VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ALGORITMY ROZVRHOVÁNÍ VÝROBY S DYNAMICKÝMI REKONFIGURACEMI A ÚDRŽBOU

PROJECT SCHEDULING WITH DYNAMIC RECONFIGURATIONS AND MAINTENANCE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. MARIÁN HALČIN**
AUTHOR

VEDOUCÍ PRÁCE **Ing. MARTIN HRUBÝ, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/19966/2016/xhalci00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Halčin Marián, Bc.**
Obor: Informační systémy
Téma: **Algoritmy rozvrhování výroby s dynamickými rekonfiguracemi a údržbou**
Project Scheduling with Dynamic Reconfigurations and Maintenance
Kategorie: Modelování a simulace

Pokyny:

1. Prostudujte metody rozvrhování průmyslové výroby, algoritmy generování rozvrhů a optimalizace rozvrhů. Prostudujte způsoby zadávání dynamických rekonfigurací strojů a preventivní údržby.
2. Navrhněte metody generování rozvrhů s prvky rekonfigurací a údržby. Proveďte klasifikaci výrobních úloh a diskutujte vhodnost použití metod pro různé typy úloh.
3. Implementujte navržené rozvrhovací metody a algoritmy optimalizace výroby.
4. Sestavte sadu ukázkových příkladů úloh s různými typy rekonfigurací a údržeb.
5. Na sadě úloh proveďte experimentální zhodnocení vašich algoritmů.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- První dva body.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hrubý Martin, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Diplomová práca sa zaoberá témou počítačového rozvrhovania výroby s dynamickými rekonfiguráciami a údržbou. Problém je formálne popísaný matematickým modelom s názvom Resource Constrained Project Scheduling Problem rozšíreným o prvky dynamickej rekonfigurácie a údržby. Podľa tohto modelu bolo navrhnutých niekoľko rôznych rozvrhovacích algoritmov. Taktiež boli popísané metódy optimalizácie riešenia založené na základe genetických algoritmov. V experimentálnej časti je uvedená typológia výrobných príkazov, z ktorých sú vytvorené rôzne typy úloh. Výsledkom experimentov je jednoznačné doporučenie rozvrhovacieho algoritmu na daný typ úlohy. Na záver sa práca zaoberá prípadovou štúdiou voľby vhodného riešenia pre konkrétne výrobne podniky.

Abstract

Thesis deals with the topic of computational scheduling of production with dynamic reconfigurations and maintenance. The problem is formally defined by a mathematical model named Resource Constrained Project Scheduling Problem which was extended by dynamic reconfiguration and maintenance. Number of different schedule generation algorithms were proposed based on this model. Also methods of solution optimization based on genetic algorithms were described. The typology of production orders of which different task types are created was described in the experimental part. The result of the experiments is clear recommendation of scheduling algorithm for given task type. For the conclusion, thesis deals with the case study of choice of suitable solution for specific production companies.

Kľúčové slová

rozvrhovanie, plánovanie, RCPSP, riadenie výroby, genetické algoritmy, dynamická rekonfigurácia, údržba

Keywords

scheduling, planning, RCPSP, production control, genetic algorithms, dynamic reconfiguration, maintenance

Citácia

HALČIN, Marián. *Algoritmy rozvrhování výroby s dynamickými rekonfiguracemi a údržbou.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Martin Hrubý, Ph.D.

Algoritmy rozvrhování výroby s dynamickými rekonfiguracemi a údržbou

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Martina Hrubého Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Marián Halčin
24. mája 2017

Poďakovanie

V prvom rade by som chcel poďakovať môjmu školiteľovi Ing. Martinovi Hrubému, Phd. za vedenie, konštruktívne pripomienky a ochotu vždy konzultovať problémy počas písania diplomovej práce. Ďalej sa chcem poďakovať všetkým ľuďom z výrobných podnikov, ktorí prispeli k úspešnému zavedeniu rozvrhovacieho systému a poskytli cennú spätnú väzbu využitú pri tvorbe tejto práce, predovšetkým však za firmu HDF s.r.o pánovi Jánovi Franekovi a pani Lujze Kamodyovej a za firmu Metaltrim pánovi Eduardovi Ferancovi a pánovi Pavlovi Holubčíkovi. Taktiež sa musím poďakovať rodine za veľkú podporu počas písania práce. Špeciálne otcovi, ktorý ma na tému plánovania výroby naviedol a sestre Janke, ktorá viacnásobne skontrolovala moje prehrešky voči slovenskému jazyku obsiahnuté v diplomovej práci.

Obsah

Kapitola 1

Úvod

Plánovanie a rozvrhovanie je jednou z kľúčových oblastí riadenia výroby. Táto súčasť obvykle tvorí jeden z modulov ERP(Enterprised Resource Planning) systémov.[7][3] Podskupinou rozvrhovania je plánovanie kapacít. Operácie obsadzujú zdroje, typicky stroje alebo pracovníkov, a po určitom čase ich uvoľnia.

Rozvrhovanie výroby často čelí nasledujúcemu problému: Pre vyžitie zdroja na vykonanie operácie musí byť stroj nastavený v istej konfigurácii. Tento stroj je však využívaný aj na vykonanie iných operácii. Preto dochádza k rekonfigurácii zdroja. Ako príklad môžeme uviesť lisovanie plastových výrobkov. Pre výrobu výlisku potrebujeme voľnú kapacitu na lise a musí mať upnutú správnu formu. Ak mal predchádzajúci výlisok inú formu, dochádza k rekonfigurácii v podobe upnutia formy. Pre optimalizáciu nákladov je výhodné minimalizovať početnosť zmeny konfigurácie.

Ďalšia komplikácia je problém údržby zdroja. Na niektorých strojoch musí byť po určitom čase využívania vykonaná údržba. Tento aspekt môže ovplyvniť dĺžku vykonávania operácie.

Jedna z možností implementácie rozvrhovania výroby je využitie simulačných metód, ktoré využívajú počítačovú optimalizáciu. V tejto práci je skúmané využitie simulačných metód pre rozvrhovanie výroby s dynamickou rekonfiguráciou a údržbou. Algoritmy sú postavené na matematickom modeli Resource Constrained Project Scheduling Problem(RCPSP), ktorý práca rozširuje o prvky dynamickej rekonfigurácie a údržby.

Výsledkom diplomovej práce je jednoznačné doporučenie rozvrhovacieho algoritmu podľa typu výroby.

V druhej kapitole je navrhnuté rozšírenie modelu RCPSP a reprezentácia rozvrhu. Ďalej sú predstavené základné možnosti tvorby rozvrhu. Tiež sú rozobraté optimalizačné metódy, predovšetkým genetické algoritmy.

Tretia kapitola obsahuje návrh rôznych pokročilých metód tvorby rozvrhu. Algoritmy sú tiež vysvetlené na príkladoch.

Kapitola 4 popisuje situáciu v prostredí výrobných podnikov. Tiež je tam uvedený návrh programov vytvorených v rámci diplomovej práce. Ich implementácia je popísaná v kapitole 5.

Posledná kapitola 6 sa skladá z dvoch častí: Na začiatku je uvedená typológia automaticky vygenerovaných inštancií úloh pre účely experimentovania. Následne sú nad touto sadou vykonané viaceré experimenty a vyhodnotená vhodnosť využitia rôznych algoritmov podľa typu úlohy. Podľa výsledkov experimentovania je odporučený rozvrhovací algoritmus pre výrobné podniky HDF s.r.o. a Metaltrim.

Kapitola 2

Rozbor témy

V tejto kapitole je predstavený problém RCPSP s navrhnutým rozšírením o dynamickú rekonfiguráciu a údržbu. Ďalej sú uvedené matematické štruktúry popisujúce generovaný rozvrh a nad nimi formálne popísané podmienky platnosti. Je uvedená problematika kódovania rozvrhu v podobe Activity Listu a jeho transformácie na rozvrh. Na záver sú predstavené možnosti optimalizačného mechanizmu nad kandidátnymi riešeniami v podobe prehľadávania náhodných riešení a genetických algoritmov.

2.1 Charakteristika úlohy RCPSP pre dynamickú rekonfiguráciu obnoviteľných zdrojov

Úloha pozostáva z množiny operácií $J = \{j_1, ..., j_{N^j}\}$. Operácie j_1 a j_{N^j} sú prázdne operácie vyjadrujúce prvú a poslednú operáciu v rozvrhu.

Ďalej definujeme precedenčné obmedzenia. S_j je množina následníkov operácie j. P_j je množina predchodcov operácie j. Predpokladáme, že $j_1 \in P_{j_n}, n = 2, ..., N^j$ a $j_{N^j} \in S_{j_n}, n = 1, ..., N^j - 1$.

Definujeme množinu obnoviteľných zdrojov s dynamickou rekonfiguráciou $R = \{r_1, ..., r_{N^r}\}$. Predpokladajme, že každý zdroj $r \in R$ má kapacitu $m_r = 1$, tj. na jednom stroji môže súčasne prebiehať len jedna operácia.

Ďalej definujeme množinu rekonfigurácií $C = \{c_1, ..., c_{N^c}\}$. Každý zdroj môže nadobudnúť podmnožinu týchto konfigurácií. Na každej z týchto podmnožín definujeme čas potrebný na zmenu konfigurácie $d_r(c_x, c_y) : C \bigtimes C \to \mathbb{N}^0$. Pre $c_x = c_y$ je $d^r(c_x, c_y) = 0$. Každý zdroj má počiatočnú konfiguráciu $c_0(r) : R \to C$.

Pre každú operáciu $j \in J$ definujeme trvanie operácie $d(j) \in \mathbb{N}^0$. Každá operácia je vykonávaná na zdroji $res(j) : J \to R$ v konfigurácii $conf(j) : J \to C$.

2.1.1 Rozšírenie RCPSP o dynamickú údržbu

V RCPSP uvažujeme množinu zdrojov $R = \{r_1, ..., r_{N^r}\}$. Pre každý zdroj $r \in R$ ďalej definujeme trvanie údržby $d^M(r) : R \to \mathbb{N}^0$ a periódu údržby $d^T(r) : R \to \mathbb{N}^0$. Pre zdroj r musí platiť, že po každých $d^T(r)$ jednotkách času, kedy bola vykonávaná práca na zdroji sa musí vykonať údržba v trvaní $d^M(r)$ časových jednotiek.

j	$d(j)$	$res(j)$	$conf(j)$	$S(j)$
1	0	-	-	3, 4, 7
2	7	2	2	3
3	3	1	3	6
4	3	1	1	5
5	2	2	1	6
6	2	3	1	9
7	2	3	1	8
8	3	1	2	9
9	0	-	-	-

Tabuľka 2.1: Zoznam operácií v definícii úlohy

r_1	$\mathbf{c_1}$	c_2	c_3
$\mathbf{c_1}$	-	2	1
c_2	2	-	1
c_3	2	3	-

r_2	$\mathbf{c_1}$	c_2
$\mathbf{c_1}$	-	1
c_2	2	-

r_3	$\mathbf{c_1}$
$\mathbf{c_1}$	-

Tabuľka 2.2: Matice trvania rekonfigurácií pre každý zdroj

2.2 Riešenie úlohy pre dynamickú rekonfiguráciu

V predchádzajúcej časti bola predstavená statická charakteristika úlohy. Riešenie úlohy RCPSP je rozvrh, ktorý vyhovuje obmedzeniam na kapacitu zdrojov, precedenčným obmedzeniam a operácie sú rozvrhnuté v príslušnej konfigurácii. V tejto časti si definujeme rozvrh vo vhodnom tvare pre účely dynamických operácií. Nad rozvrhom definujeme funkcie a predikáty pre abstrakciu vlastností rozvrhu. Ďalej definujeme aké precedenčné obmedzenia musí rozvrh spĺňať, aby bol zároveň aj riešením úlohy.

2.2.1 Definícia rozvrhu s dynamickou rekonfiguráciou

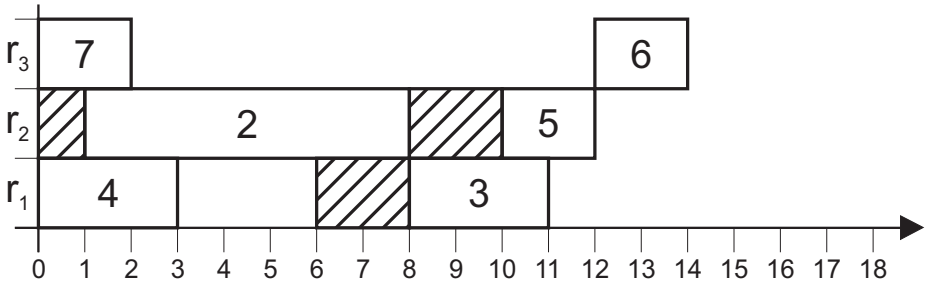
Zadefinujeme rozvrh ako usporiadanú dvojicu $s = (J_s, start(j))$, kde

- $J_s \subseteq J$ je množina operácií v rozvrhu.
- $start(j) : J_s \to \mathbb{N}$ je priradenie času začiatku operáciám v rozvrhu.

Príklad úlohy RCPSP s dynamickou rekonfiguráciou

Predstavíme príklad úlohy RCPSP s dynamickou rekonfiguráciou. Úloha obsahuje 7 operácií s dvomi prázdnymi: začiatočnou a ukončujúcou. Trvanie, zdroj, konfigurácia a následníci sú uvedení na tabuľke 2.1 Úloha ďalej obsahuje 3 obnoviteľné zdroje, ktoré pripúšťajú rôzny počet konfigurácií. Matice trvania zmeny konfigurácie sú uvedené v tabuľke 2.2. Počiatočné konfigurácie $c_0(r)$ sú uvedené tučne.

Príklad na rozvrh, vytvorený na základe definície úlohy je zobrazený ganttovým diagramom na obrázku 2.1. Operácie sú znázornené prázdnymi obdĺžnikmi s číslom. Rekonfigurácie sú šrafované obdĺžniky. Rozvrh nie je úplný, pretože neobsahuje operáciu 8.



Obr. 2.1: Ganttov diagram rozvrhu s dynamickými konfiguráciami

2.2.2 Pomocné funkcie a predikáty nad rozvrhom s dynamickou rekonfiguráciou

Ďalej zadefinujeme funkcie a predikáty pre abstrakciu v popise dynamickej charakteristiky rozvrhu. Príklady sa vzťahujú na vzorový rozvrh z obrázku 2.1.

Čas konca operácie

Čas konca operácie $j \in J$ pre rozvrh s značí funkcia $end(s, j)$, definovaná pre $j \in J_s$ ako:

$$end(s, j) = start(s, j) + d(j) \tag{2.1}$$

Príklad: $start(s, 2) = 1$, $end(s, 2) = 8$

Operácie vpravo a vľavo

$left(s, r, t)$ je najbližšia operácia naľavo od času t v rozvrhu s pre zdroj r, určená ako maximálny prvok množiny $\{j \in J_s | res(j) = r \wedge start(j) < t\}$, na ktorej je definované usporiadanie ako $x \leq y \Leftrightarrow start(x) \leq start(y)$.

$right(s, r, t)$ je najbližšia operácia napravo určená ako minimálny prvok množiny $\{j \in J_s | res(j) = r \wedge start(j) \geq t\}$ s rovnakým usporiadaním.

Príklad: $left(s, 2, 9) = 2$, $right(s, 2, 9) = 5$

Konfigurácia zdroja

Predpokladáme, že každá operácia je vykonávaná v správnej konfigurácii. Pokiaľ sú dve po sebe idúce operácie j_1 a j_2 na jednom zdroji r v rôznej konfigurácii, tak je pred zahájením operácie j_2 vykonaná rekonfigurácia, ktorej trvanie je definované v popise úlohy ako $d_r(conf(j_1), conf(j_2))$. Rekonfigurácia je teda zahájená v čase $start(j_2) - d_r(conf(j_1), conf(j_2))$ a ukončená v čase $start(j_2)$.

Na každom zdroji v rozvrhu je teda vykonávaná postupnosť rekonfigurácií $K(s, r)$, určená dvojicami nastavovanej konfigurácie c a časom začiatku rekonfigurácie t.

$$K(s, r) = \{(c_1, t_1), ..., (c_N, t_N)\} \tag{2.2}$$

Príklad: $K(s, 2) = \{(2, 0), (1, 8)\}$

Konfigurácia zdroja r pre rozvrh s v čase t je určená ako $config(s, r, t)$.

$$config(s, r, t) = \begin{cases} c_0(r) & \text{if } t < min\{t_k | (c, t_k) \in K(s, r)\} \\ \max\{(c, t_k) | (c, t_k) \in K(s, r) \wedge t_k < t\} & \text{otherwise} \end{cases} \tag{2.3}$$

Príklad: $config(s, 1, 4) = 1$

Obsadenosť zdroja

Pre určenie obsadenosti zdroja r pre rozvrh s v čase t vytvoríme predikát $empty(s, r, t)$, ktorý platí vtedy a len vtedy, ak nie je vykonávaná žiadna operácia: $\forall j \in J_s : (res(j) = r) \Rightarrow \neg(start(j) < t < end(j))$ a nie je vykonávaná žiadna rekonfigurácia: $\forall (c, t_k) \in K(s, r) : \neg(t_k < t < t_k + d_r(c_{prev}, c))$.

Obsadenie zdroja na intervale určíme pomocou predikátu $empty_i(s, r, t_S, t_E) \Leftrightarrow (\forall t_S \leq t \leq t_E : empty(s, r, t))$.

Príklad: $empty(s, 3, 4)$ a $empty_i(s, 1, 3, 6)$ platí, $empty(s, 3, 1)$ a $empty_i(s, 1, 3, 7)$ neplatí.

Platnosť rozvrhu

X je množina všetkých rozvrhov. X_f je množina všetkých uskutočniteľných rozvrhov. Uskutočniteľný rozvrh je taký rozvrh, kde sú splnené nasledujúce obmedzenia:

- Precedenčné obmedzenia:

$$\forall j_1, j_2 \in J_S : j_2 \in S_{j_1} \rightarrow end(j_1) \leq start(j_2) \tag{2.4}$$

- Obmedzenia kapacity na zdrojoch: žiadna operácia sa neprelína s inou na rovnakom stroji.

$$\begin{aligned} &\forall r \in R_s : \forall j_1, j_2 \in J_s : res(j_1) = res(j_2) = r \rightarrow \\ &end(j_2) > start(j_1) \rightarrow start(j_2) \geq end(j_1) \end{aligned} \tag{2.5}$$

- Obmedzenia na rekonfiguráciu: Medzi každými dvoma operáciami je dostatočný časový priestor pre vykonanie údržby.

$$\begin{aligned} &\forall r \in R_s : \forall j_1, j_2 \in J_s : res(j_1) = res(j_2) = r \rightarrow \\ &start(j_2) > start(j_1) \rightarrow end(j_1) \leq (start(j_2) - d_r(conf(j_1), conf(j_2))) \end{aligned} \tag{2.6}$$

Hodnotenie rozvrhu

Rozvrh je úplný, ak $J_s = J$. V množine riešení úlohy RCPSP s dynamickou údržbou X_s sú všetky úplné rozvrhy $s \in X_f$.

