ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA
ELEKTROTECHNICKÁ

BAKALÁŘSKÁ PRÁCE

Programování s omezujícími
podmínkami v Scheduling Toolboxu

Praha, 2007 Jiří Cigler

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, SW) uvedené v přiloženém seznamu.

Nemám žádný důvod proti použití tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne ..... Podpis: .....

Poděkování

Děkuji především vedoucímu práce Ing. Janu Kelbelovi, který směřoval práci ke kýženému cíli. Dále Ing. Michalu Kutilovi a Ing. Přemyslu Šůchovi za konzultace v oblasti Matlabu resp. rozvrhování.

Můj vděk patří i mým blízkým, kteří mě v práci a studiu podporovali.

Abstrakt

První část práce má za cíl přinést čtenáři ucelené informace o základních pojmech, definicích a ustálených značeních v teorii rozvrhování. Důraz je kladen zejména na 3 hlavní problémy dílen (Job-shop, Flow-shop a Open-shop).

V další části se práce zaměřuje na programování s omezujícími podmínkami – podstatu, vlastnosti, algoritmy, výhody, nevýhody a spojení rozvrhování s touto formou programovacích technik.

Následně bude práce směřovat k rozebrání problematiky řešení a implementace problémů dílen do TORSCHE Scheduling toolboxu a zhodnocení výsledků implementovaných algoritmů.

Poslední část se zabývá objektovým návrhem obecného „shopu“ v TORSCHE Scheduling toolboxu.

Abstract

First part of this work aims for familiarizing reader with basic terms, definitions and standard notations in scheduling theory emphasising on 3 main shop problems (Job-shop, Flow-shop and Open-shop).

Next part discuss constraint programming and constraint based scheduling – fundamental principles, characteristics, algorithms, advantages and disadvantages.

Problems in design and implementation of shop in TORSCHE Scheduling toolbox and results with some charts will be mentioned after more theoretical part of this work.

End of this work is about object modelling of general shop and its integration into TORSCHE Scheduling toolbox.

Obsah

1 Úvod

„Ve srovnání se schopností rozumně si rozvrhnout práci na jeden den je všechno ostatní dětskou hrou“ **Johann Wolfgang von Goethe**

Výrok německého myslitele vyzdvihuje důležitost rozvrhování a to nejen jako problém akademický, ale také problém praktický a hlavně problém každého myslícího jedince. S rozvrhováním se v běžném životě setkáme opravdu na každém kroku. Člověk si u jednoduchých problému ani neuvědomuje, že o nich nějakým způsobem přemýšlí (takových řeší denně desítky). Ale při vyšším počtu problémů, které má rozvrhnout, už musí opravdu zapřemýšlet nad svým rozvrhem a vyladit ho do podoby, která se mu bude líbit (odpovídá jeho hodnotící funkci). Přibudou-li ještě precedenční vazby mezi jednotlivými úkoly, pak složitost problému prudce stoupá a najít optimální řešení je značně komplikované. Kde končí schopnosti člověka, nastupuje výpočetní síla se speciálním programem pro řešení rozvrhovacích problémů.

Pro řešení výše zmíněných a mnohých dalších rozvrhovacích problémů se nyní na Katedře řídící techniky vyvíjí TORSCHE Scheduling Toolbox v Matlabu. Jednou z chystaných funkčností je řešení tzv. *shopů* (skládá se z několika úloh s podmínkou na vykonávající osobu nebo stroj a tyto úlohy „bojují“ o zařazení do rozvrhu). A právě návrh a implementace shopu je tématem této práce. Teoretický úvod do rozvrhování a shopům s přesnými pojmy rozeberu v kapitole 2.

K řešení výše zmíněných problémů se dá dojít různými cestami. Jednak problém můžeme reprezentovat pomocí grafu (například přiřadit úlohám vrcholy, hrany pak budou vztahy mezi nimi) a pak řešit pomocí grafových algoritmů. Další velmi rozšířenou alternativou jsou genetické algoritmy (GA). O řešení shopů pomocí GA např. v [14]. K řešení se můžeme dostat i pomocí fuzzy modelování (řešení problému Job-shop touto metodou je v [15]). Používanou metodou je také lineární celočíselné programování (ILP [13]). Z názvu vyplývá, že se jím dají popsat pouze lineární podmínky. ILP však dokáže definovat i podmínky ve tvaru $a \oplus b$ a proto se dá využít i v rozvrhování. Komplexnější podmínky se snáze vyjádří pomocí dalšího silného nástroje, kterým je programování s omezujícími podmínkami (CP). CP dokáže pracovat s podmínkami libovolného tvaru. Právě tuto techniku jsme využívali v této práci. O základních rysech omezujících podmínek bude pojednáno v kapitole 3.

Druhá část práce (kapitoly 4-6) bude navazovat na teoretické části jejich aplikací. Tedy návrhem modelu shopu (problému rozvrhování v dílně) a implementací algoritmu na jeho řešení. Dále návrhem objektu reprezentujícího Shop a objektů s ním souvisejících pro TORSCHE Scheduling toolbox v Matlabu. Následovat budou výsledky a hodnocení algoritmu.

2 Úvod do rozvrhování

2.1 Základní pojmy

V této části se nejprve seznámíme s pojmy Úloha a Zdroj jakožto základními stavebními kameny rozvrhování. Poté bude uvedena jeho definice. Kritéria optimality budou zmíněna na konci této sekce. Všechny tyto pojmy jsou dobře probrány například v [1–2].

2.1.1 Úloha

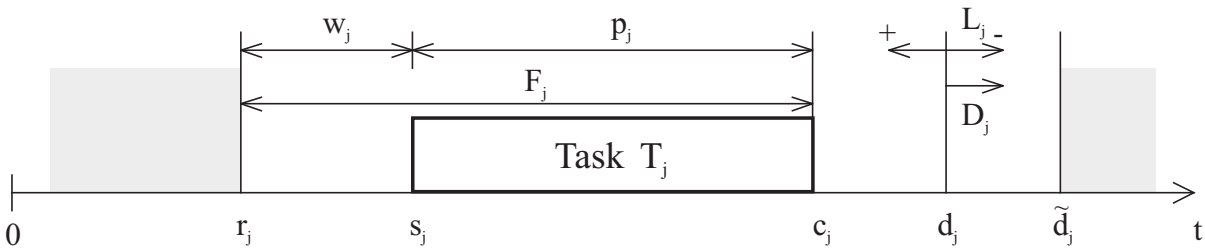
Úloha je základní prvek každého rozvrhovacího problému. Můžeme na ni nahlížet jako na jednotku práce, kterou musíme nějakým způsobem naplánovat k provedení. Setkáváme se u ní s mnoha parametry, které může mít.

- Doba vykonávání (processing time) p_j
- Začátek vykonávání (start time) s_j[1]
- Konec vykonávání (completition time) C_j
- Okamžik disponability (release date) r_j : Nejmenší čas, kdy je úloha připravena k provedení.
- Okamžik požadovaného dokončení (due date) $d_j : C_j$ by měl být menší než tato hodnota. Pokud tomu tak není, může dojít k penalizaci.
- Poslední okamžik dokončení (deadline) $\widetilde{d_j} : C_j$ musí být menší než tento čas. Pokud se C_j nevejde do stanoveného limitu, pak problém nemá řešení.
- Precedenční vazby na ostatní úlohy (precedence constraints): Velmi často dochází k situacím, kdy máme danou posloupnost úloh, které musí být vykonány v daném pořadí. Tento fakt je nutné také zachytit ve vlastnostech úlohy.
- Stroj (dedicated processor): jeden a více strojů, na kterých musí úloha běžet.
- Priorita (priority): důležitost úlohy v porovnání s ostatními. Použití nachází při přesahování d_j, kde je možné více penalizovat úlohy s vyšší prioritou.

[1] Začátek vykonávání nemusí být jen jeden okamžik. Za předpokladu preemptivního rozvrhování může nastat situace, kdy je vykonávání úlohy přerušeno jinou úlohou a další pokračování úlohy považujeme za další okamžik začátku vykonávání.

- Zpoždění (lateness) $L_j : L_j = C_j - d_j$
- Doba čekání (waiting time) $w_j : w_j = s_j - r_j$
- Překročení vymezeného času (tardiness) $D_j : D_j = \max\{C_j - d_j, 0\}$
- Doba dokončení (flow time, response time) $F_j : F_j = C_j - r_j$

Parametry úlohy jsou na obrázku 2.1. V praxi se zřídka setkáme s použitím všech parametrů úloh. Některé parametry se stejně dají vyjádřit pomocí ostatních – není je tedy nutné při implementaci zahrnovat.



Obrázek 2.1 Znázornění parametrů úlohy

2.1.2 Zdroj

Jednotka, na které může být úloha vykonávána. Rozlišujeme zde mezi zdroji, které mohou vykonávat pouze jednu úlohu v čase – unární zdroje (unary resources) a zdroji, které mají jistou kapacitu prováděných úloh (limited capacity resources, cumulatives resources). U zdrojů s kapacitou vyšší než 1 se také setkáváme s tzv. „dávkovým“ zpracováním úloh (batch processing). To znamená, že několik úloh začíná ve stejný okamžik a dokud není připravená celá dávka úloh, čeká se na další úlohy do dávky. Pod pojmem „zdroj“ si v teorii rozvrhování nemusíme představovat pouze stroj vyrábějící plošné spoje, montážního robota ve výrobně aut nebo počítač (který řeší třeba distribuovaný program). Zdroji jsou:

- (Lidská) práce
- Peníze
- Energie
- Nástroje
- Stroje, roboti

V dalším pokračování budou považovány za zdroje jenom stroje.

