ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA ELEKTROTECHNICKÁ

BAKALÁŘSKÁ PRÁCE

Rozvrhování pomocí lokálního prohledávání

Praha, 2009

Autor: Heincl Roman

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídicí techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Roman Heincl**

Studijní program: Elektrotechnika a informatika (bakalářský), strukturovaný
Obor: Kybernetika a měření

Název tématu: **Rozvrhování pomocí lokálního prohledávání**

Pokyny pro vypracování:

1. Seznamte se s pojmy z oblasti rozvrhování.
2. Nastudujte jazyk Comet pro deklarativní popis kombinatorických problémů a jejich řešení pomocí metod lokálního prohledávání.
3. Vyřešte pomocí jazyka Comet zadaný rozvrhovací problém.

Seznam odborné literatury:

Dodá vedoucí práce

Vedoucí: Ing. Jan Kelbel

Platnost zadání: do konce zimního semestru 2008/2009

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 25. 2. 2008

Prohlášení

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

V Praze dne_____ _____

podpis

Poděkování

Děkuji především vedoucímu mé bakalářské práce Ing. Janu Kelbelovi za jeho ochotu, trpělivost a cenné rady, které mi poskytoval při vedení mé práce. Dále bych chtěl poděkovat své rodině a všem svým přátelům za podporu při studiu.

Abstrakt

Cílem této bakalářské práce je seznámit se s problematikou rozvrhování pomocí lokálního prohledávání, nastudování programovacího jazyka COMET a aplikace znalostí při vlastním řešení rozvrhovacího problému. COMET je inovativní objektově orientovaný jazyk vyvinutý pro potřeby řešení rozvrhovacích problémů, jakým je např. plánování průmyslové výroby a minimalizace nákladů na ni. Hlavní výhodou jazyka COMET je separace modelu a prohledávání jako takového.

Abstract

The aim of this bachelor is to acquaint with a scheduling by local search, to learn COMET programming language and to apply this knowledge on my own scheduling problem. COMET is an innovative object-oriented language developed for scheduling problems, e.g. a scheduling of an industrial production and its cost minimalization. The main advantage of COMET language is a separation of a model and search components.

Obsah

Kapitola 1

Úvod

Rozvrhovací a optimalizační problémy jsou každodenní součástí našeho života. Běžně se s nimi setkáváme v dopravě, při plánování výroby nebo například zásobování. Rozvrhování a následnou optimalizaci provádíme, aniž bychom si to uvědomovali, ať už je to správa našeho časového harmonogramu na příští den nebo příprava večeře o více chodech. Účelem těchto aktivit je co nejvíce šetřit našimi zdroji, tj. časem, energiemi a finančními prostředky. Během posledních desetiletí technologie využívající optimalizaci významně pokročily a umožňují nám tak aktivně ovlivňovat procesy i čelit kritickým situacím.

Optimalizační nástroje pro řešení rozvrhovacích problémů byly často soustředěny okolo prozkoumávání stavového prostoru. Algoritmy, které jsou pro hledání řešení v těchto situacích používány, využívají různá kritéria při rozhodování, do které větve (resp. částí stavového prostoru) se algoritmus vydá. Princip toho řešení s sebou přináší hlavní nevýhodu, jíž je velká časová náročnost při průzkumu celého stavového prostoru, do něhož spadají i situace, které nemůžou v reálném světě nikdy nastat. Z toho předpokladu vycházejí principy lokálního prohledávaní. Hlavní úlohou je tedy definice stavového prostoru v bodě, v němž se algoritmus právě nachází, a efektivní prozkoumání jeho okolí. Lokální prohledávání využívá i tzv. skoku (úniku z lokálního minima) k tomu, aby se pokusilo nalézt výhodnější řešení daného problému. Následek toho je časově přijatelné nalezení řešení, které ovšem nemusí být to nejlepší možné a občas tyto algoritmy mohou selhat zcela. Tyto vlastnosti se využívají především při řešení problémů s rozsáhlým stavovým prostorem nebo také online problémech, kdy potřebujeme přijatelné řešení v omezeném čase. Z dosavadních zkušeností je zřejmé, že v mnoha aplikacích jsou inovativní algoritmy využívající lokální vyhledávání mnohem rychlejší a jejich výsledky se blíží nebo odpovídají optimu těchto problémů.

Kvůli potřebě implementace těchto možností do vyššího programovacího jazyka byl v roce 2001 odstartován projekt COMET. Výsledkem toho úsilí je programovací jazyk nesoucí jméno projektu. COMET je objektově orientovaný jazyk zaměřený na modelování a lokální prohledávání, který využívá především omezujících podmínek. Jeho velkou výhodou je oddělení modelu a vyhledávacího algoritmu.

Pomocí tohoto nástroje lze řešit množství kombinatorických problémů včetně problémů spojených s rozvrhováním. Do této kategorie spadají i problémy rozvrhovaní výroby. Základ problému tvoří stroje a úkoly, které vykonávají. Výroba se obvykle skládá z jistého množství jednotlivých úkonů, které lze provádět pouze na určitých strojích a v předem známém pořadí. Speciálním případem je situace, kdy máme pouze jeden stroj, jednotlivé úlohy mohou být plněny v libovolném pořadí a celkové náklady na výrobu jsou vyjádřeny pomocí cen za předčasné a zpožděné dokončení. Tento problém se nazývá rozvrhování na jednom stroji s penaltami za předčasné a zpožděné dokončení (Single Machine Earliness Tardiness Scheduling Problem (ETSP)), penalty zastupují náklady na skladování resp. sankce od zákazníka za opoždění zakázky a právě tímto případem a jeho řešením v COMETu se zabývám ve své bakalářské práci.

Tato práce je rozdělena na tři části. V první je detailněji popsána problematika rozvrhování a rozvrhovací problém OMETJSSP. V druhé části se věnuji programovacímu jazyku COMET a inovacemi, jenž přináší. Část třetí se zabývá vlastní prací, algoritmem jenž jsem vymyslel, jeho implementací v jazyce COMET a popisem jednotlivých částí zdrojového kódu.