Na množine riešení je definovaná hodnotiaca funkcia $f : X_s \rightarrow \mathbb{N}$. Príkladom hodnotiacej funkcie je makespan f_M, ktorá ako hodnotenie vracia čas, v ktorý bude dokončená posledná operácia.

$$f_M(s) = \max(\{j \in J_s | end(j)\}) \tag{2.7}$$

Príklad: $f_M(s) = 14$

Optimálnym riešením úlohy je také, ktoré minimalizuje hodnotiacu funkciu f. Nájdenie optimálneho riešenia úlohy pre hodnotiacu funkciu f_M leží v triede NP-hard[1].

Ďalej sa budeme venovať hľadaniu suboptimálneho riešenia úlohy.

Algoritmus 1 *InsertJob*

Vstup: Rozvrh $s = (J_s, start)$, pridávaná operácia $j \in J$ a čas t, na ktorý bude operácia priradená.

Výstup: Rozvrh $s' = (J_{s'}, T^J_{s'}, K_{s'}, T^K_{s'})$, do ktorého je pridaná operácia j

$J_{s'} = J_s \cup \{j\}$

$start(j) = t$

2.2.3 Operácie nad rozvrhom

Zadefinujeme elementárne funkcie nad rozvrhom, pre pridávanie operácií. Tieto operácie sú ďalej využité ako abstrakcia pre algoritmy vytvárajúce rozvrh.

V niektorých algoritmoch budú využité operácie na odstránenie operácie z rozvrhu.

Algoritmus 2 *RemoveJob*

Vstup: Rozvrh $s = (J_s, start)$, odstraňovaná operácia $j \in J$.

Výstup: Rozvrh $s' = (J_{s'}, start)$, z ktorého je odstránená operácia j

$J_{s'} = J_s \setminus \{j\}$

$start(j) = undef$

2.2.4 Activity List

Pre riešenie úlohy RCPSP je bežne využívaný tzv. Activity List na reprezentáciu kandidátneho riešenia úlohy [8]. AL je usporiadaný zoznam obsahujúci každú operáciu $j \in J$ práve jedenkrát. Tiež pre každú operáciu platí, že všetci následníci $j_s \in S_j$ sa nachádzajú v AL za j a všetci predchodcovia $j_p \in P_j$ sa nachádzajú pred j.

2.2.5 Schedule Generation Scheme

Pre konverziu AL na rozvrh bude využitý algoritmus Schedule Generation Scheme. Výstup algoritmu je deterministicky vytvorený left-justified rozvrh na základe AL. Left-justified rozvrh je taký, v ktorom nie je možné lokálne posunúť žiadnu operáciu vľavo (Wiest, 1964)[13]. Algoritmus iteruje cez všetky operácie v AL v poradí a vkladá ich do rozvrhu na najskorší možný úsek.

Algoritmus 3 *SGS*

Vstup: Activity List $l = [j_0, ..., j_N]$

Výstup: Rozvrh $s = (J_s, start)$

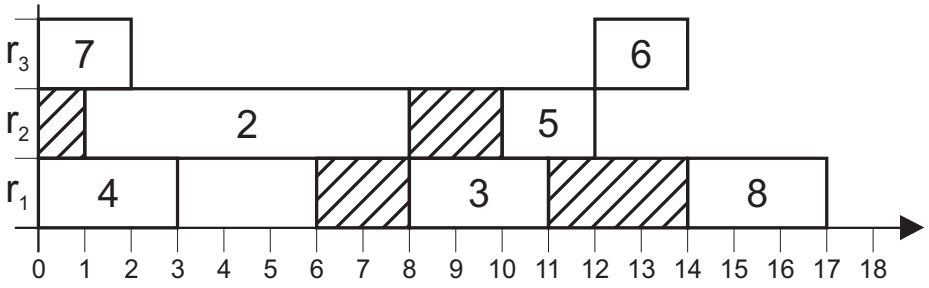
$J_s = \phi, start = \phi$

for $j_s \in l$ **do**

$s \leftarrow ScheduleJobLeft(s, j_s)$

end for

Funkcia *ScheduleJobLeft* vkladá operáciu do rozvrhu, pokiaľ možno čo najviac vľavo. Existuje viac metód, ako môže byť táto funkcia implementovaná. Na princípoch AL a SGS budú postavené všetky ďalej popísané algoritmy. Po vložení operácie musí rozvrh ostať left-justified.



Obr. 2.2: Rozvrh po pridaní operácii 8 bez ovplyvnenia zvyšku rozvrhu

2.3 Generovanie rozvrhu bez dynamických rekonfigurácií

Pre úplnosť predstavíme algoritmus vkladania operácie, ktorý neberie do úvahy rekonfigurácie. Tento algoritmus je využívaný pri riešení základnej úlohy RCPSP.

Algoritmus 4 *ScheduleJobLeft* bez rekonfigurácie

Vstup: Rozvrh s a rovzrhovaná operácia j.
Výstup: Rozvrh s', do ktorého je pridaná operácia j, na najskorší možný čas
$t_p = \max\{t_e | j_p \in P_j \wedge t_e = end(j_p)\}$ # Minimálny čas podľa predchodcov
$t_s = \min\{t | t > t_p \wedge empty_i(s, res(j), t, t + d_j)\}$ # Vyhľadanie najbližšieho voľného úseku
$s' = InsertJob(s, j, t_s)$

Vyhľadanie najbližšieho voľného úseku môže byť implementované rôznymi spôsobmi. Základná metóda je iterovanie cez zoradené operácie v rozvrhu a zisťovanie veľkosti časového úseku medzi každými dvoma operáciami, kým nie je nájdený dostatočne dlhý časový úsek. Časová zložitosť takéhoto algoritmu SGS je $\mathcal{O}(n^2)$ (Pinson)[11] Ďalšia možnosť je vytvorenie pomocnej dátovej štruktúry, ktorá mapuje nevyužité miesto v rozvrhu.

2.4 Generovanie rozvrhu s dynamickými rekonfiguráciami

Základný algoritmus vkladá operáciu aj s príslušnými rekonfiguráciami do rozvrhu tak, aby neovplyvnil ostatné operácie v rozvrhu.

Algoritmus *ScheduleJobLeftBasic* vkladá operáciu na najbližší voľný úsek v rozvrhu. Na začiatku sa určí najskorší možný začiatok podľa precedencií a následne sa vyhľadá najbližší voľný úsek o veľkosti trvania operácie d_j. Následne sa musia zistiť trvania potrebných rekonfigurácií. V prípade, že je interval dostatočne dlhý, je operácia vložená do rozvrhu. V prípade, že interval nie je dostatočne dlhý, vyhľadáva sa nasledujúci voľný interval, pokiaľ nie je operáciu možné rozvrhnúť.

Analýza zložitosti operácie ScheduleJobLeftBasic: Zistenie najneskoršieho konca predchádzajúcej operácie má lineárnu zložitosť. V prípade implementácie operácií $left$ a $right$ pomocným lineárnym zoznamom je ich zložitosť konštantná. Vyhľadanie trvania rekonfigurácie uvažujeme konštantné. Cyklus while sa vykoná prinajhoršom toľko krát, koľko je už rozvrhnutých operácií. Celková zložitosť je teda O(n).

Príklad: Po aplikácii operácie *ScheduleJobLeft* na rozvrh z obrázku 2.1 vznikne rozvrh na obrázku 2.2.

Algoritmus 5 *ScheduleJobLeftBasic*

Vstup: Rozvrh s a rozvrhovaná operácia j.

Výstup: Rozvrh s', do ktorého je pridaná operácia j, na najskorší možný čas bez presúvania ostatných operácií

```
t_p = max{t_e | j_p ∈ P_j ∧ t_e = end(j_p)}        # Minimálny čas podľa predchodcov
t_s ← t_p
while true do
  if left(s, res(j), t_s) = φ then
    c_L = c_0(res(j)); t_L = 0                      # Priradenie času konca operácie naľavo
  else
    j_L = left(s, res(j), t_s); c_L = conf(j_L) ; t_L = end(j_L)
  end if
  t_s ← max{t_p, t_L + d_r(c_L, conf(j))}
  if right(s, res(j), t_s) = φ then
    s' = InsertJob(s, j, t_s)                       # Posledná operácia
    return
  else
    j_R = right(s, res(j), t_s); c_R = conf(j_R) ; t_R = start(j_R)
  end if
  if d(j) + d_r(c_L, conf(j)) > (t_r − t_s) then
    t_s ← end(j_R)
  else
    s' = InsertJob(s, j, t_s)                       # Vloženie operácie s konfiguráciami
    return
  end if
end while
```

j	$d(j)$	$res(j)$	$S(j)$	j	$d(j)$	$res(j)$	$S(j)$
1	0	-	2,4,6,8	6	7	2	7
2	1	3	3	7	2	1	10
3	3	1	10	8	4	2	9
4	2	3	5	9	3	1	10
5	1	3	10	10	0	-	-

Tabuľka 2.3: Zoznam operácií v definícii úlohy s údržbou

r	$d^T(r)$	$d^M(r)$
1	5	2
2	5	1
3	1	0

Tabuľka 2.4: Zoznam zdrojov v definícii úlohy s údržbou

2.5 Riešenie úlohy pre dynamickú údržbu

Druhý problém riešený v tejto diplomovej práci je problém údržby.

2.5.1 Definícia rozvrhu s údržbou

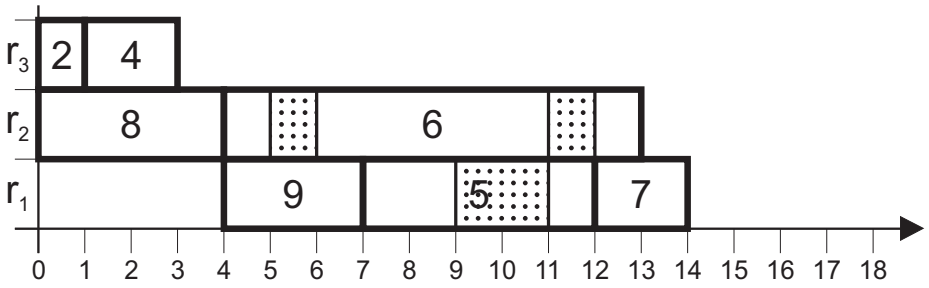
Pre riešenie problému údržby vyhovuje rovnaká definícia rozvrhu ako pri dynamickej rekonfigurácii. Rozvrh je definovaný ako dvojica $s = (J_s, start(j))$, kde

- $J_s \subseteq J$ je množina operácií v rozvrhu.
- $start(j) : J_s \to \mathbb{N}^0$ je priradenie času začiatku operáciám v rozvrhu.

Príklad úlohy RCPSP s údržbou

Predstavíme príklad úlohy RCPSP s údržbou. Úloha obsahuje 8 operácií s dvomi prázdnymi: začiatočnou a ukončujúcou. Trvanie, zdroj, a následníci sú uvedení na tabuľke 2.3.

Úloha ďalej obsahuje 3 obnoviteľné zdroje, ktoré pripúšťajú rôzny počet konfigurácií. Matice trvania zmeny konfigurácie sú uvedené na obrázku 2.2. Počiatočné konfigurácie $c_0(r)$ sú uvedené tučne.



Obr. 2.3: Ganttov diahram rozvrhu s údržbou

2.5.2 Pomocné funkcie a predikáty nad rozvrhom s údržbou

Údržba

Časy začiatku údržby sú určené podľa časov začiatkov naplánovaných operácií. Údržba je vykonávaná vždy počas trvania operácie. Pre zistenie, kedy je údržbu nutné vykonať v rámci operácie, potrebujeme poznať celkové trvanie predchádzajúcich operácií.

Funkcia $prev(s, r, t)$ vracia všetky operácie vyskytujúce sa v rozvrhu s na zdroji r pred časom t.

$$prev(s, r, t) = j \in J | res(j) = r \wedge start(s, j) < t \tag{2.8}$$

Celkové trvanie týchto operácií určuje funkcia $prev_d(s, r, t)$, ktorá pre rozvrh s a zdroj r, vráti celkové trvanie operácií pred časom t.

$$prev_d(s, r, t) = \sum_{j \in prev(s,r,t)} d(j) \tag{2.9}$$

Na základe času predchádzajúcich operácií je možné vypočítať počet potrebných rekonfigurácií počas trvania operácie.

$$maint_cnt(s, j) = (prev_d(s, res(j), start(j)) \mod d^T(res(j)) + d(j)) \; d^T \tag{2.10}$$

Koniec operácie a obsadenosť

Koniec operácie je ovplyvnený počtom údržieb vykonaných počas trvania operácie.

$$end(s, j) = start(s, j) + d(j) + maint_cnt(s, j) d^M(res(j)) \tag{2.11}$$

Koniec operácie, keby neboli vykonávané údržby označíme ako $end^O(s, j)$

$$end^O(s, j) = start(s, j) + d(j) \tag{2.12}$$

Pre určenie obsadenosti zdroja r pre rozvrh s v čase t vytvoríme predikát $empty(s, r, t)$, ktorý platí vtedy a len vtedy, ak nie je vykonávaná žiadna operácia: $\forall j \in J_s : (res(j) = r) \Rightarrow \neg(start(j) < t < end(j))$.

Operácie susedných operácií, hodnotiaca funkcia a obsadenie na intervale sú definované rovnako ako pri úlohe s dynamickými konfiguráciami.

Platnosť rozvrhu

X je množina všetkých rozvrhov. X_f je množina všetkých uskutočniteľných rozvrhov. Uskutočniteľný rozvrh je taký rozvrh, kde sú splnené nasledujúce obmedzenia:

- Precedenčné obmedzenia: definované rovnako ako v základnej úlohe.
- Obmedzenia kapacity na zdrojoch: žiadna operácia sa neprelína s inou na rovnakom stroji bez údržby.

$$\begin{aligned} &\forall r \in R_s : \forall j_1, j_2 \in J_s : res(j_1) = res(j_2) = r \rightarrow \\ &end^O(j_2) > start(j_1) \rightarrow start(j_2) \geq end^O(j_1) \end{aligned} \tag{2.13}$$

- Obmedzenia na údržbu: Je možné vykonať údržbu tak, aby sa operácie neprelínali.

$$\forall r \in R_s : \forall j_1, j_2 \in J_s : res(j_1) = res(j_2) = r \rightarrow \\ end(j_2) > start(j_1) \rightarrow start(j_2) \geq end(j_1) \tag{2.14}$$

2.6 Generovanie rozvrhu s údržbou

V tomto prípade je hľadaný prvý voľný úsek, tak aby bol po vložení operácie rozvrh znovu platný. V prípade rozvrhovania s konfiguráciami je porovnávaná dĺžka dostupného časového úseku so súčtom dĺžky operácie s rekonfiguráciami. V prípade rozvrhovania s údržbou môže naplánovanie operácie ovplyvniť nutnosť vykonania údržby na všetkých nasledujúcich operáciách. Preto je pri overovaní podmienky potrebné overiť pre všetky tieto operácie platnosť.

Algoritmus 6 *ScheduleJobLeftBasicM* s údržbou bez ovplyvnenia zvyšku rozvrhu

Vstup: Rozvrh s a rozvrhovaná operácia j.

Výstup: Rozvrh s', do ktorého je pridaná operácia j, na najskorší možný čas bez presúvania ostatných operácií

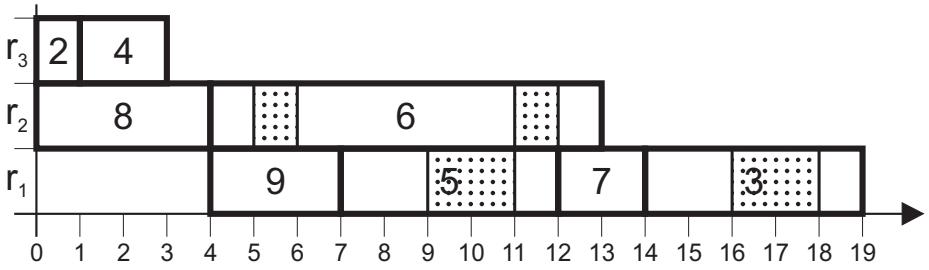
```
t_p = max{t_e | j_p ∈ P_j ∧ t_e = end(j_p)}      # Minimálny čas podľa predchodcov
t_s ← t_p
while true do
  if right(s, res(j), t_s) = φ then
    s' = InsertJob(s, j, t_s)                    # Posledná operácia
    return
  else
    j_R = right(s, res(j), t_s); t_R = start(j_R)
  end if
  s_x ← InsertJob(s, j, t_s)                     # Vloženie operácie a test platnosti
  if ∀j_1, j_2 ∈ J_s : (res(j_1) = res(j_2) = res(j)) → (end(j_2) > start(j_1) → start(j_2) ≥
  end(j_1)) then
    t_s ← end(j_R)
  else
    s' ← s_x                                     # Prijatie rozvrhu, ak je platný
    return
  end if
end while
```

Analýza zložitosti operácie *ScheduleJobLeftBasicM*: Rozdiel oproti rozvrhovaniu s údržbou spočíva v potrebe overiť platnosť rozvrhu v prípade vloženia operácie. Toto overenie má lineárnu zložitosť. Celková zložitosť bude teda $O(n^2)$.

Príklad: Po aplikácii operácie *ScheduleJobLeft* z algoritmu 6 s parametrom 3 na rozvrh z obrázku 2.1, vznikne rozvrh na obrázku 2.4.

2.7 Optimalizačné metódy

Úlohu sme si transformovali na vyhľadávanie v doméne AL, ktorá má obmedzený priestor riešení. Úlohou optimalizačných metód by malo byť teda nájsť optimálny AL. $AL_O \in AL^G$ je tzv. optimálny, ak $\nexists AL_1 \in AL^G | SGS^M(AL_1) < SGS^M(AL_O)$. AL^G je množina všetkých



Obr. 2.4: Rozvrh po pridaní operácii 8 bez ovplyvnenia zvyšku rozvrhu

platných AL, $SGS^M(AL)$ je funkcia, ktorá volá SGS, ktorý vytvorí rozvrh z AL a vyhodnotí ho hodnotiacou funkciou, ktorej hodnotu vráti.