2.1.3 Definice rozvrhování

Úkolem rozvrhování je alokovat množinu úloh $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ na množinu dostupných strojů $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ v čase při dodržení všech omezujících podmínek (zde navíc vzniká podmínka, že pokud je rozvrh nepreemptivní, pak žádná úloha nesmí být přerušena. Jinak může být přerušená právě konečným množstvím přerušení).

V praxi se setkáváme jednak s tzv. statickým rozvrhováním, kde množina úloh je známá předem. To se uplatňuje většinou ve výrobních procesech (fabriky, dílny, školy). Druhou možností je dynamické rozvrhování, kdy se množina úloh vyvíjí v čase. To nachází uplatnění naříklad v robotice, kde robot v jeden časový okomžik může provádět jednu množinu úloh a v jiný časový okamžik jinou množinu úloh. Ale dopředu není jasné, které úlohy v té množině budou.

Výsledek rozvrhování se většinou posuzuje vhodně zvolenou hodnotící funkcí. Mohou to být například funkce $f(T_1, T_2, \ldots, T_n)$, které vyjadřují :

- Nejpozději dokončenou úlohu : $C_{max} = \max_{j=1}^{n} \{C_j\}$
- Celkový součet všech dokončení úloh (total flow time): $\sum_{j=1}^{n} C_j$
- Vážený celkový součet všech dokončení úloh (weighted total flow time): $\sum_{j=1}^{n} w_j C_j$
- Maximální zpoždění: $L_{max} = \max_{j=0}^{n} \{L_j\}$
- Součet překročení vymezeného času (tardiness): $\sum_{j=1}^{n} D_j$
- Vážený součet překročení vymezeného času: $\sum_{j=1}^{n} w_j D_j$
- Vážený součet jednotkových penalizací $\sum_{j=1}^{n} w_j U_j$, kde penalizace U_j je definována jako: $U_j = \begin{cases} 0 & \text{pro } C_j < d_j \\ 1 & \text{jinak} \end{cases}$

Nejmenší čas dokončení ještě neznamená nejlepší výsledek. Záleží na okolnostech situace, pro kterou navrhujeme rozvrhovací algoritmus. Proto je důležité si před vlastní implementací rozmyslet, kterou hodnotící funkci zvolit.[2]

2.2 Grahamova notace

V praxi se setkáváme s velkým množstvím různorodých rozvrhovacích problémů. To nás vede k myšlence zavést standartní notaci. Grahamova notace je velmi rozšířená a TORSCHE Scheduling toolbox s ní pracuje také. Proto tady ve stručnosti zmíním její hlavní rysy.

Základem je trojice položek

$$\alpha|\beta|\gamma \tag{2.1}$$

První položka $\alpha = \{\alpha_1, \alpha_2\}$ popisuje zdroje. α_1 značí typ zdrojů (zda jsou stejné, předvolené), α_2 popisuje jejich počet.

Druhé pole β v sobě nese informaci o vlastnostech problému jako takového (preempce, precedenční vazby, doby trvání jednotlivých úloh, deadline apod.)

γ udává optimalizační kritérium (podsekce 2.1.3)

Podrobněji pojednává o Grahamově notaci například [1]

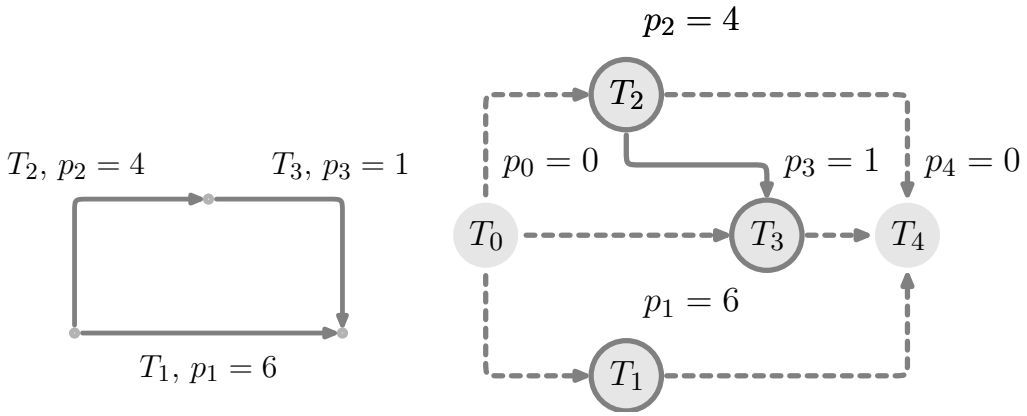
2.3 Grafické znázornění problému

Pro lepší a rychlejší pochopení zadání nebo výsledku je vhodné zobrazit celý problém pomocí diagramu. Pro zadání se používají *task-on-node* nebo *task-on-arc* grafy. Jak už název napovídá, u prvního jmenovaného jsou úlohy umístěny na vrcholech grafu zatímco u druhého na hranách grafu. Vše je přehledně vidět na obrázku 2.2, který vystihuje zadání podle zápisu 2.2. Binární operátor $\rightarrow$ představuje precedenční vazbu mezi úlohami. V tomto případě T_2 musí být provedena před T_3.

$$\mathcal{T} = \{T_1, T_2, T_3\},\ p_1 = 6,\ p_2 = 4,\ p_3 = 1,\ T_2 \rightarrow T_3 \tag{2.2}$$

U Task-on-node grafů se setkáme s dvojí možností zakreslení. Jednak se všemi úlohami a precedenčními vazbami mezi nimi (např v [1], v obrázku zakresleny pouze plnou čarou a vrcholy pouze s okrajem), ale také se můžeme setkat se situací, kdy se do grafu přidávají dvě „nadbytečné“ úlohy, které mají nulový čas vykonávání a jsou umístěny jako počáteční resp. koncová úloha. Ty se využívají k vymezení časového úseku od začátku skutečné první úlohy do konce poslední skutečné úlohy. (např v [2], v obrázku hrany čárkovaně a vrcholy bez okrajů).

[2] Hodnotící funkce nemusí být jen právě jedna z výše uvedených, ale může to být i jejich kombinace. Případně to může být funkce, kterou navrhne ekonom pro minimalizaci nákladů apod.

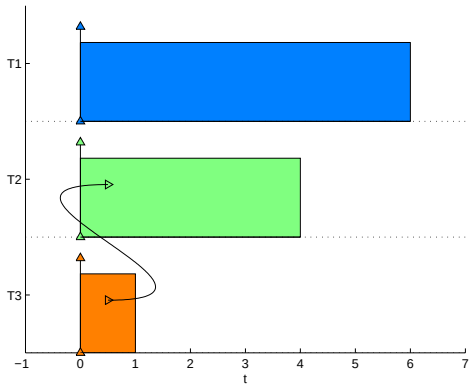


Task-on-arc graf - hrany reprezentují úlohy.

Task-on-node graf - vrcholy reprezentují úlohy.

Obrázek 2.2 Dva zůsoby zobrazení zadání rozvrhovacího problému s precedenčními vazbami.

K zobrazení výsledného rozvrhu se velmi často používají Ganttovy diagramy. Ty jsou známé spíše z manažerské oblasti, kde se často používají pro plánování projektů. V podstatě jde o sloupcový graf, který nese úplnou informaci o jednotlivých úlohách (začátek, konec, precedence, zdroj). Podrobněji o nich pojednává [6].
Hlavní rysy obou vystihuje obrázek 2.3, který představuje řešení rozvrhovacího problému zadaného zápisem 2.2.



Obrázek 2.3 Ganttův diagram[3] vygenerovaný pomocí funkce `plot` v TORSCHE toolboxu, viz [7]

2.4 Problémy rozvrhování v dílně

Dílna je místo, kde je vytvářen nějaký produkt. Proces výroby se může skládat z několika operací, které se vykonávají na strojích v dílně umístěných. Je-li dělníků v dílně více, pak může nastat situace, kdy jeden dělník čeká na dokončení práce jeho kolegy na stroji, který chce využívat. Řešením celé situace může být vhodnější

[3] Původním názvem byl vlastně jen *Harmonogram*, jehož autorem byl Karol Adamiecky. Ale ten svoje dílo po dlouhou dobu neprezentoval a proto se uchytilo pojmenování po Henry Ganttovi.

naplánování provádění úloh, aby nedocházelo k podobným situacím a tím se uspořil čas i peníze.

Právě takto se v rozvrhování chápou problémy dílen. V dalším pokračování budu nazývat tento problém „shopem“[4]. Základní stavební jednotkou shopu je job, který se skládá z množiny úloh. $\mathcal{J}_i = \{T_{i1}, T_{i2}, \ldots, T_{in}\}$. Každá z úloh $T_{ij} \in \mathcal{J}_i$ má svůj stroj, na kterém poběží $\mu_{ij} \in \{P_1, P_2, \ldots, P_m\}$. V obecném shopu platí následující podmínky:

- Žádné dvě úlohy z jednoho jobu neběží současně.
- Každý stroj zvládá v daném okamžiku zpracovávat pouze jednu úlohu (u rozšíření obecného shopu tato podmínka nemusí platit – některé stroje mohou mít kapacitu vyšší než 1. Více o tom pojednává kapitola 6)
- V shopech se vyskytují precedenční omezení, která musí být splněna.

O třech základních shopech budou pojednávat další podsekce. Jejich základní znaky lze pozorovat na obrázku 2.4, který zobrazuje jejich Ganttovy diagramy. O shopech se lze dočíst například v [1–3].

2.4.1 Job-shop

Job-shop rozšiřuje obecný shop o sadu precedenčních omezení, které vnucují pořadí úloh $T_{ij} \in \mathcal{J}_i$ do řetězce.

$$T_{i1} \rightarrow T_{i2} \rightarrow \ldots \rightarrow T_{in} \tag{2.3}$$

To znamená, že se zde nevyskytují jakékoliv precedence mezi úlohami v jednotlivých jobech, ale pouze vrámci jobu.

