FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Frameworky pro řešení rozvrhovacích úloh
Student:	Anežka Batěčková
Vedoucí:	Ing. Ondřej Harcuba
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

1. Vypracujte přehled frameworků či knihoven, které podporují optimalizaci rozvrhovacích úloh. Soustřeďte se zejména na ty, které jsou vhodné pro rozvrhování průmyslové výroby.
2. Na základě informací, které dodá vedoucí práce, specifikujte konkrétní zadání optimalizace rozvrhování průmyslové výroby.
3. Stanovte kritéria pro porovnání vybraných frameworků či knihoven.
4. Po konzultaci s vedoucím práce zvolte několik vhodných metod a na datech, které dodá vedoucí práce, je otestujte.
5. Diskutujte výsledky a doporučte vhodné frameworky či knihovny pro nasazení.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 18. prosince 2018

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Bakalářská práce

Frameworky pro řešení rozvrhovacích úloh

Anežka Batěčková

Vedoucí práce: Ing. Ondřej Harcuba

10. května 2019

Poděkování

Tímto bych ráda poděkovala Ing. Onřeji Harcubovi za skvělé vedení této práce. Za jeho rady a postřehy i za poskytnutí testovacích dat.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 10. května 2019 .....

Odkaz na tuto práci

Batěčková, Anežka. *Frameworky pro řešení rozvrhovacích úloh.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato práce uvádí čtenáře do problematiky rozvrhovacích problémů, seznamuje ho s pojmy z tohoto odvětví a nastiňuje přístupy k jejich řešení. Cílem je vypracování přehledu frameworků řešících a optimalizujících tento typ úloh. Soustředí se především na Javovské knihovny vhodné pro rozvrhování v průmyslové výrobě, a to konkrétně CPLEX, Choco, OR-Tools a JaCoP. Ty jsou testovány na sadě dat s ohledem na čas a výsledek, který nakonec podají. Porovnávají se také s ohledem na API a uživatelskou přívětivost, dokumentaci, uživatelskou podporu a typ licence.

Klíčová slova rozvrhování, Java, framework, CPLEX, Choco, OR-Tools, JaCoP

Abstract

This thesis introduces problems of scheduling tasks, scheduling terms and shows some approaches for solving these tasks. The goal of this thesis is to create an overview of frameworks solving and optimizing this kind of tasks. The thesis focuses on Java libraries which are feasible for solving scheduling tasks in manufacturing, namely CPLEX, Choco, OR-Tools and JaCoP. These frameworks are tested on a data set considering time complexity and the actual output. They are also compared considering API and user-friendliness, documentation, support and a type of their license.

Keywords scheduling, Java, framework, CPLEX, Choco, OR-Tools, JaCoP

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Tématem této práce jsou frameworky pro řešení rozvrhovacích úloh.

Rozvrhování úzce souvisí s plánováním. To se zabývá rozdělením zdrojů mezi jednotlivé úkoly, jednotlivými závislostmi mezi úkoly a udává pořadí, ve kterém se musí dané úkoly vykonávat, aby vznikl výsledný produkt. A rozvrhování přiděluje jednotlivým úkolům čas, kdy se mají vykonávat. Tyto procesy jsou používané v běžném životě každého z nás. Nastavíme si budík, abychom se stihli obléct, nasnídat a přišli včas do práce. Je dobré se nejříve obléknout, pak najíst a pak až jít pracovat. My jsme zdrojem a toto jsou naše každodenní úkoly, které je dobré plánovat v určitém pořadí. Neoblečení a hladoví nemůžeme jít do práce, práce jakožto úkol je tedy závislá na dokončení předešlých dvou úkolů. Teď může přijít na řadu rozvrhování, kdy se rozhodneme, že vstaneme v 7:00, 7:00–7:15 se budeme oblékat, 7:15–7:45 budeme snídat a v 7:50 vyrazíme do práce.

Plánování a rozvrhování ale nebývá vždy takto jednoduché. Tato práce se zabývá převážně rozvrhováním v průmyslové výrobě, kde je o mnoho více zdrojů, více úkolú, více závislostí, omezení a více faktorů, pro které je potřeba najít optimální řešení. Každý úkol může mít určitou prioritu, čas, kdy se nejdřív může začít na úkolu pracovat, a čas, do kdy musí být hotov. Někdy se hledá optimální řešení vzhledem k času, kdy nejdřív se dají všechny úkoly splnit. A někdy se minimalizuje počet potřebných pracovníků. Problémy plánování bývají různorodé, stejně tak jako přístupy k jejich řešení. Frameworků řešících tento problém je mnoho a proto je potřeba sestavit jejich přehled.

Tato práce nejdříve seznámí čtenáře s problematikou plánování a rozvrhování. Poté popíše různé Javovské knihovny řešící tyto problémy. A nakonec tato řešení porovná.

Cíle

Prvním cílem této práce je lehké uvedení čtenáře do problematiky rozvrhovacích úloh a přístupů k jejich řešení.

Druhým cílem práce je výběr několika Javovských frameworků řešících tento typ problémů a seznámení se s nimi. V každém z těchto frameworků bude napsán program řešící jednu konkrétní rozvrhovací úlohu a tento program bude spuštěn na různě velkých datech. Nakonec budou všechny frameworky porovnány na základě jednotlivých výsledků běhu programu, API, dokumentace, uživatelské podpory a typu licence.

Hlavním cílem práce je sestavení přehledu frameworků řešících rozvrhovací problémy a doporučení vhodného frameworku pro nasazení v průmyslové výrobě.

Kapitola 1

Terminologie

Plánovací a rozvrhovací problémy patří do skupin NP, NP-úplných a některé dokonce NP-těžkých problémů. Mnohé z nich se tedy nedají řešit v polynomiálním čase a u některých se ani nedá v polynomiálním čase ověřit, že je daný výsledek řešením. Alespoň ne za pomoci Turingova modelu a programování, jak ho známe dnes. Tyto problémy mohou být hodně komplexní a tak se pro zobecnění a zjednodušení zavádějí následující pojmy.

Prvním je „*úloha*“. Jedná se o komplexní problém, který se může skládat z mnoha menších podproblémů – „*úkolů*“. Tyto úkoly se pak řeší pomocí „*strojů*“ a jejich „*operací*“. Operace může vyžadovat konkrétní stroj, může mít různou délku trvání a nemusí úkol splnit úplně. Pro splnění úkolu může být vyžadováno operací více. Úlohou může být například vyrobit 20 automobilů, úkolem pak vyrobit karoserii pro první auto a operací samotné lakování karoserie prvního auta na konkrétním lakovacím stroji.

Stroje patří mezi „*zdroje*“. Zdroj lze chápat jako cokoliv, co pomáhá při zpracování vstupu na požadovaný výstup. V reálném světě jsou zdroji zaměstnanci, materiál, již zmiňované stroje a nebo třeba kolej na vlakovém nádraží. Stroje jsou specifickým případem zdrojů, které daný vstup mění přímo na požadovaný výstup. K těmto strojům navíc často přibývají „*lidské zdroje*“ – zaměstnanci, které je potřeba ke stroji na daný čas přiřadit, aby mohl fungovat.

Ke komplexnosti problémů přispívají různé „*omezující podmínky*“, které omezují zpracovávaní úkolů. Každý úkol má nějaké parametry. Čas, který zabere jeho zpracování. Čas, kdy nejdřív se na daném úkolu může začít pracovat. Čas, do kdy musí být úkol splněn. Určitou váhu, prioritu... Mezi úkoly mohou existovat také nějaké závislosti. Není možné začít pracovat na daném úkolu, dokud nejsou dokončeny všechny ostatní úkoly, na kterých je daný úkol závislý. A například tyto závislosti a omezení se vyjadřují pomocí omezujících podmínek.

U problémů, kde se navíc hledí na optimálnost řešení, je důležitým pojmem „*cíl*“. Tedy kritérium, vůči kterému se optimální řešení hledá. Může se jednat

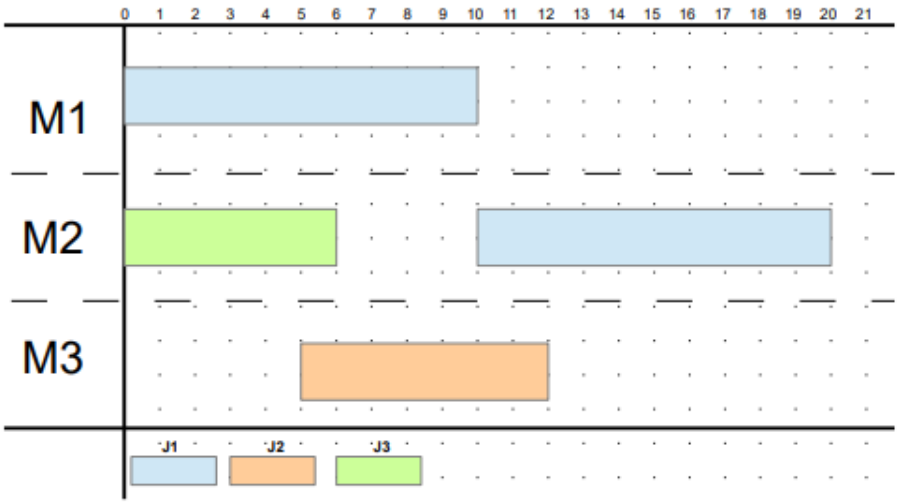
například o minimalizaci času, nákladů nebo třeba minimalizaci potřebných lidských zdrojů.

1.0.1 Ganttův diagram

K vizualizaci řešení plánovacích a rozvrhovacích problémů se používá tzv. Ganttův diagram. Ten může vypadat různě podle konkrétního problému. V této práci budu dále používat Ganttův diagram, kde horizontální osa značí čas a vertikální jednotlivé zdroje. Úkoly, respektivě jednotlivé operace, ze kterých se úkoly skládají, budou zobrazeny v podobě různě barevných pruhů. Každý úkol bude mít svou barvu. Pod diagramem bude vždy jasně definováno, která barva patří kterému úkolu.

