VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV MATEMATIKY
INSTITUTE OF MATHEMATICS

MODELOVACÍ JAZYKY PRO OPTIMALIZACI
MODELLING LANGUAGES FOR OPTIMIZATION PROBLEMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Jaroslav Talpa**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. Pavel Popela, Ph.D.**
SUPERVISOR

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání bakalářské práce

Ústav:	Ústav matematiky
Student:	**Jaroslav Talpa**
Studijní program:	Aplikované vědy v inženýrství
Studijní obor:	Matematické inženýrství
Vedoucí práce:	**RNDr. Pavel Popela, Ph.D.**
Akademický rok:	2017/18

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma bakalářské práce:

Modelovací jazyky pro optimalizaci

Stručná charakteristika problematiky úkolu:

Student prostuduje aktuální postupy matematického modelování a řešení složených inženýrských optimalizačních úloh. Využije osvojené poznatky matematické analýzy, lineární algebry a diskrétní matematiky a zkušenosti se softwarovými nástroji. Samostatně se seznámí s aktuální problematikou modelovacích optimalizačních jazyků (AIMMS, GAMS, aj.). Pro zvolenou třídu složených optimalizačních úloh navrhne pomocí vybraného modelovacího jazyka a vhodného programátorského nástroje efektivní implementaci pro jejich modelování a řešení. Její funkčnost ověří testovacími výpočty a svůj postup srozumitelně zdokumentuje.

Cíle bakalářské práce:

1. Přehledně bude uvedena vybraná problematika složených optimalizačních úloh.
2. Čtenář práce bude seznámen s problematikou modelovacích optimalizačních jazyků.
3. Pro zvolenou třídu složených optimalizačních úloh autor navrhne efektivní implementaci pro jejich modelování a řešení.
4. Autor popíše své původní softwarové řešení pomocí vybraného modelovacího jazyka a vhodného programátorského nástroje.
5. Řešení bude otestováno na konkrétních příkladech.

Zpracovaný text práce uvede uživatele jazyka do problematiky modelování. Důraz bude kladen na srozumitelnost a přehlednost výkladu a snadné seznámení čtenáře s tématem na základě vlastních zkušeností studenta. Text bude dále využit pro další rozvoj spolupráce mezi ústavem matematiky a odbornými pracovišti, které mají zájem se seznámit s nejnovějším modelovacím softwarem pro vybranou třídu aplikací.

Seznam doporučené literatury:

KLAPKA, Jindřich, Jiří DVOŘÁK a Pavel POPELA. Metody operačního výzkumu. Vyd. 2. Brno: VUTIUM, 2001. ISBN 80-214-1839-7.

WILLIAMS, H. Paul. Model building in mathematical programming. 5th ed. Hoboken, N.J.: John Wiley & Sons, 2013. ISBN 978-1-118-44333-0.

PARDALOS, Panos M. a Mauricio G. C. RESENDE (eds.). Handbook of applied optimization. Oxford: Oxford University Press, 2002. ISBN 0195125940.

BIRGE, John R. and François LOUVEAUX. Introduction to Stochastic Programming. Springer Verlag, 1997. ISBN: 978-1-4614-0236-7.

WALLACE, Stein W. and Alan KING. Modeling with Stochastic Programming. Springer Verlag, 2012. ISBN 978-0-387-87816-4.

KALL, Peter and Stein W. WALLACE. Stochastic Programming. New York: John Wiley & Sons, 1993. ISBN 978-0471951582.

GAMS Modelling Language Manuals. GAMS Inc, 2017.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2017/18

V Brně, dne

L. S.

prof. RNDr. Josef Šlapal, CSc.
ředitel ústavu

doc. Ing. Jaroslav Katolický, Ph.D.
děkan fakulty

Abstrakt

Tato bakalářská práce se zabývá problematikou modelování a řešení optimalizačních úloh, a to především pomocí modelovacích jazyků. Pro řešení dvou vybraných typů složených optimalizačních úloh, analýzy citlivosti a navazující optimalizace, jsou dále v práci popsána dvě původní softwarová řešení v jazyce Java, využívající modelovací jazyk GAMS, která jsou dále testována a aplikována na reálné problémy. V práci je také uvedena problematika paralelizace, kterou navrhované programy implementují.

Summary

This bachelor thesis is concerned with the area of modeling and solving of optimization problems, mainly by the use of modeling languages. For the solving of two selected types of complex optimization problems, sensitivity analysis and multistage optimization, two original software solutions in Java using the GAMS modeling language, are described in the paper, which are further tested and applied to real problems. The thesis also describes the issue of parallelism which is implemented by the proposed programs.

Klíčová slova

optimalizace, modelovací jazyky, složená optimalizace, analýza citlivosti, paralelizace

Keywords

optimization, modeling languages, composed optimization, sensitivity analysis, paralellization

TALPA, J.*Modelovací jazyky pro optimalizaci*. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2018. 68 s. Vedoucí RNDr. Pavel Popela, Ph.D.

Prohlašuji, že jsem bakalářskou práci na téma *Modelovací jazyky v optimalizaci* vypracoval samostatně pod vedením RNDr. Pavla Popely, Ph.D., a to s použitím odborné literatury a pramenů uvedených v seznamu literatury.

Jaroslav Talpa

Na tomto místě bych chtěl poděkovat především svému vedoucímu RNDr. Pavlu Popelovi, Ph.D. za cenné připomínky a rady k obsahu této práce, dále pak kolegům Ing. Františkovi Janošťákovi a Ing. Radovanu Šomplákovi, Ph.D. za ochotnou spolupráci na testování mých programů, jenž jsou výstupem této práce, na reálných úlohách a poskytnutí výpočetního času na výkonném počítači Ústavu procesního inženýrství FSI VUT.

Jaroslav Talpa

Obsah

Úvod **13**

I Teoretická část **15**

1 Optimalizační úlohy **17**
1.1 Matematický model optimalizační úlohy 17
1.2 Základní pojmy 19
1.3 Typy optimalizačních úloh 20
1.4 Složené úlohy 21
1.4.1 Analýza citlivosti 22
1.4.2 Navazující optimalizace 24
1.5 Nástroje pro řešení optimalizačních úloh 25
1.5.1 Řešiče 25
1.5.2 Knihovny pro vyšší programovací jazyky 26
1.5.3 Optimalizační nástroje výpočetních systémů 27
1.5.4 Modelovací jazyky 28

2 Paralelizace **31**
2.1 Multitasking a multithreading 31
2.1.1 Vícejádrový procesor 32
2.1.2 Paralelizace versus konkurence 32
2.2 Zrychlení pomocí paralelizace 33
2.2.1 Amdahlův zákon 33
2.3 Multithreading v programovacím jazyce Java 34
2.3.1 Třída Thread 35
2.3.2 Synchronizace a její úskalí 36
2.3.3 Pokročilé API 37

II Praktická část **41**

3 Program ParallelGams **43**
3.1 Funkce programu 43
3.2 Popis implementace 44
3.3 Běh programu na testovacích datech 45
3.4 Aplikace programu na reálný problém 46

4 Program ComposedOptimization **49**
4.1 Funkce programu 49
4.1.1 Typy uzlů 51
4.2 Popis implementace 54
4.2.1 Generování stromu projektu 54
4.2.2 Procházení stromu projektu 55
4.3 Použití programu na reálném problému 58

Závěr **63**

Literatura **65**

Úvod

V technických aplikacích optimalizace jsou řešené úlohy většinou rozsáhlejšího charakteru, jejich popis bývá zpravidla složitý a výpočet náročný. Z toho důvodu existuje pro řešení takových úloh řada různých, k tomu určených, modelovacích jazyků. U některých z těchto úloh vyvstává otázka, zda-li by se nedalo využít některých jejich specifických vlastností pro urychlení jejich řešení. Proto se tato práce zaměřuje na původní implementaci paralelizace využívající modelovací jazyk GAMS a testování její účinnost.

V teoretické části této práce je v Kapitole 1 uvedena problematika optimalizačních úloh a přehled vybraného softwaru pro jejich řešení, včetně modelovacích jazyků. Kapitola 2 se pak zabývá problematikou paralelních výpočtů a jejího přínosu pro jejich urychlení. Dále jsou zde popsány způsoby, jak lze paralelizace dosáhnout v programovacím jazyce Java, v němž jsou napsána softwarová řešení, která jsou výstupem této práce.

V praktické části je pak popsán původní software pro řešení výše popsaných problémů, a to včetně jeho aplikace na reálné problémy či testovací data. V Kapitole 3 se jedná o program ParallelGAMS, který uživateli umožňuje paralelní řešení optimalizačního modelu modelovacího jazyka GAMS s různými vstupními daty, čehož lze využít např. pro urychlení analýzy citlivosti daného modelu. Kromě náhodně generovaných testovacích dat je program také aplikován na reálnou optimalizační úlohu. V Kapitole 4 je pak popsán druhý program této práce, ComposedOptimization, který navazuje na zkušenosti z vývoje a následné aplikace programu ParallelGAMS a umožňuje uživateli strukturovaným způsobem řešit optimalizační úlohy rozložitelné na více podúloh. Uvedené programy byly testovány ve spolupráci s pracovníky Ústavu procesního inženýrství na dvou testovacích optimalizačních úlohách s reálnými daty řešenými v rámci projektu SPIL na uvedeném ústavu[1].

[1]Podrobné informace o projektu SPIL lze najít např. na http://netme.cz/cs/spil/

Část I

Teoretická část

Kapitola 1

Optimalizační úlohy

1.1 Matematický model optimalizační úlohy

Jako úplně obecný popis optimalizační úlohy nám poslouží pojem jejího *matematického modelu* [1].

Definice 1.1. Nechť $S \subset \mathbb{R}^n$ a $f : \mathbb{R}^n \longrightarrow \mathbb{R}$. Pak definujeme matematický model optimalizační úlohy takto

$$\min_{\mathbf{x}}\{f(\mathbf{x})|\mathbf{x} \in S\}. \tag{1.1}$$

Tedy v obecné úloze minimalizujeme nějakou funkci $f(\mathbf{x})$ vzhledem k její proměnné $\mathbf{x}$, ne však na celém jejím definičním oboru, ale pouze na jeho podmnožině, omezené podmínkami vyplývajícími ze zadání dané úlohy. Spíše než samotná funkční hodnota $f(\mathbf{x})$ nás však zajímá jí odpovídající argument - hodnota proměnné $\mathbf{x}$. Hledání řešení optimalizační úlohy tedy můžeme přeformulovat z výrazu (1.1) za předpokladu dodržení stejných podmínek takto

$$? \in \operatorname{argmin}_{\mathbf{x}}\{f(\mathbf{x})|\mathbf{x} \in S\}, \tag{1.2}$$

kdy za řešení úlohy považujeme jednu nebo více hodnot proměnné $\mathbf{x}$ z množiny přípustných řešení S, pro něž $f(\mathbf{x})$ dosahuje svého globálního minima (v některých případech je postačující i minimum lokální) [1]. Optimalizační úloha nemusí být pouze minimalizační, (1.1) a (1.2) jde ekvivalentně přepsat pro $\max_{\mathbf{x}}$ a $\operatorname{argmax}_{\mathbf{x}}$, kdy poté hledáme globální (lokální) maximum $f(\mathbf{x})$. [1] Záleží na zadání konkrétní úlohy, zda se snažíme např. snížit náklady, anebo zvýšit zisky.

Příklad 1.2. Vyrábíme pracovní stoly a kancelářské židle. Na výrobu spotřebováváme dva zdroje, dřevo (dřevotřísku) a technické komponenty (šrouby,

pojistné podložky, atd.), těch však máme k dispozici omezené množství: 90 jednotek dřeva a 50 jednotek komponentů. Oba typy výrobků potřebují na výrobu jednoho kusu jednu jednotku komponentů, stůl je však náročnější na dřevo: tři jednotky oproti jedné jednotce na výrobu židle. Jeden stůl prodáváme za 1100, židli za 400. Naším úkolem je určit, kolik máme vyrobit stolů a židlí tak, aby byl celkový výdělek největší. Poptávku trhu zanedbáme, co vyrobíme se určitě prodá.

Abychom mohli úlohu převést na matematický model, zavedeme proměnné x_1 a x_2 odpovídající počtu vyrobených stolů a židlí. Celkový výdělek, který označíme jako z, pak můžeme určit následovně

$$z = 1100x_1 + 400x_2. \tag{1.3}$$

Výraz (1.3) označujeme jako *účelovou funkci* [1, 2]. Jedná se o funkci odpovídající $f(\mathbf{x})$ z (1.1), pouze místo minima hledáme maximum (největší výdělek). Triviální řešení takového modelu odpovídá hodnotám proměnných x_1 a x_2 rostoucích nade všechny meze. Vzhledem k omezeným zdrojům však nemůžeme hledat řešení libovolně v celém $\mathbb{R}^2$, ale pouze na nějaké jeho podmnožině S. Pokud bychom vyráběli pouze jeden výrobek (hodnota druhé proměnné by byla rovna nule), tak maximální počet vyrobených stolů x_1 by byl roven 30, protože máme dostupných pouze 90 jednotek dřeva a na výrobu jednoho stolu potřebujeme tři. U židlí by byly zase limitujícím faktorem komponenty, kterých máme dostupných pouze 50, a proto i $x_2 = 50$. Pokud uvažujeme výrobu obou druhů nábytku současně, můžeme limitující velikost zdrojů popsat následujícími dvěma nerovnicemi

$$\begin{aligned} 3x_1 + x_2 &\leq 90 \\ x_1 + x_2 &\leq 50. \end{aligned} \tag{1.4}$$

Nerovnice (1.4) nazýváme *omezeními* [1, 2] daného modelu a dohromady nám určují množinu S všech přípustných řešení. Dalším omezením je nezápornost počtu vyrobených kusů jednotlivých výrobků $x_1, x_2 \geq 0$. Samozřejmě také nemůžeme vyrobit např. půl stolu, proměnné x_1, x_2 jsou tedy z množiny celých čísel $\mathbb{Z}$. Souhrnně můžeme model naší úlohy podle (1.1) zapsat takto

$$\max_{x_1,x_2}\{1100x_1 + 400x_2 | 3x_1 + x_2 \leq 90; x_1 + x_2 \leq 50; x_1, x_2 \geq 0; x_1, x_2 \in \mathbb{Z}\}. \tag{1.5}$$

Pokud označíme ceny jednotlivých výrobků jako c_j, proměnné jako x_j, koeficienty soustavy nerovnic (1.4) a_{ij} a její pravou stranu, tedy množství zdrojů,

jako b_i, dostaneme pro počet proměnných n a počet omezení m obecný sumačně indexový zápis modelu [1]

$$\max_{x_j}\{\sum_{j=1}^{n} c_j x_j | \sum_{j=1}^{n} a_{ij} x_j \leq b_i; i = 1, 2, \ldots, m; x_j \geq 0; x_j \in \mathbb{Z}; j = 1, 2, \ldots, n\}. \tag{1.6}$$

Pokud dále zavedeme vektory $\mathbf{c} = (c_1, \ldots, c_n)^T, \mathbf{x} = (x_1, \ldots, x_n)^T$ a $\mathbf{b} = (b_1, \ldots, b_m)^T$ a matici $\mathbf{A}$ jako matici prvků a_{ij}, dostaneme obecný maticově vektorový zápis modelu [1]

$$\max_{\mathbf{x}}\{\mathbf{c}^\top \mathbf{x} | \mathbf{A}\mathbf{x} \leq \mathbf{b}; \mathbf{x} \geq 0; \mathbf{x} \in \mathbb{Z}^n\}. \tag{1.7}$$

1.2 Základní pojmy

Nyní bude zavedeno několik pojmů a vlastností týkajících se oblasti optimalizačních úloh, a to především problematiky existence a hledání jejich řešení.