Optimálny rozvrh je však možné s určitosťou vytvoriť len tak, že prehľadávame všetky platné AL. Tento problém je však NP-hard a pri úlohách s veľkým množstvom operácií je táto metóda prakticky neuskutočniteľná. Hľadáme preto suboptimálny AL, teda taký, ktorý sa čo najviac približuje optimálnemu riešeniu, prípadne je sám optimálny. Ďalej sa budeme zaoberať metódami, ktoré vedú k hľadaniu suboptimálneho riešenia.

2.7.1 Prehľadávanie náhodných riešení

Táto metóda je najjednoduchšia optimalizačná metóda. Úlohou je len nájsť najlepší rozvrh tak, že náhodne vygenerujeme určité množstvo Activity Listov a výsledným riešením bude ten, podľa ktorého vygenerujeme najlepšie ohodnotený rozvrh. Platný náhodný AL môžeme vygenerovať algoritmom 7.

Algoritmus 7 Generovanie náhodného AL

Vstup: $V = j_1, ..., j_N$
Výstup: $\lambda = (k_1, k_2, ..., k_N)$
  $J \leftarrow \{j_1\}$
  $i \leftarrow 1$
  while $J \neq \emptyset$ **do**
    $j_r \leftarrow$ náhodný prvok z J
    $J = J - \{j_r\}$
    $\lambda[i] \leftarrow j_r$
    $J = J \cup \{j \in S_{j_r} | j \notin \lambda \wedge \forall j_p \in P_j : j_p \in \lambda\}$
    $i \leftarrow i + 1$
  end while

Dôležitú úlohu môže zohrať aj náhodný generátor, ktorý by nám mal produkovať dostatočne variabilné výsledky. Študovanie rôznych vhodných generátorov však presahuje rozsah tejto práce.

2.7.2 Genetické algoritmy

Genetické algoritmy(GA) predstavené Hollandom v roku 1975[6], sú často využívanou metódou na hľadanie suboptimálneho riešenia a pri vhodnej implementácii a nastavení parametrov produkuje v obmedzenom čase oveľa lepšie výsledky ako jednoduché prehľadávanie náhodných rozvrhov. Sú inšpirované mechanizmom v prírode, kde informácia o jedincovi je

uchovaná v chromozóme a vďaka kríženiu jedincov, mutáciám a následnému prirodzeného výberu sa dosahuje lepšia schopnosť populácie prežiť.

V GA nachádzame podobnosť s týmto princípom. Chromozóm je dátová štruktúra(typicky zoznam), podľa ktorej je vytváraný jedinec. Schopnosť jedinca prežiť je určená výsledkom hodnotiacej funkcie. V prípade GA pre optimalizáciu výrobných rozvrhov chromozóm reprezentuje AL, jedinec je potom vygenerovaný rozvrh pomocou SGS algoritmov. Výber metódy tvorby rozvrhu a výber objektívnej funkcie potom teda zásadne ovplyvňujú GA. Na jedincov sú následne aplikované operátory kríženia a mutácie, pričom podľa výsledkov hodnotiacej funkcie je riadený výber jedincov, na ktorých sú tieto operátory aplikované.

Formalizujeme si prvky, ktoré nám vstupujú do GA. Chromozóm je pre nás totožný s pojmom AL. Potom populácia P je podmnožina AL^G.

$$P = \{\lambda_1, \lambda_2, ..., \lambda_p\} \subset AL^G$$

Populácia P obsahuje p chromozómov, v našom prípade AL. Vytvorenie jedinca z chromozómu a vyhodnotenie bude reprezentované už použitou funkciou $SGS^M(\lambda)$. V GA sa ďalej musí vyskytovať proces reprodukcie dvoch chromozómov.

$$(\lambda_1', \lambda_2') = O_{repro}(\lambda_1, \lambda_2)$$

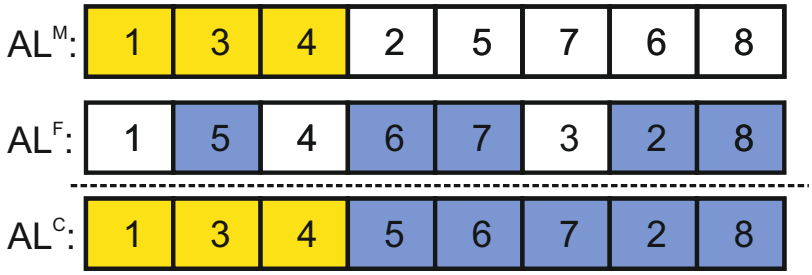
Proces zahŕňa operátory mutácie a kríženia, ďalej ho však budeme chápať ako bližšie nešpecifikovaný proces, ktorý má na vstupe dva rodičovské chromozómy a výsledkom sú dvaja potomkovia. Všeobecný genetický algoritmus potom bude mať tvar zobrazený v algoritme 8.

Algoritmus 8 Genetický algoritmus

Vstup: $t_{max}, V = \{j_1, ..., j_N\}$
Výstup: P_fin

```
t ← 0
P ← náhodne vygenerovaná populácia
vyhodnotenie populácie P
while t < t_max do
    Q ← ∅
    while |Q| < |P| do
        vyber náhodne 2 chromozómy λ1, λ2 ∈ P s nízkou hodnotou SGS^M(λ)
        if random < P_repro then
            (λ1', λ2') = O_repro(λ1, λ2)
        else
            (λ1', λ2') = (λ1, λ2)
        end if
        Q = Q ∪ {λ1, λ2}
    end while
    P ← Q
    vyhodnotenie populácie P
    t ← t + 1
end while
```

Na začiatku je náhodne vygenerovaná prvá populácia P_0. Pre jej vygenerovanie je použitý algoritmus pre generovanie náhodného AL. Počet týchto AL, teda veľkosť populácie



Obr. 2.5: Jednobodové kríženie

$|P|$ je dopredu definovaná. Máme dopredu určený počet generácií t_{max}, ktorý znamená koľko populácií GA vyprodukuje, pričom nasledujúca populácia je vždy vygenerovaná z aktuálnej populácie. Pri generovaní chromozómov do nasledujúcej populácie je časť presunutá priamo do nasledujúcej populácie a časť je vygenerovaná reprodukčnou funkciou $O_{repro}(\lambda_1, \lambda_2)$. Spôsob selekcie najlepších jedincov na presun do nasledujúcej operácie a reprodukciu závisí od konkrétnej implementácie algoritmu.

Operátor kríženia

Kríženie prebieha vždy medzi dvoma jedincami. Týmto môžeme dosiahnuť to, že ak sa stretnú dostatočne dobre ohodnotení jedinci, ich potomok môže zdediť dobré vlastnosti oboch rodičov a vzniknúť tak lepší jedinec. V rámci tejto práce som implementoval jednobodové a dvojbodové kríženie, ktoré sú v literatúre spomínané najčastejšie.

Jednobodové kríženie Na jednobodové kríženie dvoch AL sa využíva nasledujúci postup, prezentovaný Hartmannom v roku 1998[5]: Nech máme dva rodičovské AL, a to *materský* $\lambda^M = ({j_1}^M, {j_2}^M, ..., {j_N}^M)$ a *otcovský* $\lambda^F = ({j_1}^F, {j_2}^F, ..., {j_N}^F)$, z ktorých chceme vyprodukovať *synovský* AL $\lambda^C = ({j_1}^C, {j_2}^C, ..., {j_N}^C)$. Ďalej vygenerujeme náhodné celé číslo q, kde $1 \leq q < J$. Operácie na pozíciách $1, ..., q$ sú presunuté z *materského* AL nasledujúcim spôsobom:

$$ {j_i}^C = {j_i}^M \text{ pre } i = 1, ..., q $$

Operácie na pozíciách $i = q + 1, ..., N$ sú presunuté z *otcovského* AL tak, že presunieme všetky operácie, ktoré ešte nie sú presunuté z *materského* AL, pričom poradie zostane zachované:

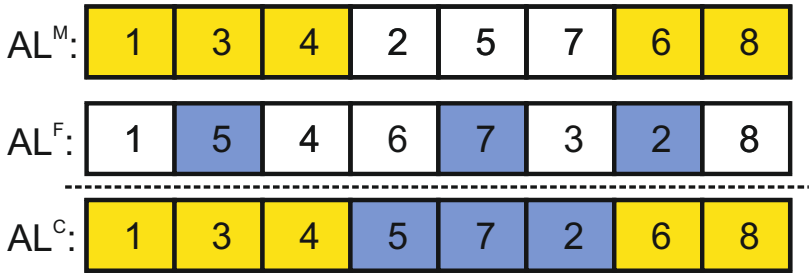
${j_i}^C = {j_k}^F$ kde k je najnižší index taký, že ${j_k}^F \notin \{{j_1}^C, {j_2}^C, ..., {j_N}^C\}$.

Druhý potomok je vytvorený rovnakým spôsobom iba vymeníme rodičov a vykonáme rovnakú operáciu so zachovaním indexu q.

$$ {\lambda_1}^M = \lambda^F; {\lambda_1}^F = \lambda^M $$

Vytvorený *synovský* AL je naďalej platný AL(Hartmann 1998)[5]. Príklad jednobodového kríženia je uvedený na obrázku 2.5.

Dvojbodové kríženie V tomto prípade vygenerujeme 2 náhodné celé čísla q_1, q_2, kde $1 \leq q_1 < q_2 < J$. Operácie na pozíciách $1, ..., q$ sú presunuté z *materského* AL nasledujúcim spôsobom:



Obr. 2.6: Dvojbodové kríženie

$$j_i{}^C = j_i{}^M \text{ pre } i = 1, ..., q_1$$

Operácie na konci AL sú taktiež presunuté z *materského* AL:

$$j_i{}^C = j_i{}^M \text{ pre } i = q_1 + 1, ..., N$$

Operácie na pozíciách $i = q_1 + 1, ..., q_2$ sú potom presunuté z *otcovského* AL tak, že presunieme všetky operácie, ktoré ešte nie sú presunuté z *materského* AL, pričom poradie zostane zachované:

$j_i{}^C = j_k{}^F$, kde k je najnižší index taký, že $j_k{}^F \notin \{j_1{}^C, j_2{}^C, ..., j_N{}^C\}$.

Dvojbodové kríženie rovnako generuje z hľadiska precedencie platné AL[5]. Príklad dvojbodového kríženia je uvedený na obrázku 2.6.

Operátor mutácie

Mutácie v genetických algoritmoch zabezpečujú dostatočnú variabilitu v populácii. Bez použitia mutácií, by sa vďaka selekcii najlepších jedincov na kríženie po niekoľkých iteráciách vytvorila taká populácia, v ktorej sú všetci jedinci veľmi podobní a krížením by už nebolo možné dosiahnuť lepšie výsledky. Úlohou mutácie je spôsobiť malú zmenu v Activity Liste, aby k tomuto javu nedochádzalo. Ak chceme zvýšiť silu mutácie, môžeme ju aplikovať viacnásobne na jeden prvok. V rámci tejto práce som implementoval dve metódy mutácie.

Výmena dvoch prvkov Máme AL $\lambda = (j_1, j_2, ..., j_N)$, na ktorý chceme aplikovať operátor mutácie. Vygenerujeme 2 náhodné celé čísla q_1, q_2, kde $1 < q_1 < q_2 < J$. Mutácia prebehne tak, že vymeníme prvok na pozícii q_1 a q_2. Výstupný AL tak bude $\lambda_O = (j_1, j_2, ..., j_{q_1-1}, j_{q_2}, j_{q_1+1}, ..., j_{q_2-1}, j_{q_1}, j_{q_2+1}, ..., j_N)$. Pri tejto výmene však môže nastať porušenie precedencie a je nutné skontrolovať či je výstupný AL platný. Ak precedencia neplatí, musíme operáciu mutácie opakovať na pôvodnom AL.

Tento druh mutácie je náročný na výpočtový výkon, pretože je nutné vykonať náročné overenie platnosti zmeny. Navyše, ak je operácia neplatná, mutácia ani nemôže byť vykonaná. Ak sa to stáva často, náročnosť sa zásadne zvyšuje. Tento prípad nastáva pri rozsiahlom grafe precedencie, kde operácie majú vysoký počet predchodcov. Výhoda tejto metódy je však veľká sila mutácie, kde sa prvok dokáže posunúť o mnoho miest.

Posúvanie prvku Máme AL $\lambda = (j_1, j_2, ..., j_N)$, na ktorý chceme aplikovať operátor mutácie. Vygenerujeme náhodné celé číslo q, kde $1 < q < J$. Výstupný AL bude $\lambda_O =$

$(j_1, j_2, ..., j_{q-1}, j_{q+1}, j_q, j_{q+2}, ..., j_N)$. Tiež overíme, či po výmene bude Activity List platný, pričom stačí overenie vzťahu dvoch vymieňaných prvkov. Následne môžeme index q inkrementovať alebo dekrementovať podľa náhodne vygenerovaného smeru posúvania a mutáciu opakovať.

Tento druh mutácie je jednoduchší oproti výmene dvoch prvkov a nie je tak výpočtovo náročný. Zlyhanie overenia platnosti výmeny je tiež oveľa menej pravdepodobné, pretože kontrolujeme len to, či dve operácie nemajú priamu následnosť. Mutácia však nespôsobuje zásadné zmeny v Activity Liste, čo však môžeme nahradiť viacnásobným použitím tejto mutácie.

Kapitola 3

Pokročilé metódy tvorby rozvrhu

V rozbore úlohy bol predstavený základný algoritmus SGS, ktorý operácie vkladá do rozvrhu postupne podľa poradia v AL so striktným dodržaním všetkých obmedzení. Tento postup však nemusí generovať uspokojivé výsledky.

V tejto kapitole sú predstavené pokročilé metódy tvorby rozvrhu. Rozdiel spočíva v pripustení dočasného zjemnenia podmienok na platnosť a uvažovanie možnosti presúvania rozvrhnutých operácií počas procesu generovania. Produkované výsledky algoritmov budú ďalej skúmané a porovnávané v nasledujúcich kapitolách.

3.1 Generovanie rozvrhu s dynamickými rekonfiguráciami

Pri tvorbe rozvrhu s dynamickými rekonfiguráciami musíme dodržať navyše obmedzenie na konfigurácie. Obmedzenie musí byť platné vždy pri výslednom vytvorenom rozvrhu. Pri niektorých postupoch však môžeme pripustiť dočasné zneplatnenie rozvrhu. Algoritmy preto rozdeľujeme na:

- SGS so striktne platným rozvrhom
- SGS s dočasne neplatným rozvrhom

3.1.1 SGS so striktne platným rozvrhom s rekonfiguráciami

V tejto kategórii predstavíme dva algoritmy. Základný algoritmus vkladá operáciu aj s príslušnými konfiguráciami do rozvrhu tak, aby neovplyvnil ostatné operácie v rozvrhu. V druhom prípade algoritmus môže manipulovať s ostatnými operáciami.

Rozvrhovanie bez ovplyvnenia zvyšku rozvrhu

Do tejto kategórie patrí základný algoritmus predstavený v kapitole 2.

Rozvrhovanie s posunutím operácií

Druhá možnosť je namiesto vyhľadania úseku, kde má operácia dostatočne dlhý časový úsek aj pre rekonfigurácie, manipulovať s ostatnými operáciami tak, aby rozvrhovaná operácia bola rozvrhnutá na prvý dostatočne dlhý úsek pre operáciu bez rekonfigurácií.

Algoritmus 9 funguje na rovnakom princípe ako algoritmus bez ovplyvnenia zvyšku rozvrhu 5, akurát po nájdení časového úseku, operáciu môže vložiť, napriek tomu, že sa

Algoritmus 9 *ScheduleJobLeftShift*

Vstup: Rozvrh s a rozvrhovaná operácia j.

Výstup: Rozvrh s', do ktorého je pridaná operácia j, na najskorší možný čas s prípadným presúvaním ostatných operácií

```
t_p = max{t_e | j_p ∈ P_j ∧ t_e = end(j_p)}        # Minimálny čas podľa predchodcov
t_s ← t_p
while true do
  if left(s, res(j), t_s) = φ then
    c_L = c_0(res(j)); t_L = 0                      # Priradenie času konca operácie naľavo
  else
    j_L = left(s, res(j), t_s); c_L = conf(j_L) ; t_L = end(j_L)
  end if
  t_s ← max{t_p, t_L + d_r(c_L, conf(j))}
  if right(s, res(j), t_s) = φ then
    s' = InsertJob(s, j, t_s)                       # Posledná operácia
    return
  else
    j_R = right(s, res(j), t_s); c_R = conf(j_R) ; t_R = start(j_R)
  end if
  if d(j) + d_r(c_L, conf(j)) > (t_r − t_s) then
    if altCondition(j, j_L, j_R) then
      s' = Mount(s, j, t_s, j_L, j_R)               # Alternatívne vloženie operácie
    else
      t_s ← end(j_R)
    end if
  else
    s' = InsertJob(s, j, t_s)                       # Vloženie operácie s konfiguráciami
    return
  end if
end while
```

operácia nemusí dať naplánovať bez posunutia ostatných operácií. Toto vloženie je vykonané operáciou *AltMount*. Podmienku, či môžeme operáciu takto vložiť predstavuje predikát *AltCondition*

Ako *AltCondition* môžeme využiť napríklad nasledujúce predikáty:

- ***ignoreConfAltConditon***: operáciu je možné vložiť bez rekonfigurácie.

$$end(j_L) + d(j) \leq start(j_R)$$

- ***sameConfAltCondition***: operácia má rovnakú konfiguráciu, ako niektorá z vedľajších operácií a existuje voľný časový úsek.

$$end(j_L) + d(j) \leq start(j_R) \wedge (conf(j_R) = conf(j) \vee conf(j_L) = conf(j))$$

Pri rozširovaní intervalu môžu byť v najhoršom prípade posunuté všetky operácie. Ako základné riešenie predstavíme postup, kedy sú v prvom kroku odstránené všetky operácie kauzálne závislé na operácii ohraničujúcej interval sprava. Tieto operácie sú následne vkladané späť do rozvrhu v poradí podľa času začiatku pred odstránením.