Pro příklad si uveďme Ganttův diagram pro rozvržení tří úkolů J_1, J_2 a J_3 mezi tři stroje M_1, M_2 a M_3. Úkol J_1 se skládá ze dvou operací, J_2 a J_3 pouze z jedné. Podle diagramu 1.1 budou stroje zpracovávat úkoly následovně:

- Na stroji M_1 se bude v čase 0–10 zpracovávat 1. operace úkolu J_1
- Na stroji M_2 se bude v čase 0–6 zpracovávat úkol J_3 a pak v čase 10–20 2. operace úkolu J_1
- Na stroji M_3 se bude v čase 5–12 zpracovávat úkol J_2



Obrázek 1.1: Příklad Ganttova diagramu

1.0.2 Grahamova klasifikace

Pro zjednodušení a zobecnění rozvrhovacích problémů se používá Grahamova klasifikace. V tomto textu je zmiňována hlavně pro zlepšení představy, jak moc složité mohou rozvrhovací problémy být a co všechno v sobě mohou skrývat. Text je z většiny složen z parafrází knihy Scheduling Algorithms [1] a její kapitoly o Grahamově klasifikaci obohacené o scripta od Hanky Rudové [2]. Pro další účely této práce se budeme zabývat ale o něco jednoduššími problémy a z většiny věcí níže uvedených budeme potřebovat maximálně tak povědomí o tom, že existují. Přesto je dobré si Grahamovu klasifikaci představit.

Problém se podle Grahamovy klasifikace zapisuje ve tvaru $\alpha|\beta|\gamma$, kde:

- α popisuje charakteristiku strojů (přiřazení úkolů strojům);
- β popisuje charakteristiku úkolů;
- γ popisuje cíle.

1.0.2.1 Charakteristika strojů

Charakteristika strojů je podle [1] specifikována množinou α obsahující maximálně dva prvky α_1 a α_2.

α_1 může mít mnoho podob:

- α_1 je vynechána v případě, že každý úkol musí být zpracován na pro sebe vyhrazeném stroji.
- $\alpha_1 = P$ znamená, že máme identické stroje, na kterých může paralerně probíhat zpracování úkolů.
- $\alpha_1 = Q$ značí to samé, co P, akorát že tyto stroje mají navíc nějakou vlastní rychlost, kterou mohou zrychlit nebo zpomalit zpracování úkolu. Tedy čas, za který je úkol zpracován, se dělí ještě rychlostí strojů.
- $\alpha_1 = R$ znamená to samé, co Q, akorát rychlost strojů se může pro každou operaci lišit.
- $\alpha_1 = PMPM$. První písmeno P znamená to samé, co samotné $\alpha = P$. MPM – z anglického „Multi-Purpose Machines“ – značí, že stroje mohou zvládat více různých druhů úkolů.
- $\alpha_1 = QMPM$ značí to samé, jako $PMPM$, akorát tyto stroje mají – jak už nám Q napovídá – svoje vlastní rychlosti zpracování.
- $\alpha_1 = G$ v případě, že se jedná o tzv. general shop.
- $\alpha_1 = X$ v případě, že se jedná o tzv. mixed shop.
- $\alpha_1 = O$ v případě, že se jedná o tzv. open shop.

- $\alpha_1 = J$ v případě, že se jedná o tzv. job shop.
- $\alpha_1 = F$ v případě, že se jedná o tzv. flow shop.

α_2 udává počet strojů. Tedy pokud $\alpha_2 \in \mathbb{N}$, pak máme k dispozici právě α_2 strojů. Jestliže $\alpha_2 = k$, pak je počet strojů libovolný, ale fixně daný. Pokud je α_2 vynechána, pak je počet strojů libovolný.

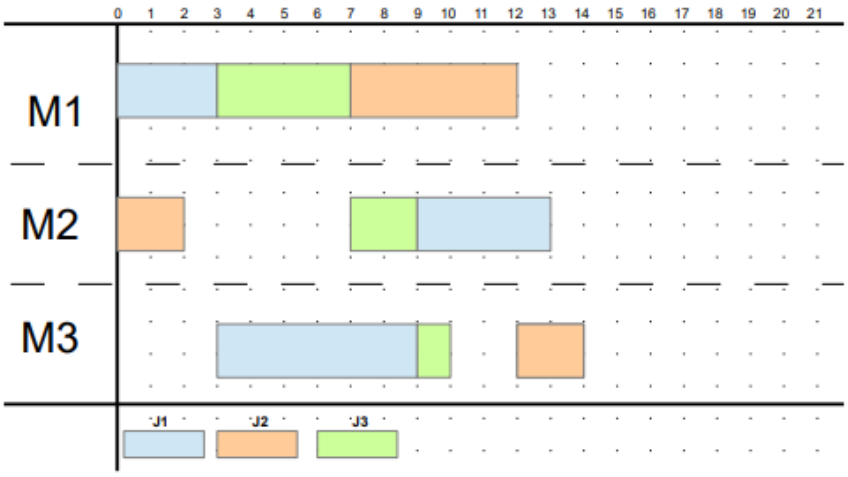
1.0.2.2 Shopy

Hana Rudová [2] popisuje shopy jako multi-operačními problémy, kde se každý úkol skládá z více operací a každá z těchto operací může vyžadovat jiný stroj. Takto obecný a nejjednodušší příklad se v [1] nazýva „general shop“ a ten se dále rozděluje na různé specifické případy: job shop, open shop, mixed shop a flow shop.

Job shop je v [1] popisován jako multi-operační problém s více stroji, kde úkoly mají na každém stroji vykonat právě jednu operaci (podle [2] může být i nulová) v předem určeném pořadí. Pokud je povoleno stroje opakovat, jedná se o speciální případ job shopu s povoleným opakováním. V tabulce 1.1 je příklad job shop problému a na obrázku 1.2 jeho možné řešení vyjádřené za pomoci Ganttova diagramu.

Úkol	Operace ve tvaru požadovaný stroj(doba trvání)		
J_1	$M_1(3)$	$M_3(6)$	$M_2(4)$
J_2	$M_2(2)$	$M_1(5)$	$M_3(2)$
J_3	$M_1(4)$	$M_2(2)$	$M_3(1)$

Tabulka 1.1: Příklad job shop problému

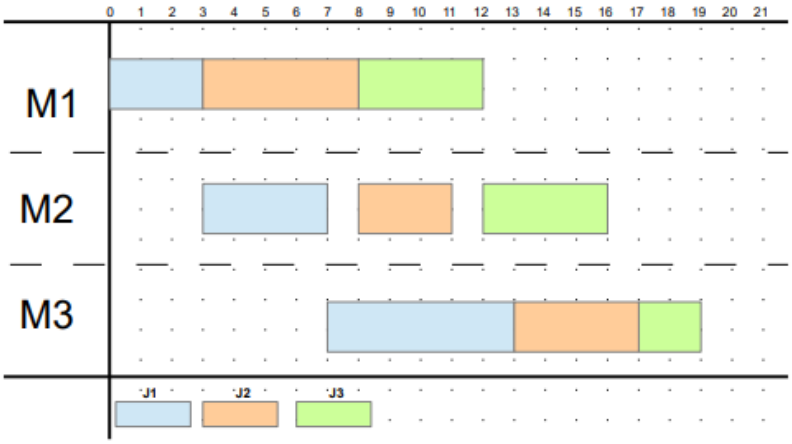


Obrázek 1.2: Příklad job shop problému

Flow shop je podle [1] a [2] speciálním případem job shop problému. Úkoly jsou zpracovávány postupně na všech strojích od 1. po m. ve fixním – pro všechny úkoly stejném – pořadí. Existuje i flexibilní varianta kde existují paralerní stroje vykonávající stejnou funkci. V takovém případě stačí, aby byla operace úkolu zpracována jen na jednom z těchto paralerních strojů. Níže je k dispozici tabulka 1.2 s příkladem flow shop problému a obrázek 1.3 možného řešení vyjádřeného Ganttovým diagramem.

Úkol	Operace ve tvaru požadovaný stroj(doba trvání)		
J_1	$M_1(3)$	$M_2(4)$	$M_3(6)$
J_2	$M_1(5)$	$M_2(3)$	$M_3(4)$
J_3	$M_1(4)$	$M_2(4)$	$M_3(2)$

Tabulka 1.2: Příklad flow shop problému

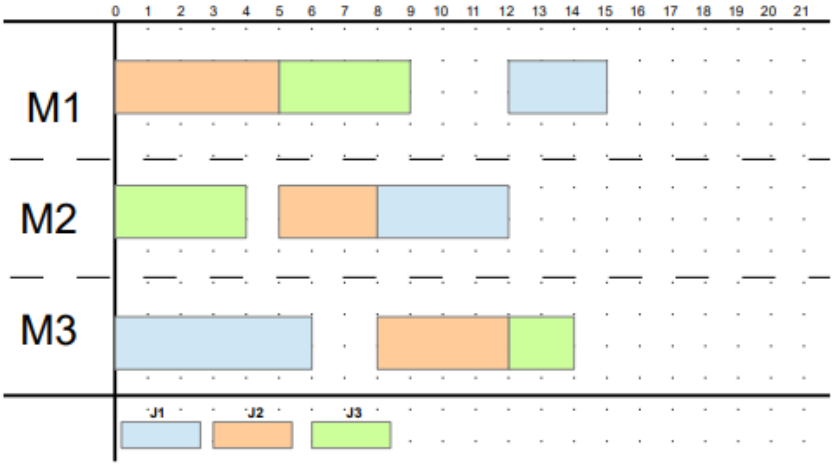


Obrázek 1.3: Příklad flow shop problému

Open shop je v [1] definován jako flow shop, akorát odpadají závislosti na ostatních operacích. Podle [2] si toto pořadí může určit rozvrhovač libovolně. A mixed shop je podle [1] kombinací open, flow a job shopu. Níže je tabulka 1.3 a Ganttův diagram 1.4 popisující a řešící příklad open shop problému.

Úkol	Operace ve tvaru požadovaný stroj(doba trvání)		
J_1	$M_1(3)$	$M_2(4)$	$M_3(6)$
J_2	$M_1(5)$	$M_2(3)$	$M_3(4)$
J_3	$M_1(4)$	$M_2(4)$	$M_3(2)$

Tabulka 1.3: Příklad open shop problému



Obrázek 1.4: Příklad open shop problému

1.0.2.3 Charakteristika úkolů

Brucker [1] charakterizuje úkoly jako specifickou množinou β obsahující maximálně šest prvků β_1, β_2, β_3, β_4, β_5 a β_6. Kterýkoliv z těchto prvků může být z množiny β vynechán.

$\beta_1 = pmtn$[1] znamená, že je povoleno přerušení zpracování úkolu. Zpracování úkolu může být tedy pozastaveno a znovu obnoveno na místě, kde skončilo. A to klidně na jiném stroji a opakovaně.

$\beta_2 = prec$[2] znamená, že úkoly mají mezi sebou závislosti, které se dají znázornit acyklickým orientovaným grafem s úkoly jako vrcholy a závislotmi jako hranami. Pokud se závislosti dají vyjádřit nějakým speciálním typem grafu, tak se tato skutečnost může zapsat také jako $\beta_2 = tree$, $\beta_2 = chain\ldots$

$\beta_3 = r_i$ značí, že se některé úkoly nemohou začít zpracovávat před určitým časem pro něj určeným.