Zadání pro jobshop se dá vyjádřit ve tvaru dvou matic. Matice doby trvání úloh T a matice strojů P, na kterých úlohy poběží.

$$\begin{aligned} T &= \begin{bmatrix} p(\mathcal{J}_1) \\ p(\mathcal{J}_2) \end{bmatrix} = \begin{bmatrix} p(T_{11}) & p(T_{12}) \\ p(T_{21}) & p(T_{22}) \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \\ P &= \begin{bmatrix} \mu_{11} & \mu_{12} \\ \mu_{21} & \mu_{22} \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \end{aligned} \tag{2.4}$$

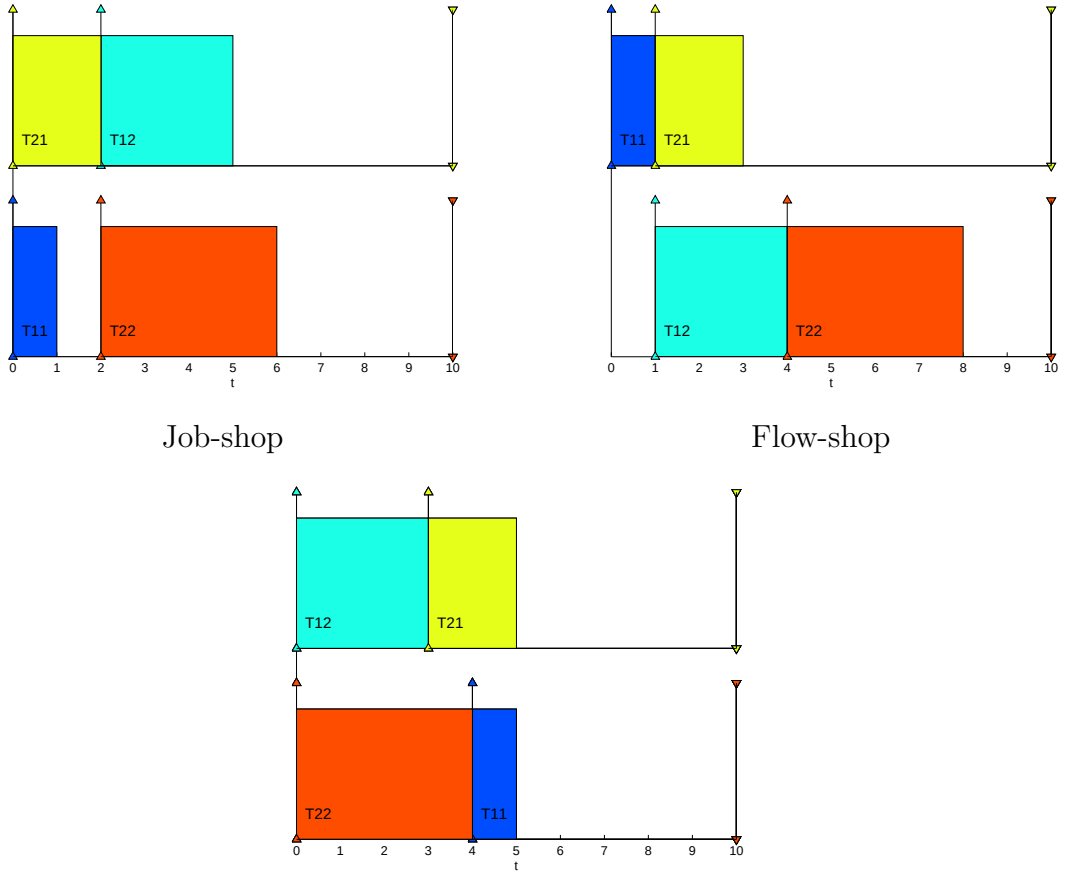
[4] Terminologie teorie rozvrhování má v českém jazyce ještě rezervy a většinou se používají slova přejatá z angličtiny. Je to především proto, že není rozšířená literatura psaná v českém jazyce pojednávající o tomto tématu.

2.4.2 Flow-shop

Flow-shop je zvláštní případ Job-shopu, kde j-té úlohy v jednotlivých jobech běží na stejných strojích. V případě platnosti $\mu_{ij} = j$ stačí k zadání problému pouze matice T z (2.4).

2.4.3 Open-shop

Slovo „Open“ už napovídá, že jednotlivé úlohy v jobech nebudou mít žádné precedenční vazby. V ostatním se shoduje s Flow-shopem.



Obrázek 2.4 Rozvrhy pro Job-shop, Flow-shop a Open-shop zadané maticí (2.4).

3 Úvod do omezujících podmínek

„Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.“

Eugene C. Freuder, Constraints, Duben 1997

Programování s omezujícími podmínkami (Constraint programming, CP) je nástrojem pro řešení kombinatorických (optimalizačních) úloh. Základem je vytvoření modelu problému, který se skládá z proměnných svázaných libovolnými *podmínkami*. Ve většině případů mají proměnné konečně velký obor hodnot(doménu), pak mluvíme o CSP (Constraint satisfaction problem). Cílem CSP je najít takové ohodnocení proměnných, které splňuje všechny podmínky a navíc patří do oboru hodnot. Rozšířením CSP o hodnotící funkci (např. z části o rozvrhování 2.1.3) se dostáváme k CSOP (Constraint satisfaction optimisation problem), kde nás zajímá právě to „nejlepší“ řešení.

Základní pojmy jsou shrnuty v [5, 8] a podrobně o CP pojednává [9].

Vlastnosti omezujících podmínek jsou:

- Vyjadřují částečnou informaci ($X > 3$, hodnota X není určena jednoznačně)
- Poskytují lokální pohled na celý model problému (svazují jen několik proměnných – ne všechny najednou)
- Mohou být heterogenní (domény proměnných mohou být různé)
- Nejsou směrové ($X = Y + 1$ lze použít pro výpočet X i Y).
- Jsou deklarativní – neurčují výpočtovou proceduru pro své splnění.
- Jsou aditivní – pořadí podmínek nehraje roli. Důležitá je jejich konjunkce.

CSP se dá reprezentovat pomocí hypergrafu, kde vrcholy značí proměnné a hrany podmínky. Hypergraf se dá převést na graf a ten pak řešit pomocí prohledávacích algoritmů.

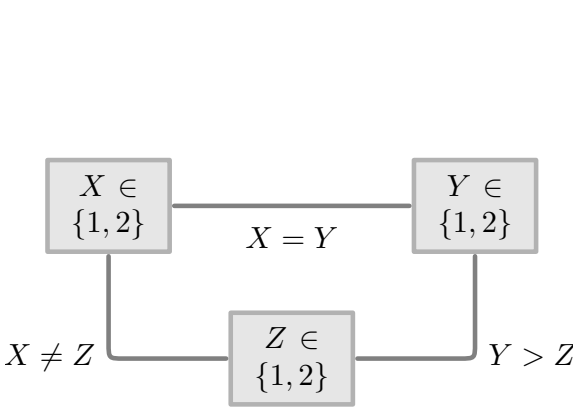
Základními stavebními kameny CP jsou konzistenční techniky (část 3.1) a prohledávací algoritmy (část 3.2).

3.1 Konzistenční techniky

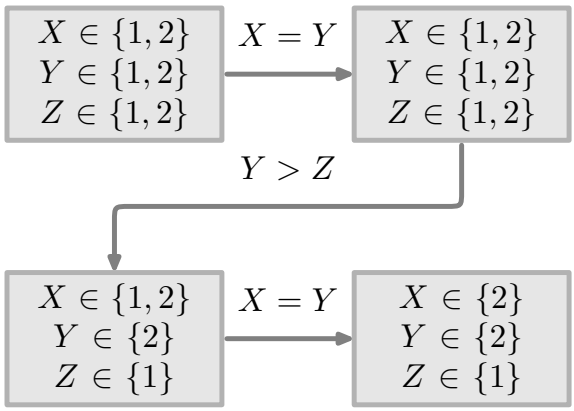
Konzistenční techniky (algoritmy) se starají o to, aby hodnoty všech domén splňovaly všechny omezující podmínky. Pokud tomu tak není, aktivně prořežou jednotlivé

domény, aby předchozí tvrzení platilo. Tomuto procesu se říká *propagace omezujících podmínek*.

Propagace podmínek tedy pracuje tak, že každá podmínka má svoji filtrovací funkci, kterou dokáže prořezávat domény proměnných, které podmínka svazuje. Jakmile se jedna z proměnných v podmínce změní, pak se volá filtrovací funkce. Tím se zaručí konzistence CSP. Propagaci podmínek ukazuje obrázek 3.1. Propagace podmínek redukuje domény proměnných opakovaným voláním filtrovacích algoritmů. Tento proces se provádí až do okamžiku dosažení tzv. „fix-point“, kdy je CSP *hranově* i *vrcholově konzistentní*.



CSP reprezentovaný grafem obsahující proměnné s jejich doménami a omezujícími podmínkami.

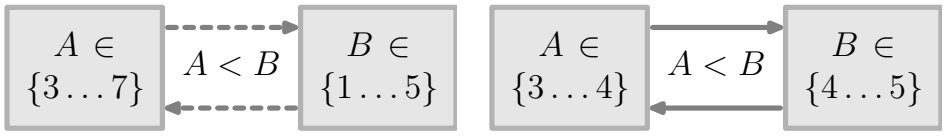


Propagace omezujících podmínek.

Obrázek 3.1 Zadání problému a následná propagace podmínek.

Vrcholová konzistence Vrchol reprezentující proměnnou X je *vrcholově konzistentní (node consistent)*, právě když každá hodnota z aktuální domény D_x splňuje všechny unární podmínky na X.
CSP je *vrcholově konzistentní* právě tehdy, když je každý vrchol vrcholově konzistentní.