Nejprve bude uveden pojem samotného minima funkce:

Definice 1.3. Nechť $S \subset \mathbb{R}^n$ a $f : S \longrightarrow \mathbb{R}$ a dále $O_\varepsilon(\mathbf{x})$ značí epsilonové okolí bodu $\mathbf{x} \in S$. Pak definujeme $\mathbf{x}_{min} \in S$ jako bod *lokálního minima* [1] funkce f na S právě tehdy, když

$$\exists O_\varepsilon(\mathbf{x}_{\min}) : \forall \mathbf{x} \in S \cap O_\varepsilon(\mathbf{x}_{\min}) \backslash \{\mathbf{x}_{\min}\} : f(\mathbf{x}_{\min}) \leq f(\mathbf{x}). \tag{1.8}$$

Definice 1.4. Nechť $S \subset \mathbb{R}^n$ a $f : S \longrightarrow \mathbb{R}$. Pak definujeme $\mathbf{x}_{min} \in S$ jako bod *globálního minima* [1] funkce f na S právě tehdy když

$$\forall \mathbf{x} \in S \backslash \{\mathbf{x}_{\min}\} : f(\mathbf{x}_{\min}) \leq f(\mathbf{x}). \tag{1.9}$$

Jestliže neostrou nerovnost v podmínce (1.8), resp. (1.9), nahradíme nerovností ostrou, hovoříme pak o *ostrém lokálním*, resp. *globálním*, *minimu* [1].

Pro řadu úloh je výpočtově jednodušší, či z hlediska existence algoritmů přímo nutné, hledat pouze minimum lokální, zvláště, pokud jich účelová funkce na množině přípustných řešení dosahuje více. [1, 2, 3]

O existenci globálního minima hovoří *Weierstrassova věta* [1]:

Věta 1.5 (Weierstrass)**.** Nechť množina $S \subset \mathbb{R}^n$ je neprázdná a kompaktní[1] a funkce $f : S \longrightarrow \mathbb{R}$ je spojitá na S, pak matematický model $\min_x\{f(x)|x \in S\}$ dosahuje svého globálního minima.

[1] tedy je zároveň *ohraničená* a *uzavřená*, tedy zároveň platí, že: $\exists \mathbf{x} \in S, \exists \varepsilon > 0 : S \subset O_\varepsilon(\mathbf{x})$ a S je rovna svému *uzávěru* [1]

Z hlediska hledání minima (řešení modelu) je výhodné pracovat s konvexní množinou přípustných řešení a konvexní účelovou funkcí, protože dle věty 1.8 je pak každé lokální minimum globální (a často i jediné), a proto lze pro hledání globálního optima použít i algoritmy a postupy pro hledání extrémů lokálních.[1, 2, 3]

Definice 1.6. Nechť $S \subset \mathbb{R}^n$. Říkáme, že množina S je *konvexní* [1, 2, 3] právě tehdy, když $\forall \mathbf{x}_1, \mathbf{x}_2 \in S, \forall \lambda \in \langle 0; 1\rangle : \lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2 \in S$.

Definice 1.7. Nechť množina $S \subset \mathbb{R}^n$ je neprázdná a konvexní a $f : S \longrightarrow \mathbb{R}$. Říkáme, že funkce f je *konvexní* [1, 2, 3] na S právě tehdy, když $\forall \mathbf{x}_1, \mathbf{x}_2 \in S, \forall \lambda \in (0; 1) : f(\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) \leq \lambda f(\mathbf{x}_1) + (1-\lambda) f(\mathbf{x}_2)$.

Věta 1.8. Nechť množina $S \subset \mathbb{R}^n$ je neprázdná a konvexní a funkce $f : S \longrightarrow \mathbb{R}$ je konvexní na S. Dále nechť $\bar{\mathbf{x}} \in \text{arglocmin}_x\{f(x)|\mathbf{x} \in S\}$, pak $\bar{\mathbf{x}} \in \text{argglobmin}_x\{f(x)|\mathbf{x} \in S\}$.[1, 3]

1.3 Typy optimalizačních úloh

Optimalizační úlohy lze podle vlastností jejich omezení či podle požadavků na číselný obor jejich proměnných rozdělit na několik typů. S těmi jsou spjaty i různé přístupy a algoritmy pro jejich řešení.

Lineární programování Úlohou/modelem *lineárního programování* (LP) [1, 2, 3, 4, 5] je taková úloha/model, jejíž všechna omezení i účelová funkce obsahují pouze lineární vztahy mezi proměnnými. Jedná se tedy o soustavu lineárních rovnic a nerovnic, kterou lze vhodným doplněním převést pouze na soustavu lineárních rovnic[2] a celý model obecně zapsat jako

$$\min_{\mathbf{x}}\{\mathbf{c}^\top \mathbf{x} | \mathbf{A}\mathbf{x} = \mathbf{b}; \mathbf{x} \geq \mathbf{0}\}, \tag{1.10}$$

kde n je počet proměnných a m počet omezení, $\mathbf{A}$ je matice soustavy dimenze $m \times n$, někdy označovaná jako *strukturální*, $\mathbf{b}$ vektor pravé strany dimenze m, někdy označovaný jako vektor *kapacitních limitů*, a $\mathbf{c}$ vektor koeficientů účelové funkce dimenze n, někdy označovaný jako *cenový* vektor. [3] Příklad 1.2 je tedy úlohou LP. Dalším příkladem úlohy LP může být úloha toku sítí.

[2] soustavu nerovnic $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ lze převést na soustavu rovnic $\mathbf{A}\mathbf{x} + \mathbf{y} = \mathbf{b}$; $\mathbf{y} \geq 0$ doplněním o *doplňkové* proměnné $\mathbf{y}$ (angl. *slack* variables) [1, 3, 5]

Nelineární programování Pokud omezení či účelová funkce modelu obsahují nelineární výrazy, jedná se o úlohu *nelineárního programování* (NLP) [1, 3], zapsanou obecně takto:

$$\min_{\mathbf{x}}\{f(\mathbf{x})|\mathbf{g}(\mathbf{x}) \circ \mathbf{0}; \mathbf{x} \in S\}, \tag{1.11}$$

kde $\circ$ představuje obecně znaménka $=$, $\leq$ a $\geq$. Jestliže jsou účelová funkce f a množina přípustných řešení S konvexní, hovoříme o *konvexní* [2] úloze NLP, jinak se obecně jedná o *nekonvexní* [2] úlohu NLP. Pro konvexní úlohy je díky Větě 1.8 každé lokální optimum globální, což je velmi výhodné, protože většina algoritmů je schopná najít právě pouze lokální extrémy. Této vlastnosti lze využít i u úloh LP, protože ty lze vždy považovat zároveň za úlohy konvexní. [2]

Kvadratické programování Jedná se o druh NLP, kdy je za model *kvadratického programování* (QP) [1] považován model ve tvaru:

$$\min_{\mathbf{x}}\{\frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} + \mathbf{c}^\top\mathbf{x}|\mathbf{A}\mathbf{x} \leq \mathbf{b}; \mathbf{x} \in S\}, \tag{1.12}$$

kde n je počet proměnných a m počet omezení, $\mathbf{A}$ je matice soustavy dimenze $m \times n$, $\mathbf{b}$ vektor pravé strany dimenze m, $\mathbf{c}$ vektor koeficientů účelové funkce dimenze n, a $\mathbf{Q}$ je symetrická matice dimenze $n \times n$.

Celočíselné programování U *celočíselného programování* (IP)[3] [1, 3, 2] je množina přípustných řešení omezena pouze na celá čísla $\mathbb{Z}$:

$$\min_{\mathbf{x}}\{f(\mathbf{x})|\mathbf{x} \in S; \mathbf{x} \in \mathbb{Z}^n\}. \tag{1.13}$$

Pokud je podmínka celočíselnosti vyžadována pouze u některých proměnných, jedná se pak o tzv. *smíšené* [1] celočíselné úlohy, např. MILP a MINLP[4].

1.4 Složené úlohy

Za složené úlohy budou v této práci dále považovány[5] takové úlohy, které se skládají z několika dalších podúloh nebo jsou na ně rozložitelné. Dále jsou popsány dva typy složených úloh, a to analýza citlivosti a navazující optimalizace.

[3] z angl. integer programming

[4] mixed integer linear programming a mixed integer non-linear programming

[5] po konzultaci s odbornou literaturou a s vedoucím práce

1.4.1 Analýza citlivosti

Při řešení reálných optimalizačních úloh může uživatele zajímat, kromě samotného řešení úlohy, také to, jak se bude hodnota optima měnit v závislosti na změně parametrů úlohy. Tyto parametry, i když jsou v rámci řešení modelu konstantní, se tedy mohou měnit. To může reflektovat např. změnu cen zdrojů či jejich dostupnost v čase, kdy za nejlepší řešení lze považovat spíše to, které se příliš nemění v rámci malých změn vstupních parametrů, na rozdíl od řešení, které je optimální pouze pro konkrétní hodnoty parametrů, ale pro jiné je silně suboptimální. Také lze zkoumat citlivost řešení v závislosti na přidání nových či odebrání některých stávajících omezení. [5, 6]

Jedním ze způsobů, jak provádět analýzu citlivosti, je otestovat všechny kombinace potenciálních změn a sledovat, jak se mění hodnota optima a samotné řešení pro jednotlivé scénáře. Spojité parametry lze nahradit nějakým vhodným dělením na intervalu změny, tím se však může ztratit informace o některých významných (zlomových) bodech citlivostní závislosti (viz dále). Takovou analýzu citlivosti, a k ní hledání celkového optimálního řešení, lze považovat za složenou optimalizační úlohu, i když na sobě jednotlivé podúlohy přímo nezávisí. Pokud je počet testovaných změn tzv. scénářů velký, je jistě zřejmé, že celkové řešení je časově náročné. Pro urychlení řešení by mohla posloužit paralelizace výpočtu jednotlivých podúloh s využitím jejich nezávislosti (viz sekce 2.2).

Pro modely LP (viz sekce 1.3) existují také analytické metody, umožňující určit pro řešení jeho tzv. *interval stability* [5], tedy oblast, ve které se může měnit hodnota účelové funkce, řešení však zůstává stejné[6]. Teprve až na okraji intervalu se mění i samotné řešení. Pro úlohu LP (viz (1.10)) lze určovat intervaly stability řešení u dvou případů:

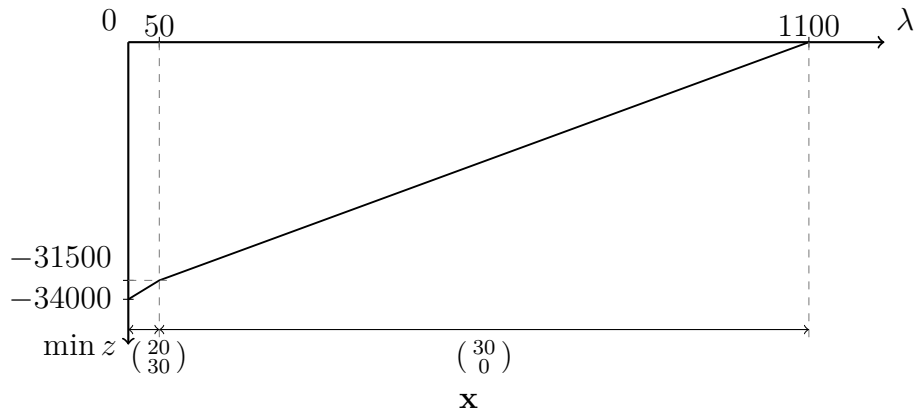
- Vychýlení cenového vektoru $\mathbf{c}$ ve směru vektoru $\mathbf{c}'$, tedy nahrazení $\mathbf{c}$ výrazem $\mathbf{c} + \lambda\mathbf{c}'$ pro reálné $\lambda \geq 0$.

- Vychýlení vektoru pravé strany $\mathbf{b}$ ve směru vektoru $\mathbf{b}'$, tedy nahrazení $\mathbf{b}$ výrazem $\mathbf{b} + \lambda\mathbf{b}'$ pro reálné $\lambda \geq 0$.