Kapitola 2

Úvod do problematiky rozvrhování

2.1 Význam rozvrhování

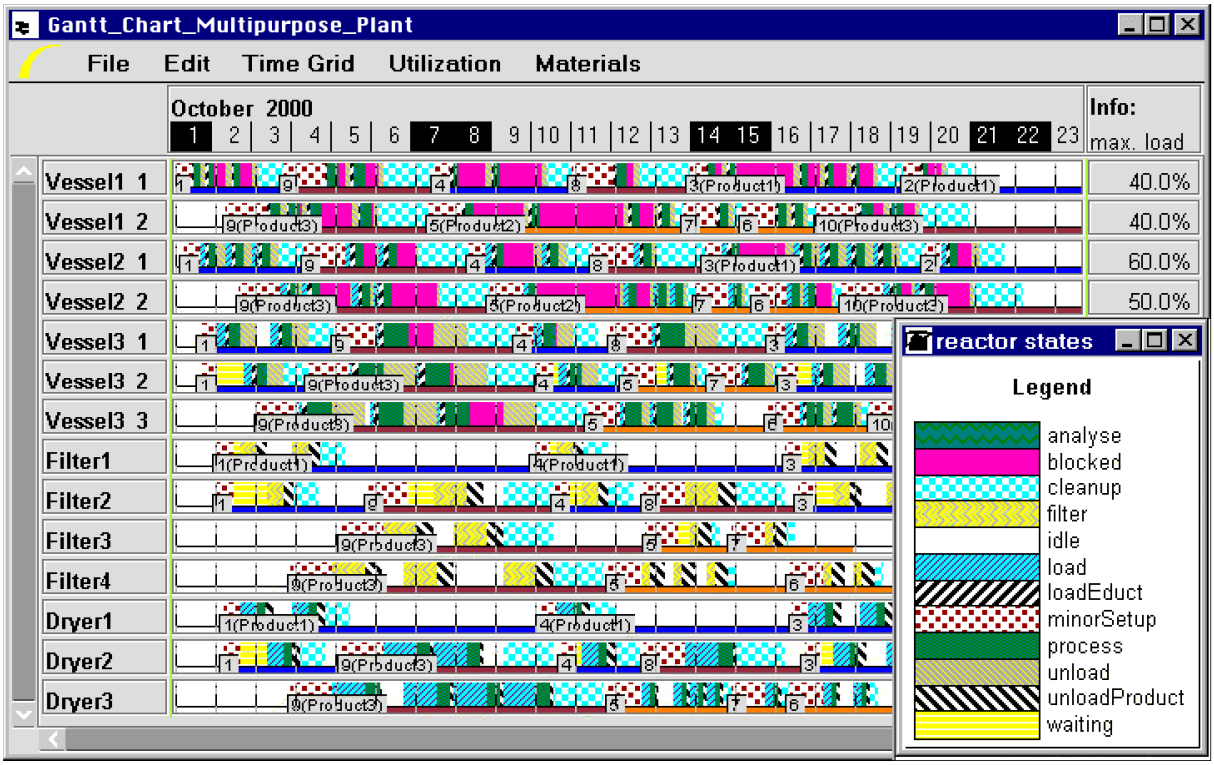
Rozvrhování s použitím kritických cest (CPA) je velice aktuální téma. Rozvrhování bylo definováno jako vědní obor v roce 1956 ale používáno bylo již mnohem dříve. Sun Tzu psal o rozvrhování a strategii z vojenského úhlu pohledu již před 5000 lety. Pyramidy jsou přes 3000 let staré a transkontinentální železnice byla vybudována před více jak 200 lety. Žádná z těchto aktivit by nemohla být dokončena bez jisté formy rozvrhu, tj. porozumění aktivitám a jejich řazení[1]. Úkolem rozvrhování je co nejefektivněji využít naše zdroje k čemuž nám dopomáhají rozvrhovací algoritmy. Díky nim jsme schopni optimalizovaně plánovat velmi složité výrobní procesy a šetřit tak našimi zdroji. Jsou to zásadní faktory, které rozhodují o úspěchu podniků.

2.2 Rozvrhování

Rozvrhování patří mezi kombinatorické problémy. Je definováno jako přiřazení omezeného počtu zdrojů aktivitám v průběhu času[2]. Rozvrhovací problémy mají více kategorií mezi ně patří i ty spojené s výrobou, podrobněji se jim věnuji v kapitole 2.2.1

2.2.1 Výrobní problémy

Velkou kapitolou v rozvrhovacích problémech jsou ty, které se zabývají výrobou a jejím rozvrhováním. Plánování výroby se z počátku provádělo ručně. Při nejjednodušších problémech ani není zapotřebí jakéhokoliv zpracování. Při složitějších aplikacích již musíme použít jisté algoritmy. Ovšem díky velkému výrobnímu rozmachu a automatizaci celého procesu nám starší metody přestávají splňovat požadavky a my jsme nuceni hledat nové. Výrobní procesy jsou v dnešní době natolik komplikované, že v některých případech ani s výpočetním výkonem, který máme k dispozici, nejsme schopni najít přijatelná řešení v krátkodobém časovém horizontu. Ukázka rozvrhování výroby je na obrázku 2.2[3].



Obrázek 2.2 příklad rozvrhování výroby

2.3 Lokální prohledávání

Princip lokálního prohledávání vychází z hlavních nedostatků klasického prohledávání založených na úplných algoritmech. Máme-li stavový prostor a používáme-li úplný algoritmus, který jej prochází celý při hledání našeho řešení, narazíme na problém, že pokud je stavový prostor rozsáhlejší, prohledání tohoto prostoru je časově velice náročné. Lokální prohledávání provádí zkoumání dalšího pohybu ve stavovém prostoru jen v nejbližším okolí bodu, ve kterém se algoritmus právě nachází. Díky tomu může nalézt požadované řešení velmi rychle. Avšak rychlost má svou cenu a může se stát, že algoritmem nalezené řešení je pouze lokální minimum, případně algoritmus při hledání řešení selže zcela. Aby se tak stalo v co nejméně případech, existují metody jak tomu předejít. Z lokálních minim se můžeme pokusit uniknout tak, že provedeme náhodný skok do okolí bodu, ve kterém se algoritmus momentálně nachází. Při porovnání výsledků úplných algoritmů a těch využívajících lokálního prohledávání jsou hlavní dvě kritéria. Prvním je úspěšnost inovativních algoritmů v nalezení, popř. v přiblížení optimálnímu řešení známému díky úplným algoritmům a druhým je porovnání časové náročnosti obou způsobů řešení.