Hranová konzistence Hrana (V_i, V_j) je *hranově konzistentní (arc consistent)*, právě když pro každou hodnotu x z aktuální domény D_i existuje hodnota y v aktuální doméně D_j tak, že ohodnocení $V_i = x$ a $V_j = y$ splňuje všechny binární podmínky nad V_i, V_j.
CSP je *hranově konzistentní*, právě když je každá jeho hrana (V_i, V_j) hranově konzistentní (v obou směrech).



Hranově nekonzistentní (existuje hodnota v A i B, která nesplňuje podmínku)

Hranově konzistentní (všechny hodnoty v A i B splňují podmínky.

Obrázek 3.2 Hranově nekonzistentní a konzistentní CSP

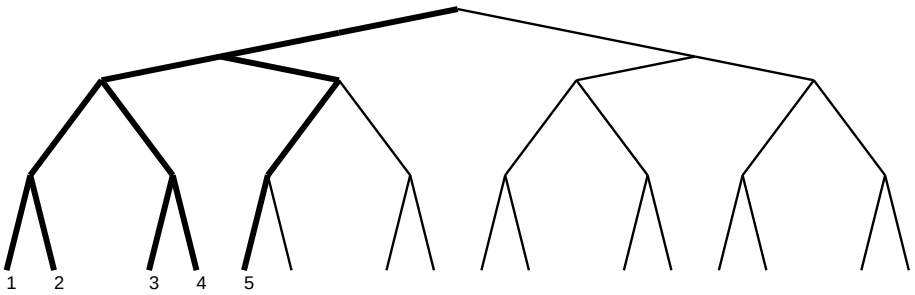
3.2 Prohledávací algoritmy

V této části budou rozebrány 3 hlavní prohledávací algoritmy, které nabízí knihovna Gecode, kterou budeme v naší práci používat a bude jí věnována část 3.4. O informovaných i neinformovaných algoritmech prohledávání pojednává [11–12].

3.2.1 Depth first search (DFS)

DFS (slepé prohledávání do hloubky) patří mezi neinformované metody prohledávání. Algoritmus pracuje se seznamy otevřených a prošlých vrcholů. Prohledávání začíná expanzí kořene a umístěním následníků na začátek seznamu otevřených vrcholů. Pak se expanduje vždy první se seznamu otevřených (jeho následníci se opět uloží na začátek seznamu otevřených) až do okamžiku kdy je nalezeno řešení nebo je seznam otevřených vrcholů prázdný (řešení nenalezeno). Obrázek 3.3 ukazuje, jak DFS funguje na příkladu.

Tato metoda je *úplná*, tj. pokud existuje řešení, pak bude vždy nalezeno. Cenou za tento fakt je možnost expanze neúměrně velkého počtu vrcholů.



Obrázek 3.3 DFS prochází postupně celou větev z kořene stromu a pokud nenarazí na řešení provádí backtracking.

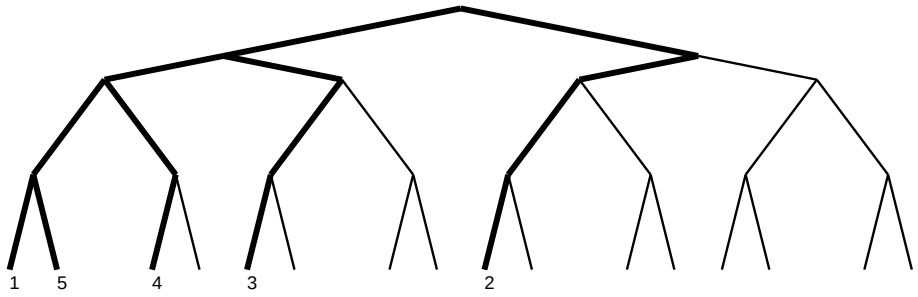
3.2.2 Limited discrepancy search (LDS)

Limited discrepancy search je v constraint-based rozvrhování velmi užívaným algoritmem. Jeho autoři se v [10] zabývali vlastnostmi heuristik a došli k dvěma poznatkům, které zde uplatnili.

- Heuristika je méně spolehlivá na počátku prohledávání (u kořene stromu) než na konci.
- Dobrá heuristika chybuje málo.

Zavedli pojem „diskrepance“, který znamená právě jedno porušení heuristiky.

Algoritmus LDS(n)[5] postupuje podle heuristiky, ale může ji právě n-krát porušit. Porušení heuristiky provádí co nejdříve (kvůli první vlastnosti heuristiky). Pokud nenajde řešení nastává backtracking, který probíhá až do okamžiku, kdy pro zadané n byly všechny větve prohledány. Pak se zvýší n o jedna a hledá znovu. Průchod větvemi prohledávacího stromu ukazuje obrázek 3.4.



Obrázek 3.4 LDS prochází nejdříve větve s nejmenším počtem diskrepancí a upřednostňuje větve, kde je diskrepance blíže kořenu stromu.

3.2.3 Branch and bound search (BAB)

Algoritmus Branch and bound (větví a mezí) má za cíl najít optimální řešení. V podstatě pro svoji činnost využívá *uspořádané prohledávání*[6]. Díky němu najde řešení a pak se snaží nalézt řešení s nižším ohodnocením (lepší). Přitom vyškrtne ze seznamu otevřených vrcholů všechny prvky, které mají ohodnocení vyšší než má nalezené řešení.

---

[5] Pro řešení problému se může použít i více heuristik; pokud je jedna heuristika neúspěšná např. pro n diskrepancí, pak se použije jiná pro $m < n$ počet diskrepancí. Je-li i ta neúspěšná, zvyšuje se dovolený počet diskrepancí a na řadě je opět původní heuristika.

[6] Informovaná metoda prohledávání. Pracuje podobně jako DFS (podsekce 3.2.1). Rozdíl je v tom, že každý vrchol má svoje ohodnocení $g(i)$ a neexpanduje se první vrchol ze seznamu otevřených vrcholů, ale právě ten s nejnižším ohodnocením $g(i)$.

Výhodou BAB oproti DFS je možnost přidávání podmínek za běhu a vlastnost, že najde vždy optimální řešení (pokud řešení existuje). Naopak nevýhodou oproti DFS je možná expanze vrcholů, které mají nízké ohodnocení, ale nevedou k cíli. Ty jsou pak prohledávány zbytečně.

3.3 CP v rozvrhování

Rozvrhovací problémy jako takové spadají do oblasti kombinatorických a optimalizačních úloh (navíc mnohdy patří mezi NP-těžké) a tím pádem se dají dobře popsat omezujícími podmínkami (CSOP).

Nejdříve je nutné vytvořit model, kterým budeme reprezentovat jednotlivé úlohy a stroje a relace mezi nimi.

Úlohy jsou představovány jako proměnné s danou doménou – ta určuje určitý časový okamžik. V obecném případě modelujeme pro každou úlohu $T_j \in \mathcal{T}$ tři proměnné.

- $start(T_j)$: začátek vykonávání s_j
- $end(T_j)$: konec vykonávání C_j
- $p(T_j)$: doba vykonávání p_j

Často se ale setkáváme s případy, kdy je doba vykonávání p_j konstantní a známá předem. Potom pro ni nemusíme vytvářet proměnnou stejně jako pro C_j, který se dá dopočítat.

Podmínky, se kterými se často setkáváme:

- Vztah mezi začátkem a koncem vykonávání úlohy popsaný rovnicí[7] 3.1.
- Precedenční vazby jsou popsány vztahem 3.2

$$start(T_j) + p(T_j) = end(T_j) \tag{3.1}$$

$$end(T_i) \leq start(T_j) \tag{3.2}$$

Stroje se modelují proměnnou $processor(T_j)$ jen v tehdy, když úlohy nemají přiřazený stroj (dedicated processor), na kterém mají běžet. Pak tato proměnná říká, na který stroj byla daná úloha rozvržena.

Jsou-li T_i a T_j úlohy, které mají běžet na stroji s kapacitou 1, pak platí rovnice 3.3.

$$end(T_i) \leq start(T_j) \vee end(T_j) \leq start(T_i) \tag{3.3}$$

[7] U nepreemptivního rozvrhování neplatí.

3.3.1 Edge finding

Velmi dobrým filtrovacím algoritmem v rozvrhování s omezujícími podmínkami je „edge finding“. Jeho charakteristickou vlastností je, že hledá precedenci mezi jednou úlohou a skupinou úloh. Z toho se pak dá usoudit, zda úloha bude provedena před nebo po skupině úloh.

Mějme stroj s jednotkovou kapacitou, úlohu A a skupinu úloh Ω, $A \notin \Omega$. Potom $p(\Omega)$ značí celkový čas potřebný k vykonání všech úloh z množiny Ω.

$$p(\Omega) = \sum_{X \in \Omega} p(X) \tag{3.4}$$

Předpokládejme dále, že úlohy z $\Omega \cup \{A\}$ nezačínají úlohou A. Pak vykonávání úloh musí začít některou úlohou z Ω. Tím pádem nejmenší začátek vykonávání z Ω je

$$\min(start(\Omega)) = \min_{X \in \Omega}\{start(X)\} \tag{3.5}$$

Sečteme-li dobu vykonávání všech úloh s nejmenším začátkem vykonávání a porovnáme s největším koncovým časem všech úloh, dojdeme k závěru, že A musí začínat dříve než jakákoliv úloha z Ω.

$$\min(start(\Omega)) + p(\Omega) + p(A) > \max(end(\Omega \cup \{A\})) \Rightarrow A << \Omega \tag{3.6}$$

Podobně lze zjistit, že úloha A bude následovat po všech úlohách z Ω. Blíže o tomto algoritmu v [5, 8].