β_4 udává restrikce týkající se času zpracování každého úkolu. Např. pokud je čas zpracování všech úkolú stejný, v β_4 se to označí jako $p_i = p$.

$\beta_5 = d_i$ značí, že každý úkol může mít svůj deadline, do kterého musí být dokončen.

β_6 se přidává, pokud potřebujeme úkoly zpracovávat po várkách[3]. Tyto várky se mohou chovat při zpracování různě a mohou obsahovat 1 až n úkolů. Buď se tato várka zpracuje za součet času zpracování jednotlivých úkolů v ní obsažených, pak $\beta_6 = s\text{-}batch$ a jedná se o tzv. „s-batching problem“. A nebo se celá várka zpracuje za čas jednoho (toho nejpomaleji zpracovaného) úkolu, pak se jedná o tzv. „p-batching problem“ a $\beta_6 = p\text{-}batch$.

1.0.2.4 Charakteristika cílů

Cíl je v oblasti plánování a rozvrhování popsán funkcí a tato funkce se nazývá funkce cíle. Při hledání optimálního řešení rozvrhovací úlohy jde zpravidla o to najít takové řešení, které danou funkci cíle minimalizuje. Pokud tedy v úloze existuje taková funkce cíle f, pak $\gamma = f$. V následujících řádcích je uvedeno pár příkladů funkcí cíle podle [1].

Označme čas dokončení úkolu J_i jako C_i, váhu úkolu jako v_i a deadline jako d_i. Nejběžnějším je asi funkce hledící na čas dokončení poslední úlohy

$$max\{Ci|i = 1, \ldots, n\}$$

celkový čas

$$\sum_{i=1}^{n} C_i$$

[1] zkratka z anglického „preemption“

[2] z anglického „precedence relations“

[3] v anglických zdrojích "batch

a nebo vážený celkový čas

$$\sum_{i=1}^{n} v_i C_i$$

Dalším příkladem může být funkce hledící na největší zpoždění

$$max\{C_i - d_i\}$$

a nebo počet zpožděných úkolů

$$\sum_{i=1}^{n} U_i \text{ kde } U_i = \begin{cases} 0, & C_i \leq d_i \\ 1, & \text{jinak} \end{cases}$$

Kapitola 2

Přístupy k řešení

Plánování je velmi komplexní problém, na jehož řešení a optimalizacích se dodnes pracuje. Existují ale dva hlavní přístupy, které se osvědčili: Constraint programming a Linear programming. Oba se hojně využívají a tak i ve frameworcích v této práci zastoupených jsou zahrnuty oba z nich.

2.0.1 Constraint programming

Constraint programming, zkráceně také CP, je do češtiny často překládán jako „Programování s omezujícími podmínkami“. Jedná se o způsob hledání vhodného (ale ne vždy nejlepšího) řešení, kde je problém popsán pomocí omezujících podmínek, které musí řešení splňovat. CP zvládá najít vhodná řešení i mezi obrovským množstvím možných kandidátů. [3]

Problémy řešené CP se nazývají Constraint Satisfaction Problems (CSP). V [4] je CSP definováno následovně:

„*Mějme množinu proměnných* $Y = \{y_1, \ldots, y_k\}$ *a konečnou množinu hodnot (doménu)* $D = D_1 \cup \ldots \cup D_k$. *Omezující podmínka c definována na* Y *je podmnožinou* $D_1 \times \ldots \times D_k$ *(tj. vztah) a omezuje hodnoty, kterých mohou proměnné nabývat v jeden čas.*“ a dále:

„*Mějme konečnou množinu proměnných:* $X = \{x_1, \ldots, x_n\}$, *konečnou množinu hodnot* $D = D_1 \cup \ldots \cup D_n$ *a konečnou množinu omezujících podmínek* $C = \{c_1, \ldots, c_m\}$ *(každá z nich definovaná na podmnožině z* X*). Pak CSP je definováno jako trojice* (X, D, C).“

Příklad CSP:

- proměnné: A, B, C
- hodnoty: $\{1, 2, 3, 4\}$ pro A, $\{2, 3, 4\}$ pro B a $\{0, 2, 4\}$ pro C
- podmínky: $A < B$, $A \neq C$

Řešení CSP je přiřazení hodnot všem proměnným tak, aby byly splněny všechny podmínky, tedy například: $A = 1$, $B = 2$ a $C = 4$.

K CSP může být přidána ještě funkce f definovaná na proměnných z X. Pak se jedná o optimalizační problém, kde se snažíme f minimalizovat. Přidejme si k předešlému příkladu funkci $f = C - A$. Pak $A = 1, B = 2, C = 4$ je sice správným, ale již ne optimálním řešením. Tím bude $A = 3$, $B = 4$ a $C = 0$.

Podle [5] se CSP řeší systematickým procházením možných řešení. To může znamenat procházení prostoru částečných a nebo náhodné prohledávání prostoru celých řešení. Problém se tedy převádí do grafů a ty se pak pomocí velmi chytrých algoritmů a heuristik prohledávají.

2.0.2 Linear programming

LP přístup pohlíží na plánování a rozvrhování jako na čistě matematický problém. A jako matematický problém jej také řeší. Zaměřuje se na nalezení nejlepšího řešení problému podaného jako sada linearních rovnic. [6]

Některé frameworky, jako např. CPLEX[4], používají k řešení tzv. simplexovou metodu. Tato metoda je hojně v LP využívaná. V následujících řádcích si vysvětlíme základní principy této metody podle [7].

Simplexová metoda spočívá v přeložení problému do nerovnic o několika neznámých a funkce f. Pro tyto nerovnice se hledá řešení, které maximalizuje hodnotu f. V plánovacích problémech potřebujeme často řešení minimalizovat. Převod mezi minimalizačním a maximalizačním problémem je ale jednoduchý:

$$min(f(x)) = -max(-f(x))$$

Simplexova metoda bude vysvětlena na příkladu. Mějme:

- $f = 8x_1 + 7x_2$
- $3x_1 \leq 6$
- $x_1 + 2x_2 \leq 20$
- $x_i \geq 0$

[4] ten byl podle simplexové metody dokonce pojmenován

Tento problém si nejdříve musíme převést z nerovnic na rovnice.

- $f = 8x_1 + 7x_2$
- $3x_1 + pom_1 = 6$
- $x_1 + 2x_2 + pom_2 = 20$
- $x_i \geq 0$, $pom_i \geq 0$

Pak ho přepíšeme do simplexové tabulky, kde sloupce odpovídají koeficientům jednotlivých proměnných a řádky odpovídají jednotlivým rovnicím. Do posledního řádku přijde naše funkce, kterou se snažíme maximalizovat, a její koeficienty vynásobené -1.

x_1	x_2	pom_1	pom_2	b
3	0	1	0	6
1	2	0	1	20
-8	-7	0	0	0

V posledním řádku sloupce b je hodnota funkce za předpokladu, že jsou všechny nebázové proměnné nulové (ty, jež nejsou součástí jednotkové matice).

Nyní si zvolíme sloupec, na kterém celá naše funkce závisí nejvíce. A to je ten, který má nejvyšší multiplikativní koeficient v naší funkci. V simplexové tabulce to je ten, který má v posledním řádku nejnižší záporné číslo. Záporné proto, že jakmile nejnižší číslo posledního řádku není záporné, tak jsme skončili a hodnota f je maximální možná. V našem případě si tedy vybereme sloupec patřící k x_1, který má koeficient ve funkci 8 a v posledním řádku -8.

Z tohoto sloupce si potřebujeme vybrat pivotní hodnotu. Tato hodnota se vybírá jako ta, která je kladná a zároveň dává nejmenší podíl s hodnotou v b sloupci. Tedy $min\{\frac{b_i}{a_{i,j}}, a_{ij} > 0\}$.V našem případě to je řádek 1. s hodnotou 3, protože $\frac{6}{3} = 2 < \frac{20}{2} = 10$. V případě, že bychom našli dvě hodnoty se stejným podílem, vybrat si můžeme libovolnou z nich.

Nyní přichází čas na elementární řádkové operace, pomocí nichž můžeme tabulku převést na tvar pro nás užitečný, aniž bychom změnili množinu řešení soustavy. Mezi elementární operace patří vynásobení řádku nenulovou konstantou a přičtení řádku vynásobeného libovolnou konstantou k jinému řádku. Tyto operace provedeme tak, aby námi vybraná pivotní hodnota vyšla 1 a celý zbytek vybraného sloupce byl nulový. Proměnná daného sloupce se tak stane bázovou.

x_1	x_2	pom_1	pom_2	b
1	0	1/3	0	2
0	1	-2/3	1	16
0	-7	8/3	0	16

První řádek jsme vynásobili $\frac{1}{3}$. K druhému řádku jsme přičetli $-\frac{2}{3}$ násobek řádku prvního. A ke třetímu řádku jsme přičetli $\frac{8}{3}$ násobek prvního řádku.

Řešení vypadá zatím takto:
$x_1 = 2$, $x_2 = 0$, $pom_1 = 0$, $pom_2 = 0$ a $f = 16$.
Tyto hodnoty se vyčtou z posledního řádku ze sloupce, kde se vyskytuje 1 dané bázové proměnné. Všechny ostatní – nebázové – proměnné jsou rovny 0. Toto řešení ale není ideální.

Celý proces vybírání sloupce, řádku a úpravy tabulky pomocí elementárních operací opakujeme, dokud v posledním řádku zbývají nějaké záporné hodnoty. Vybereme si sloupec s nejnižším číslem v posledním řádku – sloupec x_2 s hodnotou -7. V něm si vybereme pivota – řádek 2. s hodnotou 1 je jediný > 0. A přičteme ke třetímu řádku 7 násobek druhého řádku.

x_1	x_2	pom_1	pom_2	b
1	0	1/3	0	2
0	1	-2/3	1	16
0	0	-2	7	128

$x_1 = 2$, $x_2 = 16$, $pom_1 = 0$, $pom_2 = 0$ a $f = 128$.

Sloupec s nejnižší hodnotou je sloupec pom_1 s hodnotou -2. Za pivota si vybereme řádek 1. s hodnotou $\frac{1}{3}$ protože je jako jediná v daném sloupci > 0. První řádek vynásobíme 3. K druhému řádku přičteme 2 násobek prvního řádku. A k třetímu řádku přičteme 6 násobek řádku prvního.

x_1	x_2	pom_1	pom_2	b
3	0	1	0	6
2	1	0	1	20
6	0	0	7	140

$x_1 = 0$, $x_2 = 20$, $pom_1 = 6$, $pom_2 = 0$ a $f = 140$.