2.4 Rozvrhování na jednom stroji s penaltami za předčasné a zpožděné dokončení

Speciálním případem rozvrhovacího problému je rozvrhování na jednom stroji s penaltami za předčasné a zpožděné dokončení (One Machine Earliness Tardiness Scheduling Problem). Tento problém je relativně nový. V dřívějších dobách, kdy byla výroba plánována podle faktorů jako procento zpožděných aktivit, celkové zpoždění nebo průměrná doba dokončení, nebyla potřeba řešení OMETSP. Cena za předčasné dokončení nebyla brána v potaz. Tento požadavek vznikl až na základě změny v průmyslové výrobě, kdy se postupně začala objevovat i její varianta Just-In-Time (právě včas). Právě pro tento druh výroby se stala cena za předčasné dokončení stejně důležitým faktorem jako cena za zpožděné dokončení vzhledem k tomu, že pokud se výrobek vyrobí dříve, než je požadavek zákazníka, je nutné jej skladovat a to stojí také nějaké zdroje.

Základem OMETSP je množina J n aktivit $\{J_1, \dots, J_n\}$ a množina R m strojů $\{R_1, \dots, R_m\}$, která je v tomto případě jednoprvková [4].

Každá z aktivit problému má 5 parametrů:

- d - due date (termín dokončení)
- p – processing time (dobu trvání)
- C – completion time (koncový čas)
- α – earliness factor (cena za předčasné dokončení)
- β – tardiness factor (cena za zpožděné dokončení)

Doba trvání jednotlivých aktivit musí být pozitivní. Problém je omezen disjunktivním pravidle, tzn. že v jeden okamžik může být na stroji vykonávána pouze jedna aktivita.

Jakmile jsou tyto podmínky splněny, jednotlivým aktivitám je přiřazován jejich termín dokončení. Z toho také vychází cena jednotlivých aktivit e dá vyjádřit následující rovnicemi.

Pokud je koncový čas menší než termín dokončení, aktivita je dokončena předčasně a je penalizována podle vztahu:

$$\alpha(d_j - C_j)$$

Pokud je koncový čas větší než termín dokončení, aktivita je dokončena se zpožděním a je penalizována podle vztahu

$$\beta(C_a - d_a)$$

Cena aktivity, která je součásti OMETSP, pak odpovídá vztahu

$$f_j = \max\left(\alpha(d_j - C_j), \beta(C_j - d_j)\right)$$

Celková cena rozvrhu se skládá z dílčích cen aktivit a dá se vyjádřit funkcemi:

$$1|d_j|\sum \alpha_j E_j + \beta_j T_j$$

$$1|\ \ |\sum_j T_j$$

Cíl algoritmu, její minimalizace odpovídá:

$$\min \sum_{j\epsilon J} \mathrm{f_j}$$

Jak již z logiky problému vyplývá, nejnižší celkové cena, cena nulová, nastává pokud se koncový čas všechny dílčí aktivity shoduje s jejich termínem dokončení. V takovémto případě není nucen výrobce skladovat výrobky ani není vystaven postihům ze strany zákazníka za zpožděné dodání zakázky.

Problém rozvrhování na jednom stroji s earliness/tardiness penaltami je NP-těžký, protože je rozšířením problému rozvrhování na jednom stroji s minimalizováním celkového zpoždění dle $1|\ \ |\sum_j T_j$, který je NP-těžký[5].

Model OMETSP se zabývá statickým rozvrhováním. Jedná se o situaci, kdy všechny aktivity jsou předem známy včetně všech jejich parametrů a mohou být zahájeny kdykoliv v průběhu času. Existuje více variant toho problému. Shrnuje je článek [4]. Mou variantou problému, kdy každá aktivita může mít různé ceny ze předčasné a zpožděné dokončení, se zabývali v minulosti v minulosti i Bagchi (1985), Sullivan a Chang (1987), Cheng (1987), Quasddus(1987), Bector, Gupta a Gupta (1988) a Hall a Posner (1989). Dosavadním řešením tohoto problému se zabývají také autoři článku [6]. Základním polynomial-time algoritmem pro řešení OMETSP se zabývají Verma a Dessouky a jejich algoritmus umí řešit případy, kdy $\alpha_1 \leq \cdots . \leq \alpha_n$ a $\beta_1 \leq \cdots . \leq \beta_n$. Dalším krokem byl tzv. branch-and-bound algoritmus, kterým poprvé použil Fry a byl navržen pro speciální případ, kdy jsou všechny ceny za předčasné dokončení stejné stejně jako se sobě rovnají i ceny za zpožděné dokončení. Tento algoritmus uměl vyřešit až dvanáct aktivit. Dalším případem, kdy $\alpha = \beta = 1$, se zabýval Kim a Yano, kteří navrhli algoritmus pro řešení až 25 aktivit. Následující krok udělal Chang pro případ symetrických penalizací $\alpha_j = \beta_j$. Jeho algoritmus byl schopen vyřešit rozvrhovací problém s až 35 aktivitami, v některých případech dokonce 40 resp. 45 aktivitami. Nakonec autoři článku poskytují vlastní algoritmus pro řešení OMETSP. Jedná se o tzv. lower bound založený na Legrangeanově povolení časově koncipovaných

pravidel. Jedná se o dočasné povolení možnosti vícenásobného výskytu aktivit v rozvrhu a o uvolnění pravidel pro stroje. Tento algoritmus je v kapitole 4.7 použit pro srovnání mých výsledků.

Kapitola 3

COMET

3.1 Charakteristika jazyka COMET

Jak již bylo zmíněno v úvodu, COMET je objektově orientovaný jazyk zaměřený na modelování a lokální prohledávání, který využívá především omezujících podmínek a vznikl jako výsledek stejnojmenného projektu založeného v roce 2001 na americké universitě v Massachusetts. Jeho velkou výhodou je oddělení modelu a vyhledávacího algoritmu. Materiály publikované v této kapitole jsou ze zdroje [6].

3.2 Nástroje jazyka COMET

COMET nám nabízí spoustu nástrojů, díky kterým můžeme řešení rozvrhovacích problémů značně zjednodušit a urychlit. Mezi základní patří invarianty, skupina tzv. diferencovatelných objektů (differentiable objects), mezi tyto spadají omezující podmínky a kritéria, a další nástroje pro ovládání, jako jsou selektory, řešení, události a kontrolní body. Detailnější popis se nachází dále v této kapitole.

3.2.1 Local Solver

Local Solver (volně přeloženo místní řešitel) je základní datový objekt jazyka COMET, který nám uchovává hodnoty veškerých proměnných, které jsou v něm deklarovány.