3.4 Knihovna Gecode

Gecode (www.gecode.org) je open-source knihovna podporující programování s omezujícími podmínkami. Její výhodou je „otevřenost“, díky které si uživatel může připsat libovolný modul dle vlastních představ. Dalším jejím plusem jsou velmi dobré výsledky v benchmark testech na rychlost řešení a paměťovou náročnost v porovnání s ostatními systémy (http://www.gecode.org/benchmarks.html). Stále je ve vývoji a tedy hlavním problémem je nedostatek dokumentace[8].

[8] dokumentace je pouze formou okomentovaných kódů a několika vzorových programů

4 Řešení problému Job-Shop

Kapitola se bude zabývat návrhem modelu pro CSP a jeho implementací (spojení s knihovnou podporující CP Gecode a Matlabem).

4.1 Model problému

Prvním krokem k využívání programování s omezujícími podmínkami je návrh modelu, který se pak předloží „řešiči"(*constraint solveru*). Jeho výstupem je pak řešení modelu. Jedním modelem se samozřejmě dá reprezentovat více problémů[9]

4.1.1 Reprezentace Job-shopu

Job-shop popsaný v části 2.4.1 je nejznámějším a nejužívanějším *shopem* z uvedených v části 2.4. Přesto jsme chtěli, aby se stejnými objekty dal namodelovat jak Job-shop, tak i Flow-shop resp. Open-shop.

Každá úloha v jednotlivých jobech je vlastně proměnná v diskrétním čase; tím pádem se jednotlivé joby rozloží na jednu množinu úloh, mezi kterými existují precedenční vazby v závislosti na daném problému.

Pro úlohu jsme si vytvořili datovou strukturu, která sice obsahuje pouze zlomek vlastností, které byly jmenovány v části 2.1.1, ale pro tento případ je dostačující. Úloha má vlastnosti:

- Doba vykonávání p_i
- Stroj (dedicated processor)
- Seznam úloh, které musí být provedené před jejím začátkem

Vytvořili jsme seznam všech úloh, který slouží k zadání pro constraint solver. Každý prvek tohoto seznamu koresponduje s jednou proměnnou CSP a má svoji doménu (tu maji ze začátku všechny úlohy stejnou) a precedenční vazby se také transformují na podmínky mezi proměnnými CSP. Navíc jsme vytvořili imaginární („dummy") úlohu, která značí konec celého rozvrhu a má v seznamu předcházejících všechny úlohy.

4.1.2 Vstupní data

Vstup do programu jsme se snažili přizpůsobit standartnímu zápisu pomocí matic podle části 2.4.

[9] Problémy nemusí být ani stejného (např. rozvrhovacího) charakteru.

4.2 Hlavní algoritmus řešení

Chování algoritmu je vidět z vývojového diagramu na obrázku 4.1. Jednotlivé části bych rozvedl.

- Kontrola platnosti formátu dat: Vstupní data z Matlabu se testují, zda mají formát podle vzorů v maticích 2.4. Navíc doby běhu musí být kladné a musí být zvolen druh problému (Flow-shop vs. Job-shop vs. Open-shop). Neprojde-li kontrolou, pak vypíše program chybové hlášení.

- Minimální čas rozvrhu: Zde je nutné uvést hlavní myšlenku algoritmu. V začátcích naší práce jsme zkoumali, zda je výhodnější zvolit si čas, do kterého se rozvrh určitě vejde a hledat řešení v čase o jedna menším nebo vzít malou hodnotu času (určitě menší nebo rovnu optimálnímu času řešení) a hledat řešení pro daný časový interval, který se v případě neúspěchu zvýší o jednu časovou jednotku a hledá se až do okamžiku nalezení prvního řešení. V obou případech máme zaručené optimální řešení (1. případ:pokud pro čas T_d bylo řešení nalezeno a pro $T_d - 1$ nebylo, pak T_d je optimum; 2. případ: pokud pro čas T_u nebylo nalezeno řešení a pro $T_u + 1$ bylo hledání úspěšné, potom je rozvrh určitě optimální). Ukázalo se, že mnohem rychleji dojdeme k řešení aplikováním druhého přístupu, přestože je první přístup obvyklejším.

  V rovnici 4.1 jsme si nadefinovali proměnnou minTime, která značí nejmenší čas, do kterého by bylo možné rozvrh naplánovat.

$$\text{minTime} = \max \left\{ \max_i \sum_j T_{ij},\ \max_k \sum_{\mu(T_{ij})=k} p(T_{ij}) \right\} \tag{4.1}$$

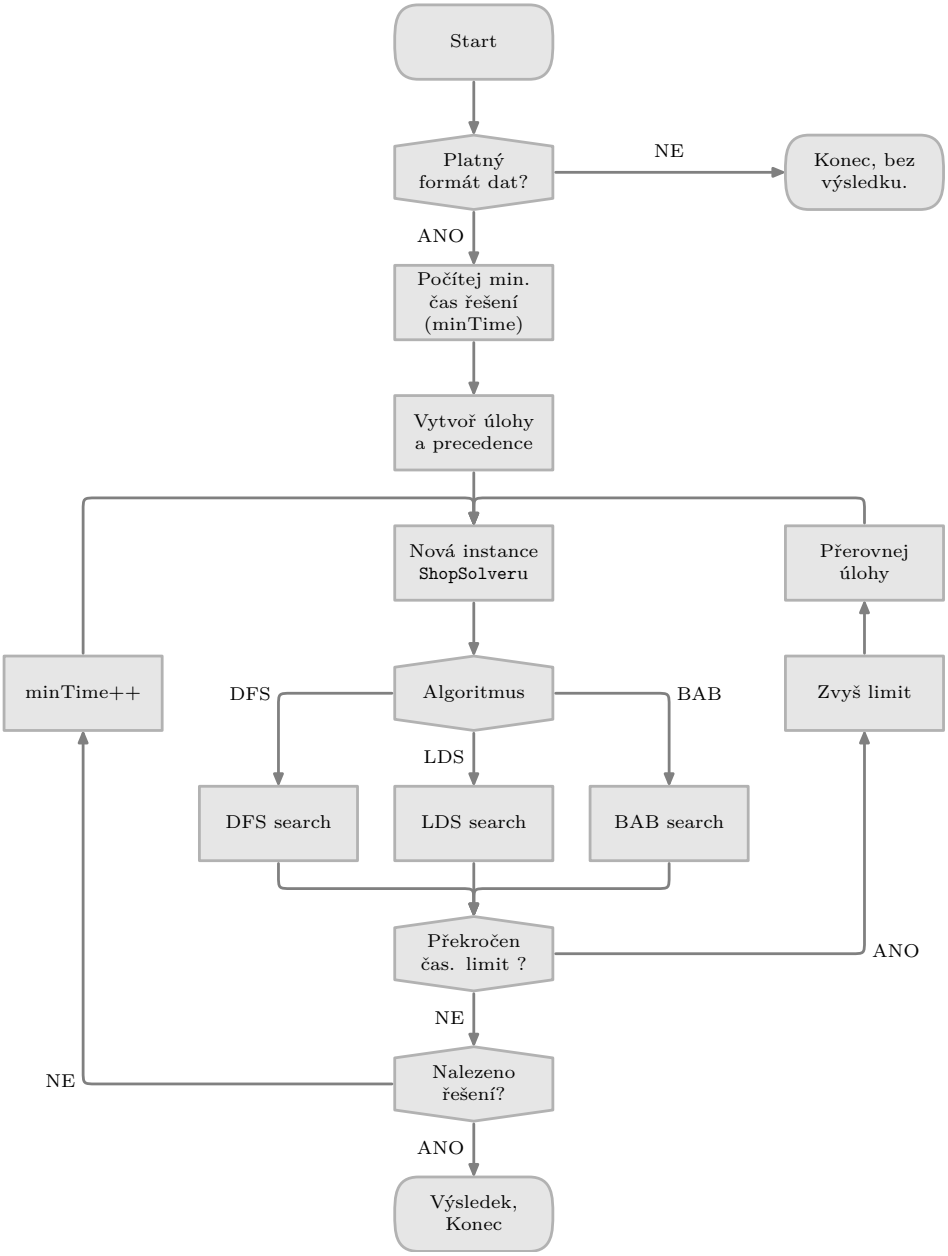
- Algoritmus: knihovna Gecode nabízí 3 prohledávací algoritmy, bohužel nejnadějnější z nich LDS se nepodařilo rozběhnout. Chybu jsme zkoumali a nejspíše je uvnitř knihovny.

  BAB se v našem přístupu k řešení problému redukuje na prostý DFS, protože nedochází k žádnému přidávání nových podmínek za běhu. Výsledky obou algoritmů (v další kapitole) na tento fakt ukazují.

- Překročení časového limitu a přerovnání úloh: Prohledávací algoritmus se čas od času dostane do takové části stromu, která zdaleka nevede k řešení a protože je strom velký, prohledával by velmi dlouho. Vložili jsme do celého prohledávání časové omezení na dobu hledání. Pokud nedojde k nalezení řešení dřív, přistupuje se k zvýšení časového limitu (tím pádem nehrozí zacyklení celého algoritmu) a k přerovnání úloh při konstantní hodnotě minimálního času. Tato funkčnost znovu oživí hledání a ve většině případů dojde k řešení rychleji než bez přerovnání.

$$\text{limit}(x) = n \cdot m \cdot x + \text{limit}(x-1) \tag{4.2}$$
$$\text{limit}(0) = n \cdot m$$

x značí iteraci - kolikrát se dosáhlo časového limitu. n, m je velikost matice T.