Algoritmus 10 *shiftJobsMount* s posunutím operácií

Vstup: Rozvrh s, pridávaná operácia $j \in J$,čas začiatku t_s, operácia vpravo j_R, operácia vľavo j_L

Výstup: Rozvrh s', do ktorého je pridaná operácia

$(s_r, D) = RemoveAfter(s, j_R)$
$s' = InsertJob(s_c, j, t_s)$
$N = |D|$
$l = [j_0, ..., j_N]$, kde $(j_i, t_i) \in D$ triedené podľa t_i
for $j_s \in l$ **do**
  $s' \leftarrow ScheduleJobLeft(s_r, j_s)$
end for

V algoritme 10 shiftJobsMount je volaná abstraktná funkcia *ScheduleJobLeft* na opätovné vloženie odstránených operácii. Ako konkrétnu implementáciu môžeme využiť *ScheduleJobLeftBasic*, alebo rekurzívne volať ScheduleJobLeftShiftJobs.

Analýza časovej zložitosti: Vloženie jednej operácie má rovnakú zložitosť ako pri základnom algoritme, teda $O(n)$. Pri využití *ScheduleJobLeftBasic* na opätovné vloženie operácie, bude potrebné znovu rozvrhnúť maximálne n operácií. Teoretická zložitosť tejto varianty je teda $O(n^2)$. Pri rekurzívnom volaní *ScheduleJobLeftShiftJobs* teoreticky môže byť potrebné rozvrhnúť až n^n operácií a zložitosť je teda $O(2^n)$.

Táto zložitosť je však len teoretické horné ohraničenie a v tejto práci nie je skúmaná existencia vstupu, pri ktorom je potrebný takýto vysoký počet rozvrhnutí. Skutočnú časovú zložitosť algoritmu ďalej experimentálne overíme.

Príklad: Pridávame operáciu 8 do rozvrhu na obrázku 2.1. V prvej fáze sú odstránené operácie, ktoré obmedzujú voľný časový úsek, čo je zobrazené na obrázku 3.1. Následne sú tieto operácie vložené do rozvrhu v poradí ako sa v ňom vyskytovali predtým. Výsledný rozvrh je zobrazený na obrázku 3.2.

3.1.2 SGS s dočasným neplatným rozvrhom s rekonfiguráciami

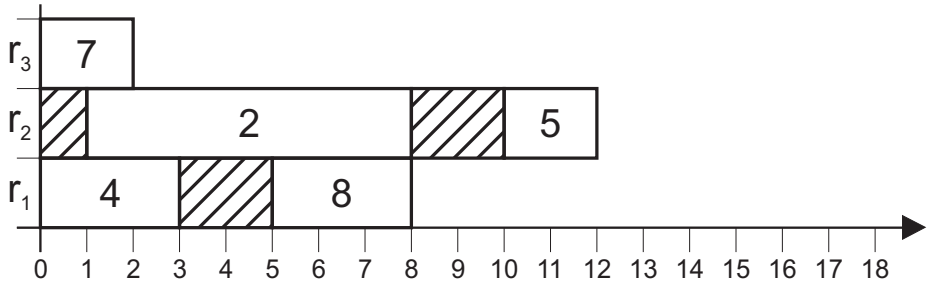
Druhá možnosť je vložiť operáciu do rozvrhu napriek porušeniu obmedzení na konfigurácie. Ak bude možné operáciu vložiť do rozvrhu bez konfigurácií, tak ju naplánujeme. Takýto

Algoritmus 11 *RemoveAfter*

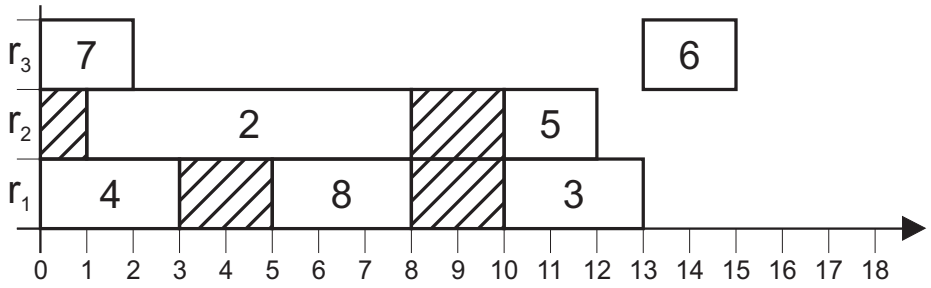
Vstup: Rozvrh s, odstraňovaná operácia $j \in J$.

Výstup: Rozvrh s', z ktorého sú odstránené operácie závislé na j a odstránené operácie s príslušnými časmi začiatku $D = \{(j_1, t_1), .., (j_N, t_N)\}$.

$j_R = right(s, res(j), end(s, j))$
if $j_R = \phi$ **then**
  $l = \phi$
  $s_a = s$
else
  $(s_n, l) = RemoveAfter(s, j_R)$
end if
for all $j_s \in S_j$ **do**
  $(s_s, l_s) = RemoveAfter(s_a, j_R)$
  $s_a \leftarrow s_s$
  $l \leftarrow l \cup l_s$
end for
$l \leftarrow l \cup (j, start(s, j))$
$s' = RemoveJob(s_a, j)$



Obr. 3.1: Rozvrh po odstránení následníkov, a pridaní operácie 8



Obr. 3.2: Rozvrh po pridaní operácie 8 s posunutím následníkov

rozvrh využívame ďalej aj pri plánovaní ostatných operácii. Až po rozvrhnutí všetkých operácií vyriešime neplatnosť rozvrhu.

Pri tejto možnosti použijeme taktiež operáciu *ScheduleJobLeft* z algoritmu 9. Postup sa bude líšiť implementáciou operácie *Mount*. Túto operáciu nazveme *ignoreconfMount* a bude len jednoducho vkladať operáciu do rozvrhu bez kontroly obmedzení.

Algoritmus 12 *ignoreconfMount*

Vstup: Rozvrh s, pridávaná operácia $j \in J$,čas začiatku t_s, operácia vpravo j_R, operácia vľavo j_L

Výstup: Rozvrh s', do ktorého je pridaná operácia

$s' = InsertJob(s, j, t_s)$

Post-processing neplatného rozvrhu

Takto vytvorený rozvrh však nespĺňa konfiguračné obmedzenie. Riešením je vytvoriť nový rozvrh tak, že je rešpektované poradie operácií. Algoritmus vytvorenia nového rozvrhu funguje podobným spôsobom ako SGS. Všetky operácie sú však plánované za poslednú operáciu na príslušnom zdroji a nevznikajú tak konflikty.

Na rozvrh vzniknutý v predchádzajúcom kroku aplikujeme operáciu *sgsPostProcess* popísanú v algoritme 13. Táto operácia vytvorí nový AL zo vstupného rozvrhu, a podľa neho vytvorí platný rozvrh.

Algoritmus 13 *sgsPostProcess*

Vstup: Rozvrh $s = (J_s, T_s^J, K_s, T_s^K)$ s porušenými obmedzeniami.

Výstup: Rozvrh $s' = (J_{s'}, T_{s'}^J, K_{s'}, T_{s'}^K)$, ktorý spĺňa obmedzenia, pričom rešpektuje poradie operácii v rozvrhu s.

$l = [j_0, ..., j_N]$, kde $j_i \in J_s$ triedené podľa $start(s, j_i)$

$J_{s'} = \phi, T_{s'}^J = \phi, K_{s'} = \phi, T_{s'}^K = \phi$

for $j_s \in l$ **do**

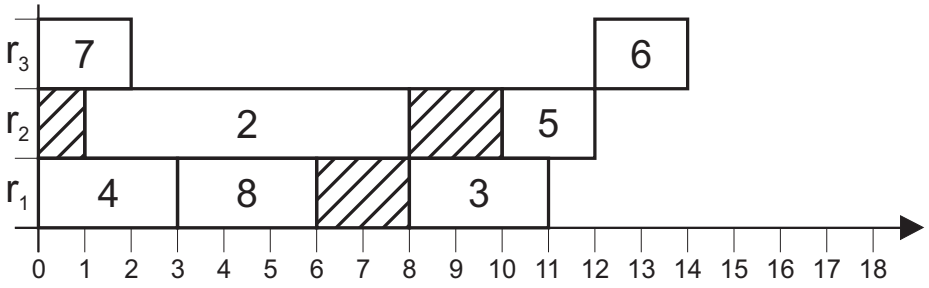
$s \leftarrow ScheduleJobLeftBasic(s, j_s)$

end for

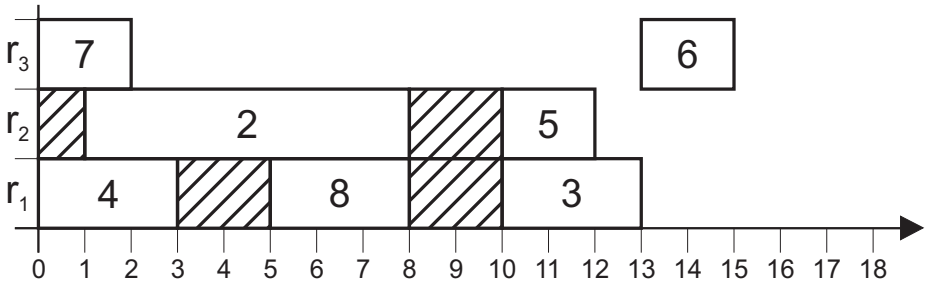
Príklad: Vytvárame rozvrh s AL [1,2,3,4,5,6,7,8,9]. Na začiatku je vytvorený rozvrh zobrazený na obrázku 3.3. Tu si môžeme všimnúť porušenie obmedzenia na konfiguráciu. Medzi operáciami 4 a 8 nie je naplánovaná potrená rekonfigurácia dĺžky 2. Pri aplikácii post-processingu vznikne AL [1,7,4,2,8,3,5,6,9]. Z tohto AL je vytvorený konečný rozvrh zobrazený na obrázku 3.4.

3.2 Generovanie rozvrhu s údržbou

Na generovanie rozvrhu s údržbou budú taktiež využité AL a SGS. Využijeme tiež rovnaké kategórie algoritmov ako pri rozvrhovaní s dynamickou rekonfiguráciou. Niektoré časti sú však mierne modifikované, aby výsledok odpovedal rozvrhu s údržbami. Popis bude zameraný hlavne na rozdielne prvky oproti rozvrhovaniu s rekonfiguráciami. Taktiež budú predstavené príklady pre každú z metód.



Obr. 3.3: Rozvrh vytvorený s povolením porušenia obmedzení



Obr. 3.4: Rozvrh po aplikácii post-processingu

3.2.1 SGS so striktne platným rozvrhom s údržbou

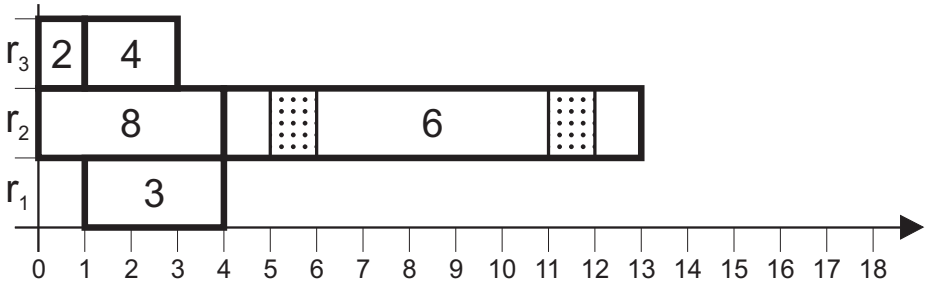
Rozvrhovanie bez ovplyvnenia zvyšku rozvrhu

Do tejto kategórie patrí základný algoritmus predstavený v kapitole 2.

Rozvrhovanie s posunutím operácií

Zmena *ScheduleJobLeftM* je rovnaká ako pri rozvrhovaní s konfiguráciami. Operácia alternatívneho rozvrhnutia operácie *Mount* je rozdielna len v tom, že pri opätovnom pridávaní odstránenej operácie sa využíva *ScheduleJobBasicM* namiesto *ScheduleJobBasic*.

Príklad: Pridávame operáciu 3 do rozvrhu na obrázku 2.1. V prvej fáze sú odstránené operácie, ktoré obmedzujú voľný časový úsek, čo je zobrazené na obrázku 3.5. Následne sú tieto operácie vložené do rozvrhu v poradí, ako sa v ňom vyskytovali predtým. Výsledný rozvrh je zobrazený na obrázku 3.6.



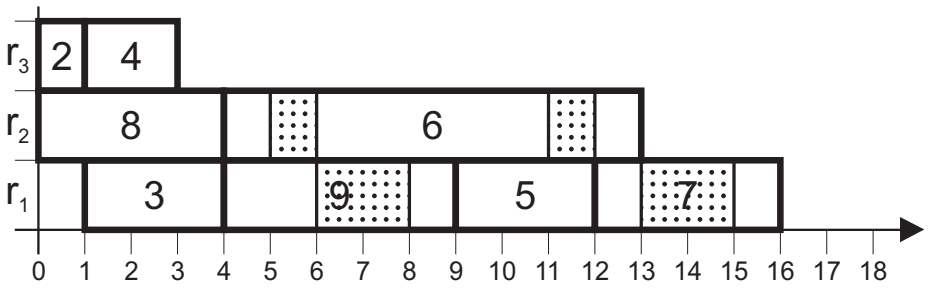
Obr. 3.5: Rozvrh po odstránení následníkov a pridaní operácie 3

Algoritmus 14 *ScheduleJobLeftShiftM* s údržbou a posunutím operácií

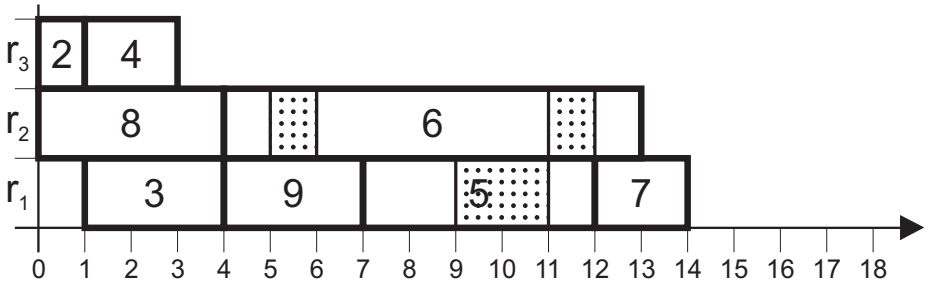
Vstup: Rozvrh s a rozvrhovaná operácia j.

Výstup: Rozvrh s', do ktorého je pridaná operácia j, na najskorší možný čas bez presúvania ostatných operácií

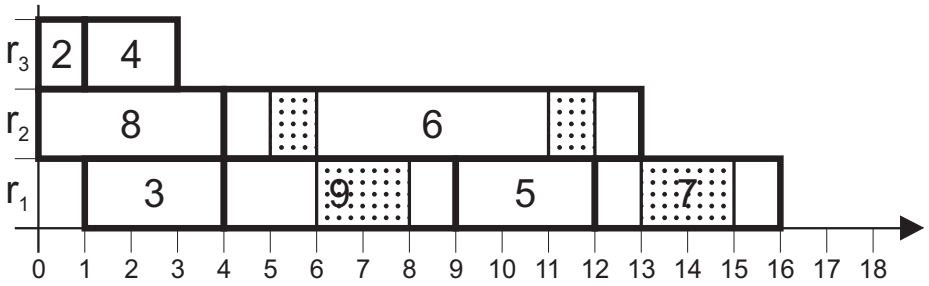
```
t_p = max{t_e | j_p ∈ P_j ∧ t_e = end(j_p)}        # Minimálny čas podľa predchodcov
t_s ← t_p
while true do
  if right(s, res(j), t_s) = φ then
    s' = InsertJob(s, j, t_s)                      # Posledná operácia
    return
  else
    j_R = right(s, res(j), t_s); t_R = start(j_R)
  end if
  s_x ← InsertJob(s, j, t_s)                       # Vloženie operácie a test platnosti
  if ∀j_1, j_2 ∈ J_s : (res(j_1) = res(j_2) = res(j)) → (end(j_2) > start(j_1) → start(j_2) ≥
  end(j_1)) then
    if altCondition(j, j_L, j_R) then
      s' = Mount(s, j, t_s, j_L, j_R)              # Alternatívne vloženie operácie
    else
      t_s ← end(j_R)
    end if
  else
    s' ← s_x                                       # Prijatie rozvrhu, ak je platný
    return
  end if
end while
```



Obr. 3.6: Rozvrh po pridaní operácie 3 s posunutím následníkov



Obr. 3.7: Rozvrh vytvorený s povolením porušenia obmedzení



Obr. 3.8: Rozvrh po aplikácii post-processingu

3.2.2 SGS s dočasným neplatným rozvrhom s údržbou

Postup je takmer totožný pri rozvrhovaní s rekonfiguráciami.

Príklad: Vytvárame rozvrh s AL [1,2,4,8,9,5,7,6,3,10]. Na začiatku je vytvorený rozvrh zobrazený na obrázku 3.7. Tu si môžeme všimnúť porušenie obmedzenia na údržbu. Zdroj 1 je obsadený od času 1 po čas 9 po dobu 8 časových jednotiek, bez vykonania údržby. Maximálna perióda pre údržbu je podľa zadania 5 časových jednotiek. Pri aplikácii post-processingu vznikne AL [1,2,8,3,4,9,6,5,7,10]. Z tohto AL je vytvorený konečný rozvrh zobrazený na obrázku 3.8.

Kapitola 4

Návrh riešenia

Táto kapitola sa zameriava na praktické riešenie zadania diplomovej práce. Na začiatku je problematika upresnená o skutočné požiadavky výroby. Ďalej je popísaný návrh programu implementujúceho algoritmy spomínané v teoretickej časti a spôsob interakcie s užívateľom.

4.1 Spresnenie problematiky do situácie výrobných podnikov

Model RCPSP bol pre účely tejto práce rozšírený predovšetkým pre modelovanie procesov výroby. Tu sa často vyskytuje problém potreby rekonfigurácie a údržby strojov. Návrh programu, ktorý vypočítava rozvrh a príslušné testovacie dáta sú ďalej prispôsobené na požiadavky výroby. Táto práca bola vyhotovená v spolupráci s podnikmi HDF Nižná, lisovňou na výrobu plastových výrobkov a Metaltrimom Oravská Jasenica, strojárskym podnikom na obrábanie kovov.