Žádná záporná hodnota posledního řádku indikuje, že jsme došli nakonec a řešení nemůže být lepší. Bázovými proměnnými jsou tedy x_2 a pom_1. x_1

a pom_2 jsou nulové. $x_2 = 20$ a $pom_1 = 6$.
$f = 8x_1 + 7x_2 = 8 \times 0 + 7 \times 20 = 140$ je maximální možné řešení.

Kapitola 3

Příklad rozvrhovacího problému

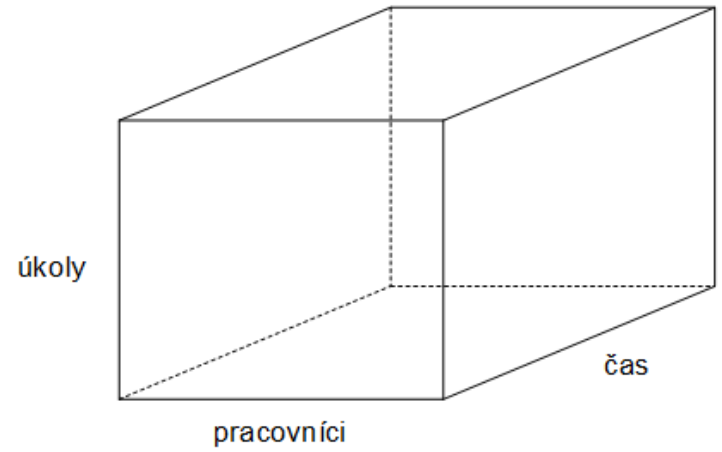
Po teorii získané v předchozích kapitolách může tato práce přejít k popisu konkrétního příkladu rozvrhovacího problému.

Mějme T úkolů každý se svým časem trvání a W pracovníků. Každý z těchto úkolů vyžaduje jednu konkrétní dovednost a každý z těchto pracovníků právě jednu dovednost ovládá. Úloha spočívá v tom, přiřadit každému úkolu pracovníka s danou dovedností a zpracovávání úkolů rozvrhnout do určitého časového intervalu. Úkoly mohou být navzájem na sobě závislé a každý z nich může požadovat více pracovníků se stejnou dovedností. Tito pracovníci budou na daném úkolu pracovat paralelně v jeden čas. Jeden pracovník nemůže pracovat na více než jednom úkolu najednou. Přerušení úkolů není povoleno. Cílem je minimalizovat celkový čas potřebný na dokončení všech úkolů. Pracovníci se dají v tomto problému chápat jako stroje a celý problém se pomocí Grahamovy klasifikace zapíše následovně:

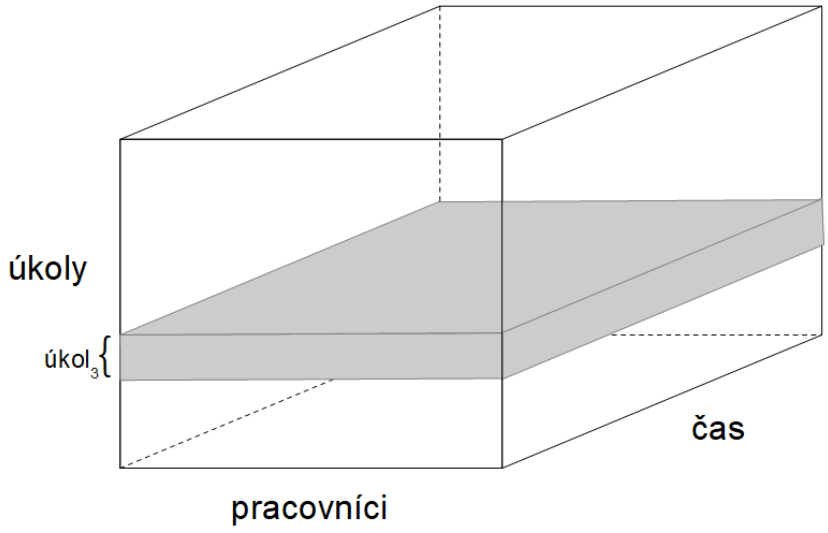
$$k|\text{prec}; d_j = d|\max\{Ci|i = 1, \ldots, n\}$$

3.0.1 Řešení

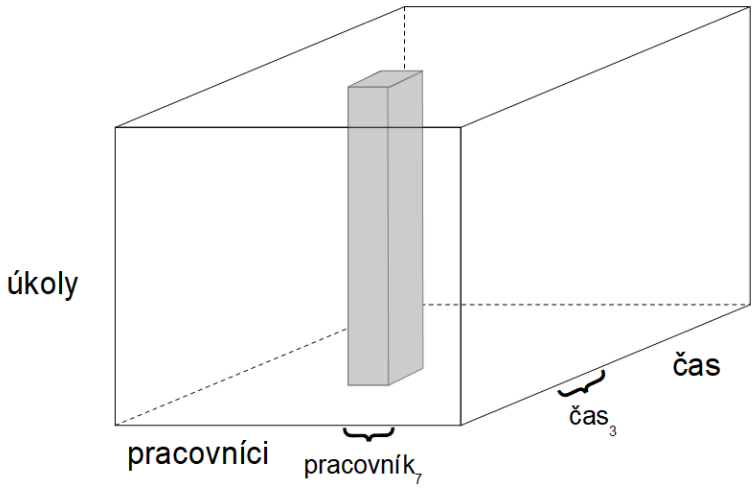
Prostor řešení může být reprezentován jako 3D pole s časem v jednom rozměru, pracovníky v druhém a úkoly ve třetím (viz. obrázek 3.1). Prvek tohoto pole může obsahovat 1 nebo 0 podle toho, zdali daný pracovník v daný čas pracuje na daném úkolu či nikoliv. Celkový čas strávený všemi pracovníky na jednom úkolu je pak 2D polem (viz. obrázek 3.2) a podle zadání se součet všech jeho hodnot musí rovnat: (počet potřebných pracovníků) × (trvání úkolu). Druhou omezující podmínkou je daný počet úkolů, ke kterým může být pracovník přiřazen v jeden čas. Ten se dá znázornit 1D polem (vic. obrázek 3.3) a součet jeho hodnot musí být roven 1.



Obrázek 3.1: Řešení jako 3D pole



Obrázek 3.2: Čas strávený na jednom úkolu



Obrázek 3.3: Úkoly pracovníka v jednom čase

Tento model se dá vylepšit uvědomíme-li si, že úkoly nemohou být přerušovány. Tudíž není potřeba si v modelu udržovat přiřazení úkolu pracovníkům pro každý jeden bod v čase, ale stačí nám znalost začátku a doby trvání úkolu. S tím přicházejí itervaly, které se mohou spojit s úkolem do jednoho objektu. Každý úkol má mít totiž přiřazen právě jeden interval, ve kterém bude zpracováván. Interval je objekt o třech proměnných: začátek, konec a trvání. Pro tyto tři intervalové proměnné musí platit, že začátek + trvání = konec. Prostor řešení se tedy zmenší na jednoduché 2D pole s úkolo-intervaly v dimenzi jedné a pracovníky v dimenzi druhé.

Přiřazení pracovníka s potřebnou dovedností k úkolu, který tuto dovednost požaduje, se v tomto případě, kdy každý pracovník ovládá právě jednu dovednost a úkol právě jednu dovednost požaduje, vyřeší rozdělením prostoru řešení pro každou dovednost zvlášť. Každá dovednost bude mít své vlastní pole ve kterém bude přiřazovat své pracovníky svým úkolům.

S tímto návrhem už zbývají jen tři omezující podmínky a optimalizační cíl, které je potřeba implementovat:

1. Na každém úkolu pracuje přesně daný počet pracovníků.

2. Pracovník nemůže být přiřazen ke dvěma úkolům v jeden čas.

3. Vztahy mezi úkoly musí být dodrženy. Tedy pokud je to vyžadováno, daný úkol může začít až po konci (nebo začátku) úkolu druhého.

4. Čas dokončení posledního úkolu má být nejnižší možný.

Máme-li v našem poli ve sloupcích jednotlivé úkolo-intervaly, v řádcích jednotlivé pracovníky a hodnoty zůstávají 1 a 0 podle toho, zdali je daný pracovník přiřazen k danému úkolu. Pak pro splnění první omezující podmínky musí platit, že součet hodnot v každém sloupci je roven počtu požadovaných pracovníků.

Pro popsání druhé omezující podmínky je potřeba projít si řádky našeho pole. Pokud je na jednom řádku více hodnot rovno 1, tak se jejich intervaly nesmí překrývat. Dva intervaly i_1 a i_2 se nepřekrývají, pokud $i_1.začátek \geq i_2.konec$ a nebo $i_1.konec \leq i_2.začátek$.

U některých knihoven je navíc k dispozici funkce Cumulative, pomocí níž se dá tato omezující podmínka popsat taktéž. Cumulative je metoda, která pro danou množinu intervalů, množinu požadavků a danou kapacitu vytvoří omezující podmínku, která zabraňuje, aby se v jednom čase zpracovávalo více požadavků, než je povolená kapacita. V tomto případě se může tato metoda zavolat pro každého pracovníka. Předájí se jí všechny intervaly všech úkolů dané dovednosti, řádky pole reprezentující přiřazení daného pracovníka k jednotlivým úkolům (pořadí musí odpovídat pořadí intervalů z předchozího parametru) a kapacita o velikosti 1.

Třetí omezující podmínka je jednoduchá. Stačí si projít všechny úkoly, a pokud některý z nich má začít až po konci druhého úkolu, pak jeho začátek $\geq$ konec úkolu druhého. A pokud má začít až po začátku úkolu druhého, pak jeho začátek $\geq$ začátek úkolu druhého.

Pro popsání optimalizačního cíle potřebujeme znát konec posledního úkolu. Tedy je potřeba zavést ještě jednu proměnnou, pro kterou bude platit omezující podmínka, že pro všechny intervaly všech úkolů je její hodnota $\geq$ než hodnota konce tohoto intervalu. A tato proměnná se bude v konečném výsledku minimalizovat.

3.0.2 Implementace

Náhled do implementace daného problému bude demonstrován s využitím knihovny OR-Tools CP-SAT.