Obrázek 4.1 Vývojový diagram algoritmu

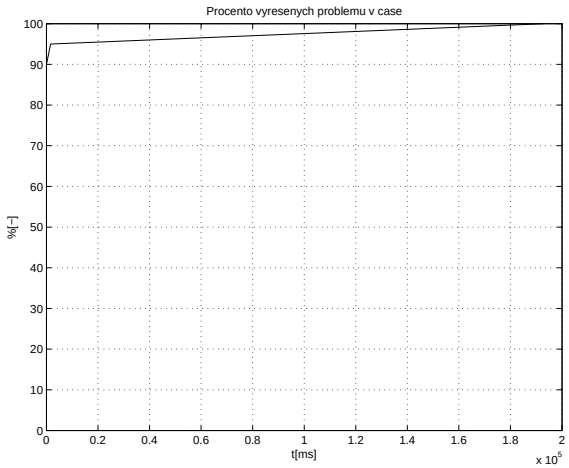
5 Výsledky

V této kapitole budou zmíněny a analyzovány výsledky, bude zde uvedeno několik grafů, na kterých bude tato analýza založena.

Grafy jsou založeny na náhodně vygenerovaných vstupních maticích s rovnoměrným rozložením pravděpodobnosti. V každém případě se graf sestává z 30 vzorků pro jednu velikost problému[10]. Testy jsem prováděl na počítači s procesorem Genuine Intel T1350 (1.86GHz) s OS linux (jádro 2.6.20-15), verzí Matlabu 7.3.0.298 (R2006b) a program byl přeložen pomocí gcc-3.4.

5.1 Job-shop

Závislost doby běhu výpočtu na velikosti Job-shop problému je na obrázku 5.2. Graf není neklesající funkcí, jak by se dalo očekávat. Je to způsobeno tím, že na řešení některých problémů (řádově do 5%) se spotřebuje velmi velké množství času, zatímco ostatní jsou hotovy hned (prořezávání domén je otázkou několika ms). Tento jev je dobře vidět na obrázku 5.1, kde se 2 problémy řešily dlouho a ostatní byly vyřešeny hned. S takovou situací se střetáváme při většině velikostí problémů větších než 5.

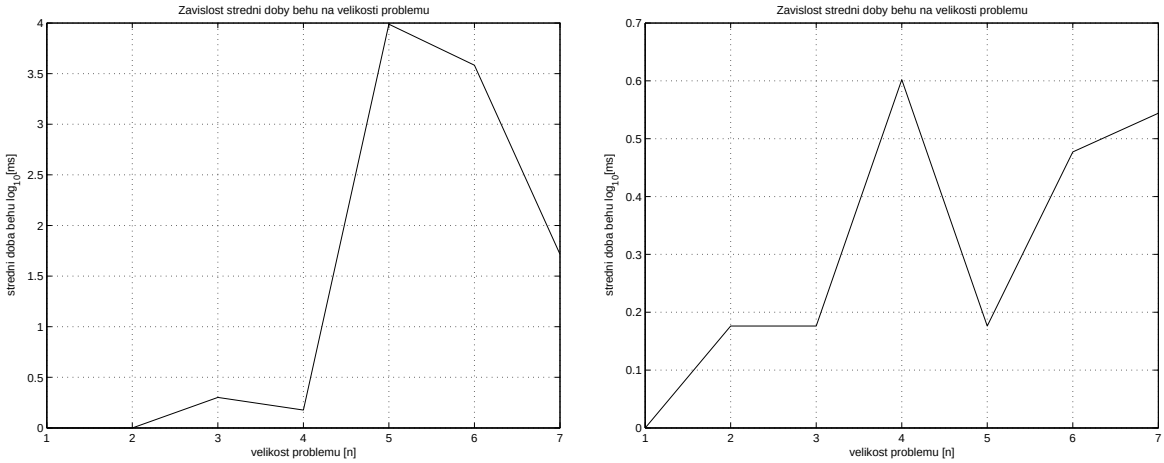


Obrázek 5.1 Procento vyřešených Job-shop problémů velikosti 5 v čase.

Jednoznačným závěrem je, že algoritmus není robustní. A to hlavně kvůli tomu, že se prohledávání dostává do takových oblastí grafu, kde to není zapotřebí. V knihovně gecode bohužel není naimplementován Edge-finder, který by výpočet nejspíše dost urychlil, ale hlavně by pak nedocházelo k tak velkému kolísání v uvedených grafech. Na druhou stranu tento algoritmus vyřeší v celku rychle komplexní benchmarky - například ft6 za 300ms.

DFS i BAB jsou v našem případě shodnými algoritmy a výsledky na to ukazují. Oba dva řešily problém řádově stejně rychle a robustnost byla podobná. Otázkou je, jak by se choval algoritmus LDS.

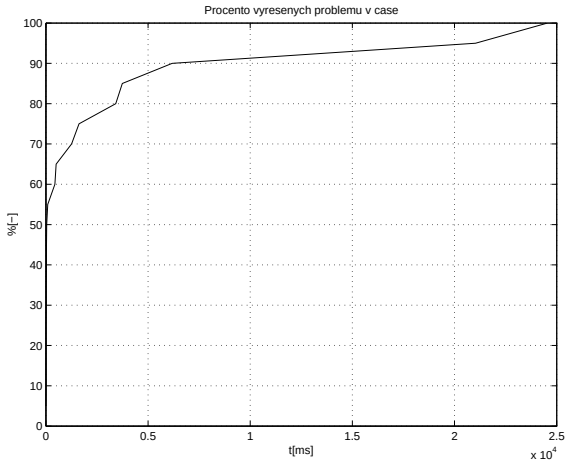
[10] Velikost problému je dána počtem řádků matice. V našem případě je matice vždy čtvercová. Tedy počet úloh rostl s druhou mocninou velikosti problému.



DFS BAB

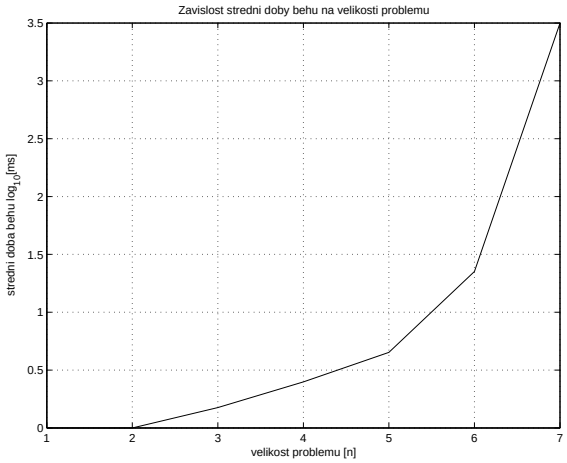
Obrázek 5.2 Porovnání doby řešení různě velkých Job-shop problémů různými prohledávacími algoritmy (časy jsou v logaritmech).

5.2 Flow-shop

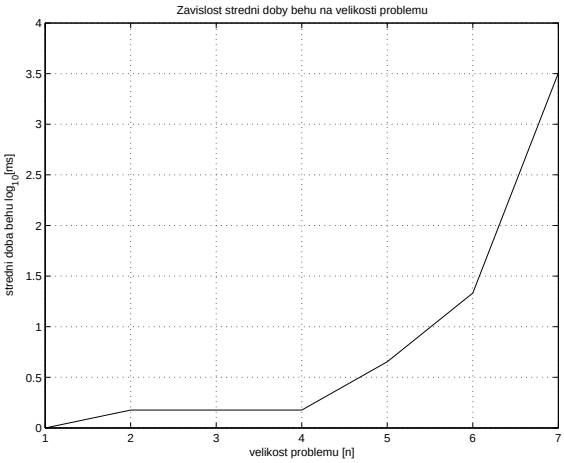


Obrázek 5.3 Procento vyřešených Flow-shop problémů velikosti 7 v čase.

Porovnání algoritmů DFS a BAB (obrázek 5.4) pro tento případ dopadne stejně. Oba algoritmy řeší Flow-shop zhruba stejně rychle. Bez větších výkyvů a křivka je podle předpokladu rostoucí.
I rozptyl doby trvání řešení se snížil v porovnání s Job-shopem. Ukazuje to obrázek 5.3.



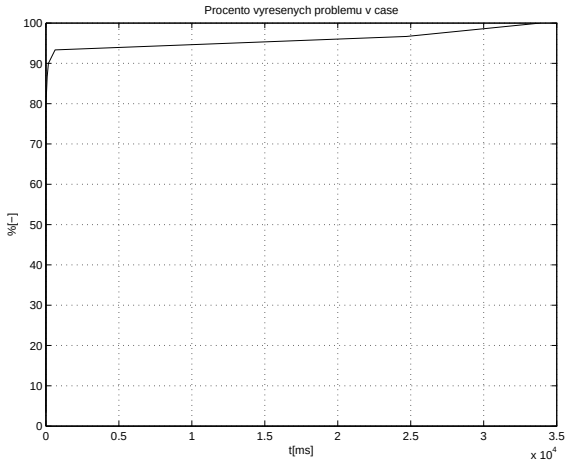
DFS



BAB

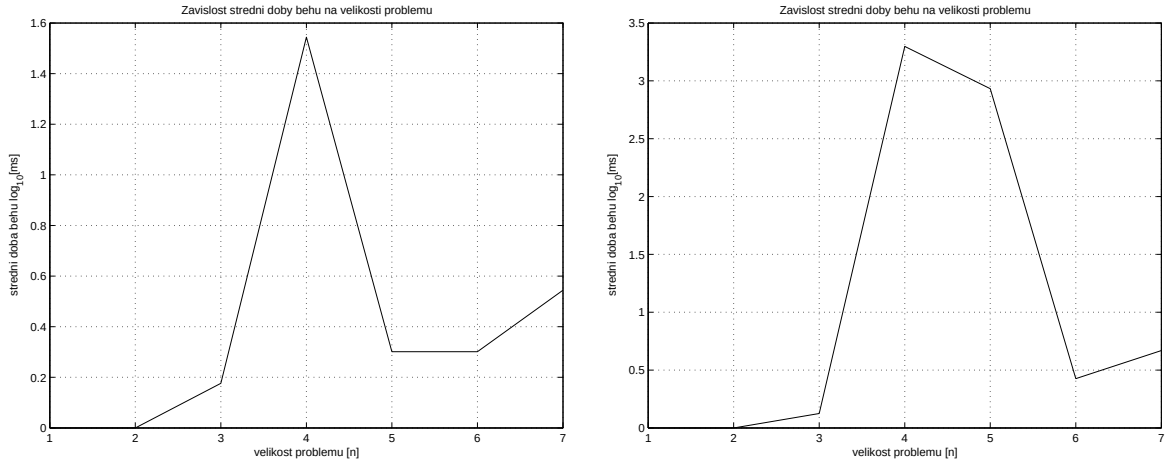
Obrázek 5.4 Porovnání doby řešení různě velkých Flow-shop problémů různými prohledávacími algoritmy (časy jsou v logaritmech).

