

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 ukázků 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

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 zkratek	7
Seznam obrázků	8
Seznam tabulek	9
1 Úvod	10
2 Použité technologie a architektura	13
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
3 Monotone Circuit Value Problem	16
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 Ukládání a načítání MCVP obvodů	22

4 Kombinatorická hra	23
4.1 Teoretický základ	23
4.2 Formát vstupu a reprezentace	24
4.3 Algoritmus analýzy	24
4.4 Krokové vyhodnocení	25
4.5 Interaktivní editace grafu	26
4.6 Generování náhodných her	26
4.7 Vizualizace grafu	27
4.8 Propojení s MCVP	27
4.9 Ukládání a načítání her	28
5 Prázdnost bezkontextových gramatik	29
6 Závěr	30
Literatura	31
Přílohy	31
A Dlouhý zdrojový kód	32

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
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
JSON	– JavaScript Object Notation – Objektový zápis JavaScriptu
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

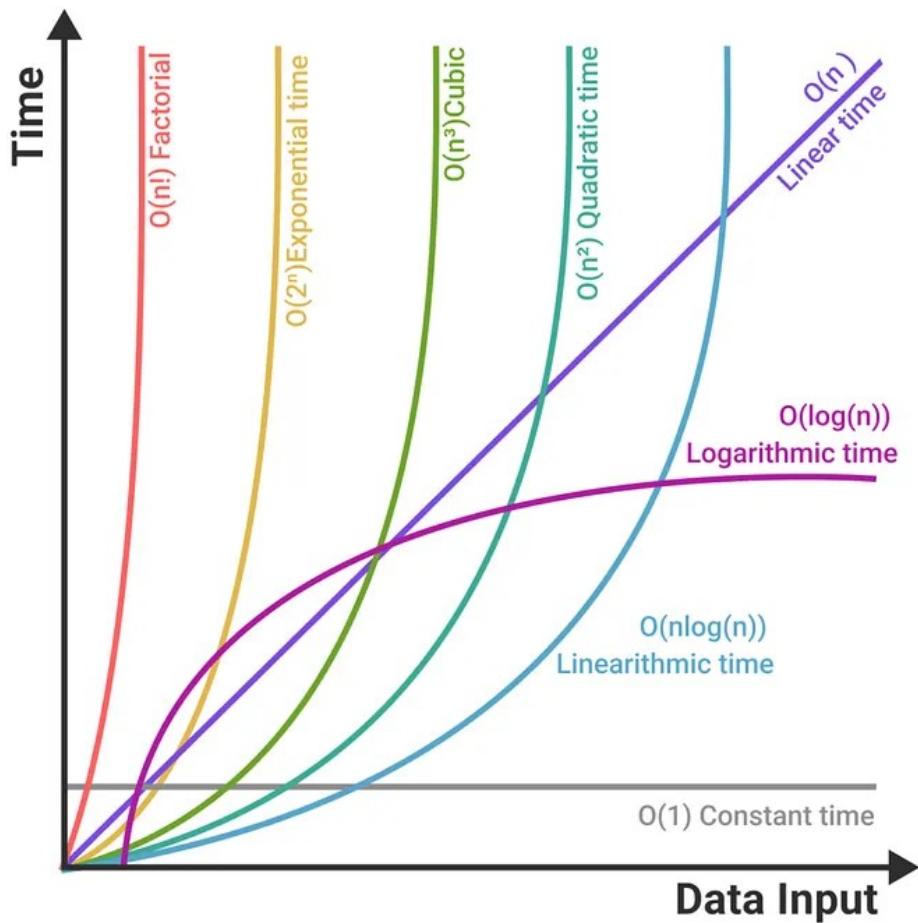
Teoretická informatika poskytuje způsob jak zkoumat, porozumět a optimalizovat možnosti a limity výpočetních systémů. Hlavním bodem tohoto zkoumání je teorie složitosti, která klasifikuje problémy na základě zdrojů potřebných pro jejich vyřešení, jako je čas a paměť. Jednou z nejvýznamnějších složitostních 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 paměťovou 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 nespadají 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 rámc) 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 *reactivity* – 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 zjednoduší 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 jendotné nastavení grafů byly vytvořeny pomocné moduly v adresáři `src/Hooks` – modul `useGraphColors` spravuje barvy použité v grafických vizualizacích a modul `useGraphSettings` obshahuje 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 varování 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 větví repozitáře dojde automaticky k novému nasazení aplikace. Toto řešení výrazně zjednoduší 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 bud vstupní proměnné (listy) nebo logická hradla (vnitřní uzly). Hrany reprezentují tok logických hodnot – přenášejí výsledky vyhodnocení z jednoho uzlu jako vstupy do uzel 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 uzel 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.