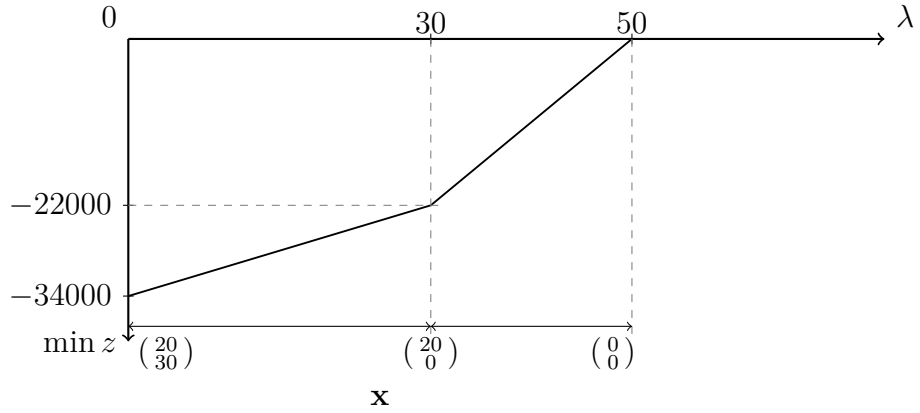
V obou dvou případech se optimální hodnota účelové funkce stává závislou na parametru vychýlení λ. [5] Pro Příklad 1.2 a zvolené směry vychýlení $\mathbf{c}' = \binom{1}{1}$ a $\mathbf{b}' = \binom{-1}{-1}$ jsou tyto závislosti znázorněny na Obrázcích 1.1 a 1.2. Pro potřeby výpočtu byla původní maximalizační úloha převedena na úlohu minimalizační, optimální hodnoty účelové funkce jsou tedy záporné. Jak je

[6] "stejné" zde nemusí nutně znamenat stejné hodnoty, ale pouze stejné „bázické řešení", více viz [5]

patrné z obrázků, obě funkce jsou po částech lineární. Dále funkce pro výchylku vektoru **c** je konkávní a funkce pro výchylku vektoru **b** je konvexní. Tyto vlastnosti platí obecně a ne jenom pro uvedený příklad, viz [5] Poznamenejme, že pokud by byla takováto analýza citlivosti provedena přístupem řešení úlohy pro „síť“ konkrétních hodnot λ, tak z ní může být obtížné přesně určit zlomové body závislostí, u složitějších úloh však analytické řešení nemusí být vždy možné.



Obrázek 1.1: Závislost optima $\min z$ na parametru λ pro vychýlení $\mathbf{c} + \lambda\binom{1}{1}$. Hodnoty řešení $\mathbf{x}$ se mění skokově a jsou zde uvedené pro jednotlivé intervaly.



Obrázek 1.2: Závislost optima $\min z$ na parametru λ pro vychýlení $\mathbf{b} + \lambda\binom{-1}{-1}$ Hodnoty řešení $\mathbf{x}$ se mění lineárně a jsou zde uvedené pro krajní body a bod „zlomu“.

1.4.2 Navazující optimalizace

Mezi problémy navazující optimalizace lze také zařadit úlohy dynamického programování (DP) [7, 8, 9], které se skládají z posloupnosti více jednodušších podúloh, také nazývaných rozhodovací stupně. Typicky se jedná o optimalizaci procesů, které zahrnují několik rozhodnutí postupně na sebe navazujících v čase.

Pro popis modelu problému navazující optimalizace slouží vícestupňový rozhodovací model (MSD[7]) [7, 8, 9]:

Definice 1.9. MSD modelem se rozumí šestice (T, Y, X, g, Y_1, f), kde:

1. T je *počet stupňů* a $t \in \tau = \{1 \cdots T\}$ je index označující *pořadí stupně*.
2. $Y = \bigcup_{t=1}^{T+1} Y_t$ je množina všech *stavů*, kde Y_t představuje množinu všech stavů pro daný stupeň t a Y_{T+1} množinu všech *koncových stavů*.
3. X je zobrazení $X : \tau \times Y \longrightarrow 2^D$, které přiřadí každé dvojci stupeň-stav (t, y_t) množinu *přípustných rozhodnutí* $X(t, y_t)$, kde $D \neq \emptyset$ je prostor rozhodnutí.
4. g je *přechodová funkce* $g : \tau \times Y \times D \longrightarrow Y$, která přiřadí každé dvojici stupeň-stav (t, y_T) a rozhodnutí x_t následující stav stupně $t + 1$, y_{T+1} je pak koncovým stavem.
5. $Y_1 \neq \emptyset$ je množina *počátečních stavů*[8].
6. $f : Y_1 \times D^T \longrightarrow \mathbb{R}^n$ je účelovou funkcí modelu.

Hodnota účelové funkce je tedy přiřazena každé posloupnosti přípustných rozhodnutí vedoucích od počátečního stavu ke konečnému. Pro usnadnění řešení takovýchto modelů lze využít vlastnosti rozložitelných funkcí.

Definice 1.10. Říkáme, že funkce více proměnných f je *rozložitelná* [7] na f_1 a f_2, pokud platí rovnost $f(x, \mathbf{y}) = f_1(x, f_2(\mathbf{y}))$ a dále, pokud je f_1 neklesající vzhledem ke svému druhému argumentu.

Pro optimum takovýchto funkcí pak platí důležitá rovnost:

Věta 1.11. Nechť f je funkce více proměnných, rozložitelná na f_1 a f_2 a platí $f(x, \mathbf{y}) = f_1(x, f_2(\mathbf{y}))$, pak pro hodnoty optim (minim resp. maxim) funkcí f, f_1 a f_2 platí rovnost [7]:

$$\operatorname*{Opt}_{x,\mathbf{y}}\{f(x, \mathbf{y})\} = \operatorname*{Opt}_{x}\{f_1(x, \operatorname*{Opt}_{\mathbf{y}}\{f_2(\mathbf{y})\})\} \tag{1.14}$$

[7] multistage decision model

[8] často může být jediný, tzn. $Y_1 = \{y_1\}$

Řešení nového problému, sestávajícího z podúloh vzniklých z původní úlohy rozložením, je pak zpravidla jednodušší, než řešení úlohy samotné.

1.5 Nástroje pro řešení optimalizačních úloh

Pro řešení reálných optimalizačních úloh, vzhledem k velkému počtu jejich proměnných a omezení, existuje celá řada softwarových nástrojů. Ty lze rozdělit do několika kategorií.

1.5.1 Řešiče

Řešiče představují ucelené balíčky procedur pro řešení konkrétních typů optimalizačních úloh. Většinou jsou koncipované jako „černé skříňky“, tedy pro jejich použití není nutné znát vnitřní implementační detaily, pouze způsob předávání dat řešiči a formát jeho výstupu. Některé jsou dostupné společně s rozhraním pro nějaký vyšší programovací jazyk, a jsou tedy využitelné jako podklad pro pokročilejší optimalizační systémy. Existují komerční i volně dostupné varianty. [3]

- **BDMLP** [10] je základní řešič pro modelovací jazyk GAMS určený pro úlohy LP a MIP. Jako takový je dostupný v rámci každé verze systému GAMS.

- **CBC** [11] je open-source řešič pro MIP problémy, napsaný v jazyce C++. Pro samotné řešení je implementován algoritmus branch-and-bound.

- **CONOPT** [12] je řešič pro NLP problémy implementující pro řešení metodu GRG[9]. Je dostupný buď v rámci modelovacích systémů AIMMS, AMPL, GAMS, LINGO a MPL, anebo jako knihovna jazyka Fortran.

- **CPLEX** [10, 13] je komerční řešič pro problémy LP, MIP a QP vyvíjený firmou IBM. Slouží jako základ pro vývojové prostředí IBM ILOG CPLEX Optimization Studio obsahující vlastní programovací jazyk. Existují také rozhraní pro programovací jazyky C++, Java, Python a pro .NET platformu. Kromě toho je také dostupný jako jeden z řešičů pro modelovací jazyk GAMS.

[9] generalized reduced gradient method

- **GLPK** [14] je komunitním balíčkem procedur napsaných v jazyce ANSI C pro řešení úloh LP, MIP. Dostupnými algoritmy jsou primární a duální verze simplexové metody, metoda vnitřního bodu a metoda branch-and-cut. Jednotlivé procedury lze použít přímo, balíček však také obsahuje nástroje na zpracování modelovacího jazyka *GNU MathProg* - verze jazyka AMPL. Dostupná je také verze pro akademické použití.
- **GUROBI** [15] je komerční řešič pro problémy typu LP, QP, MILP, QCP, MIQP a MIQCP[10] s rozhraním pro programovací jazyky a nástroje C++, Java, Python, MATLAB, R a platformu .NET, dále také pro modelovací jazyky AIMMS, AMPL, GAMS a MPL.
- **LINDO** [16] je řešič na pozadí modelovacího jazyka LINGO pro řešení problémů IP, LP, NLP, stochastického programování a hledání globálního optima. Pro řešič LINDO také existuje API pro programovací jazyky C/C++, C# a Java a dále rozhraní pro Matlab a rozšíření „What's best" pro Excel, umožňující používat řešič přímo z těchto nástrojů. Jedná se o komerční software, některé verze samotného řešiče jsou však dostupné i pod výukovými licencemi.

1.5.2 Knihovny pro vyšší programovací jazyky

Pro řadu vyšších programovacích jazyků existují knihovny, které umožňují řešit optimalizační úlohy prostřednictvím API v daném jazyce. Samotné řešení buď sestává z volání knihoven řešičů nějakého nižšího jazyka, nebo jsou optimalizační metody implementované přímo v rámci jazyka samotného API. Využít je lze např. jako součást větších informačních či řídicích systémů, kde může být uživatelský přístup k optimalizačním nástrojům přizpůsobený konkrétnímu problému či oblasti použití.

Java

- **JOptimizer** [17] je open-source knihovna pro řešení minimalizačních konvexních úloh. Je implementovaná čistě v jazyce Java, s cílem snadnějšího použití v rámci prostředí JavaEE serverů. Z algoritmů nabízí metody Newtonova typu, dále metody vnitřního bodu, a to bariérovou metodu a primární-duální metodu.
- **JOM**[11] [18] je open-source knihovna (pod LGPL[12] licencí) určená pro

[10] quadratic constrained programming, mixed integer quadratic programming a mixed integer quadratic constrained programming

[11] zkratka pro Java Optimization Modeler

[12] GNU Lesser General Public License

výuku a výzkum. Model se specifikuje v syntaxi podobné MATLABu, kterou zpracuje vestavěný interpreter. Pro samotné výpočty vyžaduje uživatelem předinstalované řešiče GLPK nebo CPLEX pro MILP problémy a IPOPT pro nelineání problémy.

Python

- **scipy.optimize** [19] je souborem numerických metod pro účely optimalizace a je součástí většího ekosystému vědeckého a open-source softwaru ScyPy, vedle např. NumPy pro operace s maticemi a lineární algebru a Matplotlib pro snadnou vizualizaci dat. Samotné scypy.optimize poskytuje celou řadu algoritmů pro vázané a nevázané extrémy, jako jsou Nelder-Mead, Newtonovy sdružené gradienty, metoda BFGS[13], nebo SLSQP[14]. Dále stochastické metody, jako je např. metoda diferenční evoluce, nebo speciální metody pro minimalizaci součtu nejmenších čtverců, či metody pro hledání kořenů soustav nelineárních rovnic.
- **PYOMO** [20] je open-source softwarový balíček pro modelování a řešení široké řady úloh LP, NLP, MILP a MINLP, dále úloh stochastického programování a úloh MPEC[15]. Zadávání modelu je koncipováno formou modelovacího jazyka v rámci samotného prostředí jazyka Python. Samotné řešení modelů je zprostředkováno pomocí rozhraní s řešiči ASL, CBC, CPLEX, GLPK, GUROBI a PICO.

1.5.3 Optimalizační nástroje výpočetních systémů

Excel V rámci programu Excel je dostupný nástroj Řešitel (Solver) [21], který umožňuje aplikaci optimalizačních algoritmů na model popsaný v rámci prostředí programu. Tento přístup není vhodný pro větší problémy, protože data musí být zadaná přímo v Excelu, což znemožňuje použití řídkého formátu dat, běžného pro reálné problémy. Nabízenými metodami pro řešení jsou GRG pro NLP problémy a simplexová metoda pro LP problémy. Řešitel dále nabízí alternativu v podobě evolučního algoritmu.

MATLAB Pro řešení optimalizačních úloh v jazyce prostředí MATLAB existuje rozšíření *Optimization Toolbox*, které obsahuje funkce pro řešení problémů LP, MILP, QP a NLP. Model je popisován přímo v jazyce MATLABu pomocí symbolických výrazů a výstupní data jde dále MATLABem zpracovat a vizualizovat.

[13] Broyden–Fletcher–Goldfarb–Shanno algoritmus
[14] Sequential Least Squares Programming
[15] mathematical programs with equilibrium constraints

1.5.4 Modelovací jazyky

Pro popis složitějších problémů a rozsáhlejších úloh slouží algebraické modelovací jazyky. Model, popsaný pomocí konstruktů blížících se jeho matematickému zápisu, je přeložen na vstup pro jednotlivé řešiče, čímž je oddělena samotná logika problému od implementace jeho řešení.

Jako příklady modelovacích jazyků lze uvést jazyk AMPL [23], dále jazyk LINGO [16], postavený okolo řešiče LINDO, systém AIMMS [24], poskytující kromě modelovacího jazyka také grafické rozhraní a systém GAMS [10], na němž bude problematika modelovacích jazyků popsána podrobněji. Všechny uvedené jazyky umožňují načítání uživatelských dat z datových souborů, excelovských tabulek, či z nějakého databázového systému a také poskytují přístup k modelovacímu procesu z několika vyšších programovacích jazyků. Kromě deklarativního popisu modelu také obsahují nějakou formu imperativního stylu programování, jako jsou cykly, logické větvení, anebo možnost uživatelských procedur.

GAMS Pro popis modelu obsahuje modelovací jazyk GAMS několik základních konstruktů [10]:

- Příkaz `Set` umožňuje nadefinovat sadu množin, na kterých jsou pak zavedené parametry a proměnné modelu. Může se jednat například o množiny výrobních materiálů a výrobků, či o seznam měst pro nějakou logistickou úlohu.