3.2.2 Invarianty

Invarianty, jinak také jednostranná omezení, jsou to základní nástroj jazyka COMET a jsou definovány například takto:

```
var{int} c(m) <- a;
```

Jak je zřejmé z příkladu níže, invarianty slouží k jednoduchému pojmenování složitějšího vztahu. Jejich vlastností se dá s velkou výhodou využít při zjišťování dopadu změny některých proměnných na například celkovou cenu našeho rozvrhu. COMET sám zajišťuje obsluhu invariant a pokud se změní hodnota některé z proměnných, pomocí kterých jsou definovány, invarianty jsou automaticky zaktualizovány. Z příkladu také vyplývá, že proměnné, pomocí nichž je stálá proměnná definována, musejí být součástí nějakého Local Solveru

```
include "LocalSolver";

// deklarace modelu
LocalSolver m();


// deklarace invariant
var{int} c(m) <- a;
var{int} b(m) <- a + c;

// uzavření modelu
m.close();

// přiřazení hodnoty proměnné a vypsání hodnot
a := 3;

cout << "a=" << a << endl;
cout << "b=" << b << endl;
cout << "c=" << c << endl;


// znovupřiřazení hodnoty proměnné a vypsání hodnot
a := 4;

cout << "a=" << a << endl;
cout << "b=" << b << endl;
cout << "c=" << c << endl;
```

```
// výsledek programu
a=3
b=6
c=3
a=4
b=8
c=4
```

3.2.3 Diferencovatelné objekty

K těmto objektům patří dvě důležité třídy, omezující podmínky (constraints) a kritéria. Omezující podmínka je datový typ obsahující nějakou naši proměnnou {var} a definici jejího omezení. Součástí toho objektu je 5 základních metod, které můžeme využít v našem programu.

- **getVariables()** navrátí proměnnou
- **isTrue()** zda-li je omezení dodrženo nebo porušeno
- **violations()** navrátí proměnnou, pokud dojde k nedodržení omezení při přiřazení nové hodnoty proměnné
- **getAssingDelta()** navrátí rozdíl hodnoty nedodržení, dojde-li k přiřazení nové hodnoty proměnné
- **getSwapDelta()** navrátí rozdíl hodnoty nedodržení, dojde-li k přehození hodnot dvou proměnných

Změníme-li například hodnotu naši proměnné {var} x, můžeme za pomoci příkazu x.isTrue() zjistit, zda-li se přiřazením nové hodnoty neporušilo naše omezení.

3.2.4 Nástroje pro vyhledávání a ovládání algoritmu

Prvním nástrojem spadající do této kategorie jsou **selektory.** Hlavním úkolem selektorů je vybrat prvek ze sady, který splňuje nebo alespoň nejlépe vyhovuje našemu kritériu. Selektor si může i sami nadefinovat, k dispozici máme ale také 4 předdefinované selektory:

- **selectMax**
- **selectMin**
- **selectFirst**
- **selectCircular**

Příklad použití selektoru je:

```
SelectMax(i in S, diff=a+i)(diff)
```

Tento selektor vybere hodnotu i z množiny S takovou, aby její součet s hodnotou proměnné a byl co největší.

Dalším významným prvkem v této skupině jsou **řešení.** Jejich inicializace se provádí příkazem:

```
Solution s= new Solution(m);
```

Jedná se o prostředek k uložení hodnot lokálních proměnných v jistém čase. Dá se tedy využít například k uložení našeho nejlepší zatím dosazeného výsledku.

Posledním významným nástrojem v této skupině je **kontrolní bod.** Kontrolní bod je součásti LocalSolver a slouží k přepnutí LocalSolver do módu, ve kterém jsou uloženy veškeré hodnoty proměnných, omezení a datových prvků. Po provedení operací, které nevedly k vylepšení našeho řešení, jsme díky kontrolnímu bodu schopni veškeré hodnoty objektů obnovit do jejich původního stavu.

3.3 Ukázkový příklad v jazyce COMET a jeho popis

Jako ukázkový příklad jsem zvolil problém n-královen. Program pochází od tvůrců COMETu a zdrojový kód je volně dostupný jako součást distribuce jazyku COMET[9]. Pro přehlednost uvádím celý zdrojový kód a posléze vysvětlím význam jeho jednotlivých částí.

```
include "LocalSolver";

int t0 = System.getCPUTime();

LocalSolver m();
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);
var{int} queen[i in Size](m,Size) := distr.get();
ConstraintSystem S(m);
S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));
m.close();

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
   selectMax(q in Size)(S.violations(queen[q]))
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
   it = it + 1;
}
int t1 = System.getCPUTime();

cout << queen << endl;
cout << "violations: " << S.violations() << endl;
cout << "iterations: " << it << endl;
cout << "time \(total\) : " << t1 - t0 << endl;
```

Na začátku dokumentu zahrnujeme potřebné objekty.

```
include "LocalSolver";
```

Jak již bylo napsáno v kapitole 3.2.1, i součástí toho programu je Local Solver. Objekt, který nám slouží pro ukládání proměnných, invariant stejně jako systému omezujících podmínek.

Příkazy uvedené níže slouží pro zjištění časové náročnosti algoritmu při řešení toho problému. Do proměnné t0 se uloží systémový čas při spuštění programu, do proměnné t1 se uloží čas před skončením programu. Rozdíl těchto dvou časů se vypíše na konci programu příkazem `cout`.

```
int t0 = System.getCPUTime();

int t1 = System.getCPUTime();

cout << "time \(total\) : " << t1 - t0 << endl;
```

Další část kódu slouží k inicializaci samotného objektu Local Solver.

```
LocalSolver m();
```

Dále inicializujeme proměnné, nejdřív proměnnou n typu integer s hodnotou 16, posléze proměnnou jazyka COMET typu pole hodnot integer, které má velikost určenou rozmezím Size, v našem případě je to oněch 16. Zároveň se také deklaruje objekt distr typu UniformDistribution, který nám poskytuje metodu get() rovnoměrně umístí královny na hrací pole.

