České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídící techniky

Bakalářská práce

Optimalizace výroby s relativními omezeními

Vypracoval: Michael Záruba **červen 2006**
Vedoucí práce: Ing. Přemysl Šůcha

Prohlášení

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

V Praze dne ………………………. ……………………………..…
podpis

Poděkování

Rád bych poděkoval Ing. Přemyslu Šůchovi za odborné vedení a cenné rady, které mi pomohly při psaní této práce. Dále bych chtěl také poděkovat svým rodičům za poskytnutí kvalitního zázemí.

Obsah

Prohlášení 2
Poděkování 3
Obsah 4
1. Anotace 5
2. Úvod 6
3. Formulace problému s relativními omezeními 8
  3.1. Rozvrhovací problém 8
  3.2. Rozvrhovací problém s relativními omezeními 9
4. Algoritmus 11
  4.1. Struktura algoritmu 11
  4.2. Alfa procedura 12
    4.2.1. Alfa procedura - inicializace 12
    4.2.2. Alfa procedura - metoda větví a mezí 14
    4.2.3. Alfa procedura - hledání disjunktních párů 16
    4.2.4. Alfa procedura - test na kladný cyklus 18
    4.2.5. Alfa procedura - test rozvrhnutelnosti 19
  4.3. Funkce okamžitého výběru 22
  4.4. Beta procedura 23
5. Implementace programu 26
  5.1. Obsluha programu 26
  5.2. implementační poznámky 27
6. Experimentální výsledky 30
7. Závěr 35
Literatura 36
Příloha 1 – Notace podle Graham-Błażewicz 37
Příloha 2 - Program na CD .... Na vnitřní straně desky.

1. Anotace

Optimalizace výroby s relativními omezeními

Tématem bakalářské práce je implementace algoritmu pro plánování (rozvrhování) výrobních procesů s relativními omezeními. Rozvrhovací problém se zabývá otázkou, jak rozvrhnout jednotlivé operace výroby (úlohy) na jednom stroji (procesoru) tak, aby celková doba vykonávání všech úloh byla co nejkratší. Časová omezení jsou zadávána orientovaným grafem *G*. Uzly grafu odpovídají jednotlivým úlohám. Váhy hran v grafu *G* udávají minimální nebo maximální čas, který uplyne mezi začátky vykonávání dvojice úloh (positive and negative time lags). Váhy uzlů udávají čas, po který je úloha vykonávána (processing time). Cílem práce bylo implementovat vybraný algoritmus v jazyce ANSI C a dále ho napojit na TORSCHE Scheduling Toolbox pro Matlab.

Optimalizacion manufacture with relative limitation

Topic of bachelor thesis is an algorithm implementation for scheduling manufacturing processes with relative limitations. The problem deals with scheduling of manufacturing processes (tasks) on one machine (processor) while the objective is to minimize the schedule length. Time limitations are represented by graph *G*. Nodes of graph *G* represent tasks. Weights of edges in graph *G* represent minimum or maximum time, which elapse between starts of two tasks (positive and negative time lags). Weights of nodes in graph *G* represent processing time of tasks. The purpose of the bachelor thesis is the implementation of chosen algorithm in ANSI C and its interconnection with TORSCHE Scheduling Toolbox for Matlab.

2. Úvod

Model s relativními omezeními je výhodný pro jeho univerzální použití. Programy pro rozvrhování lze použít například v průmyslu, kde máme nějaké operace výroby, mezi kterými existuje následnost.

Představme si výrobní proces, při kterém se provádí galvanické pokovování. Součástku, kterou chceme pokovovat, ponoříme do lázně, kde je předepsán minimální a maximální čas, po který může součástka v lázni zůstat. Tento čas je relativní vůči okamžiku, kdy je součástka ponořena do lázně. Operace (úloha) „vytažení součástky z lázně" je relativně omezena vůči operaci „ponoření součástky do lázně". Toto je jeden z problémů, který řeší model s relativními omezeními. Jsou tu kladná a záporná omezení (positive and negative time lags). Kladné omezení je v našem případě minimální čas, po který je součástka v lázni. Záporné omezení je v tomto případě maximální čas ponoření součástky v lázni. Využití v praxi je široké, např. rozvržení paralelně probíhajících procesů na jednotlivých procesorech PC a mnoho dalších.