- Pomocí příkazů `Parameter` a `Variable` se definují parametry a proměnné. Ty mohou být závislé na některé ze `Set` množin a představovat tak její vlastnosti, např. vektor dostupných množství materiálů, nebo vektor proměnných, představující počet vyrobených výrobků. Závislost na více `Set` množinách popisuje matici[16] vztahů mezi nimi, např. matici hran mezi jednotlivými městy síťové logistické úlohy, případně lze uvažovat obecnější vícerozměrné struktury. Pokud parametr, resp. proměnná, nezávisí na ničem, jedná se o konstantu, resp. skalární proměnnou, označující např. hodnotu účelové funkce.

- Jednotlivá omezení modelu se definují pomocí příkazu `Equation` a následně se seskupí příkazem `Model`. Řešení samotného modelu se pak provede příkazem `Solve`, u kterého se navíc specifikuje účelová funkce a typ řešeného modelu.

[16] pro snažší zadávání matic lze také využít příkaz `Table`

- Výpis řešení lze provést buď v základním formátu příkazem `Display`, anebo ve vlastní režii příkazem `Put`.

Dále je uvedena ukázka jednoduchého zdrojového souboru, který popisuje model z počátečního příkladu 1.2. Symboly `=e=` resp. `=l=` představují rovnost $=$ resp. nerovnost $\leq$, příkaz `sum(j, ...)` představuje sumu přes množinu `j` z výrazu v druhém parametru. Příkaz `Positive` u zavedení proměnné ji definuje jako nezápornou.

```
Sets j / stul, zidle /, i / drevo, komp /;

Table A(i,j)
         stul zidle
drevo    3    1
komp     1    1    ;

Parameters
b(i) / drevo 30, komp 0 /
c(j) / stul 1100, zidle 400 /;

Variable z;
Positive Variable x(j);

Equations ucel_fce, soustava(i);

ucel_fce ..      z =e= sum(j, c(j) * x(j));
soustava(i) ..   sum(j, A(i,j) * x(j)) =l= b(i);

Model priklad / all /;

Solve priklad maximize z using LP;
Display z.l, x.l;
```

Kapitola 2

Paralelizace

2.1 Multitasking a multithreading

Připomeňme zprvu, že *Programovatelný počítač* (angl. *stored-program computer*) je takový počítač, který má uložené své *strojové instrukce* v elektronické paměti. S konceptem programovatelného počítače přišel jako první v roce 1945 matematik John von Neumann. V jeho modelu se program skládá z posloupnosti instrukcí, které se postupně sekvenčně vykonávají. Tento tradiční způsob běhu programu zhruba odpovídá výkonu instrukcí na jednom *vlákně* (angl. *thread*). [25]

V dnešních operačních systémech se však s jedním vláknem vystačí jen velmi těžko. Běžný uživatel očekává např. možnost být připojen na internet a k tomu mít současně otevřených několik programů, navíc sám operační systém může na pozadí spouštět procesy např. antivirového software nebo sběru telemetrických dat. Díky pokrokům v technologickém vývoji však umožňují moderní procesory běh více vláken a tedy i procesů zaráz. Běhu více procesů na jednom stroji se říká *multitasking* [26]. Z vlákna se tedy stává jakási základní stavební jednotka, ze které je vystavěn celý běh operačního systému a jeho procesů. Každému procesu odpovídá alespoň jedno vlákno, tzv. *hlavní vlákno* (angl. *main thread*) [26]. Jen u jednoho vlákna však nemusí zůstat. Je běžné, aby program vykonával naráz několik odlišných činností, např. internetový prohlížeč přehrávající streamované video může zvlášť zpracovávat informaci o obraze a o zvuku a k tomu ještě vykreslovat ostatní prvky zobrazované webové stránky a reagovat na uživatelské události. Situaci, kdy jeden program/proces vykonává svou činnost na více vláknech, se říká *multithreading* [26].

Multithreading i multitasking lze realizovat na procesorech s jediným jádrem, vzhledem k důvodům popsaným dále se ale běžně užívá procesorů

vícejádrových. [25, 26]

2.1.1 Vícejádrový procesor

Vícejádrový procesor je takový procesor, který obsahuje dva a více nezávislých procesorových prvků, většinou označovaných jako jádra. [25] Jaký je vlastně důvod použití více jader? Nestačí pouze zvyšovat taktovací frekvenci jediného jádra a tím dostát rostoucím požadavkům na výpočetní výkon? Jedním z problémů jsou zvětšující se rozdíly mezi procesorovými a paměťovými rychlostmi, dále také komplikace při paralelismu na úrovni procesorových instrukcí. Hlavním důvodem, který firmu Intel přiměl v roce 2004 upustit od jednojádrové architektury ve prospěch procesorů vícejádrových, je však problém neúměrně narůstajících energetických a tepelných nároků procesoru vzhledem ke zvyšování jeho frekvence.[27]

Z energetického hlediska lze na strukturu procesoru pohlédnout zjednodušeně jako na systém sestávající z mnoha kapacitorů o celkové kapacitě C, které jsou periodicky nabíjeny a vybíjeny s frekvencí f pomocí napájecího napětí U. Celkový ztrátový výkon P procesoru jde tedy přibližně vyjádřit jako

$$P = CU^2 f. \tag{2.1}$$

Nabíjení a vybíjení kapacitorů procesoru je nezbytné pro provádění výpočetních operací a chod samotného procesoru. Frekvence f přímo odpovídá jeho výpočetnímu výkonu. Při rozdělení procesoru na víc jader lze frekvenci f pro jednotlivá jádra snížit a stále tak zachovat požadovaný výpočetní výkon. Dále každé hodnotě napětí U odpovídá nějaká maximální frekvence f, která je ještě pro dané napětí realizovatelná, potřebné napětí tedy lze pro vícejádrový procesor snížit.[27]

2.1.2 Paralelizace versus konkurence

U problematiky běhu programu na více vláknech, je dobré zdůraznit rozdíl mezi paralelními a konkurentními výpočty.

- *Konkurence* obecně znamená zpracovávání instrukcí na více vláknech. Reálně však mohou být všechna tato vlákna klidně svázána s jediným fyzickým vláknem (např. jádro procesoru). V tomto případě je mezi jednotlivými vlákny v rychlém sledu přepínáno, dochází k tzv. *změně kontextu*, a vykonávají se vždy pouze příkazy vlákna aktivního, zatímco ostatní musí čekat. Při změně kontextu se načítá a ukládá stav procesoru, tedy každé vlákno pracuje pouze se svým stavem, jedná se však

o výpočetně náročnou operaci. Konkurence tedy vytváří pouze jakousi iluzi paralelního běhu procesů.[25, 27]

- *Paralelizace* nastává tehdy, když jsou vykonávány instrukce na více fyzických vláknech (více jádrech). Jednotlivá vlákna tedy skutečně pracují zároveň a jejich instrukce mohou být prováděny po celý čas trvání výpočtu.[25, 27]

Reálně se používá kombinace obou dvou přístupů, kdy procesor obsahuje několik jader, ale probíhajících procesů je mnohem více a musejí se tedy v rámci jednotlivých jader vykonávat konkurentně.

2.2 Zrychlení pomocí paralelizace

Jak již bylo nastíněno dříve, pralelizace programu může zkrátit jeho celkovou výpočetní dobu. Jedním způsobem, jak tuto vlastnost popsat, může být porovnání výpočetní doby pro nejlepší sekvenční algoritmus pro daný problém oproti době paralelního řešení. Takové zrychlení S potom určíme jako

$$S(n) = \frac{T_s}{T_p(n)}, \tag{2.2}$$

kde T_s je doba výpočtu pomocí nejlepšího sekvenčního algoritmu a T_p doba výpočtu při implementaci paralelizace, závislá na počtu fyzických vláken n v ní použitých.[25]

2.2.1 Amdahlův zákon

Vzhledem k závislosti (2.2) na počtu fyzických vláken přirozeně vyvstává otázka, jaké je maximální zrychlení S daného programu, vzhledem k zvyšování počtu fyzických vláken n. Toto teoretické maximální zrychlení vlivem paralelizace popisuje *Amdahlův zákon*. Pro program, jehož část P je zrychlena paralelizací (zrychlení S_P) a zbylá část $1 - P$ zůstává sekvenční, určíme celkové zrychlení takto [25]

$$S = \frac{1}{(1 - F) + (\frac{F}{S_P})}. \tag{2.3}$$

Například pokud je zrychlena třetina programu paralelizací o 50 procent, tak bude dosaženo celkového zrychlení

$$S = \frac{1}{(1 - 1/3) + \frac{1/3}{1,5}} = 1,125,$$

tedy celkového zrychlení o jednu osminu.

Předpokládejme, že počet použitých jader n přímo odpovídá paralelnímu zrychlení[1]

$$S(n) = \frac{1}{(1-P) + (\frac{P}{n})}. \tag{2.4}$$

Z tohoto výrazu, pro n blížící se limitně k nekonečnu, pak dostáváme teoretické maximální zrychlení S_{max} jako

$$S_{max} = \lim_{n\to\infty} S(n) = \frac{1}{1-P}. \tag{2.5}$$

Tedy jestliže program stráví v sériové části kódu např. 10 procent celkového času, pak celkové zrychlení pomocí paralelizace je nejvýše desetinásobné, nehledě na počtu použitých fyzických vláken. Paralelní programování je tedy především výhodné pro programy s velkým časem P stráveným v paralelním kódu. [25]

V tomto znění Amdahlova zákona se předpokládá, že lze počet fyzických vláken libovolně navyšovat, toho je však v realitě velmi obtížné dosáhnout. Vlivem synchronizace a jiné komunikace mezi jednotlivými vlákny programu, běžné ve většině rozsáhlejších reálných aplikací, dochází k časovým ztrátám. Pokud do (2.4) přidáme prvek $H(n)$ zahrnující režii (angl.*overhead*) pro výše popsanné komplikace a režii samotného systému, dostáváme zrychlení pro reálné systémy [25]

$$S(n) = \frac{1}{(1-P) + (\frac{P}{n}) + H(n)}. \tag{2.6}$$

Pokud je faktor $H(n)$ velký, může převýšit výkonostní zisk z paralelizace a pro dostatečně velké hodnoty může být dokonce výsledné zrychlení S menší než jedna (dojde tedy ke zpomalení). Proto je důležité navrhovat aplikace tak, aby byla cena paralelizace co možná nejmenší. [25]

2.3 Multithreading v programovacím jazyce Java

Java je silně typovaný objektově orientovaný programovací jazyk, vyvíjený tak, aby byl co nejméně závislý na iplementaci. Aplikace psané v jazyce Java se kompilují do javového bytecodu, který může být poté interpretován na

[1] např. pro dvě jádra bude výpočetní čas poloviční a zrychlení tedy dvojnásobné

libovolném virtuálním stroji pro jazyk Java (JVM) a to nezávisle na architektuře systému. K vláknům konkrétního systému se tedy přistupuje přes nějakou objektově orientovanou abstrakci, která je na pozadí realizována skrze JVM.

2.3.1 Třída Thread

Každé vlákno aplikace je v Javě reprezentováno instancí třídy `java.lang.Thread` [28], nebo jednou z jejích podtříd. Program začíná s jedním *hlavním vláknem* v metodě `main`. Hlavní vlákno může vytvářet vlákna další pomocí instanciace třídy `Thread`, kdy se do konstruktoru předá instance rozhraní `java.lang.Runnable`, představující kód (úlohu), kterou má vlákno vykonat. Od verze jazyka Java 8 lze použít pro vytvoření instance `Runnable` funkcionální *lambda* notaci, a to přímo v konstruktoru vlákna, viz následující příklad.

```
Thread thread = new Thread( () -> {
        // Kód, který se má provést
        System.out.println("Hello world in another thread!");
        ...
});
```

Další možností, jak specifikovat prováděný kód, je rozšířit samotnou třídu `Thread` a překrýt její metodu `run`, která je zodpovědná za provedení samotného kódu.

Životní cyklus vlákna lze ovlivňovat několika metodami třídy `Thread`. [28]

`start`: Spustí běh vlákna. Vlákno nemůže být spuštěno vícekrát, opakované volání na stejné vlákno vznese výjimku.

`Thread.sleep`: Přeruší běh aktuálního vlákna na dobu specifikovanou v argumentu. Volání této metody se provádí na statický kontext třídy `Thread` přímo z vlákna, které se má uspat.

`join`: Čeká na ukončení jiného vlákna. Používá se např. tehdy, pokud potřebujeme posbírat výsledky nějakého paralelního výpočtu. Vlákno odpovědné za zpracování výsledků čeká pomocí `join` na vlákna výpočetní a až po jejich ukončení začne zpracovávat výsledky.

Vlákno lze vytvořit jako tzv. *daemon* vlákno. Toto je důležité pro ovlivnění ukončení běhu celého programu, protože aplikace je ukončena za jedné ze dvou podmínek. Byla zavolána metoda `Runtime.exit`, nebo všechna vlákna, která nejsou daemon, skončila. Vytvořením daemon vlákna se tedy signalizuje JVM, že jeho aktivita nemá blokovat konec celé aplikace. [28]

2.3.2 Synchronizace a její úskalí

Synchronizace je způsob, jak upravit pořadí vykonávání jednotlivých vláken a umožňuje řešit konfliktní situace mezi nimi, které by mohly vést k nechtěnému chování programu. Pomáhá např. koordinovat běh vláken nebo zabezpečit přístup ke sdíleným datům a vyhnout se tak chybám z důvodu nekonzistence paměti. [25]

Typ synchronizace použitý v jazyce Java je založený na principu *vzájemné exkluze* [25], kdy jedno vlákno zablokuje přístup k nějaké *kritické sekci* kódu a ostatní vlákna, pokoušející se danou sekci provést, musejí čekat, dokud ji blokující vlákno neopustí. Tím je zaručeno, že kód kritické sekce je prováděn nejvýše jedním vláknem ve stejnou dobu. Například pokud jedno vlákno pracuje s prvky nějaké kolekce (seznamu, mapy atd.) a jiné vlákno se pokusí některé prvky z ní odstranit, může dojít v případě kolize k nepředvídatelným výsledkům. Pokud je však přístup k dané kolekci synchronizován, tak se těmto problémům předejde, ale za cenu ztráty výpočetního času čekajícího vlákna.