Nyní přichází na řadu popis systému omezujících podmínek. Nejprve se inicializuje jako součást Local Solveru m a posléze se do něj vloží jednotlivé podmínky. První podmínka vyjadřuje, že královny nesmějí ohrožovat ty druhé ve stejném sloupci a řádku, další dvě podmínky pak představují úhlopříčky. Jakmile máme vše zapsáno do Local Solveru, můžeme jej uzavřít.

```
ConstraintSystem S(m);

S.post(alldifferent(queen));

S.post(alldifferent(all(i in Size) queen[i] + i));

S.post(alldifferent(all(i in Size) queen[i] - i));

m.close();
```

Níže uvedená část kódu pak představuje samotný vyhledávací algoritmus. Skládá se z cyklu, jenž se bude opakovat, dokud nebude systém omezujících podmínek plně dodržen a zároveň dokud počet opakování cyklu nepřesáhne padesáti násobek rozměru pole. Uvnitř cyklu jsou dva selektory. První vybere královnu, která má nejvíce kolizí se systému omezujících podmínek a poté vybere sloupec v, ve kterém bude mít královna kolizí nejméně. Jakmile je taková hodnota *v* nalezena, je přiřazena relevantní královně.

```
int it = 0;
while (S.violations() > 0 && it < 50 * n) {
   selectMax(q in Size)(S.violations(queen[q]))
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
   it = it + 1;
}
```

Poslední část zdrojového kódu už jen obslouží výstup programu, kdy se vytiskne seznam královen, počet nedodržení systému omezujících podmínek a počet opakování prohledávacího algoritmu.

```
cout << queen << endl;
cout << "violations: " << S.violations() << endl;
cout << "iterations: " << it << endl;
```

Kapitola 4

Vlastní práce

4.1 Popis zadaného problému

Problém, jehož řešením se v této bakalářské práci zabývám, byl podrobně popsán v kapitole 2.5. Jedná se o One Machine Earliness Tardiness Shop Scheduling Problem s disjunktivním omezením, tj. na stroji m se může v jeden okamžik vykonávat pouze jedna aktivita, a s cílem co nejmenší celkové ceny rozvrhu definované rovněž v kapitole 2.5.

4.2 Vývojové prostředí

Program pro řešení OMETJSSP jsem vyvíjel pod operačním systémem Linux. COMET je nainstalován ve verzi přímo určené pro Debian Etch v 32bitovém provedení. Jeho název je OMETSP_Heincl_Roman_2009.co. Programovací jazyk je také k dispozici pro operační systém Windows a Mac OS.

4.3 Vstupy a výstupy

Vstupem do systému je textový soubor Input.txt uložený v adresáři s kompilerem programu COMET /Comet/Compiler. Musí mít přesně specifikovaný formát. Detailní popis je uveden dále a příklad uvádí obrázek 4.1. Důležité je dodržení mezery na začátku každého řádku!

R1: první řádek souboru je vyhrazen pro komentář

R2: druhý řádek souboru obsahuje číslici představující počet aktivit, odpovídá rozdílu R_N-2

R3: třetí řádek obsahuje údaje o první aktivitě v pořadí

- due date
- duration
- earliness factor
- tardiness factor

R4: např. 120 50 1 3

…..

R_N: řádek poslední, n-tý, představuje údaje o poslední aktivitě

```
·Zdrojovy·soubor·pro·OMETJSSP¶
·5¶
·201·71·3·4¶
·162·37·5·1¶
·158·51·5·1¶
·151·15·1·3¶
·245·39·4·2¶
```

Obrázek 4.1 Vzorový vstupní soubor

Výstupem programu je počet aktivit, počáteční cena rozvrhu, koncová cena rozvrhu, počet iterací nastavených v programu a systémový čas výpočtu.

4.4 Datový koncept programu

Program OMETJSSP_Heincl_Roman_2009.co se sestává z dvou tříd a hlavního programu. První třída ACT zastupuje jednotlivé aktivity. Druhá třída OMETSP pak představuje instanci OMETJSSP rozvrhovacího problému. Jejich obsah je uveden v tabulce 4.2.

Název třídy	Typ proměnné	Název proměnné
ACT	Int	_duedate
	Int	_duration
	Int	_finish_date
	Int	_alfa
	Int	_beta
	int	_act_id
	Int	_jobcost
	Int	order
	Int	updated
	Int	cost
OMETSP	range	Activities
	ACT[]	myJobs
	string	fName
	int	totalCost
	Int	diff
	int	nbActivities