Manuálně

Generovat

Načíst ze sady

Interaktivně

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.

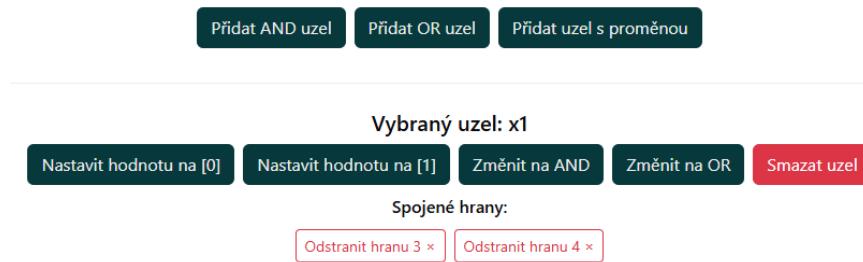
Komponenta `StepByStepTree` zobrazuje každý krok graficky – aktuálně vyhodnocovaný uzel je zvýrazněn a vedle grafu je zobrazena informace o hodnotách vstupů a výsledku operace.

3.4 Interaktivní editace obvodu

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

- **Přidávat uzly:** Pomocí tlačítka 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

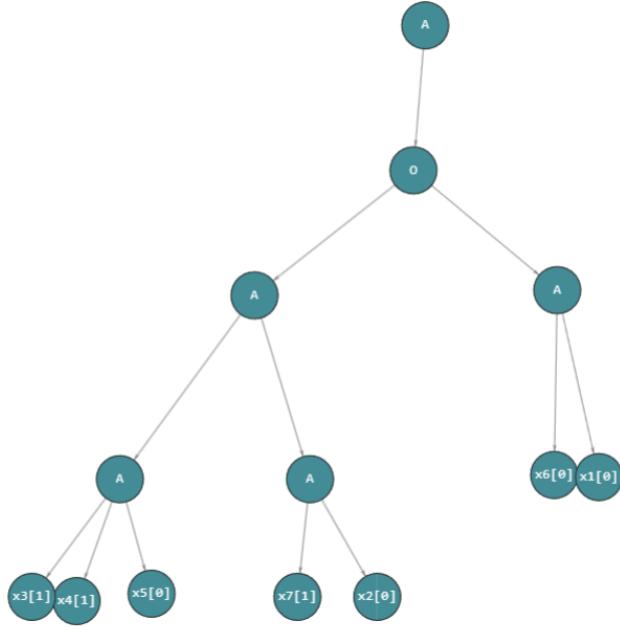
Aplikace obsahuje generátor náhodných obvodů v modulu `Generator.js`. Uživatel může upravit dva parametry:

- **Počet hradel:** Kolik uzelů s operacemi (hradel AND a OR) bude obvod obsahovat.
- **Počet proměnných:** Kolik uzelů 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 uzelů 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 uzelů jako potomky a náhodně zvolí operaci (AND nebo OR) a přidá se do množiny použitelných uzelů pro potomky. Poslední hradlo spojí všechny zbývající uzly a stane se kořenem. Výsledný graf je platný DAG.

3.6 Vizualizace obvodu

Vizualizace monotonného 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.js` 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 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 [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 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

3.9.1 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.