Ochrana kritické sekce je zprostředkována pomocí synchronizačního prvku zvaného *zámek* (angl. *lock*, někdy také *mutex* či *monitor*) [25, 29]. Pokud chce vlákno vstoupit do synchronizovaného bloku chráněného zámkem, musí tento zámek nejprve *získat*. Teprve pak může pokračovat v kódu uvnitř chráněného bloku a při jeho opuštění zámek zase *uvolní*. Po dobu mezi získáním a uvolněním zámku říkáme, že vlákno zámek *vlastní*, a tedy jej nemůže jiné vlákno získat. [25, 29] V jazyce Java je s každým objektem asociovaný jeho vlastní zámek, pro zamknutí chráněného bloku se tedy používá přímo reference objektu, jehož zámek je vyžadován. Buď lze vytvořit nový objekt pouze za cílem synchronizace, anebo jako zámek použít přímo objekt, který je potřeba chránit, jako v následujícím příkladu. [29]

```
// Seznam referencovaný proměnnou 'list'
// je použit jako zámek synchronizovaného bloku
synchronized(list) {

        // Uvnitř bloku se provede v daném programu
        // nebezpečná operace odstranění pruku ze seznamu
        list.remove(...);
}
// Po opuštění bloku je zámek zase uvolněn
```

Synchronizovat lze také celou metodu přidáním klíčového slova `synchronized` do její hlavičky. Celé tělo metody se pak chová jako synchronizovaný blok, kde zámkem je ten objekt, jehož synchronizovaná metoda je

volána. [29]

Stejný zámek lze použít ve více synchronizovaných blocích zároveň, kdy stále platí stejné pravidlo: zámek může vlastnit pouze nejvýše jedno vlákno. Pokud vlákno narazí na blok chráněný zámkem, který již vlastní, může pokračovat normálně dál. Tímto způsobem lze dosáhnout pokročilejší logiky synchronizace kódu, s tím však mohou být spjaty některé komplikace. [29]

Nejběžnějším synchronizačním problémem je deadlock. Jakožto *deadlock* [25, 29] chápeme situaci, kdy dvě vlákna navzájem čekají na své zámky, které však nikdy nemohou uvolnit, a zůstanou tak čekat navždy. Jak k deadlocku dochází, lze popsat následovně:

- Vlákno A vstoupí do synchronizovaného bloku a získá jeho zámek l_1.
- Vlákno B vstoupí do jiného synchronizovaného bloku a získá jeho zámek l_2.
- Uvnitř tohoto bloku narazí B na další blok, chráněný zámkem l_1, ten již však vlastní A, a B tedy čeká.
- Vlákno A, ještě před opuštěním původního bloku (a uvolněním l_1), narazí na blok chráněný zámkem l_2. V tuto chvíli je zřejmé, že došlo k deadlocku, protože ani jedno z vláken není schopné uvolnit zámek, který vlastní a na který čeká vlákno druhé.

Deadlock je velmi obtížné odhalit, zvláště u rozsáhlejších aplikací. Nejedná se totiž o chybu jako takovou, při které by došlo k vznesení výjimky, která by pomohla odhalit problémové místo, či alespoň upozornit na to, že vůbec k něčemu došlo. [29]

2.3.3 Pokročilé API

Výše popsané prostředky pro práci s více vlákny se doporučuje používat jako takové pouze v programech menšího rozsahu. [29] Pro realizaci složitějších konkurentních systémů a řešení problémů s nimi spojených nabízí Java pokročilejší nástroje, především sadu tříd a rozhraní nacházejících se v balíčku `java.util.concurrent`. Tyto pomocné třídy většinou skrývají základní synchronizaci a práci s vlákny pod vyšší vrstvu abstrakce, a tím umožňují přehledněji a bezpečněji řešit daný problém. [29]

Rozhraní `java.util.concurrent.locks.Lock` a jeho implementace jsou alternativou k synchronizačnímu bloku. Kromě základních akcí, prováděných pomocí metod `lock` a `unlock`[2] nabízí `Lock` také např. metodu `tryLock`, která,

[2]ekvivalenty získání a uvolnění zámku syncronizovaného bloku

v případě, že zámek není volný, čeká na jeho uvolnění pouze stanovenou dobu, nebo na něj nečeká vůbec. Rozhraní `...locks.ReadWriteLock` zase umožňuje jemněji rozlišovat mezi jednotlivými kritickými sekcemi v kódu. `ReadWriteLock` obsahuje, dvojici zámků `read lock` a `write lock`, kdy zámek pro čtení může získat více vláken naráz, ale pouze pokud není držen zámek pro zápis. Zámek pro zápis může být držen zároveň pouze jedním vláknem. Pomocí `ReadWriteLock` lze elegantně vyřešit častý problém synchronizovaného přístupu ke sdíleným zdrojům, například k nějakému seznamu dat. Čtení ze seznamu více vlákny naráz není problematické, ale zápis již může být. [28]

Pro složitější práci s vlákny, pro jejich vytváření a spravování jejich běhu slouží implementace rozhraní `...concurrent.Executor`. Rozhraní samotné obsahuje pouze jednu metodu, a to `execute`, která v argumentu přijímá `Runnable`, představující úlohu, která se má někdy v budoucnu vykonat. Způsob a čas takového vykonání již záleží na jednotlivých implementacích[3]. Pomocná třída `...concurrent.Executors` poskytuje mj. několik statických metod pro vytváření základních implementací rozhraní `Executor`. [28]

> `newFixedThreadPool`: Exekutor s pevně daným počtem vláken[4], které mají dané úlohy vykonávat. Pro nově příchozí úlohu není vytvořeno vlákno nové, místo toho je jí přiřazeno nějaké již existující pracovní vlákno, které zrovna nic neprovádí. Pokud jsou všechna pracovní vlákna zaneprázdněná, úloha je zřazena do fronty, kde čeká na uvolnění některého z vláken.
>
> `newSingleThreadExecutor`: Stejné jako předchozí, všechny úlohy se však vykonávají pouze na jediném vlákně. Využíván je tedy pouze mechanizmus fronty úloh.
>
> `newScheduledThreadPool`: Exekutor umožňující spouštět danou úlohu periodicky pro definovaný časový interval nebo po uplynutí stanovené doby, nebo obojí.

Další užitečnou implementací `Executor` je `...concurrent.ForkJoinPool`, který společně s `...concurrent.ForkJoinTask` umožňuje efektivně řešit situace, kdy úlohy během své činnosti vytváří další podúlohy. Z vlákna spravovaném instancí `ForkJoinPool` lze na úlohu typu `ForkJoinTask` zavolat její metodu `fork` pro přidání této úlohy do zastřešujícího `ForkJoinPool`. Následným voláním `join` lze počkat na výsledek. Voláním statického `ForkJoinTask`

[3]Striktně vzato nemusí vůbec dojít k asynchronnímu běhu úlohy. Ta může být vykonána přímo na tom vlákně, které metodu `execute` zavolá.

[4]Vlákna nemusí být vytvořena všechna naráz, většinou vznikají nová až podle potřeby.

`.invokeAll` lze spustit všechny úlohy specifikované v argumentu a počkat na jejich dokončení. V tomto případě může být volající vlákno použito pro běh jedné z těchto úloh, takže není zbytečně plýtváno omezeným počtem vláken, které jsou k dispozici. V opačném případě by volající vlákno pouze nečinně čekalo na dokončení jednotlivých úloh. [28]

Poznamenejme, že na poznatky z oblasti paralelizace uvedené v této kapitole bude navázáno v praktické části této práce.

Část II

Praktická část

Kapitola 3

Program ParallelGams

Jak již bylo zmíněno dříve, z Amdahlova zákona (2.5) vyplývá, že paralelizaci je dobré, za účelem zrychlení výpočtu, aplikovat především na problémy s velkým podílem paralelizovatelné části. Jako výborným kandidátem pro takovéto zrychlení je problém analýzy citlivosti konkrétního optimalizačního modelu. Jednotlivé testované scénáře sdílí stejný model, s rozdílnými parametry modelu pro různé scénáře, jejich samotné řešení však na sobě vzájemně nijak nezávisí. Pro tento případ byl napsán v jazyce Java program pro paralelní řešení modelů popsaných v modelovacím jazyce softwaru GAMS.

3.1 Funkce programu

ParallelGams je konzolová aplikace sloužící pro paralelní spuštění řešení konkrétního modelového souboru GAMS (soubor s příponou .gms) s různými vstupními daty, specifikovanými ve vlastních modelových souborech. Datové soubory se do hlavního modelu přidávají pomocí příkazu `$include`, kód modelu tedy musí obsahovat vhodně[1] umístěné následující dva řádky.

```
$if not set incdata $abort
        'no include file name for data file provided'
$include %incdata%
```

Cesta ke konkrétnímu datovému souboru je programem ParallelGams nastavena do kompilační proměnné[2] `'incdata'`. První řádek kódu kontroluje, zda-li byla proměnná `'incdata'` řádně nastavena, v opačném případě zastaví kompilaci modelu. Druhý řádek připojí k hlavnímu modelu programem

[1] většinou pod deklarací měněných parametrů

[2] viz https://www.gams.com/latest/docs/UG_GamsCall.html
#UG_GamsCall_DoubleDashParametersEtc_CompileTimeVars

specifikovaný soubor s konkrétními daty. Programu lze nastavit pomocí konfiguračního souboru `.properties` cestu k hlavnímu modelovému souboru, cestu k adresáři, obsahujícímu soubory pro jednotlivé scénáře, a počet vláken, na kterých má paralelizace probíhat. Důležitým nastavením je také cesta ke kořenové složce distribuce samotného programu GAMS.

3.2 Popis implementace

Pro snadnější komunikaci se softwarem GAMS a práci s jeho modely existuje pro jazyk Java oficiální API[3] obsažené v archivu GAMSJavaAPI.jar, který je součástí distribuce programu GAMS. Obsah tohoto archivu je pak přibalen do výsledného jaru k hlavnímu programu ParalelGams jakožto knihovna.

Před tím, než program přejde k samotnému řešení modelu, provede několik akcí:

1. Načte uživatelské nastavení ze souboru `.properties`.
2. Přidá nativní `.dll` knihovny, potřebné pro komunikaci s GAMS, do Java classpath, aby je mohl JVM nalézt a načíst.
3. Vytvoří instanci třídy `com.gams.api.GAMSWorkspace`, která slouží jako výchozí bod pro další práci s GAMS API.

Po úspěšném provedení těchto kroků se provede samotná paralelizace modelu, viz výňatek ze zdrojového kódu[4] na následující straně.

Popis kódu:

1 Vytvoření exekutoru s požadovaným počtem vláken.

2-8 Vytvoření seznamu všech datových souborů v uživatelem zadané složce.

9-10 Pro každý soubor z tohoto seznamu se exekutoru předá nová paralelní úloha.

14-15 V rámci této úlohy se vytvoří instance typu `GAMSJob` pro hlavní model,

16 dále se modelu nastaví kompilační proměnná `incdata` na konkrétní datový soubor

25 a nakonec se provede řešení modelu.

[3] API existuje také pro další programovací jazyky, např. python nebo C#

[4] ve výňatku je vynechán výpis do konzole a ochranné `try...catch` bloky, které jsou podružné pro samotnou funkci programu; celý zdrojový kód je dostupný v CD příloze této práce

29-30 Čekání hlavního vlákna na vyřešení všech modelů.

```
ExecutorService pool = Executors.newFixedThreadPool(threads);

Files.find(dataDir, Integer.MAX_VALUE, (path, attr) -> {
    return attr.isRegularFile()
    && !attr.isDirectory()
    && path.getFileName().toString()
            .endsWith(GAMSGlobals.GAMS_FILE_EXTENSION);
    })
    .forEach(path -> {
        pool.execute(() -> {
            String name = path.getFileName().toString();
            name = name.substring(0, name.length()-4);

            GAMSJob job =
                ws.addJobFromFile(props.getProperty(MODEL), name);
            GAMSOptions opt = ws.addOptions();

            opt.defines(
                "incdata", path.toAbsolutePath().toString());

            Path dir = path.subpath(0, path.getNameCount() - 1);
            opt.setPutDir(dir.toAbsolutePath().toString());
            opt.setOptDir(dir.toAbsolutePath().toString());

            job.run(opt);
        });
    });

pool.shutdown();
pool.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
```

Program průběžně vypisuje údaje o délce řešení jednotlivých modelů a nakonec zobrazí i celkový čas běhu hlavní části programu (paralelizace modelu).

3.3 Běh programu na testovacích datech

Jako testovací model byl použit obecný problém lineárního programování s 1000 proměnnými a 100 omezeními (viz Kapitola 1). Měnícími se parametry byly matice soustavy $\mathbf{A}$, vektor pravé strany $\mathbf{b}$ a vektor vah proměnných $\mathbf{c}$.