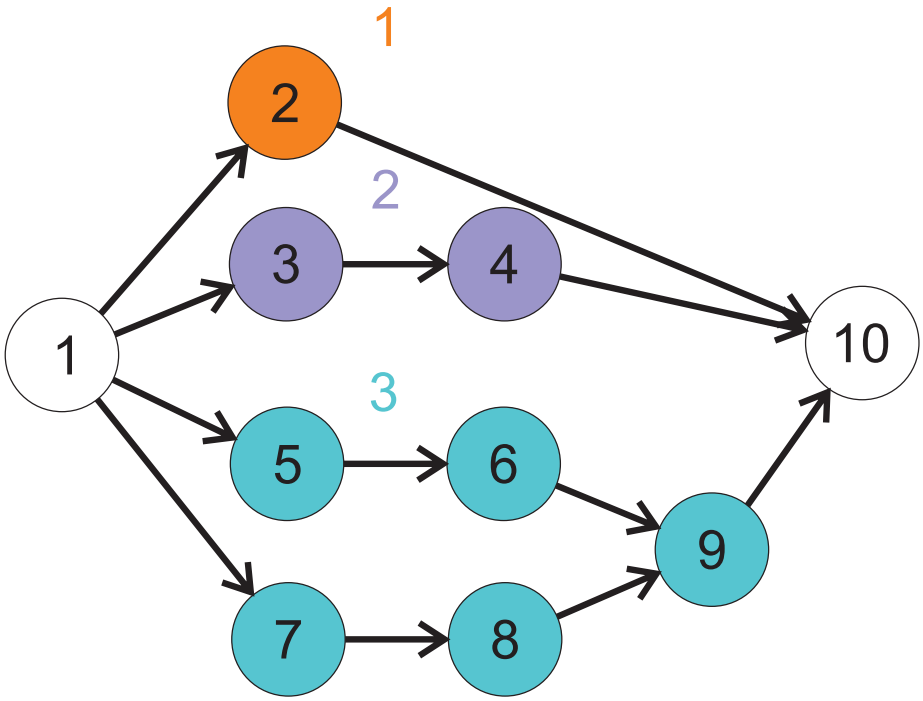
4.1.1 Výroba

Jednotkou výroby výrobkov je typicky zákazka alebo výrobný príkaz, ktorý je požiadavkou na výrobu určitého počtu výrobkov s požadovaným termínom. Precedenčný graf potom pozostáva zo skupín nezávislých operácií, ktoré vedú k produkcii výrobkov na zákazke. Jednoduchý výrobný príkaz pozostáva zo sekvencie operácií. Pri zložitejšej výrobe sa môže vyskytnúť operácia montáže dvoch alebo viacerých výrobkov. Táto operácia je závislá na predchádzajúcich operáciách. Príklad precedenčného grafu je zobrazený na obrázku 4.1.

4.2 Návrh programu

V rámci diplomovej práce bol navrhnutý program na riešenie problému RCPSP s konfiguráciami a údržbou, ktorý bol následne implementovaný jazyku Haskell.

Program pozostáva z nasledujúcich komponent: Hlavnou časťou návrhu je algoritmus SGS, popísaný v predchádzajúcich kapitolách, transformujúci AL na rozvrh. Vyprodukovaný rozvrh je následne ohodnotený hodnotiacou funkciou. Optimalizátor pracuje s kandidátnymi riešenia a ich hodnoteniami s účelom vyprodukovať čo najlepšie hodnotený rozvrh. Počiatočné náhodné kandidátne riešenia sú produkované v generátore.



Obr. 4.1: Príklad precedenčného grafu úlohy, ktorá obsahuje 3 výrobné príkazy.

SGS	**Skr.**	**AltConditon**	**Mount**	**PostProc.**
sgsBasic	B	no	-	-
sgsPostProcess	P	spaceWithoutConf	ignoreConf	sgsBasic
sgsPostProcessIgnoreM	PM	spaceWithoutM	ignoreConf	sgsBasic
sgsShiftJobs	S	spaceWithoutConf	shiftJobsBasic	-
sgsShiftJobsRecursive	SR	spaceWithoutConf	shiftJobs	-
sgsShiftJobsIgnoreM	SM	spaceWithoutM	shiftJobsBasic	-
sgsShiftJobsRecursiveIgnoreM	SMR	spaceWithoutM	shiftJobs	-

Tabuľka 4.1: Implementované varianty SGS

4.2.1 Návrh univerzálneho generátora rozvrhu

V kapitole 3 sú popísané rôzne varianty algoritmu SGS oddelene pre problém RCPSP s údržbou a rekonfiguráciou. Pri implementácii algoritmu predpokladáme úlohy, ktoré súčasne riešia RCPSP s rekonfiguráciou aj údržbou. Výsledkom je univerzálny SGS s jednotnou štruktúrou. Tento generátor je parametrizovateľný alternatívnou podmienkou, alternatívnou metódou vloženia operácie a prípadným post-processingom. Všetky uvažované kombinácie týchto parametrov sú popísané v tabuľke 4.1. Celkovo je teda implementovaných 7 variácií SGS.

Podmienku na alternatívne rozvrhnutie operácie **AltCondition** je pre spojený algoritmus s rozvrhovaním aj údržbou potrebné znovu definovať:

- **noAltCondition**: nikdy neplatí, operácia je rozvrhnutá vždy štandardne.

- **spaceWithoutConfAltCondition**: operácia je rozvrhnutá alternatívne, ak je ju možné vložiť s porušením podmienky na rekonfiguráciu alebo údržbu.
- **spaceWithoutMAltCondition**: operácia je rozvrhnutá alternatívne, ak je ju možné vložiť s porušením podmienky na rekonfiguráciu alebo údržbu.

4.2.2 Návrh optimalizátora

Pre optimalizáciu bol implementovaný parametrizovateľný genetický algoritmus založený na princípoch popísaných v druhej kapitole. Jeho chovanie definujú nasledujúce zameniteľné súčasti:

- **SGS**: Tento parameter určuje, ktoré z implementovaných SGS sa použije.
- **Generátor AL**: AL generovaný podľa zákaziek
- **Mutácia**: Vo všetkých prípadoch je momentálne využívaná mutácia s posúvaním prvku. Parametrom je možné ovplyvniť maximálnu vzdialenosť posunutia.
- **Kríženie**: Vo všetkých prípadoch je momentálne využité jednobodové kríženie. Bod kríženia je náhodne vybratý. Kríženie dokáže pracovať s oboma typmi navrhnutých kandidátnych riešení.
- **Hodnotiaca funkcia**

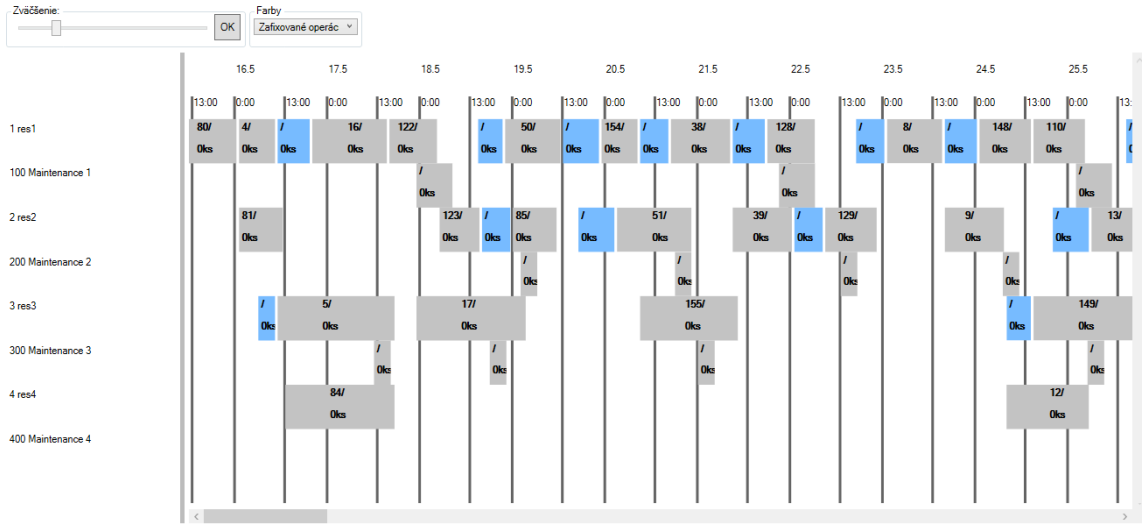
Samotný genetický algoritmus ďalej ovplyvňujú nasledujúce parametre, ktoré určujú predovšetkým zloženie nasledujúcej populácie:

- **GenerationCount**: Určuje počet generácií, tj. koľko iterácii GA pobeží.
- **TopSize**: Počet presunutých najlepších jedincov do nasledujúcej generácie.
- **CrossSize**: Počet jedincov, ktorí vzniknú krížením najlepších. Na takto vygenerované AL je následne aplikované kríženie.
- **TopCross**: Hranica najlepších jedincov na kríženie.
- **RandBotSize**: Počet jedincov, ktorí vzniknú krížením a následnou mutáciu spomedzi všetkých prvkov populácie.
- **NewSize**: Počet novo vygenerovaných AL v každej populácii.

4.2.3 Návrh vstupného súboru s popisom úlohy

Pre účely načítania úlohy bol navrhnutý vstupný formát RCPC. Navrhnuté kódovanie rozširuje formát rcp využívaný v knižnici psplib. V hlavičke boli pridané riadky popisujúce informácie o zdrojoch a ich rekonfiguračných časoch. Operácie sú rozšírené o 2 stĺpce: konfigurácia a číslo zákazky.

Súbor je tvorený celými číslami v textovom tvare a oddeľovačmi tabulátor a nový riadok. Formát je nasledovný:



Obr. 4.2: Vizualizácia rozvrhu v informačnom systéme ISIT ganttovým diagramom. Modro znázornené operácie značia rekonfigurácie. Zdroje s pomenovaním maintenance zobrazujú, kedy je vykonávaná údržba.

```
jobcount rescount
[res_conf_count defcount default_configuration maint_duration maint_period
[c1 c2 duration
](defcount)
](rescount)
(res order conf duration succcount [succ ](succcount)
)jobcount
```

Tvar `[content](count)` značí, že položka `content` sa opakuje `count` krát. Príklad vstupného súboru sa nachádza v prílohe B.

4.2.4 Vizualizácia generovaného rozvrhu a výstup

Pre účely zobrazenia rozvrhu v podobe Ganttovho diagramu je využitý program, ktorý som vyvinul v rámci bakalárskej práce. Vygenerovaný rozvrh na platforme Haskell je možné uložiť na výstup v podobe SQL scriptu. Po spustení skriptu nad SQL databázou informačného systému ISIT je možné rozvrh otvoriť vo vizualizačnej aplikácii. Príklad výstupu je zobrazený na obrázku 4.2.

Kapitola 5

Implementácia

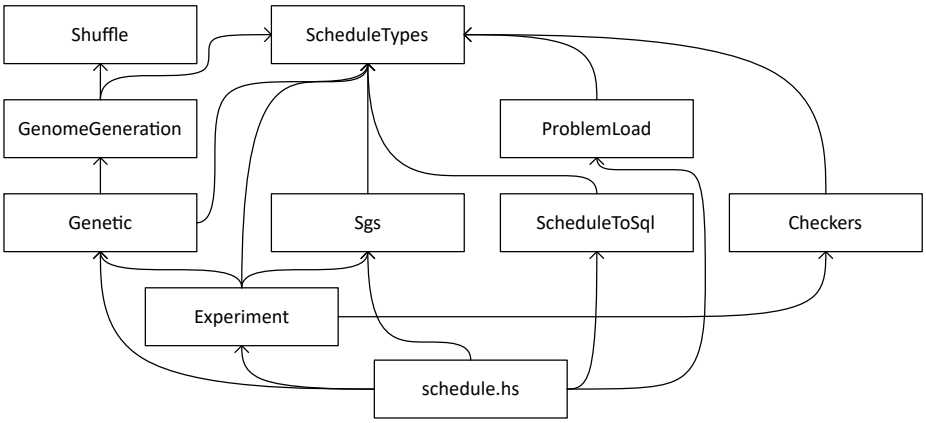
Hlavným výstupom diplomovej práce mimo technickú správu je počítačový program implementujúci navrhnuté postupy riešenia RCPSP s dynamickou rekonfiguráciou a údržbou. Druhým vytvoreným programom je generátor inšancií úlohy. V tejto kapitole budú ilustrované kľúčové súčasti programu z implementačného hľadiska.

5.1 Program na riešenie RCPSP s dynamickou rekonfiguráciou a údržbou

Pre implementáciu navrhnutých riešení bol zvolený programovací jazyk Haskell. Toto riešenie bolo zvolené, pretože problém je podrobne popísaný matematickým modelom a funkcionálny jazyk poskytuje formálnu bázu na popis takýchto modelov. Medzi ďalšie výhody patrí silná typovanosť a immutabilita premenných[10].

5.1.1 Moduly programu

Program v jazyku Haskell je na najvyššej úrovni tvorený množinou modulov. Na obrázku 5.1 je znázornený diagram modulov a ich závislostí. Hlavný program je tvorený modulom **Main** v súbore **schedule.hs**, ktorý využíva ostatné moduly. Matematické štruktúry popísané v kapitole 2 sú v programe reprezentované typmi implementovanými v module **Sche-**



Obr. 5.1: Diagram modulov a ich závislostí pre program na riešenie RCPSP s dynamickou rekonfiguráciou a údržbou.

duleTypes. Tieto typy sú využívané takmer všetkými ostatnými modulmi. Algoritmy na generovanie rozvrhu z AL sú implementované v module **Sgs**. Generátor náhodných AL a optimalizátor sú implementované v moduloch **Shuffle**, **GenomeGeneration** a **Genetic**. Ďalej budú popísané najdôležitejšie implementované typy a funkcie rozdelené podľa modulov.

5.1.2 Modul ScheduleTypes

Tento modul definuje základné typy využívané v celom programe. Najdôležitejšie z nich sú:

- **Job**: Typ nesúci informácie o operácii popísané v zadaní úlohy
- **Resource**: Popisuje parametre zdroja: Trvanie a periódu údržby a maticu trvania zmeny konfigurácie.
- **Problem**: Obsahuje celý popis úlohy s operáciami a zdrojmi.
- **Schedule**: Typ popisujúci rozvrh, ako je definovaný v teórii. Atribút *jobs* je reprezentovaný poľom všetkých operácií, vyjadrený ďalej popísaným typom **ScheduleJob**. Druhý atribút *rows* slúži na určenie, ktoré operácie sa v rozvrhu nachádzajú a pre účely zefektívnenia práce s rozvrhom. Obsahuje pole zoznamov obsahujúcich id operácie, organizované podľa zdroja a poradia operácií na zdroji.
- **ScheduleJob**: Operácia s priradeným časom začiatku. Pre zjednodušenie niektorých operácií je tu uložený aj čas trvania rekonfigurácie.
- **ActivityList**: Je reprezentovaný zoznamom s prvkami typu **Job**.

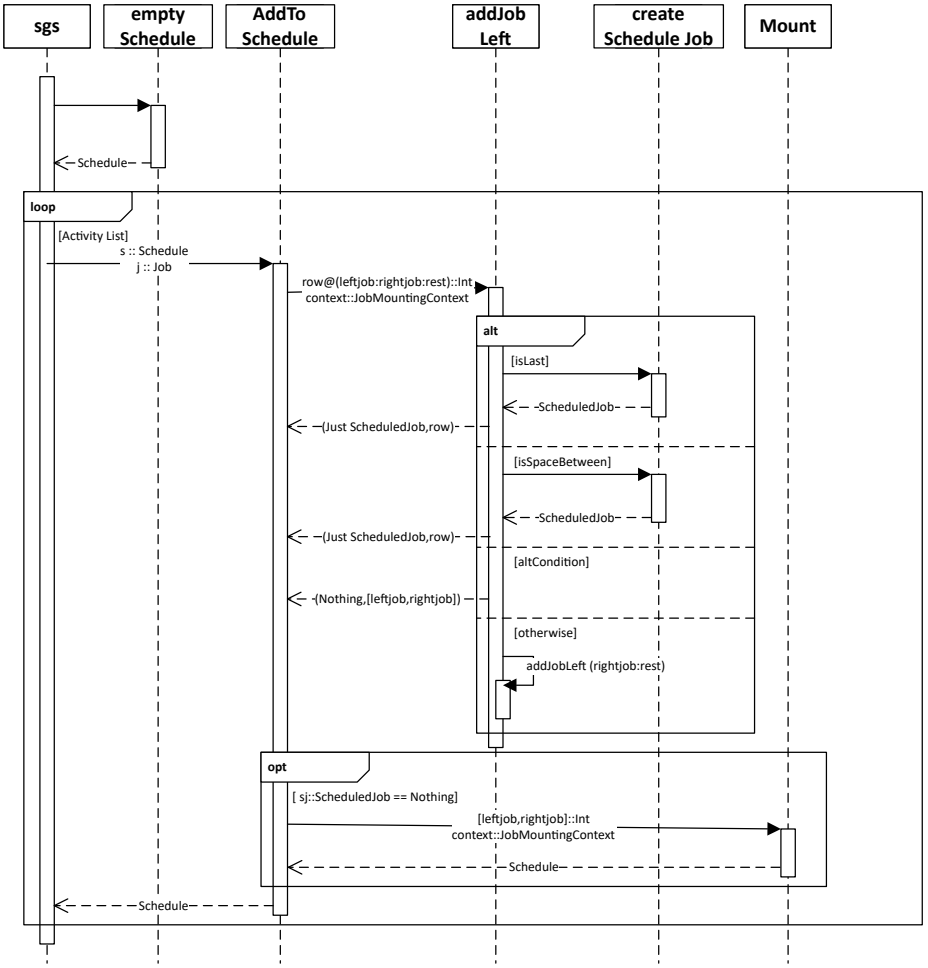
  Tento modul navyše obsahuje aj funkcie na zistenie základných vlastností rozvrhnutej operácie ako čas ukončenia so započítaním trvania rekonfigurácie a údržby.

5.1.3 Modul Sgs

Modul pozostáva z algoritmov SGS zobrazených v tabuľke v kapitole 4. Všetky tieto algoritmy majú spoločnú kostru. Funkcia slúžiaca ako SGS má 2 vstupné parametre: **Problem** a **ActivityList**. Výstupným parametrom je **Schedule**. Zovšeobecnené fungovanie implementácie SGS je zobrazené na sekvenčnom diagrame 5.2.

Popis sekvenčného diagramu

Na diagrame 5.2 vidíme, že na začiatku je vytvorený prázdny rozvrh a následne sú doňho pridávané operácie v poradí, v akom sa vyskytujú v AL. Pridanie operácie má za úlohu funkcia **addToSchedule**. Tá inicializuje **JobMountingContext**, objekt slúžiaci na zapuzdrenie informácií o spôsobe rozvrhovania operácie, a volá funkciu **addJobLeft** s parametrom riadku rozvrhu **ScheduleRow**, do ktorého pridávame operáciu. **addJobLeft** potom rekurzívne prechádza celý riadok a vyhľadáva voľné miesto. Ak je možné operáciu vložiť so základnými podmienkami, tak je vytvorená funkciou **createScheduleJob** a vrátená ako výsledok spolu s upraveným riadkom rozvrhu. V prípade, že bola vyvolaná *altCondition*, vráti za len miesto, kam chceme operáciu vložiť. Vloženie je vykonané funkciou mount, volanou z funkcie **addToSchedule**.