Vytvoření modelu:

```
CpModel model = new CpModel();
```

Přidání proměnných do modelu:

```
for (int y = 0; y < numberOfTasks; y++) {
    TaskInterval taskInterval =
        taskWorkerPool.get(skill).tasks.get(y); // pro kazdy ukol
        dane dovednosti
    taskInterval.start =
        model.newIntVar(station.getStartTime().intValue(),
        station.getEndTime().intValue(), "start_" +
        taskInterval.task.getName()); // prida do modelu dve
        celociselne promenne, ktere se mohou pohybovat v rozsahu
        uplneho zacatku prace na uloze az uplneho konce. Vyjadrovat
        budou zacatek a konec prace na danem ukolu.
    taskInterval.end =
        model.newIntVar(station.getStartTime().intValue(),
        station.getEndTime().intValue(), "end_" +
        taskInterval.task.getName());

    for (int x = 0; x < numberOfWorkers; x++) {
        taskInterval.workers.add(model.newBoolVar(taskInterval.task.getName()
            + "_worker_" + x)); // inicializuje pole pracovniku pro
            kazdy ukol a jeho prvky prida do modelu. V tomto poli jsou
            booleanovske promenne (tedy promenne nabyvajici hodnoty 0
            nebo 1).
    }
}
```

První omezující podmínka se přidá do modelu pomocí následujícího kódu:

```
for (int y = 0; y < numberOfTasks; y++) {
    TaskInterval theTask = taskWorkerPool.get(skill).tasks.get(y);
    // pro kazdy ukol dane dovednosti
    long workersNeeded =
        theTask.task.getSkill().get(0).getCount().intValue();
    // se zjisti pocet potrebnych pracovniku

    IntVar[] workersArray = new IntVar[theTask.workers.size()];
    // zde se jen prevadi List na pole
    workersArray = theTask.workers.toArray(workersArray);
    model.addLinearSumEqual(workersArray, workersNeeded);
    // prida do modelu omezujici podminku, kde soucet vsech prvku
        pole workersArray se musi rovnat workersNeeded
}
```

Druhá omezující podmínka řešená pomocí Cumulative:

```
IntVar one = model.newConstant(1);
for (int w = 0; w < numberOfWorkers; w++) {
// pro kazdeho pracovnika zvlast
    IntervalVar[] workersTasks = new IntervalVar[numberOfTasks];
    // vytvori pole vsech intervalu dane dvednosti
    IntVar[] demand = new IntVar[numberOfTasks]; // vytvori pole,
        ve kterem budou 1 a 0 podle toho, zdali dany pracovnik
        v intervalu danem predchozim polem na stejnem indexu pracuje,
        nebo nikoliv
    for (int y = 0; y < numberOfTasks; y++) { // projde vsechny ukoly
        dane dovednosti
        TaskInterval theTask = taskWorkerPool.get(skill).tasks.get(y);
        IntervalVar interval =
            model.newOptionalIntervalVar(theTask.start,
            theTask.task.getDuration().intValue(), theTask.end,
            theTask.workers.get(w), theTask.task.getName() + "_" + w);
            // prida do modelu promennou typu vylepseneho intervalu,
            ktery se bere v uvahu pri vypoctech cumulative jen a pouze
            tehdy, pokud je predposledni argument
            (theTask.workers.get(w)) roven 1.
        workersTasks[y] = interval;
        demand[y] = theTask.workers.get(w);
    }
model.addCumulative(workersTasks, demand, one); // samotne pridani
    omezujici podminky pomoci cumulative
}
```

Třetí omezující podmínka starající se o vztahy mezi úkoly:

```
for (TaskInterval task : allTaskIntervalsList) { // pro vsechny ukoly
    if (task.task.getTaskRelations().getTaskRelation() != null) {
    // pokud jsou zavisle na nejakem jinem ukolu
        for (TaskRelation relation :
            task.task.getTaskRelations().getTaskRelation()) {
        // tak pro vsechny tyto zavislosti

          String type = relation.getRelType();
          switch (type) {
              case "startAfterEnd": { // pokud se jedna o zavislost
                  typu zacni az dany ukol skonci
                  TaskInterval other =
                      nameToTask.get(relation.getName());
                  if (other != null) {
                      model.addLessOrEqual(other.end, task.start);
                      // pak prida do modelu omezujici podminku, kde
                          druhy interval nejdriv skonci, nez tento
                          zacina
                  }
              }
                  break;
              case "startAfterStart": { // pokud se jedna
                  o zavislost typu zacni az dany ukol zacne
                  TaskInterval other =
                      nameToTask.get(relation.getName());
                  if (other != null) {
                      model.addLessOrEqual(other.start, task.start);
                          // pak prida do modelu omezujici podminku,
                          kde druhy ukol nejdriv zacne, nez zacne
                          tento
                  }
              }
                  break;
          }
        }
    }
}
```

Přidání optimalizačního cíle do modelu:

```
IntVar maximumEnd =
    model.newIntVar(station.getStartTime().intValue(),
    station.getEndTime().intValue(), "theEnd"); // prida do modelu
    novou promennu
IntVar[] allEndsArray = new IntVar[allTaskIntervalsList.size()];
for (int i = 0; i < allTaskIntervalsList.size(); i++) { // vezme
    vsechny intervaly vsech ukolu, ktere jsou v modelu
  allEndsArray[i] = allTaskIntervalsList.get(i).end;
}
model.addMaxEquality(maximumEnd, allEndsArray); // a prida omezujici
    podminku, kde maximumEnd musi byt vzdy vetsi nebo roven vsem
    prvkum v poli vsech intervalu modelu
model.minimize(maximumEnd); // prida do modelu optimalizacni
    kriterium, kde maxEnd ma byt nejmensi mozny
```

Řešení modelu:

```
CpSolver solver = new CpSolver(); // vytvori tridu starajici se
    o hledani reseni
solver.solve(model); // a tuto tridu pozada o vyreseni daneho modelu
```

Kapitola 4

Frameworky pro řešení rozvrhovacích úloh

4.0.1 CPLEX

IBM ILOG CPLEX, zkráceně často též CPLEX, je softwarový balíček pojmenovaný podle jazyka C, ve kterém je napsán, a simplexové metody, kterou při svých výpočtech využívá. Zabývá se především řešením problémů pomocí LP modelovaných jako sada lineárních rovnic či nerovnic s funkcí, kterou se snaží minimalizovat nebo maximalizovat. CPLEX dokáže řešit i rozšířené problémy LP, jako je třeba Quadratic Programming (QP), kde funkce může být i kvadratická, Quadratically Constraint Programming (QCP), kde sada nerovnic obsahuje kvadratické prvky. Dokáže řešit Second Order Cone Programming problémy (SOCP), což jsou konvexní optimalizační problémy, kde jsou rovnice i funkce o něco složitější. Mixed Integer Programming (MIP) problémy, kde některé nebo všechny výsledky optimalizace LP, QP nebo QCP musí být celočíselné a kde MIP samo může obsahovat prvky Special Order Sets (SOS), kde se určuje množina proměnných, ze kterých ve výsledku jen jedna může být nenulová. Nebo třeba Network Flow problémy. [8]

CPLEX je vyvíjen společností IBM. Jedná se o komerční software dostupný jako součást balíčku IBM ILOG CPLEX Optimization Studio. IBM ale nabízí neomezené užívání tohoto softwaru zadarmo pro studenty a učitele a to přes Academic Initiative. Také nabízí zdarma balíčky s omezenou funkcionalitou. Licence komerčních balíčků se podle [9] pohybují cenově od $199/developer. K použití na Windows je potřeba k plné funkčnosti Microsoft Visual C++.

4.0.1.1 Dokumentace a Uživatelská podpora

Dokumentace CPLEXu sestává z několika ukázkových kódů, které jsou součástí instalace, tutoriálu [10] a Java reference manuálu [11]. K dispozici je i pár videí demonstrujících použití CPLEXu [12]. Uživatelská podpora je zajištěna

pomocí speciálního chatu a telefonní linky, kde je možnost vyřešit jakýkoliv problém s pracovníky IBM.

4.0.1.2 Instalace a použití

CPLEX je součástí IBM ILOG CPLEX Optimization Studia, což je rozsáhlý nástroj pro vývoj a řešení optimalizačních problémů. Instalace může být tedy o něco složitější, než pouhé uvedení knihovny v konfiguračním souboru. Po nainstalování celého softwaru je potřeba přidat celou složku obsahující cplex.exe mezi knihovny projektu a cplex.jar soubor do classpath.

Samotné použití knihovny může být o něco složitější. CPLEX postrádá typ intervalu a funkcí s intervaly spojenými. Pro použití u rozvrhovacích úloh si je tedy musí uživatel napsat sám.

Hlavní třídou knihovny CPLEX je IloCplex, jejíž instance obsahují všechny potřebné metody pro popsání a vyřešení problému. Ten se popisuje pomocí proměnných a vztahů mezi nimi. Typů proměnných je velké množství, hlavními jsou ale IloNumVar (číselná proměnná) a IloNumExpr (číselný výraz). Od těchto dvou typů dědí další jako například IloIntVar (celočíselná proměnná), IloIntVarExpr (výraz celočíselných proměnných), IloRange (číselný výraz s omezením zdola a shora), IloAnd (výraz logického and) nebo IloOr (výraz logickéh or).

IloCplex obsahuje také spoustu metod vhodných pro popsání problému, mezi než patří například:

- intVar(a, b, jméno) – vytvoří a přidá do modelu novou celočíselnou proměnnu, která může nabývat hodnot z intervalu $< a, b >$.
- boolVar() – vytvoří a přidá do modelu novou celočíselnou proměnnou IloIntVar, která může nabývat hodnot z intervalu $< 0, 1 >$
- diff(a, b) – vrací IloNumExpr vyjadřující proměnnou $c = a - b$.
- sum(pole) – vrací IloNumExpr nebo IloIntExpr vyjadřující proměnnou $a = \sum pole[i]$
- max(pole) – vrací IloNumExpr nebo IloIntExpr vyjadřující proměnnou a, která je rovna maximálnímu prvku v poli
- and(a, b) – vrací IloAnd vyjadřující proměnnou $c = a \wedge b$.
- or(a, b) – vrací IloOr vyjadřující proměnnou $c = a \vee b$.
- eq(a, b) – vrací IloRange vyjadřující proměnnou a s omezením shora i zdola hodnotou b. Popřípadě IloConstraint, není-li ani jeden z parametrů primitivního typu, vyjadřující omezení $a = b$.
- not(c) – vrací IloConstraint vyjadřující negaci c.

- le(a, b) – vrací IloRange nebo IloConstraint omezující hodnotu a shora hodnotou b.
- ifThen(a, b) – vrací IloConstraint vyjadřující $a \Rightarrow b$, kde a a b jsou omezující podmínky.
- add(c) – přidá do modelu danou omezující podmínku c.
- addEq(a, b) – přidá do modelu omezující podmínku $a = b$.
- addLe(a, b) – přidá do modelu omezující podmínku $a \leq b$.
- addMinimaze(a) – přidá do modelu optimalizační kritérium, tedy že hodnota a má být minimální možná.