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, která zpětně vyhodnocuje pozice od koncových uzelů. Implementace je podobná algoritmu vyhodnocování MCV (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 a reprezentace

Aplikace nabízí několik způsobů zadání herního grafu. Uživatel může zadat graf textově, vytvořit ho interaktivně nebo načíst z připravených sad.

4.2.1 Textový formát

Textové zadání využívá jednoduchý formát:

```
nodes: 1:1, 2:2, 3:1; edges: 1->2, 1->3
```

- **nodes**: definuje uzly ve formátu `id:player`, kde `id` je identifikátor uzlu a `player` určuje, který hráč je v této pozici na tahu (1 nebo 2).
- **edges**: definuje hrany ve formátu `source->target`, reprezentující možný tah z pozice `source` do pozice `target`.

Kromě samotného grafu musí uživatel také určit počáteční pozici.

4.2.2 Parsování vstupu

Parsování textového vstupu implementuje modul `Parser.js`. Funkce `parseExpressionToTree()` zpracovává vstupní řetězec v těchto krocích:

1. Rozdělí výraz na část s uzly a část s hranami pomocí oddělovače středníkem.
2. Zpracuje definice uzlů – pro každý uzel vytvoří objekt `GamePosition` s ID a přiřazeným hráčem.
3. Zpracuje definice hran – pro každou hranu propojí odpovídající uzly a aktualizuje seznamy potomků a rodičů.
4. Vrátí strukturu `GameGraph` obsahující všechny pozice a odkaz na počáteční pozici.

Výsledná struktura používá třídy `GamePosition` a `GameGraph` definované v modulu `NodeClasses.js`.

4.3 Algoritmus analýzy

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

4.3.1 Vyhodnocení výherních pozic

Algoritmus `computeWinner()` používá průchod do hloubky (DFS) s memoizací pro určení výherních pozic. Pro každou pozici p určuje, zda je výherní pro Hráče I:

- **Koncová pozice Hráče II:** Pokud je Hráč II na tahu a nemá žádné možné tahy, Hráč I vyhrává. Taková pozice je výherní.
- **Pozice Hráče I:** Hráč I vyhrává, pokud existuje alespoň jeden tah do výherní pozice. Stačí jedna výherní možnost.
- **Pozice Hráče II:** Hráč I vyhrává pouze tehdy, když všechny možné tahy Hráče II vedou do výherních pozic pro Hráče I. Jinak Hráč II může zvolit tah do prohrávající pozice a zabránit výhře Hráče I.

Formálně můžeme zapsat:

- Pozice p je výherní pro Hráče I, pokud:
 - p patří Hráči II a nemá následníky, nebo
 - p patří Hráči I a existuje následník q takový, že q je výherní, nebo
 - p patří Hráči II a všichni následníci q jsou výherní.

Algoritmus používá zásobník pro iterativní průchod grafem a slovník `memo` pro ukládání již vyhodnocených pozic. Detekuje také cykly pomocí množiny `processing` – cyklické pozice jsou považovány za prohrávající.

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

4.3.2 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.

4.4 Krokové vyhodnocení

Pro vzdělávací účely implementuje komponenta `StepByStepGame` krokovatelnou analýzu. Funkce `computeWinningStrategySteps()` zaznamenává každý krok vyhodnocení:

- ID analyzované pozice

- Výsledky vyhodnocení všech potomků
- Finální rozhodnutí, zda je pozice výherní
- Textové vysvětlení rozhodnutí

Uživatel může procházet kroky analýzy pomocí navigačních tlačítek. Aktuálně analyzovaný uzel je ve vizualizaci zvýrazněn a vedle grafu se zobrazuje detailní vysvětlení logiky rozhodnutí. Tato funkce pomáhá pochopit, jak algoritmus postupně buduje znalost o výherních pozicích od listů ke kořeni.

