takže uživatel může sledovat, jak se jednotlivé uzly obvodu transformují na odpovídající struktury v cílovém problému.

Tato vizualizace názorně ilustruje techniku polynomiálních redukcí a vzájemnou převoditelnost P-úplných problémů.

6.2 Vzdělávací přínos

Aplikace představuje vzdělávací nástroj pro výuku teorie složitosti s interaktivním přístupem. Krokové vyhodnocení s textovými vysvětleními umožňuje pochopit fungování algoritmů na konkrétních příkladech, zatímco vizualizace převodů demonstruje vzájemnou převoditelnost P-úplných problémů. Generování náhodných instancí pak podporuje experimentální učení a pozorování chování algoritmů na různých strukturách vstupů.

6.3 Možnosti dalšího rozvoje

Přestože aplikace poskytuje funkční implementaci všech plánovaných funkcí, existuje prostor pro další rozšíření:

- **Další P-úplné problémy:** Aplikace by mohla být rozšířena o další P-úplné problémy, jako je například vyhodnocování booleovských formulí v konjunktivní normální formě nebo problém dosažitelnosti v grafech s omezenou šířkou.
- **Více převodů:** Implementace dalších směrů převodů – například z kombinatorických her na gramatiky nebo opačným směrem z gramatik na MCVP – by poskytla kompletnější obraz vzájemné převoditelnosti těchto problémů.
- **Výkonnostní optimalizace:** Pro velmi velké instance (stovky uzlů) by mohly být implementovány optimalizace vykreslování a výpočtu, například pomocí virtualizace zobrazení nebo progresivního vykreslování.

Výsledná aplikace může sloužit jako doplněk k tradičním výukovým materiálům v kurzech teorie složitosti a teoretické informatiky, kde pomáhá studentům lépe pochopit abstraktní koncepty prostřednictvím konkrétních interaktivních příkladů.

Literatura

1. ARORA, Sanjeev; BARAK, Boaz. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. ISBN 978-0521424264.
2. 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`.
3. BOOTSTRAP TEAM. *Bootstrap – The most popular HTML, CSS, and JS library in the world* [Online]. 2024. Dostupné také z: `https://getbootstrap.com/`.
4. BOSTOCK, Mike. *D3.js – Data-Driven Documents* [Online]. 2024. Dostupné také z: `https://d3js.org/`.
5. ESLINT TEAM. *ESLint – Pluggable JavaScript linter* [Online]. 2024. Dostupné také z: `https://eslint.org/`.
6. KHADRA, Fadi. *react-toastify – React notification made easy* [Online]. 2024. Dostupné také z: `https://fkhadra.github.io/react-toastify/`.
7. META PLATFORMS, INC. *React – A JavaScript library for building user interfaces* [Online]. 2024. Dostupné také z: `https://react.dev/`.
8. MIYANO, Satoru; SHIRAISHI, Shunsuke; SHOUDAI, Takayoshi. *A List of P-Complete Problems*. 1990-12. Tech. zpr., RIFIS-TR-CS-17. Kyushu University. Dostupné také z: `https://catalog.lib.kyushu-u.ac.jp/opac_download_md/3123/rifis-tr-17.pdf`.
9. PAPADIMITRIOU, Christos H. *Computational Complexity*. Addison Wesley, 1994. ISBN 978-0201530827.
10. SAWA, Zbyněk. *Teoretická informatika – Podklady pro přednášky* [Online]. 2025. Dostupné také z: `https://www.cs.vsb.cz/sawa/ti/slides/ti-slides-08.pdf`.
11. SAWA, Zbyněk. *Úvod do teoretické informatiky – Bezkontextové gramatiky* [Online]. 2014. Dostupné také z: `https://www.cs.vsb.cz/kot/download/uti2014/uti-10-cz.pdf`.
12. SAWA, Zbyněk. *Úvod do teoretické informatiky – Turingovy stroje* [Online]. 2014. Dostupné také z: `https://www.cs.vsb.cz/kot/download/uti2014/uti-08-cz.pdf`.

13. 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`.

14. VITE TEAM. *Vite – Next Generation Frontend Tooling* [Online]. 2024. Dostupné také z: `https://vitejs.dev/`.