Tabulka 4.2 Seznam tříd a jejich proměnných

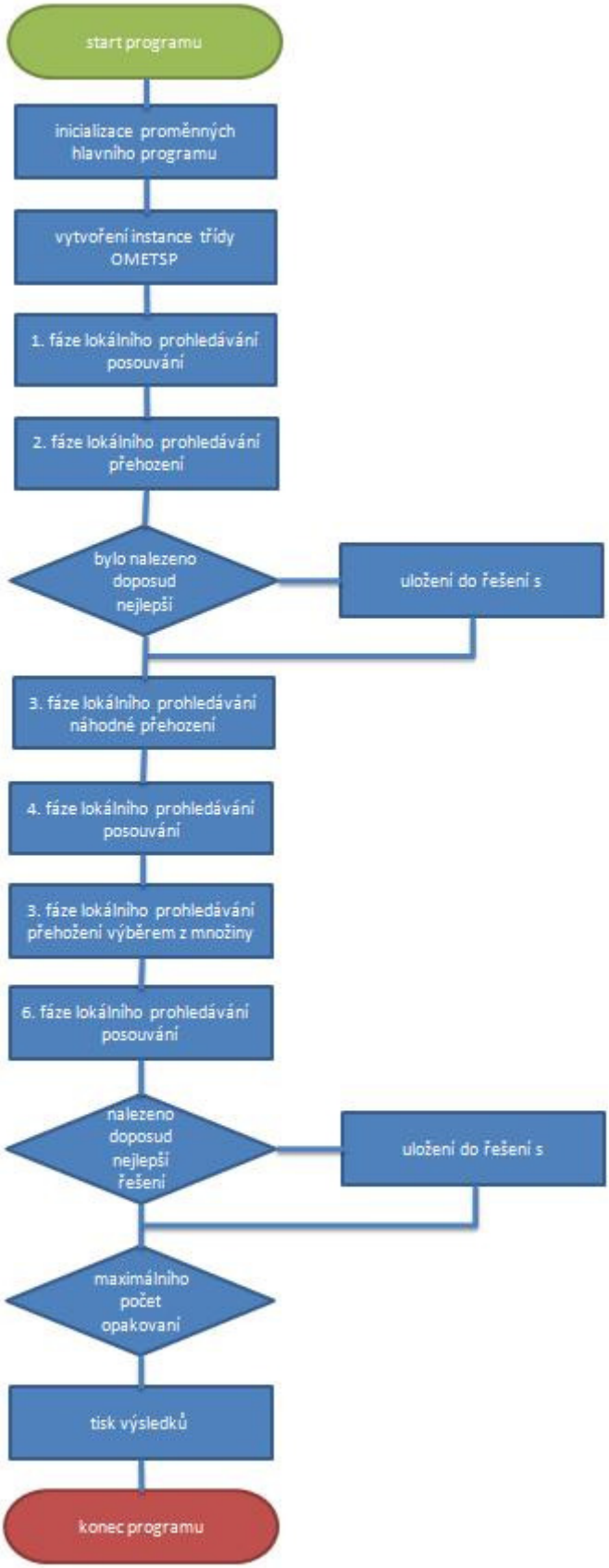
4.5 Algoritmus lokálního vyhledávání

Postup při hledání co nejlepšího řešení byl zvolen následující. Prvotní seřazení všech načtených aktivit se děje podle jejich data dokončení a to ve vzestupné tendenci. Podle takto určeného pořadí se přistoupí k jejich zařazení do rozvrhu. Začíná se aktivitou s nejmenším termínem dokončení a její koncový čas se algoritmus snaží nastavit stejný jako je tento termín. Pokud je ovšem doba trvání aktivity delší než termín dokončení, koncový čas aktivity odpovídá její době trvání, aby se minimalizovala její dílčí cena. Algoritmus takto postupně přiřadí všechny další aktivity, primární klíč pro přiřazení je

taktéž snaha o nastavení koncového času shodného s termínem dokončení, pokud to nelze, aktivita je přidána za aktivitu jí předcházející. Jakmile jsou všechny aktivity umístěny do rozvrhu přichází na řadu samotné lokální prohledávání.

Lokální prohledávání mnou navržené má 6 částí.

- Část první posouvá celý blok aktivit na časové ose vlevo popř. vpravo. Tato činnost je prováděna dokud se posouváním snižuje celková cena rozvrhu
- Algoritmus vyhledává aktivity, jejichž přehozením se co nejvíce sníží celková cena rozvrhu. Dokud takové aktivity existují, algoritmus vykonává svou činnost
- Aby se zabránilo uvíznutí algoritmu v lokálních minimech, je provedeno náhodné přeřazení dvou aktivit rozvrhu
- Opakuje se část první a blok je posouván po časové ose
- Algoritmu v páté části opět prohazuje pořadí jednotlivých aktiv. Aby se předešlo negaci dopadu kroku tři, algoritmus nevybírá dvě aktivity, jejichž prohozením se celková cena rozvrhu sníží nejvíce, ale náhodně vybere z množiny tří dvojic aktivit, které po přehození nejvíce sníží celkovou cenu rozvrhu
- Bod šestý se shoduje s body jedna a čtyři a algoritmus se opět snaží doladit celkovou cenu posouváním celého bloku aktivit po časové ose stroje



Obrázek 4.2 Vývojový diagram programu

4.6 Popis programu

Průběh programu má několik fázi. Načte se systémový čas, inicializuje se LocalSolver, deklarují se potřebné proměnné a poté se přistoupí k inicializaci třídy OMETSP. Ta spočívá v načtení seznamu aktivit z textového souboru Input.txt a následné inicializaci jednotlivých aktivit jako instancí třídy ACT. Součástí inicializace třídy OMETSP je i seřazení aktivit a jejich přiřazení do rozvrhu. Poté se již přistupuje k samotnému vyhledávání, tak jak je popsané v kapitole 4.5. Jakmile proběhne předem stanovený počet opakování vyhledávacího cyklu, je znovu načten systémový čas a spolu s výsledky zobrazen uživateli.

4.6.1 Hlavní program

deklarace řešení s

```
Solution  s;
```

inicializace Local Solveru

```
LocalSolver    m();
```

inicializace proměnné pro uložení nejlepšího výsledku v Local Solveru m a nastavení počáteční hodnoty

```
var{int}  BEST(m):=999999;
```

uzavření Local Solveru

```
m.close();
```

proměnná udávající počet opakování celého cyklu prohledávání

```
int  maxIteration=10000;
```

proměnná v níž je uloženo jméno vstupu

```
string _fName = "JS.txt";
```

inicializace celého rozvrhu

```
OMETSP shop(_fName);
```

první selektor algoritmu pro lokální prohledávání

```
selectMax(a in R2,diff=shop.moveActDiff(a))(diff)
```

druhý selektor algoritmu pro lokální prohledávání

```
selectMax(a in R1, b in R1: a!=b&&a<b,
diff=shop.swapActDiff(a,b))(diff)
```

aktualizace nejlepšího řešení

```
if (BEST>temp10) { BEST:=temp10; s=new Solution(m); }
```

náhodné prohození aktivit

```
shop.swapAct(random1,random2);
```

selektor čtvrté fáze algoritmu

```
selectMax(a in R2,diff=shop.moveActDiff(a))(diff)
```

selektor páté fáze algoritmu

```
selectMax[3](a in R1, b in R1:
a!=b&&a<b,diff=shop.swapActDiff(a,b))(diff)
```

selektor poslední fáze algoritmu

```
selectMax(a in R2,diff=shop.moveActDiff(a))(diff)
```

načtení a zobrazení nejlepšího nalezeného řešení

```
int vysledek=BEST.getSnapshot(s);
cout << "Nejlepsi nalezeny vysledek: " << vysledek << endl;
```

4.6.2 Seznam metod

Seznam metod třídy ACT

metoda pro tisk detailů aktivity č. jobNbr

```
ACT print (int jobNbr)
```

metoda getD pro zjištění doby trvání dané aktivity

```
int getD()
```

metoda getDD pro zjištění terminu dokončení dané aktivity

```
int getDD()
```

metoda getFD pro zjištění koncového času dané aktivity

```
int getFD()
```

metoda getID pro zjištění ID dané aktivity

```
int getID()
```

metoda getOrder pro zjištění pořadí dané aktivity ve stávajícím rozvrhu

```
int getOrder()
```

metoda getCost pro zjištění ceny dané aktivity ve stávajícím rozvrhu

```
int getCost()
```

metoda isUpdated pro zjištění nastavení příznaku aktivity

```
int isUpdated()
```

metoda updateFD pro aktualizaci finish date dané aktivity

```
ACT updateFD(int newFD)
```

metoda updateOrder pro přiřazení nového pořadí aktivitě

```
ACT updateOrder(int newOrder)
```

metoda updated pro nastavení příznaku aktivity

```
ACT updated()
```

metoda updatedBack pro vynulování příznaku aktivity

```
ACT updatedBack()
```

Seznam metod třídy OMETSP

metoda getTC vracející celkovou cenu rozvrhu

```
int getTC()
```

metoda getActNum vracející rozsah načtených aktivit

```
range getActNum()
```

metoda swapActDiff vracející rozdíl celkové ceny rozvrhu při prohozeni dvou aktivit rozvrhu daného pořadí

```
int swapActDiff(int JobOrd1,int JobOrd2)
```

metoda swapAct pro prohození dvou aktivit rozvrhu daného pořadí

```
OMETSP swapAct(int JobOrd1,int JobOrd2)
```

metoda moveActDiff vracející rozdíl celkové ceny rozvrhu při posunutí celého rozvrhu

```
int moveActDiff(int direction)
```

metoda moveAct pro posunutí celého rozvrhu

```
OMETSP moveAct(int direction)
```

metoda getID pro zjištění pozice aktivity s daným pořadím v poli aktivit

```
int getID(int poradi)
```