5.3 Open-shop



Obrázek 5.5 Procento vyřešených problémů Open-shop velikosti 4 v čase.

Známky, které vykazovaly algoritmy u řešení Job-shopu, jsou stejné i u Open-shopu. Velké výkyvy doby běhu, nižší robustnost - obrázek 5.6 porovnává DFS a BAB, 5.5 ukazuje procento vyřešení v čase. I zde se většinou najde jedno zadání, které trvá delší dobu.



DFS BAB

Obrázek 5.6 Porovnání doby řešení různě velkých Open-shop problémů různými prohledávacími algoritmy (časy jsou v logaritmech).

6 Objektový návrh pro Matlab

Kapitola se zabývá návrhem obecného Shopu - nového objektu, který by měl vzniknout v TORSCHE Scheduling toolboxu. Nejprve budou zmíněny požadavky, které jsou kladeny na tento objekt. Poté bude čtenář obeznámen se samotným designem.

6.1 Požadavky na Shop

Přidávání nových objektů a nových funkčností do už vyvinutých aplikací (případně do aplikací, které jsou už delší dobu vyvíjeny a nejsou doposud úplně hotovy) je vždy trochu oříšek. Návrhář musí brát ohled na to, aby nedošlo k situaci, kdy se některé objekty budou překrývat - bude docházet k redundanci. Nepříjemná je také situace, kdy se pokusí o zcelení původních objektů s novými, které vede k velmi složitým vztahům mezi objekty a tím se pak zvyšuje riziko chyby (špatného návrhu). O to horší má pak situaci další návrhář, který přidává další objekt do už tak složitého systému.

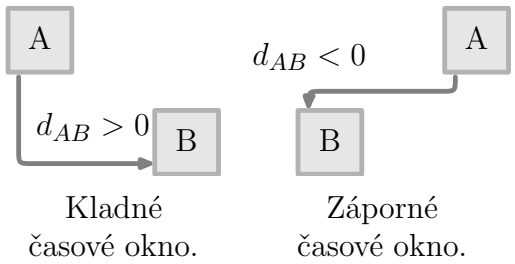
Cílem je tedy kompromis mezi všemi kritérii.

Požadavky na objekt Shop jsme získali shrnutím informací z literatury [1–5], kde jsme se setkávali s níže uvedenými pojmy, které rozšiřují teorii v kapitole 2 a zejména v části 2.4.

Positive and negative time lags (časová okna, zobecněné relace následnosti) jsou zmiňovány prakticky u všech autorů. Jedná se o časové vztahy mezi jednotlivými úlohami určující dobu mezi začátky úloh podle vztahu 6.1.

$$s_A + d_{AB} \leq s_B \tag{6.1}$$

d_{AB} značí časové okno mezi úlohami A a B. Tato doba může být obecně i záporná vše zobrazuje obrázek 6.1.



Obrázek 6.1 Dvě možnosti velikosti časového okna. Z obrázků je zřejmé, proč se časovému oknu také říká zobecněná relace následnosti.

Stejně často se setkáváme i s *batch processing* – joby se seskupují do *dávek* a ty jsou pak plánovány jako rozvrhovací jednotky.

Další poznatky směřovaly většinou do oblasti zdrojů – [2] se zmiňuje například o zdrojích s kapacitou n (může být využíván v jeden okamžik až n úlohami), obnovitelnosti zdrojů (dostupnost zdroje je funkcí času a závisí na jeho předchozí vytíženosti; zdroje mohou být úplně neobnovitelné, anebo částečně obnovitelné v jistých časových okamžicích) a jejich různých režimech (výkon zdroje je funkcí času[11]).

V [2–3] se setkáváme s pojmy *transport robots* (transportní roboti) a *limited buffers* (mezisklady s omezenou kapacitou).

Transportní roboti se vyskytují tam, kde je nezanedbatelná doba mezi koncem úlohy na jednom stroji a začátkem úlohy na jiném. Po tuto dobu právě robot přenáší výrobek mezi jednotlivými stroji. Setkáváme se s dvěmi situacemi, které mohou nastat:

- Neomezený počet robotů: Každý job má alespoň jednoho robota[12]. Tím pádem není problém s konflikty mezi joby transportní čas se modeluje jako time lag.
- Omezený počet robotů: Je méně robotů než je jobů. Pak se roboti modelují jako stroje s kapacitou 1.

Mezisklady s omezenou kapacitou budou ukázány na příkladu:

Mějme množinu jobů sestávajících se z několika úloh. Může nastat situace, kdy jedna úloha z jobu J_i je dokončena v čase t_0 na stroji μ_a. Job J_i má pokračovat další úlohou, která má být vykonávána na stroji μ_b. Shodou okolností je zrovna stroj μ_b vytížen až do času t_1. Tím pádem po dobu $\Delta t = t_1 - t_0$ blokuje z jobu J_i stroj μ_a a žádná jiná úloha na něm běžet nemůže.

Proto se zavádějí mezisklady, kam se tyto úlohy odloží a neblokují stroj. Pokud už je mezisklad plný, nezbývá než čekat na uvolnění dalších strojů, aby se mezisklad uvolnil.

Existuje několik modelů meziskladů.

- *General*: existuje n „centrálních“ meziskladů, které využívají všechny úlohy.
- *Job-dependent*: Každý job má svůj mezisklad.
- *Pairwise*: tento model svazuje stroje do dvojic a každé dvojici přiřazuje mezisklad.
- *Input*: na vstupu každého stroje je mezisklad velikosti n.
- *Output*: na výstupu každého stroje je mezisklad velikosti n.

[11] Je-li zdroj člověk, pak mívá po obědě menší pracovní výkonnost než poránu, když přijde do práce.

[12] Má-li jich přidělených více, pak jsou přebyteční; Pokud rozvineme pojem „robot“ i na člověka, pak si pod tím můžeme představit několik kopáčů sledujících jednoho kopáče, který občas kope.

V [4] se dočteme o paralelních strojích pro Job-shop a Flow-shop. Ty pak mají přízvisko *flexible*. A dále také o tzv. *supply-chain* – množina shopů je svázána do sítě s určitými precedenčními vazbami.

Souhrnem a kompromisem mezi vším výše zmíněným a dalšími našimi nápady je tento seznam požadavků na objekt Shop:

- Různé délky jobů – joby mohou obsahovat proměnný počet úloh.
- Zdroje s kapacitou > 1.
- Alternativní zdroje – uživatel bude mít možnost nadefinovat si pro danou úlohu více zdrojů, na kterých může být provedena.
- Paralelní zpracovávání úlohy na více strojích.
- Zobecněné relace následnosti
- Transportní roboti
- Mezisklady s omezenou kapacitou.

6.2 Návrh objektu Shop

Při vlastním návrhu jsme vycházeli z poznatků a požadavků v předchozí části.

Různé délky jednotlivých jobů jsme vyřešili použitím struktury TaskSet. Tím pádem je vstupem objektu Shop cell struktura, která obsahuje n TaskSetů odpovídajících n jobům. Tento přístup navíc vede k dobrému zakomponování objektu Shop do struktury celého toolboxu – TaskSet je totiž jedním z jeho pilířů.

Paralelní a alternativní zdroje se realizují přímo na úrovni objektu Task, který obsahuje proměnnou processor (matice). Hodnoty v prvním řádku matice určují alternativní zdroje; má-li matice více řádků, pak sloupce odpovídají množině zdrojů, které mohou úlohu paralelně vykonávat. Vše je na níže uvedeném příkladu (paralelní zpracovávání ještě nejsou v současné verzi Tasku k dispozici, proto není uvedený celý výstup, ale pouze předpokládané volání funkce).