Obr. 5.2: Sekvenčný diagram implementácie SGS.

Popis kľúčových častí zdrojového kódu

Pri funkcionálnom programovaní zoznam nie je možné prechádzať cyklom pre nemožnosť modifikácie premenných. Namiesto toho využívame rekurziu. Ako abstrakcia slúži funkcia **fold**, ktorá rekurzívne aplikuje zadanú binárnu funkciu na prvky zoznamu. Veľká výhoda tohto prístupu oproti klasickým procedurálnym jazykom je, že objekt typu Schedule nie je možné ovplyvniť z vonkajšieho prostredia a pri správnom fungovaní funkcie **ScheduleJobLeft** je zaručená korektnosť vytváraného rozvrhu. Všeobecná implementácia sgs je teda nasledovná:

```
sgs :: Problem -> ActivityList -> Schedule
sgs  p (ActivityList al) =
foldl' (scheduleJobLeft p) (emptySchedule p) al
```

Špecifické chovanie je dosiahnuté parametrizovateľnosťou funkcie **ScheduleJobLeft**. Jej implementácia je nasledovná:

```
scheduleJobLeft ::
--typova signatura altcondition
([Int]-> JobMountingContext -> ScheduledJobs -> Bool)
--typova signatura altMount
-> (Schedule-> [Int] -> Job -> Schedule)
-> Bool -> Problem -> Schedule -> Job -> Schedule
scheduleJobLeft altConditon altMount  useMaintenance
(Problem _ (Resources res)) s j  =
addToSchedule s altAction
(addJobLeft ((s_rows s) ! (j_res j))
--inicializacia JMC
(JobMountingContext {
jmc_job=j,
jmc_schedule=s,
jmc_soonestStart = (latestPred s (j_pred j) ),
jmc_resource = (res ! (j_res j)),
jmc_altCondition=altConditon,
jmc_useMaintenance=useMaintenance })
(s_jobs s) 0) j
```

Prvé 3 parametre slúžia na ovplyvnenie chovania všeobecnej rozvrhovacej funkcie spôsobom popísaným v TOD.

- **altcondition**: Predikát alternatívneho rozvrhnutia operácie, ktorý má na vstupe indexy operácií, medzi ktorými zisťujeme dostatok voľného miesta, **JobMountingContext** a pole vložených operácií **ScheduleJob**.
- **altMount**: Funkcia vkladajúca operáciu do rozvrhu po splnení altCondition.
- **useMaintenance**: Hodnota typu Bool indikujúca, či má výsledný rozvrh spĺňať podmienku na údržbu.

Vo funkcii je inicializovaná štruktúra **JobMountinContext**, ktorá obsahuje atribúty pre informovanie **addJobLeft** o spôsobe rozvrhovania operácie. Atribút **jmc_soonestStart** určuje minimálny začiatok podľa predchádzajúcich operácií.

SgsPostProcess funguje v dvoch fázach, kedy je prvý vytvorený rozvrh konvertovaný na AL a ten následne rozvrhnutý. Pre tento prípad môžeme využiť matematický operátor "po"na naväzovanie funkcií:

```
sgsPostProcess :: Problem -> ActivityList -> Schedule
sgsPostProcess p = (sgsBasic p) . scheduleToAL . (sgsNoConf p)
```

5.1.4 Modul Genetic

Chovanie genetického algoritmu je parametrizovateľné cez inicializáciu dátovej štruktúry **GeneticParameters**, ktorá obsahuje všetky položky popísané v sekcii 4.2.2. Na vykonanie GA slúži funkcia **genetic**, ktorá iteratívne volá funkciu **geneticStep** nad populáciou s náhodným generátorom:

```
genetic param = (foldr (.) (geneticInit param)
(replicate (gp_generationCount param) (geneticStep param)))
```

Populácia je reprezentovaná zoznamom **ActivityList**ov a ich ohodnotení. Funkcia **genetic** vracia finálnu vygenerovanú populáciu.

5.1.5 Modul ScheduleToSql

Tento modul obsahuje funkciu **scheduleToSql** so vstupnými parametrami typu **Problem** a **Schedule**, ktorá generuje SQL skript do informačného systému Key-Soft ERP ISIT.

Využívaná štruktúra je zobrazená v ER diagrame na obrázku 5.3. Využité sú nasledujúce tabuľky

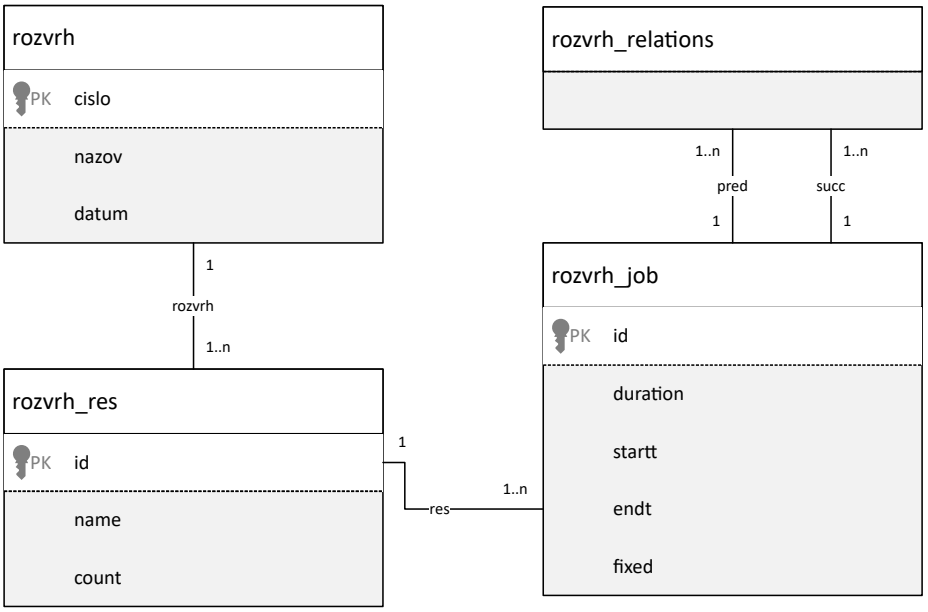
- **rozvrh**: Obsahuje jeden riadok pre každý vygenerovaný rozvrh.
- **rozvrh_res**: Zdroje definované v popise úlohy. Pre zobrazenie údržby vykonávanej počas trvania operácie, pre každý zdroj vytvoríme pomocný zdroj, na ktorom zobrazíme vykonávané údržby.
- **rozvrh_job**: Operácie s priradeným časom začiatku. Každá rekonfigurácia je tiež uložená ako operácia. Atribút **fixed** je využitý na rozlíšenie operácie od rekonfigurácie.
- **rozvrh_relations**: Pomocná tabuľka na zobrazenie precedenčných vzťahov zo zadania úlohy.

5.1.6 Modul ProblemLoad

Obsahuje funkciu **LoadRCPC** ktorá, vstupný reťazec vo formáte RCPC prevedie na **Problem**.

5.1.7 Modul Experiments

Experimentovanie prebieha nad každým RCPC súborom zo zvoleného priečinku. Výsledok experimentu nad každým súborom je uložený do dátovej štruktúry odvodenej od triedy **Exper**:



Obr. 5.3: ER diagram využitej časti databázy ERP ISIT

```
class Exper a where
exAggr :: [a] -> Int -> String
exHead :: a -> Int -> String
```

Nad každou sadou výsledkov takto môžeme definovať viacero agregačných funkcií (**exAggr**) a popisnú hlavičku (**exHead**), ktoré vracajú reťazec reprezentujúci agregovaný výsledok experimentu v podobe napríklad latexovej tabuľky.

5.1.8 Testovanie programu

Testovanie správneho fungovania programu sa zameriava na overenie platnosti rozvrhu podľa kritérií z kapitoly 2. Pre tieto účely je využitá vstavaná funkcia **trace**, ktorá má dva vstupné parametre: funkciu, ktorá je vykonaná a reťazec vypísaný na štandardný chybový výstup. **trace** má teda vedľajší efekt a nie je vhodné ju využívať pri tvorbe programu. Na testovanie a ladenie je však vhodná.

Funkcie na overenie podmienok platnosti rozvrhu majú vstupný parameter **Schedule**, ktorý vrátia ako návratovú hodnotu. Po porušení podmienky vypíšu chybovú hlášku na štandardný chybový výstup. Funkcie na overenie jednotlivých obmedzení sú nasledovné:

- **checkPrecedence**: Overuje precedenčné obmedzenia.
- **checkOverlap**: Overuje obmedzenia kapacity na zdrojoch.
- **checkConfiguration**: Overuje obmedzenia na rekonfiguráciu.
- **checkMaintAndConf**: Overuje obmedzenia na údržbu spolu s rekonfiguráciou.

Na overenie všetkých podmienok môžeme využiť funkciu **checkAll**:

```
checkAll p = (checkMaintAndConf p) . (checkConfiguration p) .
checkOverlap . checkPrecedence
```

5.1.9 Kompilácia a použitie programu

Pre preklad programu bol využitý prekladač Glasgow Haskell Compiler (ghc). Program je multiplatformný a obmedzený len na možnosti ghc. Vývoj, testy a experimenty boli vykonané na OS Microstoft Windows 10 s využitím prostredia **cygwin** na emuláciu linuxového terminálu. Využitá verzia prekladača je GHC 8.0.2.

Preklad

Na vytvorenie spustiteľného súboru je možné využiť príkaz make v prostredí cygwin alebo na unixovom termináli.

```
$ make
```

Výrazne výkonnejšiu verziu je možné vytvoriť s povolením optimalizácií

```
$ make opt
```

Použitie programu

Spustenie experimentov nad priečinkom:

```
$ ./schedule -e1 folder
$ ./schedule -e2 folder
```

Vytvorenie rozvrhu z náhodného AL s využitím zvoleného sgs a jeho vypísanie na štandardný výstup v podobe sql skriptu:

```
$ ./schedule -r inputfile.rcpc [B|P|PM|S|SR|SM|SMR]
```

Vytvorenie rozvrhu s využitím GA a jeho vypísanie na štandardný výstup v podobe sql skriptu. Je tiež potrebné nastaviť parametre GA:

```
$ ./schedule -g inputfile.rcpc [B|P|PM|S|SR|SM|SMR] generationCount topSize
crossSize randBotSize newSize topCross
```

5.2 Program na generovanie problémov RCPSP

Druhý program vytvorený v rámci diplomovej práce, slúži na generovanie súborov s inštanciami úloh problému RCPSP s rekonfiguráciami a údržbou. Generátor sa skladá z dvoch častí: generátoru zdrojov a generátoru operácií.

5.2.1 Trvanie operácie a rekonfigurácií

Generátor podporuje náhodné nastavenie trvania, ktoré je určené náhodným generátorom. Pre tento účel bola vytvorená trieda pre reprezentáciu pravdepodobnostného rozloženia:

```
convertR :: a -> Int -> Int
nextr :: a -> StdGen -> (Int,StdGen)
-- funkcia vracajuca nahodne cislo podla rozlozenia
nextr dist gen = ((convertR dist n),gen2)
where (n,gen2)=next gen
```

Každé rozloženie musí implementovať funkciu **convertR**, ktorá reprezentuje distribučnú funkciu.

V programe je vytvorená inštancia tejto triedy v podobe uniformného rozloženia.

5.2.2 Generovanie zdrojov

Pre generovanie zdrojov slúži metóda makeResources s nasledovnou typovou anotáciou:

```
makeResources :: MyRan a => [(GenRes,a)] -> StdGen ->
([String],StdGen)
```

Vstupom je generátor náhodných čísel a zoznam obsahujúci prvky s parametrami generovania jednotlivého zdroja.

5.2.3 Generovanie operácií

Pre generovanie operácií boli vytvorené funkcie generujúce jednotlivé typy zákaziek. Pre vytvorenie zákazky S2 definovanej v kapitole 6 je využitá funkcia s nasledovnou anotáciou:

```
addTypS2 :: (MyRan a) => Int -> [GenRes] -> (Int,a) -> (Int,a) ->
StdGen -> [Job] -> ([Job],StdGen)
```

Funkcia **addTypS2** pridá do zoznamu operácií typu **Job** zvolený počet zákaziek typu S2. Kľúčové sú parametre typu **(Int,a)**, ktoré určujú zdroj, na ktorý bude jednotlivá operácia pridaná a pravdepodobnostné rozloženie trvania.

5.2.4 Kompilácia a použitie generátora

Program je možné skompilovať príkazom make. Po spustení sa RCPC súbory vygenerujú do priečinku gen.

```
$ make
$ ./generate
```

Kapitola 6

Experimenty

Pre overenie skutočných vlastností navrhnutých a implementovaných algoritmov je nevyhnutné vykonať sadu experimentov. Experimenty sú zamerané na zhodnotenie skutočnej časovej zložitosti algoritmov a posúdenie objektívnych kritérií vytvoreného rozvrhu ako makespan a počet rekonfigurácií.

Na začiatku kapitoly v sekcii 6.1 je popísaná typológia inštancií úloh na experimentovanie a spôsob ich generovania. Následne je v sekcii sekcii 6.2 uvedená metodika experimentov, kde je popísané aké hlavné parametre sú sledované v experimentoch. V sekcii 6.3 sú uvedené komentované výsledky v tabuľkách. Výsledkom experimentov je jednoznačné určenie najvýhodnejšieho rozvrhovacieho algoritmu na daný typ úlohy popísané v sekcii 6.4. Podľa týchto záverov je v prípadovej štúdii v sekcii 6.5 navrhnuté riešenie pre výrobné podniky.

6.1 Testovacie dáta

Pre overenie výsledkov bolo potrebné zostrojiť vlastnú sadu testovacích inštancií úlohy, pretože neexistuje štandardná knižnica pre problém RCPSP s dynamickou údržbou a rekonfiguráciou. Jednotlivé problémy boli vytvorené automatickým generátorom.

Na experimentálne overenie programov, ktoré riešia RCPSP bez údržby a rekonfigurácie sa ako testovacia databáza využíva PSPLIB[9] [2] [12], ktorá slúžila ako inšpirácia pri tvorbe príkladov. Problémy však nebolo možné prebrať, pre výraznú odlišnosť s požiadavkami pre výrobu.

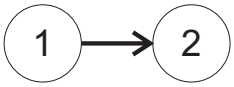
6.1.1 Typológia výrobných príkazov

Každá úloha sa skladá z určitého počtu zákaziek, nazvaných tiež ako výrobný príkaz. V testoch sú využívané rôzne typy výrobných príkazov, ktoré sa delia na niekoľko kategórií.

Jednoduché

Pri tomto type je vygenerovaný výrobný príkaz skladajúci sa z jednej operácie. Generátor operácie má nasledujúce vstupné parametre:

- **Distribučná funkcia trvania.** Podľa nej sa náhodne pridelí čas operácie. Vo všetkých prípadoch je využívané uniformné rozloženie.
- **Zdroj, ktorý operácia obsadzuje.**



Obr. 6.1: Precedenčný graf zákazky typu S2.

Počet možných konfigurácií operácie a čas potrebný na rekonfiguráciu určuje pridelený zdroj. Pri generovaní je náhodne uniformne pridelená konfigurácia.

Z dvoch operácií

Výrobný príkaz je vytvorený z dvoch precedenčne závislých operácií. Charakteristika tohto typu zákaziek môže byť výrazne rozdielna vzhľadom na to, na ktorom z dvoch strojov je vykonávaná údržba.

Zo sekvencie operácií

Zovšeobecnenie zákazky na N po sebe nadväzujúcich operácií. V tejto práci je využitá maximálne sekvencia troch operácií.

Zložené výrobné príkazy

V tomto type výrobného príkazu sa vyskytuje operácia montáže. Sekvencie operácií tak môžu tvoriť stromy. Pre účely testovania využijeme prípad len s jednou montážou.

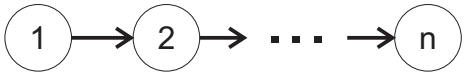
6.1.2 Balíky výrobných príkazov

Všetky operácie v úlohe pozostávajú z určitého množstva výrobných príkazov. Pre účely testovania majú všetky úlohy nasledovnú charakteristiku: celkový počet zákaziek pre každú úlohu je zvolený tak, aby mala výsledná úloha približne 200 operácií. Priemerná dĺžka každej operácie je určená tak, aby boli všetky zdroje približne rovnako vyťažené.

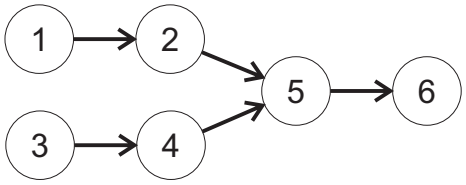
Homogénne úlohy

Homogénne úlohy sa skladajú len z jedného typu výrobných príkazov. Pre označenie, z akých výrobných príkazov sa úloha skladá zavedieme nasledujúce pomenovania:

- **S1***(S-serial)*: označuje úlohu skladajúcu sa z výrobných príkazov s jednou operáciou s údržbou a rekonfiguráciou.
- **S2**: označuje úlohu skladajúcu sa z výrobných príkazov z dvoch operácií s údržbou a rekonfiguráciou. Precedenčný graf je znázornený na obrázku 6.1.
- **S2a**: ako S2, ale pri prvej operácii nie je rekonfigurácia a údržba.
- **S2b**: ako S2, ale pri druhej operácii nie je rekonfigurácia a údržba.
- **Sn**: príkazy so sekvenciou n operácií. Precedenčný graf je znázornený na obrázku 6.2.
- **A6***(A-assembly)*: príkazy s montážou skladajúce sa zo šiestich operácií *(precedencia: 1p2p5,3p4p5,5p6)*. Precedenčný graf je znázornený na obrázku 6.3.



Obr. 6.2: Precedenčný graf zákazky typu Sn.



Obr. 6.3: Precedenčný graf zákazky typu A6.

Ak úloha neobsahuje zdroje s údržbou, použijeme označenie **nm** *(no maintenance)*. Pre experimentovanie je vygenerovaných viacero úloh s rovnakou charakteristikou. Tie rozlíšime pridaním suffixu s pomlčkou a poradovým číslom.

Príklad: Súbor s názvom **S3nm-14.rcpc** označuje úlohu s poradovým číslom 14 skladajúcu sa z výrobných príkazov z dvoch operácií s údržbou a rekonfiguráciou. V úlohe nie je využitá údržba zdrojov.