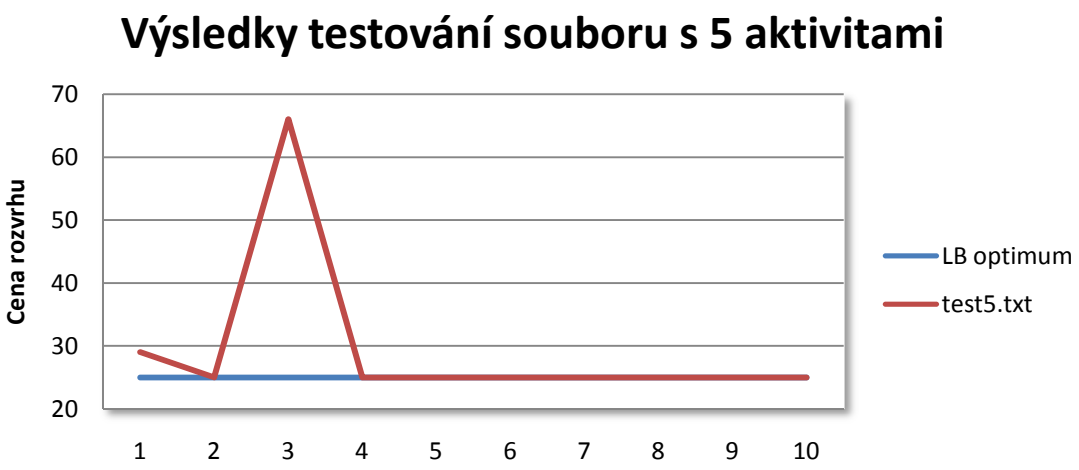
4.7 Výsledky

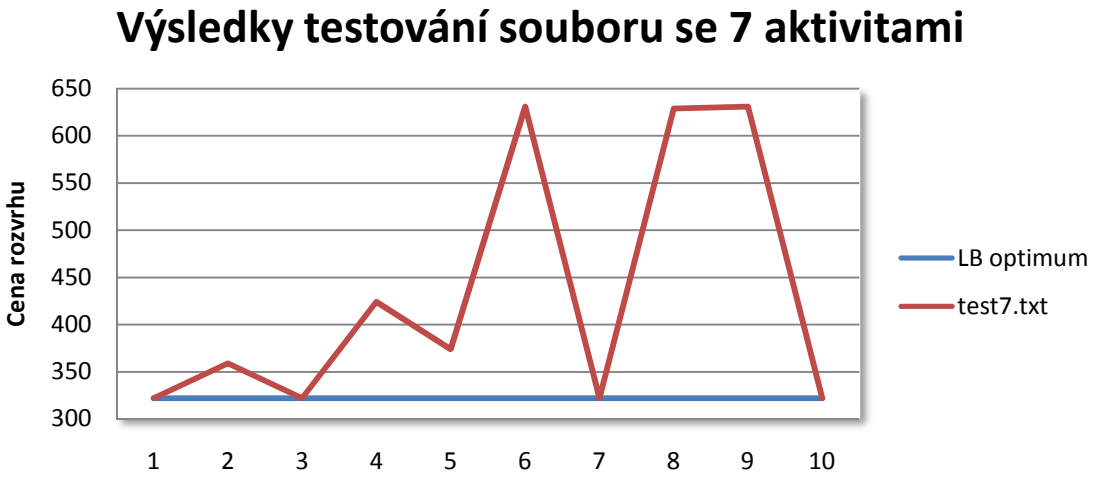
Výsledky byly zpracovány na základě následující metodiky testování. Na testovacím textovém souboru bylo vždy provedeno 10 testů, při nichž byly zaznamenány všechny výstupy programu. Záznam testování uvádí tabulka 4.1.

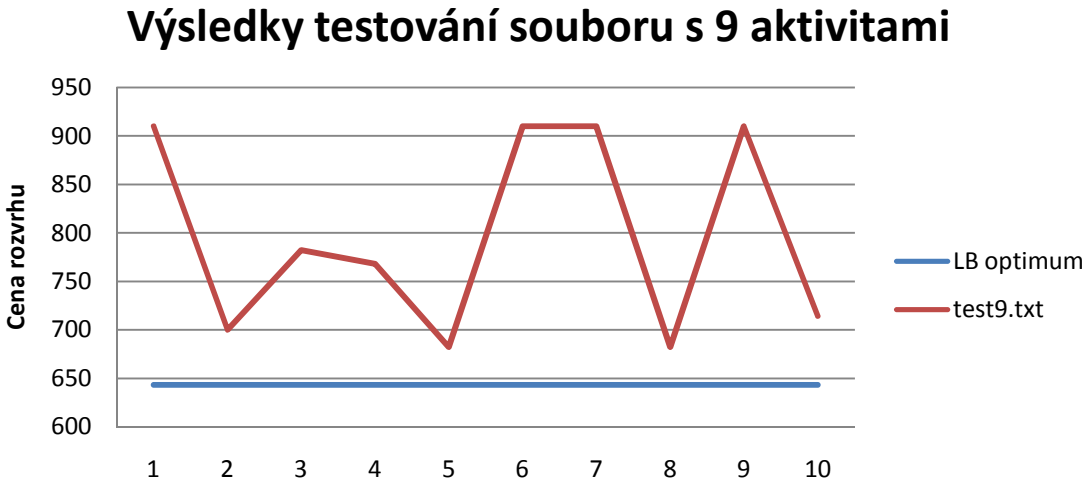
Pro nalezení optima (LB optimum) pro srovnání mých výsledku byl použit program využívající lower-bound algoritmus popsaný v kapitole 2.5. Jejich zhodnocení je součástí závěru.