Zdrojový kód

```
>> t=task(1) % vytvoření úlohy
Task ""
 Processing time: 1
 Release time:    0
>> t.Processor=[1 3]  % Úloha může být vykonána na stroji 1 nebo 3
Task ""
 Processing time: 1
 Release time:    0
```

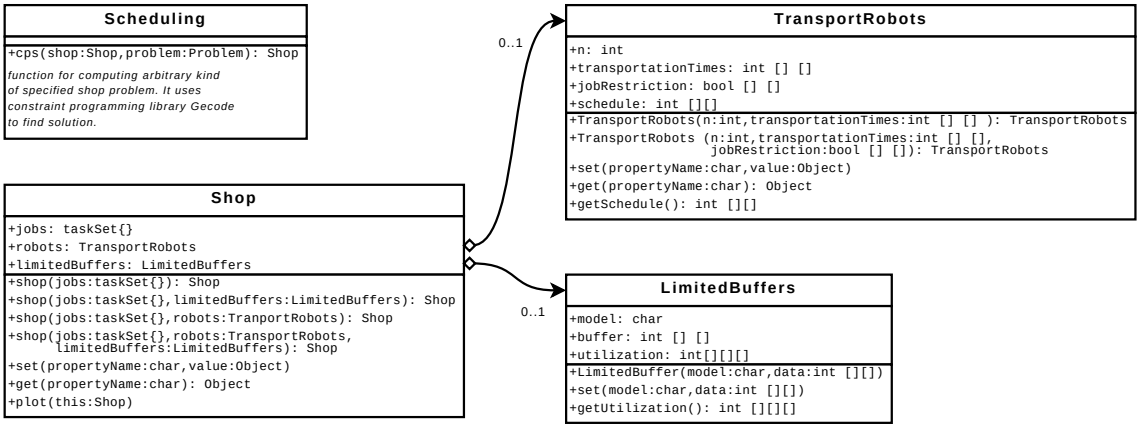
```
 Processor:       1  3
>> t.ProcTime=[2 3]  %Na stroji 1 resp. 3 bude trvat vykonávání 2 resp. 3 jednotky
času
Task ""
 Processing time: 2  3
 Release time:    0
 Processor:       1  3
>> t.Processor=[2 4;3 3] % Úloha může být vykonána na stroji 2 a zároveň 3 nebo 4 a
zároveň 3.
```

Zdroje s kapacitou vyšší než 1 se budou vkládat jako parametr algoritmu řešícího rozvrhovací problém.

Zobecněné relace následnosti rozšíří objekt TaskSet a bude se zadávat do TSUserParam (nepovinný uživatelsky definovaný parametr). Tento parametr se bude zadávat pomocí matice, kde se nadefinují časové intervaly mezi jednotlivými úlohami.

Transportní roboti a mezisklady s omezenou kapacitou tvoří na jednu stranu jednotnou a ucelenou oblast problémů, ale na druhou stranu není jejich četnost výskytu tak vysoká. Hlavně proto jsme vytvořili objekty (obsahující všechny zmíněné) modely pro obě kategorie problémů, ale umístili jsme je do UserParams objektu shop (nepovinné parametry).

Vše přesně zobrazuje obrázek 6.2. Celé UML je v příloze A.



Obrázek 6.2 UML class diagram general shopu v TORSCHE Scheduling toolboxu

7 Shrnutí

V této práci jsme se seznamovali s rozvrhováním jako takovým a s programováním s omezujícími podmínkami (CP). Hlavním cílem práce bylo využít CP k řešení rozvrhovacích problémů dílen (shopů). Z počátku jsme váhali nad správností výběru knihovny Gecode, protože řešení i malých problémů trvalo velmi dlouhou dobu. Po několika zdokonaleních algoritmu proběhne výpočet ve většině případech v zanedbatelném čase. Implementace Edge-finderu (případně prohledávacího algoritmu LDS) je výzvou do budoucna k zrychlení výpočtu.

Využili jsme také vhodného návrhu k tomu, že jsme implementovali řešení i dalších shopů - řešení Open-shopu je časově podobně náročné jako Job-shop, zatímco Flow-shop pracuje nejrychleji a i nejspolehlivěji. U předchozích dvou případů dochází zhruba u 5% zadání k zdlouhavému hledání, které je řádově delší než u většiny.

V další části práce jsme se zabývali návrhem objektu Shop. Jeho struktura je kompromisem mezi všemi návrhy a čeká na implementaci.

Literatura

[1] J. Błazewicz, J. Węglarz, K. Ecker, and G. Schmidt, *Scheduling in computer and manufacturing systems.* (Springer, 1994).

[2] P. Brucker and S. Knust, *Complex Scheduling.* (Springer, 2006).

[3] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems.* (Prentice Hall, 2001).

[4] M. Pinedo, *Planning and Scheduling in Manufacturing and Services.* (Springer, 2006).

[5] P. Baptiste, C. L. Pape, and W. Nuijten, *Constraint-Based Scheduling - Applying Constraint Programming to Scheduling Problems.* (Springer, 2005).

[6] , *Gantt chart - wikipedia, the free encyclopedia* (2007).

[7] M. Kutil, P. Šůcha, M. Sojka, and Z. Hanzálek *TORSCHE Scheduling Toolbox for Matlab, User's Guide.* (2007).

[8] R. Barták, *Constraint-based scheduling: An introduction for newcomers* Tech. Rep., Institute for Theoretical Computer Science (2003).

[9] R. Barták, *Online guide to constraint programming* (1998).

[10] W. D. Harvey and M. L. Ginsberg, *Limited discrepancy search* (1995)

[11] V. Mařík, O. Štěpánková, J. Lažanský, and kol., *Umělá inteligence (1).* (ACADEMIA, 1993).

[12] S. Russell and P. Norvig, *Artificial Intelligence - A Modern Approach.* (Prentice Hall, 2003). RUS st 03:1 1.Ex.

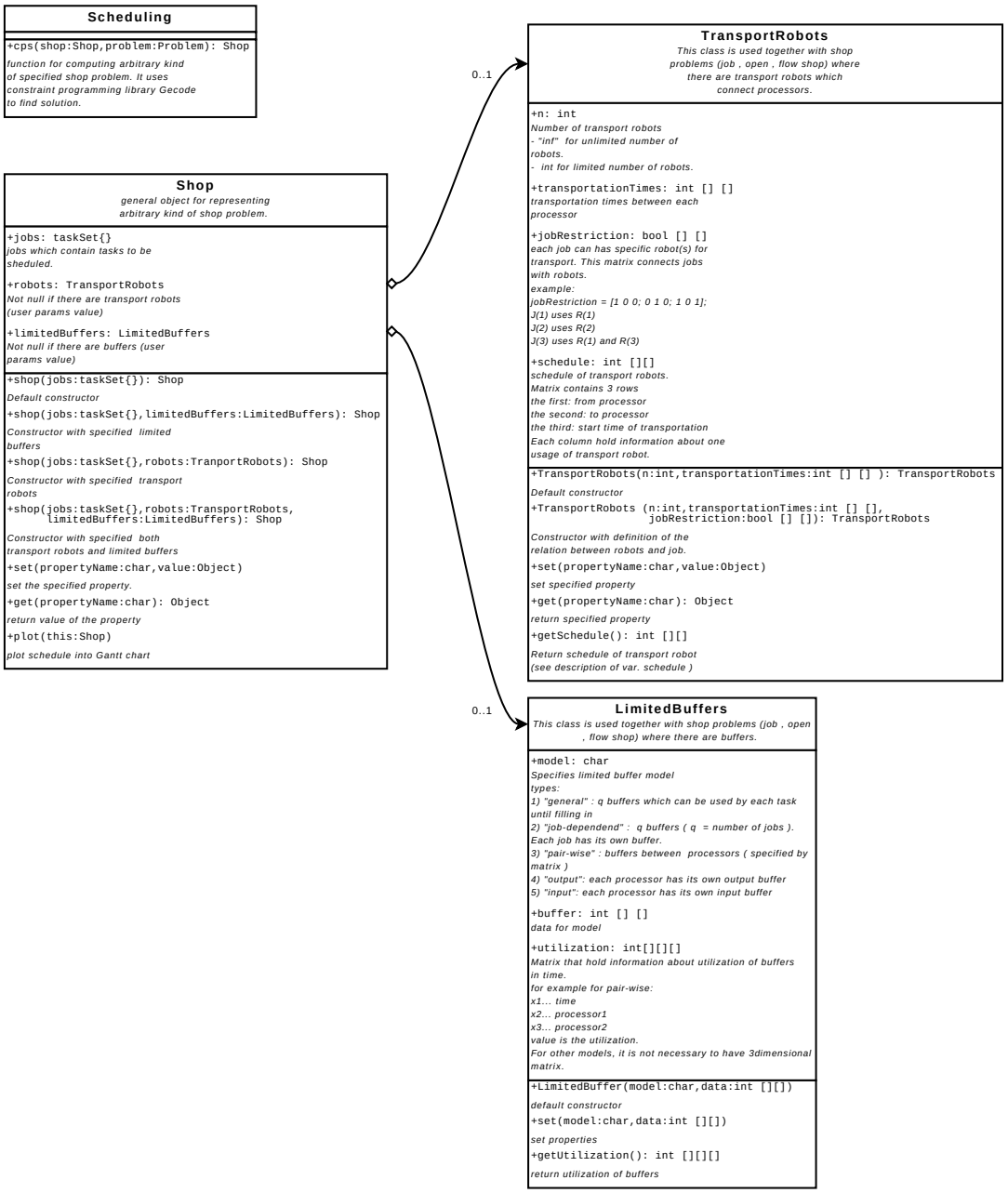
[13] P. Šůcha, *Celočíselné lineární programování* (2004)

[14] S. Khuri and S. R. Miryala, *Genetic algorithms for solving open shop scheduling problems* (1999)

[15] V. Lowndes, J. M. Carter, M. H. Wu, and S. Berry, *Fuzzy modelling applied to jobshop scheduling* (2003)

Příloha A UML diagram podrobně

Obrázek A.1 zobrazuje UML návrh s kompletně rozepsanými metodami, členskými proměnnými a jejich komentáři pro jednotlivé objekty.



Obrázek A.1 Podrobně rozepsaný UML diagram.

Příloha B Dokumentace k programu

Dokumentace k programu je na přiloženém CD. Dá se ovšem náléz i na stránkách projektu na http://code.google.com/p/jcjobshop/.
Přiložené CD obsahuje:

- Dokumetace k zdrojovým kódům (formát HTML)
- Návod na instalaci programu.
- Zdrojové kódy[13].
- Stručný návod na ovládání programu.
- Tato práce ve formátu PDF.

[13] Zdrojové kódy jsou pod licencí GNU General Public License 2.0