Heterogénne

Heterogénne úlohy sa skladajú z rôznych typov výrobných príkazov. Pri tvorbe je snaha o rovnomerné zastúpenie operácií zo všetkých obsiahnutých typov.

Príklad: Súbor s názvom **S1S2aA6-2.rcpc** označuje úlohu s poradovým číslom 2 skladajúcu sa z výrobných príkazov **S1**, **S2a** a **A6**.

6.2 Metodika experimentov

Prvá časť experimentov slúži na porovnanie vlastností samotných SGS algoritmov nad náhodnými AL. Druhá časť overuje správanie týchto algoritmov pri využití GA.

6.2.1 Charakteristika vygenerovaného rozvrhu

Tento experiment bol vykonaný nasledovne: pre každú inštanciu úlohy bolo náhodne vygenerovaných N_{iter} AL. Každý z týchto AL bol prevedený na príslušný rozvrh všetkými implementovanými algoritmami popísanými v kapitolách 2 a 3 . Vypočítané rozvrhy sú následne štatisticky zhodnotené a je vykonané odporučenie vhodného rozvrhovacieho algoritmu podľa typu úlohy.

Skutočná zložitosť

Prvou sledovanou veličinou je skutočná výpočtová náročnosť na inštanciách z reálnej výroby. Porovnávacou jednotkou je počet rozvrhnutí operácie potrebný pre výpočet rozvrhu. Toto kritérium je zvolené, pretože všetky algoritmy využívajú podobnú operáciu pre rozvrhnutie jednotlivej operácie.

V kapitole 3 bola teoreticky analyzovaná zložitosť algoritmov. V prípade základného algoritmu *sgsBasic* je každá operácia rozvrhovaná práve jeden krát. Pre algoritmus *sgsPostProcess* je každá operácia rozvrhnutá jeden krát pri prvom prechode a druhý krát

pri post-processingu. Pre *sgsShiftJobs* a odvodené algoritmy je teoretická horná hranica pomerne vysoká, avšak počet potrebných rozvrhnutí môže byť výrazne rozdielny. Dátová štruktúra reprezentujúca rozvrh obsahuje preto atribút navyše, reprezentujúci počet rozvrhnutí operácie. Tento atribút je ďalej štatisticky zhodnotený.

Hodnotiace kritériá

Ako hlavná hodnotiaca funkcia je zvolený makespan. Ďalšie kritérium určujúce kvalitu rozvrhu je počet rekonfigurácií. Tieto kritéria sú zaznamenané pri každom vytvorenom rozvrhu a ďalej rôzne agregované pre účely posúdenia algoritmov.

Porovnanie dvoch algoritmov na základe zložitosti a objektívneho kritéria.

Pre určenie lepšieho algoritmu bola navrhnutá rozhodovacia funkcia. Predpokladáme známosť nasledujúcich hodnôt:

- Priemerný počet rozvrhnutí potrebný na vytvorenie rozvrhu pre každý algoritmus mnt_A a mnt_B
- Počet alebo podiel rozvrhov, pri ktorých bolo dosiahnuté lepšie hodnotenie s algoritmom A h_A a algoritmom B h_B

Rozhodovacia funkcia bola vytvorená na základe nasledovnej intuície: Predpokladáme optimalizátor v podobe prehľadávania náhodných riešení. Ak algoritmus B má R krát vyššiu zložitosť ako algoritmus A, musíme algoritmu A povoliť vytvoriť R krát viac rozvrhov ako algoritmu B. Ako ekvivalentné zvýhodnenie môžeme uvažovať že algoritmus A namiesto h_A lepších rozvrhov vytvoril $h_A R$ lepších rozvrhov. Ak je toto číslo vyššie ako h_B, algoritmus A je výhodnejší.

Vzorec pre pomer zložitosti:

$$R_{AB} = mnt_A / mnt_B \tag{6.1}$$

Výsledná rozhodovacia funkcia:

$$D = h_A - R_{AB} h_B \tag{6.2}$$

Ak je D kladné, zvolíme algoritmus A.

Výber najvhodnejšieho algoritmu podľa typu úlohy

Hlavným cieľom experimentovania je jednoznačne určiť najvhodnejší algoritmus generovania rozvrhu na daný typ úlohy. Ako kritérium vhodnosti je schopnosť algoritmu vytvoriť rozvrh s čo najnižším makespanom a počtom rekonfigurácií pri najnižšej zložitosti. Na základe prvých výsledkov experimentov bol určený postup výberu najvhodnejšieho algoritmu.

Nastavenie parametrov

Každý typ úlohy je zastúpený 20 inštanciami. Počet vygenerovaných AL nad každou úlohou $N_{iter} = 100$. Všetky úlohy sa skladajú z 200 operácií.

6.2.2 Overenie funkcie genetického algoritmu

Druhý test má za úlohu overiť správanie rozvrhovacích algoritmov s využitím GA. Návrh vhodného optimalizátoru pre každý algoritmus však presahuje rozsah tejto práce. Tento experiment bol vykonaný pre ilustráciu fungovania cieleného optimalizátoru s prezentovanými rozvrhovacími algoritmami.

Nastavenie parametrov

Pre genetický algoritmus boli nastavené nasledujúce parametre pre celý experiment: generationCount=30, TopSize=0, CrossSize=30, RandBotSize=0, NewSize=0, TopCross=15. Nad každou inštanciou bol algoritmus spustený 3 krát.

6.3 Výsledky experimentov

V tejto sekcii sú zobrazené dôležité výsledky experimentov vo forme tabuliek, ktoré vedú k jednoznačnému porovnaniu algoritmov na základe hodnotiacich kritérií a zložitosti. Pre vysoké množstvo rôznych tabuliek bolo zvolené umiestnenie popisu jednotlivých výsledkov do nedeliteľného textu pod tabuľkou.

6.3.1 Výsledky skúmajúce charakteristika vygenerovaného rozvrhu

	S	P&PM	S		SB		SS		SBS	
úloha	mnt	mnt	mnt_{avg}	mnt_{max}	mnt_{avg}	mnt_{max}	mnt_{avg}	mnt_{max}	mnt_{avg}	mnt_{max}
S1	202	404	202.0	202.0	202.0	202.0	1816.58	2133.6	1816.675	2134.3
S2	202	404	1127.45	1402.8	1112.63	1383.25	1864.02	2308.25	1861.26	2298.45
S2a	202	404	460.68	629.3	468.1	635.4	534.08	706.55	499.44	665.5
S2b	202	404	208.08	246.0	208.08	246.0	760.66	1012.45	760.75	1012.45
S3	200	400	1207.52	1557.45	1192.04	1522.4	1519.5	1899.95	1556.87	1942.45
A2	200	400	1027.17	1403.0	1021.28	1393.55	1365.91	1857.95	1379.51	1872.1
S1S2	202	400	280.66	381.5	279.53	375.5	992.62	1342.2	920.44	1224.75
S1S2aS2b	202	404	202.05	202.55	202.05	202.55	202.08	202.75	202.084	202.75
S1S2S3A2	202	404	611.19	853.45	611.54	854.4	1015.80	1339.55	992.09	1324.45
S3A2	203	406	1490.50	2084.5	1471.83	2025.5	1888.87	2466.45	1946.07	2534.1
S2nm	202	404	355.24	468.65	353.04	459.8	202.0	202.0	202.0	202.0
S3nm	200	400	400.18	527.1	400.01	521.65	200.0	200.0	200.0	200.0
A2nm	200	400	324.49	475.25	323.12	479.9	200.0	200.0	200.0	200.0
S1S2aS2bnm	202	404	202.06	202.55	202.06	202.55	202.0	202.0	202.0	202.0
S1S2S3A2nm	202	404	303.28	395.1	303.42	397.85	202.0	202.0	202.0	202.0

Tabuľka 6.1: **Počet rozvrhnutí operácie pre vytvorenie rozvrhu z AL pre rôzne algoritmy. (mnt_{avg} – priemerný počet rozvrhnutí, mnt_{max} – maximálny počet rozvrhnutí,)** Maximálny počet rozvrhnutí je pri všetký algoritmoch a úlohách vyšší o menej ako 50 percent oproti priemernému, čiže počet rozvrhnutí pre rôzne AL je stabilný bez väčších extrémov, takže ďalej môžeme uvažovať priemernú zložitosť ako smerodatný údaj. Algoritmy S a SR resp. SM a SMR majú vo všetkých prípadoch takmer rovnakú zložitosť. Dostávame tak 4 triedy algoritmov s rovnakou zložitosťou: B, P+PM, S+SR, SM+SMR. Pri triede úloh bez údržby vedú algoritmy B PM SM SRM na rovnaký výsledok, takže B,SM a SRM majú aj rovnakú zložitosť. Pri tejto triede nemá teda zmysel uvažovať PM SM a SRM ako vhodné algoritmy.

	B			P			S		
úloha	m_{min}	m_{avg}	m_{max}	m_{min}	m_{avg}	m_{max}	m_{min}	m_{avg}	m_{max}
S3nm	1531.65	1586.34	1633.0	**1492.6**	**1543.13**	**1590.45**	1514.7	1559.85	1606.4
S3A2	1986.25	2108.29	2240.55	1774.15	1868.38	1991.1	**1721.2**	**1774.15**	**1841.55**
S3	2117.9	2276.02	2462.45	1771.25	1887.15	2016.7	**1690.35**	**1737.48**	**1789.35**
S2nm	2122.1	2177.645	2242.15	**2109.6**	**2160.31**	**2223.45**	2124.9	2178.23	2239.25
S2b	2355.2	2399.805	2444.5	**2353.7**	**2399.12**	**2443.65**	2356.45	2400.57	2444.65
S2a	2398.7	2500.79	2623.55	2252.05	2336.45	2444.45	**2211.25**	**2266.96**	**2326.2**
S2	2773.8	2929.005	3121.4	2480.75	2613.345	2773.95	**2406.9**	**2464.125**	**2522.0**
S1S2S3A2nm	1676.05	1753.30	1831.9	**1672.7**	**1750.81**	**1827.15**	1675.5	1752.19	1827.25
S1S2S3A2	1932.9	2050.875	2235.3	1906.15	1974.73	2049.6	**1905.1**	**1971.89**	**2040.6**
S1S2aS2bnm	**1846.35**	**1905.27**	**1962.6**	1847.05	1905.88	1964.4	1847.05	1905.88	1964.4
S1S2aS2b	**2120.55**	2177.48	2240.6	**2120.55**	**2176.86**	2240.15	**2120.55**	**2176.86**	**2239.9**
S1S2	2382.45	2482.39	2647.7	2355.2	2415.03	2501.7	**2354.85**	**2404.10**	**2458.6**
S1	**4852.85**	**4918.88**	**4977.85**	**4852.85**	**4918.88**	**4977.85**	**4852.85**	**4918.88**	**4977.85**
A2nm	1610.45	1683.09	1765.9	**1599.55**	**1668.74**	**1748.65**	1600.45	1671.60	1751.9
A2	2037.3	2169.7	2313.5	1815.25	1920.5	2037.7	**1745.4**	**1828.56**	**1909.55**

Tabuľka 6.2: **Porovnanie makespanu u rôznych SGS. (m_{min} – minimálny makespan, m_{avg} – priemerný makespan, m_{max} – maximálny makespan)** Tučne je zvýraznený najnižší minimálny, priemerný a maximálny makespan pre každú kategóriu. Môžeme si všimnúť, že ak je pre daný algoritmus najnižší priemerný makespan, potom je najnižší aj minimálny a maximálny makespan. Na základe tohto zistenia, môžeme brať do úvahy priemernú hodnotu ako smerodatnú pre ohodnotenie algoritmu.

Postup na výber najvhodnejšieho algoritmu pre daný typ úlohy

Tento postup bol zvolený na základe zistení z tabuliek 6.1 a 6.2.

V prípade že nám nezáleží na zložitosti zistíme algoritmus s najlepšou priemernou hodnotou hodnotiaceho kritéria (makespanu alebo počtu rekonfigurácií) a zvolíme ho ako najvýhodnejší algoritmus. Vhodnejšie je však vyhodnotiť algoritmus aj na základe zložitosti. Postup je nasledovný:

1. Zistíme algoritmus s najlepšou priemernou hodnotou hodnotiaceho kritéria. Ak je tento algoritmus z množiny {B,P,PM}, zvolíme ho ako najvýhodnejší a ukončíme ďalšie skúmanie, pretože dvojnásobný nárast počtu rozvrhnutí je prijateľný a nemenný s rôznym počtom operácií. Tento bod je vykonaný podľa tabuliek 6.3 a 6.4.

2. Zvolíme najlepší algoritmus podľa hodnotiaceho kritéria za každú zo skupín {S,SR} a {SM,SRM}. Tieto algoritmy následne porovnáme podľa kritéria 6.2 a zvolíme najvýhodnejší algoritmus z tejto kategórie. Porovnanie je vykonané v tabuľke 6.8.

3. Algoritmus rozvrhovania zvolený v poslednom kroku porovnáme s najlepším zo skupiny {B,P,PM} a určíme odporučený algoritmus. Toto porovnanie je vykonané v tabuľkách 6.6 a 6.7.

typ úlohy	B	P	PM	SR	S	SMR	SM
S1	4918.88	4918.88	4816.53	4918.88	4918.88	**4816.15**	4816.19
S2	2929	2613.34	**2387.77**	2464.12	2470.26	2391.44	2390.43
S2a	2500.79	2336.45	**2168.61**	2266.96	2267.7	2202.09	2232.58
S2b	2399.80	2399.12	2348.87	2400.57	2400.57	2348.31	**2348.26**
S3	2276.02	1887.15	1710.515	1737.48	1745.15	1714.28	**1702.92**
A2	2169.7	1920.5	1781.87	1828.56	1833.66	1779.83	**1776.32**
S1S2	2482.39	2415.03	2361.76	2404.1	2404.06	**2353.83**	2355.62
S1S2aS2b	2177.48	2176.86	2176.47	2176.86	2176.86	**2176.4**	**2176.4**
S1S2S3A2	2050.87	1974.73	1929.59	1971.89	1972.12	1928.74	**1928.30**
S3A2	2108.29	1868.38	1734.30	1774.15	1776.95	1728.02	**1721.35**
S2nm	2177.64	**2160.31**	2176.95	2178.23	2177.36	2177.64	2177.64
S3nm	1586.34	**1543.13**	1586.12	1559.85	1560.845	1586.34	1586.34
A2nm	1683.09	**1668.74**	1682.31	1671.6	1671.67	1683.09	1683.09
S1S2S3A2nm	1753.3	**1750.81**	1751.68	1752.19	1752.19	1753.3	1753.3
S1S2aS2bnm	**1905.27**	1905.88	**1905.27**	1905.88	1905.88	**1905.27**	**1905.27**

Tabuľka 6.3: **Priemerný makespan podľa typu úlohy.** Tučne sú označené najlepšie dosiahnuté makespany pri danom type úlohy. Bez zohľadnenia zložitosti tak touto tabuľkou môžeme určiť najlepší algoritmus vzhľadom na makespan. Pri úlohách bez údržby nemusíme zložitosť ďalej skúmať, pretože víťazné algoritmy sú P alebo B. Pri úlohách s údržbou je potrebné v prípade víťazstva SM a SRM ďalej vyhodnotiť pomer zložitosti a schopnosti vytvoriť lepší rozvrh.

	SR vs. S					SRM vs. SM				
úloha	N_{same}	m(S)<	c_c(S) <	m(SB)<	c_c(SB) <	N_{same}	m(SC)<	c_c(SC) <	m(SBC)<	c_c(SBC) <
S1	100%	0%	0%	0%	0%	95%	0%	0%	0%	0%
S2	7%	33%	24%	10%	31%	0%	26%	100%	37%	0%
S2a	32%	28%	17%	29%	24%	3%	77%	84%	12%	3%
S2b	100%	0%	0%	0%	0%	94%	1%	1%	1%	1%
S3	6%	45%	29%	26%	50%	0%	33%	98%	63%	1%
A2	10%	33%	18%	14%	43%	0%	29%	82%	36%	6%
S1S2	80%	1%	3%	1%	5%	0%	14%	85%	6%	6%
S1S2aS2b	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
S1S2S3A2	50%	3%	15%	1%	21%	0%	4%	86%	5%	8%
S3A2	6%	38%	29%	28%	43%	0%	29%	92%	51%	5%
S2nm	76%	2%	5%	5%	9%	100%	0%	0%	0%	0%
S3nm	70%	11%	10%	4%	13%	100%	0%	0%	0%	0%
A2nm	84%	3%	0%	2%	9%	100%	0%	0%	0%	0%
S1S2aS2bnm	100%	0%	0%	0%	0%	100%	0%	0%	0%	0%
S1S2S3A2nm	87%	0%	3%	0%	7%	100%	0%	0%	0%	0%

Tabuľka 6.8: **Porovnanie využitia *shiftJobsRecursive* oproti *shiftJobs* pri rozvrhovaní s posunutím operácií. (N_{same} – počet totožných rozvrhov, m(X)< – podiel rozvrhov s lepším makespanom pre algoritmus X, c_c(X)< – podiel rozvrhov s nižším počtom rekonfigurácií pre algoritmus X)** Pri tomto porovnaní nie je možné povedať, že by bol niektorý prístup výrazne lepší. Pri mnohých úlohách však vznikajú takmer všetky rozvrhy rozdielne, takže má zmysel uvažovať, ktorý je pre daný typ výroby vhodnejší. Pri výbere vhodnejšieho algoritmu spomedzi porovnávaných je ale lepšie využiť priemerný makespan, pretože majú podobnú zložitosť, čo sme ukázali na tabuľke 6.1

typ úlohy	B	P	PM	SR	S	SMR	SM
S1	160.03	160.03	146.36	160.03	160.03	**146.21**	*146.21*
S2	128.45	131.59	**114.71**	133.51	133.40	115.36	124.62
S2a	49.28	50.66	**42.48**	51.71	51.82	44.23	49.19
S2b	79.81	79.91	73.56	79.97	79.97	**73.44**	**73.44**
S3	132.36	135.75	**120.51**	138.25	137.77	121.41	131.03
A2	108.96	111.03	**102.24**	111.94	111.39	102.90	106.97
S1S2	127.40	130.34	117.45	130.97	131.03	**115.25**	120.63
S1S2aS2b	49.77	49.76	**49.74**	49.76	49.76	**49.74**	**49.74**
S1S2S3A2	130.05	133.11	123.08	133.77	133.63	**122.63**	127.11
S3A2	122.16	125.12	**112.34**	126.34	125.55	113.93	120.43
S2nm	**121.19**	132.58	**121.19**	132.94	132.86	**121.19**	**121.19**
S3nm	**126.68**	138.22	**126.68**	138.65	138.55	**126.68**	**126.68**
A2nm	**106.34**	112.05	**106.34**	111.96	111.73	**106.34**	**106.34**
S1S2aS2bnm	**50.14**	50.23	**50.14**	50.23	50.23	**50.14**	**50.14**
S1S2S3A2nm	**126.84**	132.95	**126.84**	132.92	132.78	**126.84**	**126.84**

Tabuľka 6.4: **Priemerný počet rekonfigurácií podľa typu úlohy.** Tučne sú označené najlepšie dosiahnuté makespany pri danom type úlohy. Bez zohľadnenia zložitosti tak touto tabuľkou môžeme určiť najlepší algoritmus vzhľadom na počet rekonfigurácií. Pri úlohách bez údržby je vždy najvýhodnejší algoritmus B, ktorý má najnižšiu možnú zložitosť, takže výber je jasný. Pri úlohách s údržbou je tiež potrebné vykonať porovnanie ako v prípade makespanu.