4.0.2 Choco

Choco Solver 4, dále jen Choco, je open source knihovna zaměřená na řešení CP problémů. Napsán je v Javě a vydáván je pod BSD licencí. [13]

BSD licence je jedna z nejvíce benevolentních licencí, kde je podle [14] povoleno použití, úprava i redistribuce kódu bez nějakých větších omezení. Stačí jen, aby případná redistribuce obsahovala požadovaný text BSD licence a aby se nepoužívalo jméno autorů projektu pro propagaci pojektů odvozených bez předchozího písemného souhlasu.

Choco podporuje několik typů proměnných včetně reálných. Obsahuje předdefinované omezující podmínky, nastavitelné vyhledávací funkce a funkce pro řešení konfliktů. Obsahuje nástroje umožňující interakci s vyhledávacími procesy a nástroje pomáhající s modelováním problémů. Choco podporuje přidávání mnoha rozšíření, jako např. podpora FlatZinc modelovacího jazyka pomocí FlatZinc parser rozšíření. [15]

4.0.2.1 Dokumentace a Uživatelská podpora

Dokumentace Choco se skládá z přehledného uživatelskéh manuálu [15], tutoriálů [16] a javadoc [17]. Vše je přehledně dostupné hned z úvodní stránky oficiálního webu [13]. Samozřejmostí je také spousta ukázkových kódů.

Uživatelská podpora sestává z Twitterového účtu, Gitter chatu a Google groups mailing listu. Otázky na Gitteru a Google groups bývají zodpovězeny během několika dní.

4.0.2.2 Instalace a použití

Choco podporuje Maven a jeho alternativu SBT. Instalace je tedy jednoduchá a sestává pouze z upravení dependencies v pom.xml (respektivě build.sbt pro SBT) souboru projektu. K projektu se dá připojit i ručně, stačí uvést příslušný balíček v classpath.

Použití pak sestává z popisu problému v třídě Model a zavolání třídy Solver k jeho vyřešení. Samotné popisování probíhá za pomoci proměnných a omezujících podmínek. Typů proměnných je k dispozici hned několik: celočíselný typ proměnné IntVar, reálné proměnné s typem RealVar, booleanovské proměnné BoolVar a množiny SetVar.

Model obsahuje mnoho metod pro popsání problémů, jako například:

- intVar(jméno, od, do) – přidá do modelu celočíselnou proměnnou, která může nabývat hodnot z intervalu <od, do> včetně.
- intOffsetView(a, offset) – přidá do modelu celočíselnou proměnnou, která je rovna a + offset.
- boolVar() – přidá do modelu novou booleanovskou proměnnou
- taskVar(začátek, trvání) – přidá do modelu tři proměnné start = začátek, duration = trvání a end = start + duration.
- sum(pole, -", hodnota) – omezující podmínka, kde součet prvků pole = daná hodnota. Operátor -"může být nahrazen jiným operátorem jako je např. "≤"nebo »".
- arithm(a, "!=", b) – omezující podmínka, kde a je různé od b. Operátor "!="může být nahrazen, stejně jako u metody sum, jinými operátory.
- cumulative(pole_task, pole_cen, kapacita) – omezující podmínka, kde pole cen reprezentuje pod každým ze svých indexů cenu za výkonávání dané tasky. Kapacita udává maximální možnou cenu, která může být vykonávána v jeden čas. Brání tedy překrývaní intervalů task s nastavitelnou kapacitou.
- max(max, pole) – omezující podmínka, kde max musí nabýt hodnoty maxima z daného pole.
- setObjective(false, maximumEnd) – přidá do modelu optimalizační kritérium. Prvním parametr udává, zdali se jedná o maximalizaci/minimalizaci (true/false). A druhým parametrem je proměnná, která se má maximalizovat/minimalizovat.
- ifThen(podmínka$_1$, podmínka$_2$) – omezující podmínka, kde splnění první podmínky implikuje splnění podmínky druhé.

Metody přidávající omezující podmínky do modelu mohou, ale nemusejí být v konkrétním modelu uplatněny. Uplatnění omezující podmínky se vyžádá zavoláním metody post(). Pokud post() nebyl zavolán, nemusí být v konečném výsledku podmínka splněna. Další metodou volanou na podmínce je reify(). Ta vrací BoolVar s hodnotou 1, pokud je splněna a 0 jinak.

4.0.3 OR-Tools

OR-Tools je open source balíček napsaný v C++ a vyvíjený společností Google. Hodí se pro řešení optimalizačních problémů, tzv. vehicle routing problémů, toků, integer a linear programming a CP problémů. OR-Tools s sebou přinášejí vlastní solvery pro řešení CP a LP problémů CP-SAT a GLOP. Je ale možné s jejich pomocí pouze vymodelovat daný problém a řešit ho s jiným z podporovaných solverů, jako například: Gurobi, CPLEX, SCIP nebo GLP. OR-Tools se navíc se svým CP-SAT dlouhodobě udržuje na prvních příčkách v MiniZinc Challenge, což je celosvětová soutěž CP řešení. Vedle běžných programovacích jazyků jako C++, C, Python nebo Java, podporuje navíc modelovací jazyk FlatZinc. [18]

Vydávaný je pod Apache licencí. Ta podle [19] umožňuje volné používání, úpravu a redistribuci veškerého kódu. Požaduje v podstatě jen zachování autorství a aby se v případné redistribuce přidal ke kódu text obsahující mimo jiné text Apache licence, nebo při změně kódu upozornění, že byl soubor změněn oproti původnímu. Jakékoliv změny mohou být licencovány pod jinou než Apache licencí. Pod Windows je OR-Tools možné používat pouze v kombinaci s Visual Studio 2015 nebo novější. Visual Studio je už ale komerčním, placeným softwarem s cenou pohybující se okolo $45/developer za měsíc [20]. Na Linuxech žádné takovéto omezení není a tak ho tam lze používat zcela zdarma.

4.0.3.1 Dokumentace a Uživatelská podpora

Dokumentace OR-Tools se skládá z velkého množství příkladů s vysvětlením samotného typu problému s řešením psaném v Pythonu, C++ a Javě k nalezení na webu [18]. Další část dokumentace je dostupná na githubu [21] společně s ukázkovými kódy. Nalezení části dokumentace, která by byla ale užitečná, může být náročnější už jen kvůli množství. Dokumentace není všechna na jednom místě, část je na webu a část na githubu, kde se může minimálně ze začátku hůř hledat. Samotná knihovna se dělí na více jazyků a tak se části dokumentace mohou týkat jen jednoho z nich. Developeři se zaměřují možná o něco více na C++, ve kterém je celý nástroj napsán, a Python, pro který je dokumentace asi nejrozsáhlejší. Přesto je Java část udržovaná a funkční.

K uživatelské podpoře má team OR-Tools vytvořen Google groups fórum a mailing list. K zadání dotazu je nutná registrace, která ale bývá rychle potvrzena. Na samotném fóru se každý den objeví několik dotazů a většina z nich bývá ještě ten den zodpovězena.

4.0.3.2 Instalace a použití

OR-Tools oficiálně nepodporuje Maven. Nyní patří instalace k těm složitějším, kde je potřeba stáhnout si soubory a ručně je k projektu připojit. Pro Windows je navíc potřeba mít nainstalovanou danou verzi Visual Studia a při spouštění

přes příkazovou řádku používat x64 Native Tools Command Prompt, která je ve Visual Studiu k dispozici. Na začátku programu musí být ručně načtena knihovna libjniortools.so.

Na začátku programu je možno vybrat si z podporovaných knihoven. V této práci byla použita knihovna CP-SAT, což je defaultní knihovna pro řešení CP problémů. V této knihovně se nacházejí dvě hlavní třídy: CpModel a CpSolver. V modelu se popisuje problém a solveru se předává model k řešení. Problém se popisuje pomocí proměnných a omezujících podmínek. Z proměnných je k dispozici pouze IntVar – celočíselná proměnná. Jakékoliv jiné proměnné jako je například BoolVar nebo IntervalVar jsou jen vylepšené IntVar. Reálné proměnné nejsou podporovány.

Aby byly proměnné vyhodnoceny, přidávají se do modelu pomocí metod:

- newIntVar(od, do, název) – přidá do modelu novou celočíselnou proměnnou, která může nabývat hodnot z intervalu <od, do>.
- newConstant(hodnota) – přidá do modelu novou konstantu
- newEnumeratedIntVar(hodnoty, název) – přidá do modelu novou celočíselnou proměnnou, která může nabývat libovolné z vyjmenovaných hodnot
- newBoolVar(název) – přidá do modelu novou celočíselnou proměnnou, která může nabývat pouze hodnot 0 a 1
- newIntervalVar(začátek, trvání, konec, název) – přidá do modelu tři celočíselné proměnné s omezující podmínkou konec = začátek + trvání
- newOptionalIntervalVar(začátek, trvání, konec, bool, jméno) – stejně jako interval přidává do modelu tři nové proměnné se stejnou omezující podmínkou. Navíc ale přidává možnost býti ignorován některými funkcemi jako addNoOverlap a addCumulative, a to pokud je bool roven 0.

Některé z dalších metod modelu zahrnují:

- addNoOverlap(pole_intervalů) – omezující podmínka, dané intervaly se nikde nepřekrývají
- addCumulative(pole_intervalů, pole_požadavků, kapacita) – omezující podmínka. Je dána polem intervalů, stejně velkým polem požadavků, kde každý index odpovídá počtu požadavků pro daný interval, a kapacitou, která udává maximální možný počet požadavků vykonávaných v jeden čas. Tedy zabraňuje překrývání intervalů, a to po překročení dané kapacity.
- addEquality(a, b) – omezující podmínka, a = b

- addEqualityWithOffset(a, b, offset) – omezující podmínka a = b + offset
- addLessOrEqual(a, b) – omezující podmínka a ≤ b
- addLinearSumEqual(pole, hodnota) – omezující podmínka, součet prvků v poli se musí rovnat dané hodnotě
- minimaze(variable)/maximize(variable) – přidá optimalizační cíl minimalizace/maximalizace dané proměnné do modelu
- addMaxEquality(max, pole) – omezující podmínka, hodnota max se rovná největšímu prvku z pole

4.0.4 JaCoP

JaCoP[5] je Javovský plugin napsaný v Javě řešící CP problémy. Napsaný a udržovaný je hlavně dvěma lidmi: Krzysztof Kuchcinski z Lund University, Švédsko a Radoslaw Szymanek ze společnosti Crossing-Tech. Ti na něm pracují ve svém volném čase jako na svém koníčku. Upřít se ale nedá ani práce studentům dané univerzity, která minimálně na úplném začátku projektu hodně pomohla. Používán je jak na akademické půdě pro výuku a vývoj, tak i v průmyslu. Podporuje FlatZinc modelovací jazyk a nabízí také vlastní, na Scale založený, jazyk. [23]

Autoři se rozhodli používat dual-licensing. Jednak tedy software vydávají pod GNU Affero GPL licencí verze 3, ale vyhazují si právo software vydat pod licencí jinou, pokud to bude potřeba. GNU Affero GPL licence je již jednou z těch složitějších a restriktivnějších ve světě open source. Podle [22] vyžaduje mimo jiné, aby jakýkoliv odvozený software byl vydán pod stejnou licencí. Používání a redistribuce je ale povolena, a to i v případě zpoplatnění výsledného produktu.