Bylo vygenerováno 20 datových souborů, kde jsou hodnotami uvedených parametrů náhodná celá čísla z intervalu $(-50, 50)$. Počet vláken paralelizace byl postupně zvyšován z 1 na 10 a pro každou hodnotu bylo provedeno 5 měření. Program byl testován na běžném laptopu s operačním systémem Windows 10 (64bit). Počítač je vybaven procesorem Intel Core i5-7200U se čtyřmi jádry s taktovací frekvencí 2.5 GHz s možností přetaktování až na 2.7 GHz. Velikost operační paměti RAM je 8192 MB. Průměry naměřených hodnot, spolu s jejich rozptyly, jsou uvedené v Tabulce 3.1. Pro počet vlá-

počet vláken	průměrná doba řešení [s]	σ^2
1	10.203	0.0568
2	6.1022	0.0128
3	5.0786	0.0047
4	3.8898	0.0070
5	4.0682	0.0483
6	4.0782	0.0366
7	3.9108	0.0036
8	3.88	0.0095
9	3.9788	0.0012
10	3.9458	0.0175

Tabulka 3.1: Průměrná doba řešní v závislosti na počtu vláken, kde σ^2 představuje statistický rozptyl naměřených hodnot

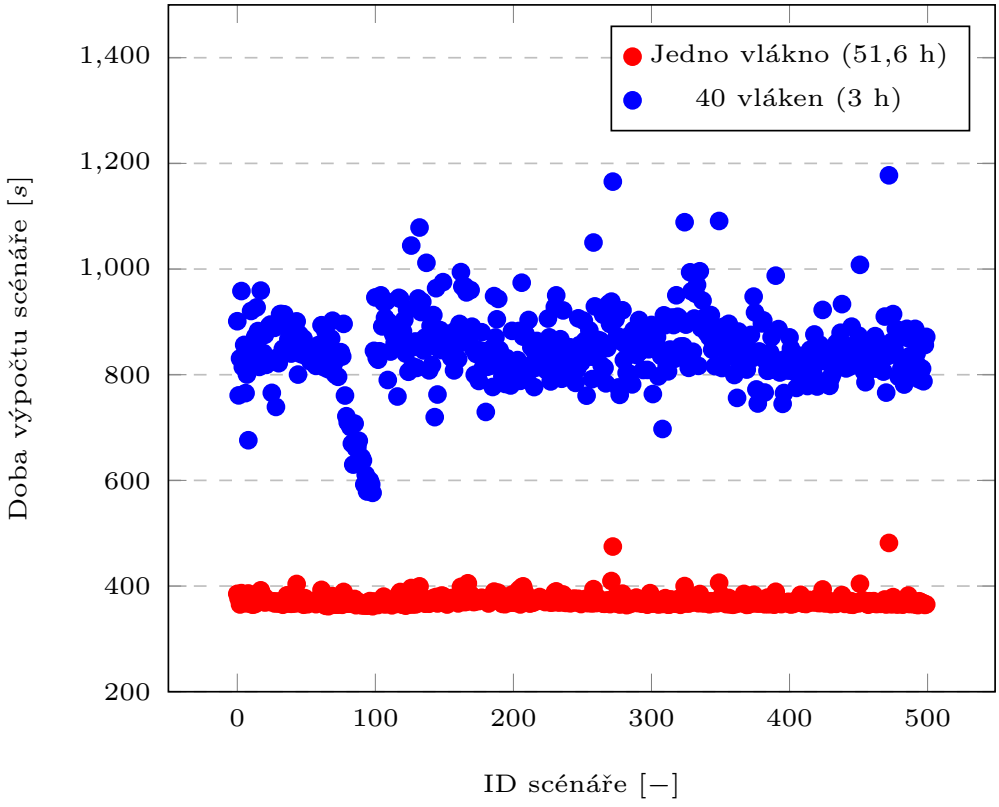
ken 1 až 4 je vidět znatelné urychlení výpočtu. Další zvyšování počtu vláken už ale žádné zrychlení nepřináší, protože fyzická vlákna (jádra) procesoru jsou již všechna využitá.

3.4 Aplikace programu na reálný problém

Ve spolupráci s Ústavem procesního inženýrství fakulty strojní Vysokého učení technického v Brně byl program dále aplikován na reálný problém. Jednalo se o testování 500 scénářů pro model PIGEON, který je součástí větší logistické úlohy pro svoz a využití odpadu řešené systémem NERUDA[5]. Výpočet probíhal na výkoném počítači výše zmíněného ústavu vybaveném procesorem Intel Xeon CPU E5-2698 v4 se 40 jádry s taktovací frekvencí 2.2 GHz a s velikostí operační paměti RAM 131072MB. Vzhledem k časové náročnosti řešení problému (přibližně 6 až 7 min na jeden scénář), byly provedeny pouze dva testy programu, a to pro počty vláken 1 a 40. Celková

[5]více informací o těchto systémech viz [30, 31, 32]

doba výpočtu byla snížena z 51,6 h (pro jedno vlákno) na 3 h při běhu na 40 vláknech. Na Obrázku 3.1 jsou v grafu znázorněny časy potřebné pro výpočty jednotlivých scénářů. Doby při paralelizaci jsou pak zhruba dvakrát tak velké, než při sekvenčním řešení celého problému.



Obrázek 3.1: Porovnání paralelního a sekvenčního výpočtu scénářů

Pro úspěšné použití programu na daný problém bylo nutné vyřešit několik technických problémů. Původní model načítal vstupní data z tabulky aplikace MSExcel v rámci jednotlivých scénářů, paralelní práce s aplikací MSExcel však, jak se ukázalo, nebyla možná. Celá tabulka tedy musela být nejprve převedena do společného .gdx souboru [10], ze kterého již byly jednotlivé scénáře schopné své parametry načíst. Dále tvorba 500ti datových souborů byla nakonec vyřešena formou automatického skriptu, stejně tak zpracování jejich výsledků. Na řešení těchto i jiných problémů navazuje další program napsaný v rámci této práce, popsaný v následující kapitole.

Kapitola 4

Program ComposedOptimization

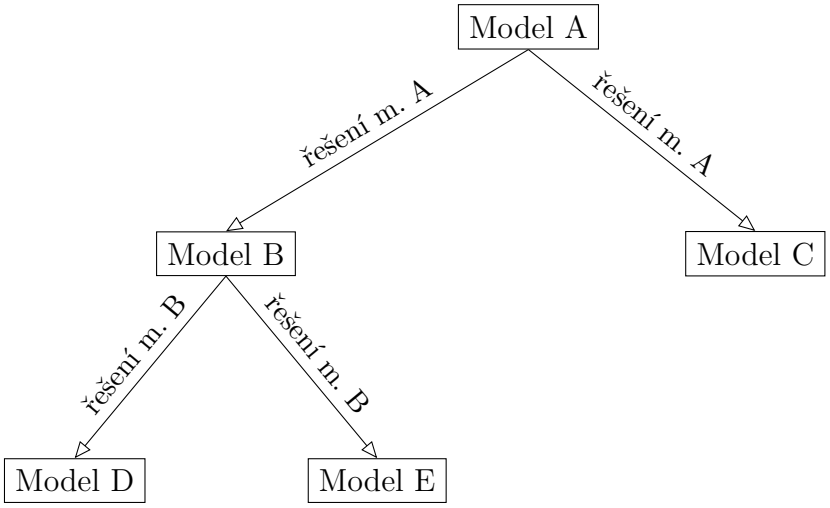
Proces řešení reálných optimalizačních úloh zahrnuje mnohem více, než jenom popis modelu a jeho následné vyřešení. Vstupní data modelu mohou být poskytnuta od zadavatele dané úlohy v řadě různých datových formátů a ty je potřeba nejprve zpracovat pro potřeby samotné úlohy a konkrétního modelovacího jazyka. Tomuto kroku se souhrnně říká *preprocessing* [33] dat. Obdobně existuje také krok *postprocessingu* [33], kdy je nutno uživateli srozumitelně prezentovat výsledky řešení nebo výstupní data předpřipravit pro další zpracování. Samotné řešení modelu se může skládat z více kroků (z více „podmodelů"), které na sobě často závisí. Problém popisu více sdružených modelů v rámci modelovacího jazyka lze vyřešit jejich sloučením do jednoho velkého modelu a sled jejich řešení popsat pomocí nějaké rozhodovací logiky daného jazyka. Výsledný model pak ale ztrácí na čitelnosti a spolupráce více lidí na jednotlivých „podmodelech" je komplikovaná.

Pro usnadnění celkového optimalizačního procesu byl napsán v programovacím jazyce Java a v jazyce Groovy program ComposedOptimization. Ten umožňuje navrhnout celou strukturu řešení (včetně pre a postprocessingu dat) v uživatelském skriptu pomocí DSL[1] verze jazyka Groovy.

4.1 Funkce programu

Řešení jednotlivých „podmodelů", zvláště v případě navazující optimalizace (viz podsekce 1.4.2), lze popsat stromovým schématem jako je např. na Obrázku 4.1. Modely nižších uzlů vyžadují řešení modelů ve stromové hierarchii nad nimi a jsou tedy řešeny postupně (např. model B čeká na řešení z A).

[1] doménově specifický jazyk; jazyk určený pro určitou oblast či problematiku



Obrázek 4.1: Schéma návaznosti řešení jednotlivých „podmodelů“

Modely na stejné úrovni na sobě nijak nezávisí a lze je tedy řešit zároveň (např dvojice modelů D a E).

Projekt programu je tedy také reprezentován stromovou strukturou, kde se kromě uzlů modelů mohou nacházet také „řídicí“ uzly, které ovlivňují průběh řešení celého modelu. Uživateli je pak poskytnuto několik typů konfigurovatelných uzlů, ze kterých vytvoří celkový strom projektu. Program začíná u kořenového uzlu a pro každý další uzel, který navštíví, provede následující dva úkony:

1. Provede akci s uzelm spojenou, pokud je specifikována[2]. Jedná se např. o provedení nějakého uživatelského kódu, nebo o spuštění řešení nějakého nadefinovaného modelu.

2. Uzel pak určí, jaké další uzly se mají navštívit. Většinou se jedná jednoduše o všechny jeho poduzly, tímto způsobem je však řešeno i logické větvení či cykly.

Navštívení dalšího uzlu je rekurzivní, jedná se tedy v základu o procházení stromu do hloubky[3].

Jednotlivé uzly a celková struktura projektu je uživatelem definována deklarativním stylem v rámci DSL verze jazyka Groovy. Kromě samotného popisu uzlů je uživateli zpřístupněn celý skriptovací jazyk Groovy, který

[2]ne všechny uzly musí mít nadefinovanou akci

[3]pokud narazí program na uzel, který jej nikam nepošle, vrátí se automaticky v zásobníku funkčních volání o úroveň výše, tzn. k předchozímu uzlu

umožňuje mj. také imperativní styl programování, který může být použit pro usnadnění procesu definování struktury projektu[4]. Pro odlišení popisných struktur od skriptovací části projektu začínají všechny „uzlové příkazy“ symbolem `$`. Následuje příklad definice uzlu:

```
$GAMS('model.gms') {
        name = 'Hlavní model'

        $GAMS('submodel1.gms')
        $GAMS('submodel2.gms')
}
```

Argumenty uzlového příkazu (v kulatých závorkách) jsou povinné parametry uzlu nutné pro jeho funkci, jako je cesta k souboru modelu v uvedeném příkladě. Následující složené závorky mohou obsahovat nastavení dalších nepovinných parametrů anebo definice poduzlů. V uvedeném příkladě se jedná o nastavení názvu hlavního modelu a definice uzlů dvou dalších submodelů.

4.1.1 Typy uzlů

$group Skupinový uzel je základním typem pro většinu ostatních uzlů. Při návštěvě přesměruje program postupně na všechny své poduzly. `$group` také umožňuje paralelní navštívení poduzlů na samostatných vláknech pomocí nastavení parametru `execution = parallel`, viz následující příklad paralelního výpočtu dvou modelů:

```
$group {
        execution = parallel

        $GAMS('model1.gms')
        $GAMS('model2.gms')
}
```

Díky tomu, že je `$group` jakýmsi „nadtypem“ většiny ostatních uzlů[5], je možnost paralelizmu poskytnuta i ostatním uzlům. Počet vláken, která má program k dispozici, lze nastavit v konfiguračním souboru `.properties` parametrem `threads`.

[4] např. cyklus vytvářející větší množství podobných uzlů

[5] s výjimkou `$later`

$later Akce, spojená s tímto uzlem, spočívá v provedení uživatelského kódu uvedeného v těle uzlu. Tím je umožněna implementace vlastních metod, např. pro pre a postprocessing dat. Uzel lze také využít jako základ pro složitější optimalizační algoritmy. Kód uvnitř uzlu je schopen přistupovat k proměnným skriptu a měnit je, viz příklad:

```
String greeting = 'Hello world'
int count = 0
$later {
        println greeting
        count++
}
```

Uzel `$later` neumožňuje specifikaci poduzlů, a tudíž ani není „podtypem“ `$group`, zatímco všechny další uvedené typy uzlů již „podtypem“ `$group` jsou.

$predicate Umožňuje „vypínat“ a „zapínat“ části stromu podle potřeby, je tedy jakýmsi ekvivalentem bloku `if` běžných programovacích jazyků. Uzel `$predicate` umožní navštívení poduzlů pouze tehdy, pokud uživatelský kód v argumentu vrátí hodnotu pravdy (`true`)[6].

```
i = 5
$predicate({return i > 10}) {
        $GAMS('model.gms')
}
```

Kód podmínky může obsahovat i více příkazů, ne jen samotný testovací výraz. Může se jednat např. o načtení výsledků nějakého předešlého modelu a vrácení hodnoty pravdy/nepravdy v závislosti na jeho hodnotách.

$repeat Slouží pro opakované navštěvování poduzlů. Pokud je v argumentu uvedena číselná hodnota, bude počet opakování pevný a to tolikrát, kolik je v parametru uvedeno, jedná se tedy o jakýsi ekvivalent cyklu `for`. Viz příklad kódu, který vypíše do konzole hodnoty 1 až 10.

[6] tato podmínka je testována pro každé navštívení uzlu

```
i = 0;
$repeat(10) {
        $later {
                println i
                i++
        }
}
```

Do argumentu lze také, podobně jako u `$predicate`, předat kód s podmínkou. Ta bude testována na začátku každého cyklu, a pokud bude vrácena hodnota nepravdy (`false`), bude celý cyklus přerušen, podobně jako u cyklu `while`. Pro testování podmínky na konci cyklu lze použít nastavení `check = after`, podobně jako u cyklu `do...while`, viz následující příklad kódu, který provádí iterační řešení modelu, dokud je rozdíl posledních dvou dosažených hodnot dostatečně velký[7]:

```
double eps = 1e-3
double diff = 0
double oldval = 0
$repeat({
        val = getval()
        diff = Math.abs(oldval - val)
        oldval = val

        return diff >= eps
}) {
        check = after

        $GAMS('iteracni_model.gms')
}
```

$GAMS Tento uzel představuje řešení nějakého modelu pomocí modelovacího jazyka GAMS. V argumentu je uvedena cesta k souboru modelu, v těle uzlu pak lze nastavit název modelu pro konzolový výpis. Pomocí parametru `options` je zpřístupněn objekt typu `GAMSOptions` z oficiálního API sloužící pro nastavení samotného řešení modelu[8]. Nastavením `print = System.out`

[7] implementace uživatelské funkce `getval` není pro potřeby příkladu důležitá, a proto není uvedena

[8] celkový výčet všech možností tohoto objektu je uvedený v oficiální dokumentaci GAMS Java API, dostupné např. na adrese: https://www.gams.com/latest/docs/apis/java/classcom_1_1gams_1_1api_1_1GAMSOptions.html

se přesměruje průběžný výstup z GAMS do konzole. Přes globální proměnnou skriptu `GAMSWrkspc` je uživateli umožněno pracovat přímo s oficiálním GAMS JAVA API prostřednictvím objektu typu `GAMSWorkspace`.