	P vs. PM					S vs. SM				
úloha	N_{same}	m(P)<	c_c(P) <	m(PM)<	c_c(PM) <	N_{same}	m(S)<	c_c(S) <	m(SM)<	c_c(SM) <
S1	0%	0%	0%	100%	100%	0%	0%	0%	100%	100%
S2	0%	0%	1%	100%	97%	0%	1%	0%	99%	100%
S2a	0%	0%	0%	99%	95%	0%	4%	1%	95%	94%
S2b	0%	0%	0%	100%	100%	0%	0%	0%	100%	100%
S3	0%	1%	2%	98%	97%	0%	27%	0%	70%	100%
A2	0%	0%	6%	99%	92%	0%	8%	3%	91%	96%
S1S2	0%	3%	1%	94%	97%	0%	0%	0%	98%	100%
S1S2aS2b	99%	0%	0%	1%	0%	99%	0%	0%	1%	0%
S1S2S3A2	0%	1%	4%	98%	95%	0%	1%	1%	98%	98%
S3A2	0%	0%	2%	99%	96%	0%	13%	1%	87%	98%

Tabuľka 6.5: Porovnanie alternatívnej podmienky *spaceWithoutConfAltCondition* a *spaceWithoutMAltCondition*. (N_{same} – počet totožných rozvrhov, m(X)< – podiel rozvrhov s lepším makespanom pre algoritmus X, c_c(X)< – podiel rozvrhov s nižším počtom rekonfigurácií pre algoritmus X) V tabuľke vidíme jednoznačné víťazstvo PM oproti P a SM oproti S v úlohách s údržbou. P a S teda určite nebudú patriť medzi odporúčané algoritmy pri type úloh s údržbou. Jedinú výnimku tvorí typ s1s2as2b, kde však vznikajú rovnaké rozvrhy. Využitie P alebo S by ale aj tak neprinieslo žiadne výhody- Úlohy bez údržby v tabuľke nie sú, pretože SM a PM sa v tom prípade správajú rovnako ako B.
0

	PM vs. SM					
úloha	N_{same}	m(PM)<	c_c(PM) <	m(SM)<	c_c(SM) <	R_{PMSM}
S1	0%	**8%**	**8%**	11%	14%	4.49
S2	0%	**43%**	**99%**	37%	0%	4.6
S2a	0%	**97%**	**88%**	2%	2%	1.23
S2b	0%	**24%**	8%	19%	**18%**	1.88
S3	0%	**45%**	**96%**	53%	2%	3.89
A2	0%	**37%**	**80%**	53%	15%	3.44
S1S2	0%	**23%**	**72%**	44%	17%	2.3
S1S2aS2b	97%	0%	0%	0%	0%	0.5
S1S2S3A2	0%	**23%**	**75%**	31%	18%	2.45
S3A2	0%	**29%**	**92%**	65%	5%	4.79

Tabuľka 6.6: **Porovnanie PM a SM (N_{same} – počet totožných rozvrhov, m(X)< – podiel rozvrhov s lepším makespanom pre algoritmus X, c_c(X)< – podiel rozvrhov s nižším počtom rekonfigurácií pre algoritmus X, R – pomer zložitosti určený podľa 6.1.)** Vidíme, že rozvrhy sa takmer vždy líšia, takže výber algoritmu bude mať zásadný vplyv na generovanie rozvrhu. Tučne je označený výhodnejší algoritmus podľa daného kritéria, pričom berieme do úvahy pomer zložitosti R a lepší algoritmus je vybratý podľa vzorca 6.2.

	PM vs. SMR					
úloha	N_{same}	m(PM)<	c_c(PM) <	m(SMR)<	c_c(SMR) <	R_{PMSMR}
S1	0%	**4%**	**5%**	11%	14%	4.49
S2	0%	48%	48%	36%	38%	4.6
S2a	0%	88%	59%	11%	21%	1.23
S2b	0%	26%	10%	20%	18%	1.88
S3	0%	51%	52%	44%	35%	3.89
A2	0%	49%	52%	41%	38%	3.44
S1S2	0%	15%	25%	**44%**	**58%**	2.3
S1S2aS2b	99%	0%	0%	0%	0%	0.5
S1S2S3A2	0%	**28%**	**43%**	31%	51%	2.45
S3A2	0%	**43%**	**55%**	50%	35%	4.79

Tabuľka 6.7: **Porovnanie PM a SMR (N_{same} – počet totožných rozvrhov, m(X)< – podiel rozvrhov s lepším makespanom pre algoritmus X, c_c(X)< – podiel rozvrhov s nižším počtom rekonfigurácií pre algoritmus X, R – pomer zložitosti určený podľa 6.1.)** Vidíme, že rozvrhy sa takmer vždy líšia, takže výber algoritmu bude mať zásadný vplyv na generovanie rozvrhu. Tučne je označený výhodnejší algoritmus podľa daného kritéria, pričom berieme do úvahy pomer zložitosti R a lepší algoritmus je vybratý podľa vzorca 6.2.

6.3.2 Experiment s využitím genetického algoritmu

Zhodnotenie experimentu je reprezentované tabuľkou 6.9 s makespanom najlepšieho rozvrhu vo finálnej populácii. Do zhodnotenia boli vybraté len najlepšie 4 algoritmy podľa predchádzajúceho experimentu. Sú dosahované výrazne lepšie výsledky ako pri rozvrhovaní bez cieleného optimalizátora zobrazené v tabuľke 6.2. Môžeme teda vyhodnotiť, že experiment dopadol úspešne.

	B		PM		SM		SMR	
úloha	m_{min}	m_{avg}	m_{min}	m_{avg}	m_{min}	m_{avg}	m_{min}	m_{avg}
S1	4559.5	4586.91	4475.0	4504.65	4478.15	4503.73	4487.5	4512.86
S2	2181.4	2196.34	2167.8	2188.03	2163.15	2178.03	2164.05	2174.99
S2a	1954.5	1971.95	1952.95	1968.65	1956.35	1974.41	1954.5	1970.16
S2b	2144.0	2162.28	2112.0	2130.16	2112.8	2131.51	2111.1	2126.58
S3	1494.65	1505.7	1496.15	1507.94	1497.55	1509.98	1495.3	1505.64
A2	1449.65	1462.75	1438.75	1453.96	1523.5	1552.25	1491.1	1520.85
S1S2	2174.2	2190.93	2146.3	2160.68	2132.8	2153.01	2146.1	2163.66
S1S2aS2b	1941.35	1959.45	1934.45	1958.05	1941.2	1955.36	1933.95	1956.93
S1S2S3A2	1717.8	1733.03	1686.1	1699.5	1693.05	1705.18	1690.0	1702.0
S3A2	1492.45	1505.89	1480.7	1497.55	1531.35	1550.31	1516.35	1536.26
S2nm	1881.25	1897.13	1878.7	1896.4	1874.05	1891.61	1875.5	1892.90
S3nm	1300.15	1310.88	1296.55	1310.65	1298.5	1308.01	1303.1	1312.21
A2nm	1260.15	1274.51	1258.05	1274.01	1261.1	1275.38	1262.6	1275.01
S1S2aS2bnm	1669.6	1689.61	1669.1	1685.03	1668.95	1686.78	1670.95	1688.91
S1S2S3A2nm	1475.95	1495.18	1480.7	1499.98	1478.9	1494.45	1482.1	1499.9

Tabuľka 6.9: Porovnanie makespanu u rôznych SGS s využitím genetického algoritmu. (m_{min} – minimálny makespan, m_{avg} – priemerný makespan)

6.4 Zhodnotenie experimentov

Hlavný prínos experimentov je odporučenie najvýhodnejšieho algoritmu pre každý typ úlohy. Výber bol vykonaný na základe postupu zo sekcie TODO. Výsledky pre rôzne kritériá sú prehľadne zobrazené v tabuľke 6.10. Vo všeobecnosti je ako najlepšie kritérium z prezentovaných brať do úvahy optimalizáciu na najlepší makespan s ohľadom na zložitosť. Makespan preto, lebo pri optimalizácii na počet konfigurácií nie sú brané do úvahy dĺžky trvania operácií a teda nízky počet rekonfigurácií nemusí viesť na nízky makespan. Je vhodné pozerať sa na zložitosť preto, lebo pri jednoduchšom algoritme môžeme pri rovnakých zdrojoch vytvoriť väčšie množstvo rozvrhov, čím v konečnom dôsledku dosiahneme lepšie najlepšie vygenerované riešenie.

	bez ohľadu na zložitosť		s ohľadom na zložitosť	
úloha	min m	min c_c	min m	min c_c
S1	SMR	SMR	**PM**	PM
S2	PM	PM	**PM**	PM
S2a	PM	PM	**PM**	PM
S2b	SM	SM	**PM**	SM
S3	SM	PM	**PM**	PM
A2	SM	PM	**PM**	PM
S1S2	SMR	SMR	**SMR**	SMR
S1S2aS2b	B	B	**B**	B
S1S2S3A2	SM	SMR	**PM**	PM
S3A2	SM	PM	**PM**	PM
S2nm	P	B	**P**	B
S3nm	P	B	**P**	B
A2nm	P	B	**P**	B
S1S2S3A2nm	P	B	**B**	B
S1S2aS2bnm	B	B	**B**	B

Tabuľka 6.10: **Tabuľka odporúčaných SGS pre daný typ úlohy.** Odporúčania sa líšia podľa hodnotiaceho kritéria požiadaviek na zložitosť. Vo všeobecnosti však môžeme odporučiť tučne označený algoritmus, kedy minimalizujeme makespan s ohľadom na zložitosť.

6.5 Prípadové štúdie vo výrobných podnikoch

Vo firmách HDF s.r.o. a Metaltrim momentálne beží rozvrhovací systém navrhnutý v rámci mojej bakalárskej práce[4]. Tento systém je žiaduce rozšíriť o možnosť modelovania plánu výroby aj s rekonfiguráciami a údržbou. Podľa výsledkov experimentovania môžeme určiť vhodný rozvrhovací algoritmus pre toto rozšírenie.

6.5.1 Prípadová štúdia podniku HDF s.r.o.

V tomto lisovacom podniku sú takmer všetky výrobné príkazy nasledovné: Na začiatku je potrebné vykonať namiešanie a prípravu zmesi, ktorá sa odohráva na pracovisku miešania. Tu nie je potrebné uvažovať žiadnu rekonfiguráciu ani údržbu. Ako druhá operácia je lisovanie, odohrávajúce sa na lisoch, na ktorých je potrebné počítať s rekonfiguráciami a údržbou. Táto výroba má teda charakter úlohy **S2a**.

Podľa tabuľky 6.10 vidíme, že odporúčaný algoritmus pre túto výrobu je *sgsPostProcessIgnoreM*. Tento algoritmus bude teda zvolený pri rozšírení programu.

6.5.2 Prípadová štúdia podniku Metaltrim

Metaltrim je kovoobrábací podnik, ktorý vyrába mnoho rôznych výrobkov pre rôznych odberateľov. Zákazky, ktoré sa tu obvykle vyskytujú sa skladajú z rôzneho počtu operácií a v niektorých prípadoch sa vyskytuje aj operácia montáže. Z prezentovaných balíkov výobných príkazov výroba najviac odpovedá balíku **S1S2S3A2**.

Podľa tabuľky 6.10 je teda vhodné zvoliť algoritmus *sgsPostProcessIgnoreM*. Za zváženie by však stálo vynechanie modelovania údržby pre jednoduchosť. Takýto model by odpovedal balíku **S1S2S3A2nm**. V tomto prípade bude najvhodnejší algoritmus *sgsBasic*.

Kapitola 7

Záver

V diplomovej práci boli prezentované metódy pre vytváranie rozvrhov s dynamickou rekonfiguráciou a údržbou, za účelom praktického využitia v priemyselnej výrobe.

V teoretickej časti bol popísaný matematický model problému RCPSP s dynamickou rekonfiguráciou a údržbou. Tiež bol predstavený problém zakódovania kandidátneho riešenia úlohy a jeho konverzia na rozvrh. Ďalej boli prezentované optimalizačné metódy v podobe prehľadávania náhodných riešení a genetických algoritmov.

Hlavným cieľom tejto práce bolo skúmanie rôznych možností implementácie algoritmov SGS, ktoré transformujú Activity List na rozvrh. Je im teda venovaný aj najväčší priestor v podobe návrhu pokročilých metód v kapitole 3, rozdelených na metódy s udržiavaním striktne platného rozvrhu a metódy s dočasným pripustením neplatnosti rozvrhu, návrhu konkrétnych funkcií SGS a ich implementácií v kapitolách 4.2.1 a 5.1.3 a následného dôsledného experimentálneho vyhodnotenia vhodnosti v rôznych prípadoch v kapitole 6.

Pre účely experimentovania a porovnávania rôznych postupov riešenia RCPSP s dynamickou rekonfiguráciou a údržbou bola vytvorená sada inštancií úlohy. Tieto príklady sú tvorené z rôznych typov zákaziek, čím je napodobnená situácia vo výrobných podnikoch. Experimentovaním sa podarilo vytvoriť jednoznačné odporučenie algoritmu generovania rozvrhu na príslušný druh výroby.

Prínos tejto práce spočíva tiež v podobe netradičnej implementácie rozvrhovacích algoritmov funkcionálnym jazykom Haskell. Jazykové konštrukcie ilustrujúce výhody tohto prístupu boli prezentované v kapitole 5.1.3.

Princípy prezentované v tejto práce budú v blízkej budúcnosti využité pre vylepšenie plánovacieho systému v lisovni plastov HDF s.r.o. Pri implementácii bude využitý algoritmus *sgsPostProcessIgnoreM*, ktorý bol zvolený na základe výsledkov experimentov. Model bude však potrebné rozšíriť o podporu multimódovosti popísanej modelom MRCPSP. Následne bude potrebné dôkladné overenie využiteľnosti modelu v skutočnom prostredí. Taktiež by bolo vhodné sa intenzívnejšie zamerať na optimalizačný mechanizmus.

Literatúra

[1] Blazewicz, J.; Lenstra, J. K.; Kan, A. R.: Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, ročník 5, č. 1, 1983: s. 11–24.

[2] Chen, R.-M.; Wu, C.-L.; Wang, C.-M.; aj.: Using novel particle swarm optimization scheme to solve resource-constrained scheduling problem in PSPLIB. *Expert systems with applications*, ročník 37, č. 3, 2010: s. 1899–1910.

[3] Gregor, M.; Mičieta, B.; Bubeník, P.: Plánovanie vỳroby, Žilinská univerzita v žilině, Žilina, 2005. Technická zpráva, ISBN 80-8070-427-9 [11] Šimon, M., Přednášky předmětu PI.

[4] Halčin, M.: *Počítačová optimalizace a vizualizace výrobních rozvrhů*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2014.

[5] Hartmann, S.: A competitive genetic algorithm for resource-constrained project scheduling. *Naval Research Logistics (NRL)*, ročník 45, č. 7, 1998: s. 733–750.

[6] Holland, J. H.: *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[7] Keřkovskỳ, M.: *Moderní přístupy k řízení vỳroby, 2. vydání.* Nakladatelství CH Beck, 2009.

[8] Kolisch, R.; Hartmann, S.: Heuristic algorithms for the resource-constrained project scheduling problem: Classification and computational analysis. In *Project scheduling*, Springer, 1999, s. 147–178.

[9] Kolisch, R.; Sprecher, A.: PSPLIB-a project scheduling problem library: OR software-ORSEP operations research software exchange program. *European journal of operational research*, ročník 96, č. 1, 1997: s. 205–216.

[10] O'Sullivan, B.; Goerzen, J.; Stewart, D. B.: *Real world haskell: Code you can believe in.* Ö'Reilly Media, Inc.", 2008.

[11] Pinson, E.; Prins, C.; Rullier, F.: Using tabu search for solving the resource constrained project scheduling problem. In *Proceedings of the 4. International Workshop on Project Management and Scheduling*, Leuven, 1994, s. 102–106.

[12] Valls, V.; Ballestin, F.; Quintanilla, S.: Justification and RCPSP: A technique that pays. *European Journal of Operational Research*, ročník 165, č. 2, 2005: s. 375–386.

[13] Wiest, J. D.: Some properties of schedules for large projects with limited resources. *Operations Research*, ročník 12, č. 3, 1964: s. 395–418.

Príloha A

Obsah CD

generate/ - zdrojové kódy programu na generovanie problémov RCPSP s údržbou
problems/ - vygenerováne inštancie úlohy RCPSP vo formáte RCPC, na ktorćh boli vykonané experimenty
schedule/ - zdrojové kódy programu na riešenie RCPSP s dynamickou rekonfiguráciou a údžbou
tex/ - zdrojový text technickej správy
praca.pdf - technická správa v elektronickom formáte

Príloha B

Príklad súboru RCPC

```
19          2
1           1           0           0           5000
1           1           0

2           4           0           5           25
1           1           0
2           1           9
1           2           7
2           2           0


1           0           1           0           8           2           4           6           8
10          12          14          16
1           1           1           13          1           3
2           1           2           15          1           18
1           2           1           20          1           5
2           2           2           13          1           18
1           3           1           15          1           7
2           3           2           20          1           18
1           4           1           20          1           9
2           4           2           13          1           18
1           5           1           17          1           11
2           5           1           16          1           18
1           6           1           15          1           13
2           6           1           12          1           18
1           7           1           10          1           15
2           7           2           16          1           18
1           8           1           12          1           17
2           8           2           17          1           18
1           0           1           0           0
```