testovaný soubor	počet iterací	číslo pokusu	LB optimum	celkový čas [ms]	nejlepší výsledek
test5.txt	10000	1	25	5700	29
test5.txt	10000	2	25	5604	25
test5.txt	10000	3	25	5656	66
test5.txt	10000	4	25	5680	25
test5.txt	10000	5	25	5504	25
test5.txt	10000	6	25	5596	25
test5.txt	10000	7	25	5696	25
test5.txt	10000	8	25	5704	25
test5.txt	10000	9	25	5544	25
test5.txt	10000	10	25	5576	25
test7.txt	10000	1	322	14568	322
test7.txt	10000	2	322	14516	359
test7.txt	10000	3	322	14228	322
test7.txt	10000	4	322	14296	424
test7.txt	10000	5	322	14512	374
test7.txt	10000	6	322	14308	631
test7.txt	10000	7	322	14336	322
test7.txt	10000	8	322	14736	629
test7.txt	10000	9	322	14520	631
test7.txt	10000	10	322	14292	322
test9.txt	10000	1	643	36934	910
test9.txt	10000	2	643	35838	700
test9.txt	10000	3	643	36310	782
test9.txt	10000	4	643	36126	768
test9.txt	10000	5	643	36326	682
test9.txt	10000	6	643	35778	910
test9.txt	10000	7	643	36666	910
test9.txt	10000	8	643	36862	682
test9.txt	10000	9	643	36378	910
test9.txt	10000	10	643	36010	714

Tabuka 4.1 záznamy testovaní


Výsledky testování souboru s 5 aktivitami
Cena rozvrhu
70
60
50
40
30
20
1
2
3
4
5
6
7
8
9
10
LB optimum
test5.txt


Výsledky testování souboru se 7 aktivitami
Cena rozvrhu
650
600
550
500
450
400
350
300
1
2
3
4
5
6
7
8
9
10
LB optimum
test7.txt


Výsledky testování souboru s 9 aktivitami
Cena rozvrhu
950
900
850
800
750
700
650
600
1
2
3
4
5
6
7
8
9
10
LB optimum
test9.txt

Kapitola 5

Závěr

Cílem této práce bylo seznámit se s problematikou rozvrhování a s principy lokálního prohledávání. Následně se naučit používat programovací jazyk COMET a pomocí něj vyřešit problém rozvrhování na jednom stroji s penaltami za předčasné a zpožděné dokončení.

Téma rozvrhování pomocí lokálního prohledáváni je velice aktuální. Lokální prohledávaní nabízí alternativní cestu při hledaní řešeni rozvrhovacích problémů, pro které je použití úplných algoritmu prohledávajících celý stavový prostor nevhodné. Hlavni výhodou je velká časová úspora při řešení rozsáhlých rozvrhovacích problémů, avšak nevýhodou je to, že nemusí nalézt zcela optimální řešení, pokud je vůbec nějaké řešení nalezeno. To vše musíme brát v úvahu při volbě řešeni rozvrhovacích problémů. Rozvrhování na jednom stroji s penaltami za předčasné a zpožděné dokončení je problém, jehož náročnost se zvyšujícím se počtem aktivit značně roste a právě z toho důvodu je vhodné použít algoritmy s lokálním.

Programovací jazyk COMET je velice silný nastroj, díky kterému se řešení kombinatorických problémů výrazně zjednodušuje. Je to hlavně díky implementovaným datovým objektům, pomocí nichž lze vhodně popsat model problému a posléze prohledávací algoritmus. Při volbě operačního systému, pod kterým budu program vyvíjet, jsem se rozhodl pro Linux z toho důvodu, že jako jediný podporuje i vizualizace. Jeho samotná instalace byla ovšem velice komplikovaná a musel jsem požádat o pomoc mého vedoucího práce, resp. vyučující z katedry řízení. To může být překážkou pro další uživatele tohoto programovacího jazyka. Dalším problémem byl i nedostatek vhodné literatury. I když jsem měl k dispozici knihu[6], nejsou v ní popsány všechny nástroje toho jazyka a jejich popis v COMET API dokumentaci[8] je velice strohý, občas chybí zcela. I tak jsem se snažil nástroje vhodně použít tam, kde to mělo smysl.

Uživatelské prostředí programy bylo voleno co nejjednodušší. Ať už se jedná o vstup programu prostřednictvím textového souboru nebo jeho výstup, který poskytuje pouze nezbytná data k vyhodnocení běhu programu a úspěšnosti algoritmu. Program má však v sobě zabudovány možnosti tisku průběžných výsledku a sledování aktuálního stavu rozvrhu. Tyto části kódu jsou momentálně ve zdrojovém kódu neaktivní, je zde ale možnost je aktivovat jejich odkomentováním.

Výsledky mnou navrženého algoritmu jsou vzhledem ke složitosti toho problému velice přijatelné. Počet opakování jsem při testech schválně nevolil příliš vysoká, aby bylo z výsledků zřetelné, jak se s rostoucím počtem aktivit mění složitost celého rozvrhovacího problému a tomu odpovídají i takto nalezená řešení. Jiná řešení toho problému, úspěšnější než můj algoritmus, jsou založena na složitých matematických modelech a do budoucna by bylo zajímavou výzvou spojení těchto modelů a nástrojů, které poskytuje jazyk COMET.

Literatura

[1] WEAVER P. A brief history of scheduling. Mosaic Project Services Pty Ltd, Australia, 2006
http://www.pmforum.org/library/papers/2006/A_Brief_History_of_Scheduling.pdf

[2] BAPTISTE, P., LE PAPE, C., NUIJTEN, W.: Constraint-Based Scheduling.
Kluwer Academic Publishers, 2001.

[3] Webová stránky společnosti AICOS Technologies AG in Basel, Switzerland
http://www.aicos.ch

[4] BAKER, K. R., SCUDDER, G. D.: Sequencing with earliness and tardiness penalties: A review. Operations Research 38 (1), 1990, 22–36.

[5] DU, J., LEUNG, J.Y.-T.: Minimizing total tardiness on one machine is NP-hard. Mathematics of Operations Research 15(3),1990, 483-495.

[6] VAN HENTENRYCK, P., MICHEL, L.: Constraint-based local search. Massachusetts Institute of Technology, 2005.

[7] SOURD, F., KEDAD-SIDHOUM, S.: An efficient algorithm for the earliness-tardiness scheduling problem

[8] COMET API documentation
http://www.cs.brown.edu/people/pvh/CometDoc/www/api/api.html

[9] The Comet Programming Language and System
http://comet-online.org/Welcome.html

Příloha A

Obsah přiloženého CD

Obsah přiloženého CD:

- bp_2009_Heincl_Roman.pdf : tato práce ve formátu pdf
- codes : zdrojové kódy
- test texts : testovací textové soubory