4.0.4.1 Dokumentace a Uživatelská podpora

Dokumentace JaCoP se skládá z několika příkladů na githubu [24], uživatelské příručky [25] a javadoc [26]. Všechny odkazy se dají najít na oficiálním webu [23]. K uživatelské podpoře je zřízeno fórum. Dotazy zde nebývají moc časté, ale když už se nějaký objeví, bývá do druhého dne zodpovězen. JaCoP nabízí také placenou podporu a konzultace pro firemní zákazníky.

4.0.4.2 Instalace a použití

JaCoP podporuje Maven, instalace tedy obnáší pouze jednoduché přidání knihovny do konfiguračního souboru.

[5] název vznikl z "Java Constraint Programming

Problém se v JaCoP modeluje pomocí třídy Store. Tato třída ale narozdíl od předchozích frameworků neobsahuje mnoho metod. Místo toho se předává konstruktorům ostatních objektů, které ji nějakým způsobem potřebují. Takovými třídami jsou například třídy typů proměnných. JaCoP podporuje celočíselné proměnné (IntVar), reálné proměnné (FloatVar), množinové proměnné (SetVar), booleanovské proměnné (BooleanVar) a intervaly (Interval). Třída Store v sobě tyto proměnné uchovává, ale i přesto je potřeba si je v programu zvlášť ukládat do pole a pak je předat SelectChoicePoint třídě, která se společně s DepthFirstSearch třídou stará o samotné řešení problému. Proměnné, které nejsou této třídě předány, nejsou vyhodnocovány. Omezující podmínky se přidávají do instance Store pomocí metody impose(podmínka).

Tyto podmínky zahrnují například:

- new SumBool(pole, -=", a) – zajistí, že $a = \sum pole[i]$. Operátor -="může být nahrazen operátorem jiným, jako například "!="nebo «".
- new Cumulative(pole_začátků, pole_trvání, pole_požadavků, kapacita) – Pole začátků a trvání dohromady určují interval. Pole požadavků na daném indexu značí počet požadavků k vyřízení na daném intervalu. Tato omezující podmínka zajistí, že tento počet nikdy nepřevýší danou kapacitu.
- new Max(pole, a) – zajistí, že a bude rovna maximu z daného pole.
- new XlteqY(a, b) – zajistí, že $a \leq b$.
- new XplusYeqZ(a, b, c) – zajistí, že $c = a + b$
- new Reified(a, b) – má za parametry dvě omezující podmínky a zajistí $a \Leftrightarrow b$

Kapitola 5

Porovnání

V následující kapitole budou porovnány všechny vybrané frameworky. Shrnutí je k dispozi na konci této kapitoly v tabukách 5.1 a 5.2. V 5.1 se nachází stručný přehled vybraných frameworků s typem jejich licence, odkazy na dokumentaci, druh uživatelské podpory a stručný postřeh ohledně jejich API. V 5.2 jsou k nahlédnutí výsledky testů jednotlivých frameworků na datech, ta se dělí do čtyr kategorií:

1. velmi malá (soubor 01_super_small.xml) – 7 úkolů
2. malá (soubor 02_small.xml) – 105 úkolů
3. střední (soubor 03_medium.xml) – 248 úkolů
4. velká (soubor 04_large.xml) – 330 úkolů

V samotné tabulce 5.2 jsou u každého frameworku dva časy. První značí čas potřebný pro nalezení prvního řešení. Druhý značí čas potřebný pro nalezení optimálního řešení. Data, výsledky i použité zdrojové kódy jsou dostupné v příloze. Data ve složce test_data/scheduler/requests a výsledky s použitými zdrojovými kódy jednotlivých frameworků jsou k nalezení ve složce framework_specific_solutions.

Mezi vybranými frameworky byly tři open source a jeden s komerční licencí. Nevýhoda komerčního softwaru spočívá krom ceny také v nedostupnosti kódu a tedy těžšímu pochopení funkcionality a vhodnosti použití jednotlivých metod. Vyváženo to ale může být lepší uživatelskou podporou, která se u placeného softwaru předpokládá. Z čistě licenčního hlediska je na tom nejlépe OR-Tools s velmi benevolentní BSD licencí. Co se ale ceny v konečném důsledku týče, uživatelé na Windows si musí koupit Visual Studio a nebo přejít na Linux. Choco s Apache licencí bez potřeby použití softwaru třetích stran vychází tedy nakonec o něco lépe. JaCoP GNU licence je už o něco restriktivnější, ale stále se jedná o volný a open source software. CPLEX je na tom s restrikcemi licenčních podmínek nejhůře.

Dokumentace je u všech frameworků velmi rozsáhlá. Choco ji má ale nejdostupnější, kde na všechny zdroje odkazuje přímo ze své úvodní stránky. OR-Tools nabízí velké množství ukázkových kódů a manuálů, bohužel ne na všechny odkazuje přímo na svých webových stránkách. A také fakt, že podporuje více jazyků, celou dokumentaci může trochu znepřehledňovat. Také se mi nepodařilo najít nikde odkaz na javadoc, který by byl užitečný, určitě ho ale půjde vygenerovat ze zdrojových kódů. Dokumentace CPLEXu byla asi nejméně přehledná, skládající se z mnoha odkazů na další odkazy. Ale fakticky obsahuje dostatečné množství informací.

Uživatelská podpora je u všech vybraných frameworků, kromě komerčního CPLEXu, řešená pomocí diskuzního fóra řízeného tvůrci aplikace. Nejvíce dotazů a odpovědí je u OR-Tools, kde denně přibývá okolo desíti dotazů a na každý z nich se dostává odpovědi do několika hodin. To je ale možná způsobeno mimo jiné nedávným vydáním nové verze, se kterou se uživatelé teprve seznamují. JaCoP fórum neoplývá velkým množstvím nových dotazů, ale ty, co tam jsou, byly vyřízeny v rozmezí jednoho až dvou dnů. Na dotazy v Choco google groups se může čekat na odpověď i několik dní. CPLEX nabízí konzultace na své zákaznické lince a chatu kdykoliv během pracovních hodin.

Nejlepší API mělo s přehledem Choco a OR-Tools. Choco se ale může chlubit o něco větším počtem metod a funkcí užitečných a přímo dělaných pro rozvrhování. OR-Tools má rozhraní hodně podobné, akorát oplývá o něco menším počtem funkcí. CPLEX API je minimálně pro použití v rozvrhování o něco složitější. Nenabízí typ intervalu a tedy i funkce na něm, které jsou u rozvrhování vcelku užitečné. JaCoP a jeho rozhraní je vcelku nešikovně pojaté, kdy se místo vytvoření metod na modelu předává model konstruktorům nových instancí tříd. Pro každou takovouto metodu tedy existuje nová třída. Nestačí přidávat proměnné do modelu, ale ještě se musí i s modelem předat solveru. Všechny proměnné tedy musí být ještě v jednom dalším poli, kam když se zapomenou vložit, nebudou v solveru řešeny a výsledek pak může vypadat neočekávaně. To může vést k vcelku složitému debugování.

Testování na datech zvládl nejlépe OR-Tools CP-SAT, kde i pro velká data bylo možné získat optimální řešení do 10 minut. U OR-Tools byly testovány dva přístupy implementace omezující podmínky pracovníků nepracujících na dvou úkolech najednou. Prvním byla funkce addNoOverlap, která se zdála být pro tento úkol vhodnou volbou. Při použití této funkce se ale časová náročnost řešení mnohonásobně zvýšila v porovnání s funcí addCumulative. Pro porovnání jsou oba výsledky dostupné v příloze a pro použití metody addNoOverlap stačí ve zdrojovém kódu nastavit proměnnou USE_addNoOverlap na true. Choco si ale vedlo nejlépe v rychlosti hledání prvního vhodného řešení. JaCoP je velmi paměťově náročný. Pro řešení problémů používá rekurzivní depth first search s backtrackingem, který nezvládá větší množství dat a tak padá se StackOverflow chybou. CPLEXu trvalo v porovnání se všemi ostatními testovanými frameworky o mnoho déle nalezení optimálního řešení

i pro malá data. Pro střední data nenašel v rozumném čase[6] řešení žádné a na velkých datech mu po chvíli došla paměť.

Na základě výsledků této práce doporučuji pro nasazení OR-Tools nebo Choco. Oba tyto frameworky jsou dobře zdokumentované, s kvalitní uživatelskou podporou, open source a zvládají řešení i větších dat. OR-Tools je možná o něco rychlejší pro nalezení optimálního výsledku, velkou roli ale hraje konkrétní problém, použité metody a způsob, jakým je problém popsán. Pro jiná data to tedy už platit nemusí. Choco obsahuje více metod a výkonostně je s OR-Tools srovnatelné, až na velká data, kde se mu nepodařilo v rozumném čase[7] najít optimální řešení.