4.2 Popis implementace

Běh programu probíhá v několika krocích:

1. Program načte uživatelské nastavení z konfiguračního souboru `.properties`[9] .
2. Provede se uživatelský skript, čímž dojde k vytvoření stromu projektu.
3. Dojde k rekurzivnímu navštívení projektového stromu.

4.2.1 Generování stromu projektu

Uživatelský skript je zpracován pomocí třídy `groovy.lang.GroovyShell`, která ho zkompiluje pro další použití. Pro snadnější implementaci DSL modifikace jazyka Groovy je jako základní třída objektu, reprezentujícího skript, použita třída `groovy.util.DelegatingScript`. Takovému skriptu pak lze předat objekt, jenž slouží jako delegát pro volání metod, které nejsou základním skriptem rozpoznány[10]. Pro implementaci jednoduché DSL modifikace pak stačí vytvořit třídu, jež obsahuje všechny funkce pro daný DSL. Tu reprezentuje v programu třída `...dsl.ProjectDSL`, která, kromě DSL funkcí, také vytváří samotný strom projektu.

Těla uzlů jsou jazykem Groovy interpretována jako objekty typu `groovy.lang.Closure`, který obecně představuje nějaký „kus kódu“[11]. Pro dosažení strukturované syntaxe definice stromu je využita ta vlastnost, že pokud je v jazyce Groovy poslední argument funkce typu `Closure`, lze jej zapsat až za kulaté závorky funkčního volání. Následující dva zápisy jsou tedy ekvivalentní:

```
function(arg1, arg2, {closure code})
function(arg1, arg2) {closure code}
```

[9] objekt typu `java.util.Properties`, obsahující tyto nastavení, je ve skriptu dostupný přes globální proměnnou `prop`

[10] jakým způsobem toto funguje (a další metaprogramovací možnosti jazyka Groovy) je popsáno např. v oficiální dokumentaci na adrese: http://groovy-lang.org/metaprogramming.html

[11] jedná se o základní prvek funkcionální výbavy jazyka Groovy

Při volání některé z „uzlových“ funkcí dojde nejprve k vytvoření objektu uzlu z předaných parametrů. Poslední parametr typu `Closure`, představující tělo uzlu, je následně proveden v DSL kontextu pro typ daného uzlu. Kontexty uzlů jsou, stejně jako samotný `ProjectDSL`, třídy obsahující metody specifické pro konkrétní typ uzlu. Volání neznámých funkcí se nejdříve delegují k nim, a teprve potom do obecnějšího `ProjectDSL`. Tímto způsobem je vytvořen a nakonfigurován strom projektu. Kromě „uzlových“ funkcí jsou samozřejmě provedeny i všechny ostatní příkazy uživatelského skriptu.

4.2.2 Procházení stromu projektu

Procházení projektu je implementačně variací návrhového vzoru *návštěvník* (angl. *visitor*)[12] a lze je popsat pomocí několika kroků:

1. Návštěvník (objekt typu `...tree.TreeVisitor`) „navštíví“ uzel prostřednictvím své metody `visit`.

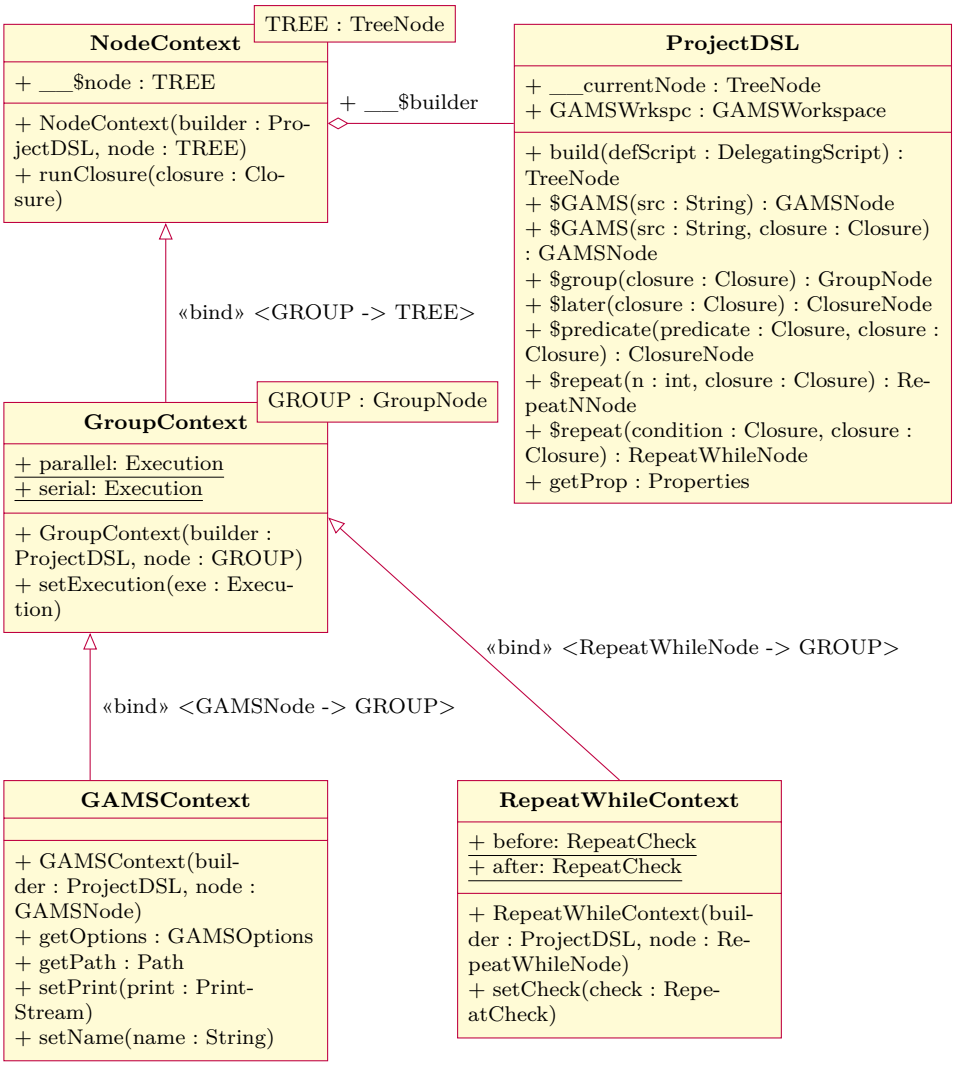
2. Pokud navštěvovaný uzel implementuje rozhraní `...tree.ActionNode`, tak je zavolána jeho metoda `run`. To představuje provedení akce spojené s daným uzlem.

3. Následným voláním metody `accept` navštěvovaného uzlu je mu předána kontrola nad tím, jaké uzly se mají navštívit dále. To obnáší další volání metody `visit`, procházení stromu je tedy rekurzivní.

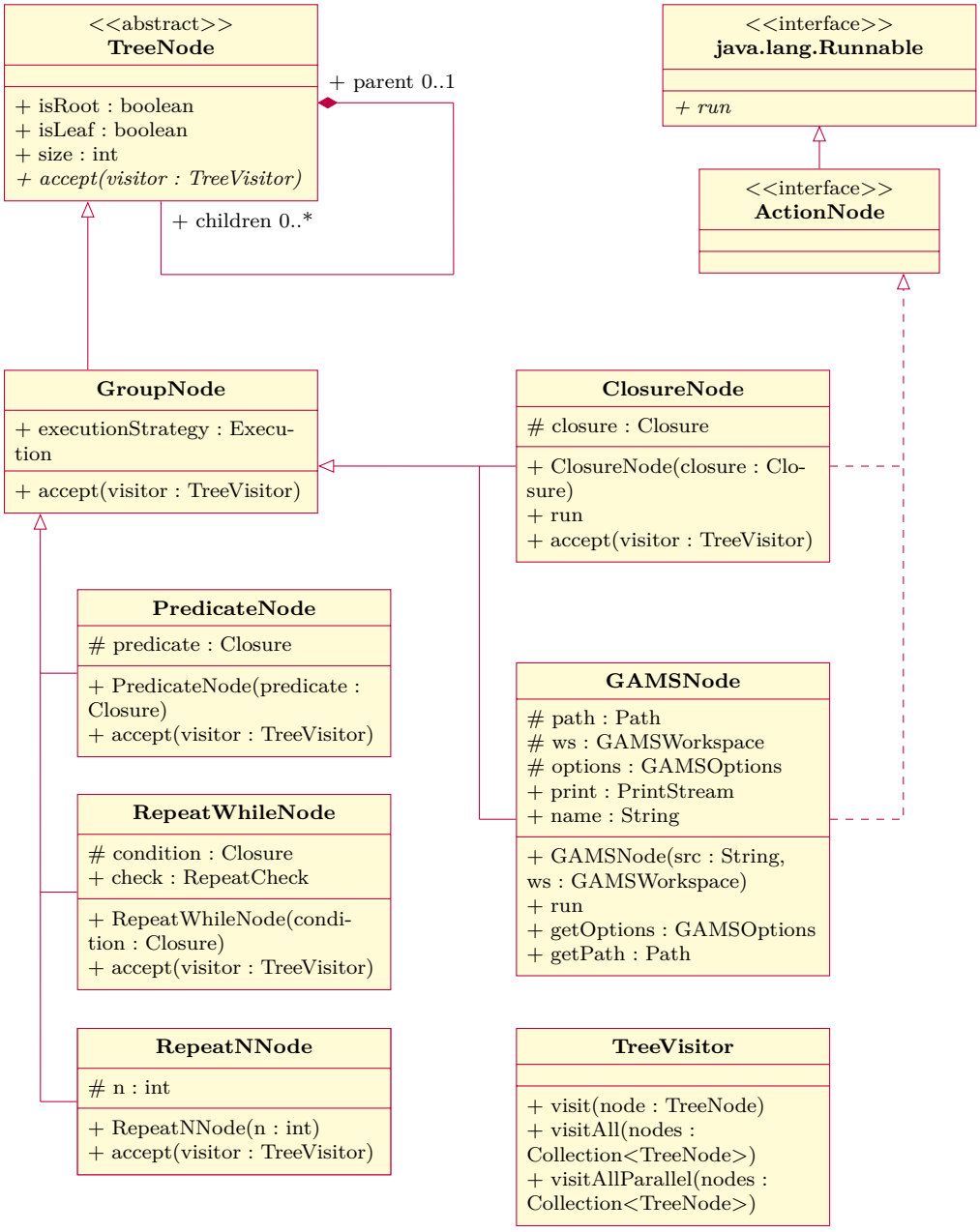
Třída `TreeVisitor` také poskytuje pro usnadnění procházení stromu dvě pomocné metody `visitAll` a `visitAllParallel`, využívané v metodách `accept` jednotlivých uzlů. Metoda `visitAll` postupně navštěvuje všechny uzly předané v argumentu. V metodě `visitAllParallel` jsou pro jednotlivé uzly vytvořeny úlohy, které jsou pak předány zastřešujícímu `ForkJoinPool`, na jehož vlákně je procházení stromu spuštěno, pro vykonání[13]. V rámci těchto úloh jsou vytvořeni noví návštěvníci, kteří následně navštíví jim přiřazené uzly.

Na následujících dvou stranách jsou diagramy popisující třídy z balíčků `tree` a `dsl`. Celý zdrojový kód programu je dostupný na CD této práce.

[12] tj. jeden z klasických návrhových vzorů objektově orientovaného programování
[13] viz podsekce 2.3.3 a metoda `ForkJoinTask.invokeAll`



Obrázek 4.2: UML diagram popisující třídy v rámci balíčku `...dsl`



Obrázek 4.3: UML diagram popisující třídy v rámci balíčku `...tree`

4.3 Použití programu na reálném problému

Program byl pro demonstraci použit na problému popsaném v diplomové práci Ing. Františka Janošťáka [34], v době psaní této práce působícího na Ústavu procesního inženýrství fakulty strojní Vysokého učení technického v Brně. Uvažovaný model se týká optimálního návrhu výstavby zařízení na energetické využití odpadu (ZEVO) u stávající teplárny. V modelu se maximalizuje hodnota vnitřního výnosového procenta (IRR) tak, aby teplárna nevykazovala žádné ztráty. Jak ve své práci Ing. Janošťák popisuje, byl původní model typu MINLP[14], což mělo za následek velmi obtížné hledání řešení. Proto, jak dále ve své práci uvádí, zparametrizováním proměnné, která popisuje kapacitu ZEVO, dosáhl modelu typu ILP[15], pro který je jednodušší nalezení optima. Následné hledání globálního optima lze chápat jako složenou maximalizační úlohu

$$\underset{C}{\operatorname{argmax}}\{IRR(C)|\ C \in \langle 10000; 40000\rangle\}, \tag{4.1}$$

kde C představuje zvolenou hodnotu kapacity ZEVO a $IRR(C)$ řešení původního zparametrizovaného modelu. Pro řešení této úlohy je pak použita metoda kvadratické interpolace.

Původní model je popsán v modelovacím jazyce GAMS, kde je implementována i samotná iterační metoda kvadratické interpolace, a celý model je tedy řešen v cyklu. Pro použití programu ComposedOptimization byla z modelu odstraněna zmíněná metoda spolu s jejím cyklem a model byl rozšířen o příkaz načítající hodnotu vstupního parametru kapacity C ze specifikovatelného datového souboru typu `.gdx` a o příkaz vypisující odpovídající hodnotu IRR do specifikovatelného textového souboru. Místo metody kvadratické interpolace byla pro jednoduchost použita metoda bisekce. Sled opakovaného řešení modelu a samotná metoda bisekce jsou pak nadefinovány v uživatelském skriptu. Pro sdílení dat mezi modelem a samotným programem jsou využity již zmíněné datové soubory. Zdrojový kód skriptu je pro ilustraci použití programu uveden celý.