4.5 Interaktivní editace grafu

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

- **Přidávat uzly:** Vytvořit novou pozici pro Hráče 1 nebo Hráče 2 pomocí tlačítka pod grafem.
- **Upravovat uzly:** Kliknutím na uzel ho označíme. Označený uzel můžeme:
 - Změnit na pozici Hráče 1 nebo Hráče 2
 - Odstranit z grafu
 - Nastavit jako počáteční pozici
 - Použít jako zdroj nebo cíl pro vytvoření hrany
- **Vytvářet hrany:** Po označení prvního uzlu aktivujeme režim přidávání hrany. Následným kliknutím na druhý uzel vytvoříme hranu mezi nimi.
- **Odstraňovat hrany:** Označíme uzel, aktivujeme režim odstranění hrany a klikneme na cílový uzel.
- **Reorganizovat graf:** Uzly lze přesouvat myší pro lepší vizuální uspořádání.

Aplikace průběžně validuje graf – upozorní uživatele, pokud není nastaven počáteční uzel nebo pokud graf obsahuje cykly (což by porušilo požadavek na DAG).

4.6 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:** Nejprve vytvoříme uzly očíslované od 0 do $n - 1$, kde n je počet pozic. Každému uzlu náhodně přiřadíme Hráče 1 nebo 2. Poté pro každý uzel $i > 0$ vytvoříme hranu z náhodného uzlu $j < i$ do uzlu i . To zajistí, že výsledný graf je acyklický (DAG) a souvislý – z uzlu 0 lze dosáhnout všech ostatních uzlů.
2. **Přidání dalších hran:** Procházíme všechny dvojice uzlů (i, j) , kde $i < j$, a s pravděpodobností odpovídající vstupu přidáme hranu $i \rightarrow j$, pokud tato hrana ještě neexistuje. Směr je vždy z nižšího indexu k vyššímu, což zachovává acykličnost.

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

4.7 Vizualizace grafu

Vizualizace herního grafu využívá komponentu `DisplayGraph`, která je postavena na knihovně `react-force-graph-2d`. Graf zobrazuje:

- **Uzly:** Pozice jsou obarveny podle přiřazeného hráče – modrá pro Hráče 1, červená pro Hráče 2. Výherní pozice mají světlejší odstín, prohrávající tmavší.
- **Počáteční pozice:** Označena výrazným rámečkem a nápisem „START“.
- **Hrany:** Možné tahy jsou zobrazeny jako šipky. Hrany patřící do optimální strategie jsou zvýrazněny tlustší čarou a jasnější barvou.
- **Popisky:** Každý uzel zobrazuje své ID a přiřazeného hráče.

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

4.8 Propojení s MCVP

Aplikace demonstrouje P-úplnost implementací převodu z MCVP na kombinatorickou hru (viz kapitola 3.7). Tento převod je realizován v modulu `ConversionCombinatorialGame.js`.

Princip převodu mapuje hradla MCVP obvodu na herní pozice:

- Hradlo OR odpovídá pozici Hráče I – Hráč I si volí, kterou větev následovat

- Hradlo AND odpovídá pozici Hráče II – Hráč II volí větev
- Proměnná s hodnotou 1 vytváří pozici Hráče II bez tahů (Hráč I vyhrává)
- Proměnná s hodnotou 0 vytváří pozici Hráče I bez tahů (Hráč I prohrává)

Výsledná hra zachovává sémantiku – Hráč I má výherní strategii právě tehdy, když MCVP obvod vyhodnotí na hodnotu 1.

4.9 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 příkladů.

4.9.1 Předpřipravené sady

Ve složce `Sady/CombinatorialGame` najdeme předpřipravené herní grafy různé velikosti a složitosti. Tyto sady slouží jako ukázkové příklady a výchozí bod pro experimentování s algoritmem.

Kapitola 5

Prázdnost bezkontextových gramatik

Kapitola 6

Závěr

Literatura

1. PAPADIMITRIOU, Christos H. *Computational Complexity*. Addison-Wesley, 1993. ISBN 978-0201530827.
2. BERNARDES, Bruno. *Understanding Big O Notation* [Medium]. 2023. Dostupné také z: <https://medium.com/@bruno-bernardes-tech/understanding-big-o-notation-0646b764797e>.
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]. 2024. 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