[6] Program běžel cca. 2 hodiny

[7] I tento program běžel cca. 2 hodiny

	CPLEX	Choco	OR-Tools CP-SAT	JaCoP
Licence	komerční	Apache License 2.0	BSD 4-Clause License	GNU AFFERO GENERAL PUBLIC LICENSE, version 3
Dokumentace	javadoc [11], tutoriály [10], ukázkové kódy, videa [12]	manuál [15], tutoriály [16], javadoc [17], zdrojové kódy, ukázkové kódy	tutoriály [18], dokumentace [21], zdrojové kódy, ukázkové kódy	manuál [25], javadoc [26], zdrojové kódy, ukázkové kódy [24]
Uživatelská podpora	chat telefonní linka	Twitter, Gitter chat, Google groups	Google groups	fórum, možnost placené podpory
API	spíše složitější, postrádající funkce pro práce s intervalem, složitější funkce si musí uživatel napsat sám	velmi jednoduché a intuitivní, třídy jako např. Task přímo určené pro rozvrhování	velmi jednoduché a intuitivní, méně metod	velmi složité, nestačí ukládání proměnných pouze do modelu

Tabulka 5.1: Tabulka Javovských frameworků

	CPLEX	Choco	OR-Tools CP-SAT	JaCoP
velmi malá data	0,09 s 0,15 s	0,05 s 0,13 s	0.01 s 0.03 s	0,02 s 1,29 s
malá data	7,46 s 133,79 s	1,87 s 78,52 s	1.65 s 27.73 s	0,60 s 1,51 s
střední data	not found	6,36 s 286,24 s	116.39 s 342.64 s	stack overflow
velká data	out-of-memory exception	10,50 s not found	234.05 s 613.76 s	stack overflow

Tabulka 5.2: Výsledky testování

Závěr

Cílem této práce bylo vypracování přehledu frameworků řešících a optimalizujících rozvrhovací úlohy. Vybrány byly čtyry frameworky: CPLEX, Choco, OR-Tools za použití CP-SAT a JaCoP.

Ačkoliv to nebylo přímo v zadání požadováno, před samotným popisováním jednotlivých knihoven a jejich zkoušením bylo potřeba se s danou problematikou seznámit. Na základě studia odborných pramenů jsem získala náhled do problematiky, který jsem se posléze snažila převést i do této práce. V první kapitole jsem popsala základní pojmy z oblasti rozvrhování a čtenář mohl získat náhled i do více formálního popisu problému pomocí Grahamovy klasifikace. Ve druhé kapitole jsem poté rozebrala dva nejčastější přístupy řešení rozvrhovacích problémů, Constraint a Linear programming.

Od třetí kapitoly jsem se již soustředila na samotné zadání. Začala jsem popisem konkrétního příkladu zadání optimalizace rozvrhovací úlohy. Rozebrala jsem zde jak zadání, tak řešení a návrh modelu s konečnou implementací s použitím jednoho z frameworků. Dále jsem ve čtvrté kapitole pokračovala popsáním a seznámením čtenáře s vybranými frameworky.

Po seznámení se s frameworky jsem pro každý z nich napsala kód, na kterém jsem spouštěla testovací data. Tento kód nemusí být optimální. Nicméně jsem se snažila projít si dostupnou dokumentaci a používat metody, které mi přišly nejvhodnější možné.

Všechny z těchto frameworků jsem otestovala na zadaných datech a v poslední kapitole jsem vypracovala jejich přehled a porovnání. Nejlépe z tohoto porovnávání vyšly frameworky OR-Tools a Choco, které také doporučuji pro nasazení na projekty zabývající se rozvrhováním v průmyslové výrobě. Naopak nejhorší výsledky podaly frameworky JaCoP a CPLEX, které nemusí být vhodné pro použití na větších datech.

Osobně pro mě byla tato práce velkým přínosem. Z nulových znalostí problematiky rozvrhování jsem získala, byť ne úplně hluboký, přehled. Seznámila jsem se s různými technologiemi řešící tento problém, naučila se vyhledávat v různých dokumentacích a naučila jsem se popsat rozvrhovací problém za

pomoci různých frameworků.

Literatura

[1] BRUCKER, Peter. Classification of Scheduling Problems In: *Scheduling algorithms*. 5th ed. New York: Springer, c2007. ISBN 978-3-540-69515-8.

[2] RUDOVÁ, Hana. *Úvod do rozvrhování* [online]. Brno: Fakulta Informatiky, Masarykova Univerzita, 2019 [cit. 2019-03-30]. Dostupné z: `https://www.fi.muni.cz/~hanka/rozvrhovani/prusvitky/prvni.pdf`

[3] *Constraint Optimization* [online]. Mountain View (Kalifornie): Google LLC, OR-Tools team, 2019 [cit. 2019-03-31]. Dostupné z: `https://developers.google.com/optimization/cp`

[4] RUDOVÁ, Hana. *Constraint Programming and Scheduling* [online]. Brno: Fakulta Informatiky, Masarykova Univerzita, 2009 [cit. 2019-03-30]. Dostupné z: `https://www.fi.muni.cz/~hanka/konstanz09/slides01_bw.pdf`

[5] BARTÁK, Roman. *Constraint Programming* [online]. Praha: Matematicko-fyzikální fakulta, Univerzita Karlova, 1998 [cit. 2019-03-30]. Dostupné z: `http://kti.ms.mff.cuni.cz/~bartak/constraints/constrsat.html`

[6] *Linear Optimization* [online]. Mountain View (Kalifornie): Google LLC, OR-Tools team, 2018 [cit. 2019-03-31]. Dostupné z: `https://developers.google.com/optimization/lp`

[7] ŠTECHA, Jan. *Optimální rozhodování a řízení: Přednášky*. Praha: Vydavatelství ČVUT, 2000. ISBN 80-01-02083-5. Dostupné také z: `https://www.algoritmy.net/article/1416/Simplexova-metoda`

[8] *IBM ILOG CPLEX Optimization Studio: Getting Started with CPLEX* [online]. Version 12 Release 8. Armonk (New York): IBM Corporation, 2017 [cit. 2019-04-15]. Dostupné z: `https://www.ibm.com/`

support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.studio.help/pdf/gscplex.pdf

[9] *IBM ILOG CPLEX Optimization Studio* [online]. Armonk (New York): IBM Corporation [cit. 2019-04-02]. Dostupné z: https://www.ibm.com/products/ilog-cplex-optimization-studio/pricing

[10] Java tutorial In: *IBM ILOG CPLEX Optimization Studio V12.9.0 documentation* [online]. Armonk (New York): IBM Corporation, 2019 [cit. 2019-04-13]. Dostupné z: https://www.ibm.com/support/knowledgecenter/SSSA5P_12.9.0/ilog.odms.cplex.help/CPLEX/GettingStarted/topics/tutorials/Java/Java_synopsis.html

[11] Overview (CPLEX Java API Reference Manual) In: *IBM ILOG CPLEX Optimization Studio V12.9.0 documentation* [online]. Armonk (New York): IBM Corporation, 2019 [cit. 2019-04-13]. Dostupné z: https://www.ibm.com/support/knowledgecenter/SSSA5P_12.9.0/ilog.odms.cplex.help/refjavacplex/html/overview-summary.html

[12] Videos In: *Developer Center: Decision Optimization on Cloud* [online]. Armonk (New York): IBM Corporation [cit. 2019-04-13]. Dostupné z: https://developer.ibm.com/docloud/blog/videos/

[13] PRUD'HOMME, Charles, Jean-Guillaume FAGES a Xavier LORCA. *Choco Documentation* [online]. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2018 [cit. 2019-04-15]. Dostupné z: http://www.choco-solver.org

[14] *The 3-Clause BSD License* [online]. Palo Alto (Kalifornie): Open Source Initiative [cit. 2019-04-02]. Dostupné z: https://opensource.org/licenses/BSD-3-Clause

[15] PRUD'HOMME, Charles, Jean-Guillaume FAGES a Xavier LORCA. *Choco Solver User Guide documentation* [online]. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2018 [cit. 2019-04-15]. Dostupné z: https://choco-solver.readthedocs.io/en/latest/?badge=latest

[16] PRUD'HOMME, Charles, Jean-Guillaume FAGES a Xavier LORCA. *Choco Tuto: Getting started with Choco* [online]. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2016 [cit. 2019-04-15]. Dostupné z: https://choco-tuto.readthedocs.io/en/latest/

[17] PRUD'HOMME, Charles, Jean-Guillaume FAGES a Xavier LORCA. *Choco-4.10.0: an Open-Source Constraint Solver 4.10.0 API* [online]. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2018 [cit. 2019-04-15]. Dostupné z: http://www.choco-solver.org/apidocs/index.html

[18] *Google OR-Tools* [online]. Mountain View (Kalifornie): Google LLC, OR-Tools team [cit. 2019-03-31]. Dostupné z: https://developers.google.com/optimization/

[19] *Apache License* [online]. Version 2.0. Forest Hill (Maryland): The Apache Software Foundation, 2004 [cit. 2019-03-31]. Dostupné z: https://www.apache.org/licenses/LICENSE-2.0

[20] *Buy Visual Studio* [online]. Redmond (Washington): Microsoft, 2019 [cit. 2019-03-30]. Dostupné z: https://visualstudio.microsoft.com/vs/pricing

[21] *Using the CP-SAT solver* [online]. Mountain View (Kalifornie): Google LLC, OR-Tools team, 2019 [cit. 2019-04-13]. Dostupné z: https://github.com/google/or-tools/tree/stable/ortools/sat/doc

[22] *GNU Affero General Public License* [online]. Version 3. Boston (Massachusetts): Free Software Foundation, 2007 [cit. 2019-04-03]. Dostupné z: http://www.gnu.org/licenses/agpl-3.0.html

[23] SZYMANEK, Radoslaw a Krzysztof KUCHCINSKI. *JaCoP* [online]. 2018 [cit. 2019-04-02]. Dostupné z: https://osolpro.atlassian.net/wiki/spaces/JACOP/overview

[24] SZYMANEK, Radoslaw a Krzysztof KUCHCINSKI. *JaCoP* [online]. 2018 [cit. 2019-02]. Dostupné z: https://github.com/radsz/jacop/tree/develop/src/main/java/org/jacop/examples

[25] SZYMANEK, Radoslaw a Krzysztof KUCHCINSKI. *JaCoP Library: User's Guide* [online]. Version 4.6. 2018 [cit. 2019-02]. Dostupné z: https://osolpro.atlassian.net/wiki/spaces/JACOP/pages/24248333/JaCoP+Guide

[26] SZYMANEK, Radoslaw a Krzysztof KUCHCINSKI. *JaCoP 4.6.0 API* [online]. 2018 [cit. 2019-02]. Dostupné z: http://jacopapi.osolpro.com/

Příloha **A**

Seznam použitých zkratek

API Application Programming Interface

BSD Berkeley Software Distribution

CP Constraint Programming

CSP Constraint Satisfaction Problem

GPL General Public License

LP Linear Programming

MIP Mixed Integer Programming

NP Nondeterministic Polynomial

QCP Quadratically Constraint Programming

QP Quadratic Programming

SOCP Second Order Cone Programming

SOS Special Order Sets

Příloha **B**

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `framework_specific_solutions` ..... adresář se zdrojovými kódy a výsledky jednotlivých frameworků
- `test_data` ..... testovací data
- `thesis`
  - `src` ..... zdrojová forma práce ve formátu LaTeX
  - `text` ..... text práce ve formátu pdf