[14] mixed integer nonlinear programming; viz sekce 1.3

[15] integer linear programming; viz sekce 1.3

```
import java.io.File

/* Konstanty */
final String model = "spalovna_teplarna.gms"
final double eps = 1e-3

/* Pomocná datová třída */
class Data {
    double c
    double irr

    String toString() {"[kapacita: $kap, irr: $irr]"}
}

/* Seznam datových bodů */
List data = [new Data(c: 10000),
             new Data(c: 40000)]

/* Pomocné Funkce */

// Zapíše kapacitu do gdx
def writeC = { String gdx, Data dat ->
    def db = GAMSWrkspc.addDatabase(gdx)
    db.addParameter("KAPHEU", 1).addRecord("1").setValue(dat.c)
    db.export()
    db.dispose()
}

// Načte irr z gdx
def readIrr = { String putfile, Data dat ->
    String put = new File(putfile).text.trim()
    dat.irr = Double.parseDouble(put)
}

// Funkce pro třídění senamu od největší hodnoty irr po nejmenší
def irrSort = { a, b -> b.irr <=> a.irr }

// Heuristika pro určení další hodnoty kapacity
def nextC = {
    data.sort(irrSort)

    if (data.size > 2)
        data.remove(2)

    double kapheu = (data[0].c + data[1].c) / 2
    println "Nová kapacita: $kapheu"
    return kapheu
}

/* Definice projektového stromu */

// Vytvoř počáteční .gdx
$later {
    writeC("kap1.gdx", data[0])
    writeC("kap2.gdx", data[1])

    println "Počáteční kapacity: ${data[0].c} a ${data[1].c}"
}
```

```
// Vyřeš (paralelně) počáteční irr hodnoty
$group {
    execution = parallel

    $GAMS(model) {
        name = "First initial point"
        options.defines("kapgdx", "kap1.gdx")
        options.defines("putirr", "irr1.txt")
    }

    $GAMS(model) {
        name = "Second initial point"
        options.defines("kapgdx", "kap2.gdx")
        options.defines("putirr", "irr2.txt")
    }
}

// Načti počáteční irr z gdx předešlých výpočtů a urči novou kapacitu
$later {
    readIrr("irr1.txt", data[0])
    readIrr("irr2.txt", data[1])

    println("Hodnoty irr pocatecnich bodu:"
     + " ${data[0].irr} a ${data[1].irr}")

    double kapheu = nextC()
    data.add(new Data(c: kapheu))
    writeC("kap.gdx", data[2])
}

// Funkce podmínky cyklu
def condition = {
    // Načti irr z gdx proběhlé iterace
    readIrr("irr.txt", data[2])
    println "Nová hodnota irr: ${data[2].irr}"

    // Ukonči cyklus, pokud nedošlo k dostatečně velkému zlepšení
    if (data[2].irr <= data[0].irr + eps)
        return false

    // Urči další kapacitu a zapiš ji do gdx
    double kapheu = nextC()
    data.add(new Data(c: kapheu))
    writeC("kap.gdx", data[2])

    // Pokračuj na další iteraci
    return true
}

// Počítadlo iterací
int i = 0

// Opakuj, dokud funkce v podmínce vrací hodnotu true
// Parametr "check" je nastaven na hodnotu "after",
//     první iterace tedy proběhne bez podmínky
$repeat(condition) {
    check = after

    $GAMS(model) {
```

```
        name = "Third point"
        options.defines("kapgdx", "kap.gdx")
        options.defines("putirr", "irr.txt")
    }

    // Navyš počet iterací
    $later {i++}
}

// Vypiš nejlepši dosaženou hodnotu
$later {
    data.sort(irrSort)
    println("Nejlepsi dosazena hodnota IRR po [$i] iteracich:"
     + " ${data[0].irr}, pro kapacitu: ${data[0].c}")
}
```

Závěr

V teoretické části této práce byla popsána problematika optimalizačních úloh a softwarových nástrojů pro jejich řešení, a to vč. modelovacích jazyků. Pro řešení úloh zvolených tříd složených optimalizačních úloh, analýzu citlivosti a navazující optimalizaci, byly navrženy, popsány a v programovacím jazyce Java implementovány vlastní softwarová řešení v podobě programů ParallelGAMS a ComposedOptimization.

Programem ParallelGAMS byl demonstrován přínos paralelních výpočtů pro analýzu citlivosti, a to urychlením řešení reálné úlohy na Ústavu procesního inženýrství FSI VUT z původních 51,3h na 3h. Při analýze naměřených dat se dále ukázalo, že při paralelním řešení byly délky výpočtů jednotlivých scénářů analýzy citlivosti zhruba dvakrát tak dlouhé, než při výpočtu sekvenčním. Tato skutečnost může být využita pro další výzkum urychlení paralelních výpočtů.

Program ComposedOptimization byl navrhnut tak, aby sloužil jako uživatelské prostředí pro strukturované řešení problémů navazující optimalizace, a to pomocí jejich rozložení na jednotlivé podproblémy. Program nabízí jako způsob řešení těchto podproblémů modelovací jazyk GAMS, vzhledem k objektově orientovanému přístupu při vývoji programu však bude snadné zakomponovat i další možnosti řešení. Uvedené prostředí bude dále vyvíjeno pro efektivní implementaci náročných optimalizačních dekompozičních algoritmů, jako jsou Bendersova dekompozice [35, 36] a Progressive Hedging algoritmus [37, 38] používané při aplikačních výpočtech na FSI. Budou rovněž zkoumány možnosti přispět k vývoji pokročilých knihoven DOP (Distributed Optimization Problems), viz např. [39, 40].

Literatura

[1] POPELA, Pavel. *Optimization I.* Brno, 2017. VUT, učební text.

[2] WILLIAMS, H. P. *Model building in mathematical programming.* 5th ed. Hoboken, N.J.: Wiley, 2013. ISBN 978-1-118-44333-0.

[3] KLAPKA, Jindřich, Jiří DVOŘÁK a Pavel POPELA. *Metody operačního výzkumu.* Vyd. 2. Brno: VUTIUM, 2001. ISBN 80-214-1839-7.

[4] PARDALOS, P. M. a Mauricio G. C. RESENDE. *Handbook of applied optimization.* New York, N.Y.: Oxford University Press, 2001. ISBN 0195125940.

[5] BAZARAA, M. S. a John J. JARVIS. *Linear programming and network flows.* New York: Wiley, c1977. ISBN 0471060151.

[6] KALL, Peter. a Stein W. WALLACE. *Stochastic programming.* New York: Wiley, c1994. ISBN 978-0471951582.

[7] SKLENÁŘ, J. *Introduction to Dynamic Programming.* University of Malta, učební text.

[8] BIRGE, John R. a Francois. LOUVEAUX. *Introduction to stochastic programming.* 2nd ed. New York: Springer, c2011. ISBN 978-1-4614-0236-7.

[9] KING, Alan J. *Modeling with stochastic programming.* New York: Springer, 2012. ISBN 978-0-387-87816-4.

[10] GAMS Modelling Language Manuals. GAMS Inc, 2017.

[11] CBC User Guide. *COIN-OR* [online]. [cit. 2018-05-23]. Dostupné z: https://www.coin-or.org/Cbc/cbcuserguide.html

[12] *CONOPT* [online]. [cit. 2018-05-23]. Dostupné z: http://www.conopt.com/

[13] CPLEX Optimizer. *IBM* [online]. [cit. 2018-05-23].Dostupné z: https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer

[14] GLPK. *GNU Project* [online]. [cit. 2018-05-23].Dostupné z: https://www.gnu.org/software/glpk/

[15] *Gurobi Optimization* [online]. [cit. 2018-05-23].Dostupné z: http://www.gurobi.com/

[16] *LINDO Systems* [online]. [cit. 2018-05-23]. Dostupné z: https://lindo.com/

[17] *Optimizer: java convex optimizer* [online]. [cit. 2018-05-23]. Dostupné z: http://www.joptimizer.com/

[18] JOM. *Net2Plan* [online]. [cit. 2018-05-23]. Dostupné z: http://www.net2plan.com/jom/

[19] Scipy.optimize. *SciPy* [online]. [cit. 2018-05-23]. Dostupné z: https://docs.scipy.org/doc/scipy/reference/optimize.html

[20] *PYOMO* [online]. [cit. 2018-05-23]. Dostupné z: http://www.pyomo.org/

[21] *Excel Solver* [online]. [cit. 2018-05-23]. Dostupné z: https://www.solver.com/

[22] Optimization Toolbox. *MathWorks* [online]. [cit. 2018-05-23]. Dostupné z: https://www.mathworks.com/products/optimization.html

[23] *AMPL* [online]. [cit. 2018-05-23]. Dostupné z: https://ampl.com/

[24] *AIMMS* [online]. [cit. 2018-05-23]. Dostupné z: https://aimms.com/

[25] AKHTER, Shameem. *Multi-core programming: increasing performance through software multi-threading.* New York: Intel Press, 2006. ISBN 0-9764832-4-6.

[26] BUYYA, Rajkumar, Christian. VECCHIOLA a S. Thamarai. SELVI. *Mastering cloud computing: foundations and applications programming.* Boston: Morgan Kaufmann, 2013. ISBN 0124114547.

[27] OSHANA, Robert. *Multicore software development techniques: applications, tips, and tricks.* Waltham, MA: Newnes, c2016. ISBN 0128009586.

[28] *Java Platform, Standard Edition 8 API Specification* [online]. [cit. 2018-05-23]. Dostupné z: https://docs.oracle.com/javase/8/docs/api/

[29] Concurrency. *Oracle Documentation* [online]. [cit. 2018-05-23]. Dostupné z: https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

[30] PUTNA, O.; JANOŠŤÁK, F.; ŠOMPLÁK, R.; PAVLAS, M. Short-time Fluctuations and their Impact on Waste-to-Energy Conceptual Design Optimised by Multi- stage Stochastic Model. In Chemical engineering transactions. *CHEMICAL ENGINEERING TRANSACTIONS*. 2017. s. 955-960. ISBN: 978-88-95608-51- 8. ISSN: 2283-9216.

[31] JANOŠŤÁK, F.; PAVLAS, M.; PUTNA, O.; ŠOMPLÁK, R.; POPELA, P. Heuristic Approximation and Optimization for Waste-to- Energy Capacity Expansion Problem. *Mendel Journal series*, 2016, roč. 2016, č. 1, s. 123-130. ISSN: 1803-3814.

[32] FERDAN, Tomáš, Radovan ŠOMPLÁK, Lenka ZAVÍRALOVÁ, Martin PAVLAS a Lukáš FRÝBA. A waste-to-energy project: A complex approach towards the assessment of investment risks. *Applied Thermal Engineering* [online]. 2015, **89**, 1127-1136 [cit. 2018-05-23]. DOI: 10.1016/j.applthermaleng.2015.04.005. ISSN 13594311. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/S135943111500321X

[33] P. POPELA, *An Object Oriented Approach to Multistage Stochastic Programming: Models and Algorithms*, Ph.D. thesis, Charles University, Prague, 1998.

[34] JANOŠŤÁK, F. *Modely toků v síti pro odpadové hospodářství*. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2016. 43 s. Vedoucí diplomové práce Ing. Martin Pavlas, Ph.D.

[35] GEOFFRION, A. M. Generalized Benders decomposition. *Journal of Optimization Theory and Applications* [online]. 1972, **10**(4), 237-260 [cit. 2018-05-23]. DOI: 10.1007/BF00934810. ISSN 0022-3239. Dostupné z: http://link.springer.com/10.1007/BF00934810

[36] KŮDELA, Jakub a Pavel POPELA. Warm-start cuts for Generalized Benders Decomposition. *Kybernetika* [online]. 1012-1025 [cit. 2018-05-23]. DOI: 10.14736/kyb-2017-6-1012. ISSN 0023-5954. Dostupné z: https://www.kybernetika.cz/content/2017/6/1012

[37] ROCKAFELLAR, R. T. a Roger J.-B. WETS. Scenarios and Policy Aggregation in Optimization Under Uncertainty. *Mathematics of Operations Research* [online]. 1991, **16**(1), 119-147 [cit. 2018-05-23]. DOI: 10.1287/moor.16.1.119. ISSN 0364-765X. Dostupné z: http://pubsonline.informs.org/doi/abs/10.1287/moor.16.1.119

[38] KLIMEŠ, Lubomír, Pavel POPELA, Tomáš MAUDER, Josef ŠTĚTINA a Pavel CHARVÁT. Two-stage stochastic programming approach to a PDE-constrained steel production problem with the moving interface. *Kybernetika* [online]. 1047-1070 [cit. 2018-05-23]. DOI: 10.14736/kyb-2017-6-1047. ISSN 0023-5954. Dostupné z: https://www.kybernetika.cz/content/2017/6/1047

[39] ROUPEC, Jan a Pavel POPELA. The Nested Genetic Algorithms for Distributed Optimization Problems. *Special edition of the World Congress on Engineering and Computer Science 2010, San Francisco, CA, USA, 20-22 October 2010: WCECS 2011 : 19-21 October, 2011, San Francisco, USA*. Melville, N.Y.: American Institute of Physics, 2011, s. 480-484. IAENG transactions on engineering technologies, v. 6. ISBN 0735409331.

[40] POPELA, P.; SKLENÁŘ, J.; MATOUŠEK, R.; ŽAMPACHOVÁ, E.; ROUPEC, J.: Advances in the Formal Framework for DOP, *17th International Conference of Soft Computing*, MENDEL 2011 (id 19255), pp.320-325, ISBN 978-80-214-4302-0, (2011), VUT, článek ve sborníku, akce: 17th International Conference of Soft Computing, MENDEL 2011, Brno University of Technology, 15.06.2011-17.06.2011