Na vývoji různých modelů a algoritmů se podílelo mnoho lidí. Například A. M. Kordon [3] používá model, který uvažoval pouze kladná časová omezení (positive time lags). Již tento problém patří do třídy takzvaných NP-úplných problémů [10]. Model, kterým se zde budeme zabývat, uvažuje kladná i záporná časová omezení. Tento model zavedl B. Roy [4].

Způsoby řešení tohoto problému jsou jednak optimální, kdy cílem je najít optimální řešení, nebo heuristické, kdy se spokojíme s řešením s horší hodnotou cílové funkce. Naproti tomu optimální algoritmy jsou použitelné pouze na malé problémy, kdežto heuristické dokáží řešit problémy výrazně větší.

Jedno z heuristických řešení od J. Hurink and J. Keuchel [2] je založeno na metodě zakázaného prohledávání (tabu search). Peter Brucker [1] navrhl optimální řešení metodou větví a mezí (branch and bound). Jedná se o rozvrhovací algoritmus, který používá model od B. Roy.

Přínosem této práce je naprogramovaný Bruckerův algoritmus v programovacím jazyce ANSI C. Bruckerův algoritmus byl vybrán z důvodu jeho vysoké výpočetní rychlosti. Předpokládá se, že jeho výpočetní čas bude kratší než výpočetní časy ostatních algoritmů, které rozvrhují stejný problém. Dalším přínosem je, že tento algoritmus je začleněn do TORSCHE Scheduling Toolbox pro Matlab [11], volně dostupného nástroje pro vývoj algoritmů pro rozvrhování (http://rtime.felk.cvut.cz/scheduling-toolbox). Implementovaný algoritmus bude zařazen do příští verze nástroje.

Práce je strukturována následovně. Model s relativními omezeními je vysvětlen v kapitole 3 a algoritmus je popsán v kapitole 4. V kapitole 5 je popsána organizace souborů, implementace a rozmístění popisovaných algoritmů v jednotlivých souborech. V kapitole 6 jsou experimentální výsledky a porovnání dvou algoritmů. Závěr je v kapitole 7. Následuje seznam literatury a Příloha 1 (popis notace rozvrhovacích problémů podle Grahama a Błażewicz), na kterou se budeme v textu odkazovat.

Pokud se bude chtít čtenář seznámit s kódem programu podrobně, odkazuji ho přímo do zdrojových souborů. Detaily jsou vysvětleny pomocí komentářů (viz Příloha 2). Předpokládá se, že čtenář má základní znalosti jazyka ANSI C.

3. Formulace problému s relativními omezeními

3.1. Rozvrhovací problém

Předmětem rozvrhování je uspořádat množinu úloh (tasks)

$$\tau = \{T_1, ..., T_n\} \quad (1)$$

na zadané procesory (stroje) tak, abychom dosáhli optimálního řešení vzhledem k požadovanému kritériu. Nejčastěji to znamená dosáhnout co nejkratší doby vykonání celé množiny úloh. Jednotlivé úlohy z množiny τ jsou obvykle charakterizovány parametry:

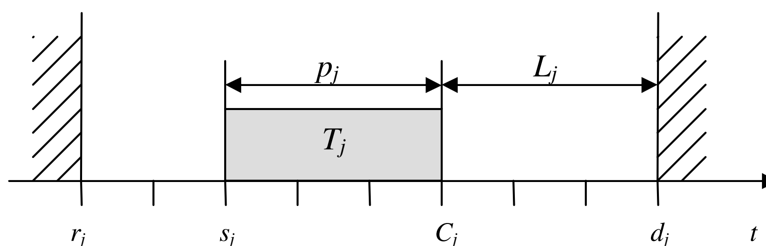
p_j - doba vykonávání úlohy (processing time)
r_j - čas, kdy nejdříve může být úloha vykonána (release date)
d_j - čas, kdy musí úkol nejpozději skončit (dead line)

Cílem rozvrhování je nalézt začátky vykonávání jednotlivých úloh T_i tak, že se jednotlivé úlohy na přiděleném procesoru nepřekrývají a všechny úlohy vyhovují zadaným omezením (např. r_j a d_j). Výsledkem rozvrhování potom je:

s_j - začátek vykonávání úlohy (start time)

Od něhož lze odvodit:

C_j - čas dokončení úlohy (completion time)
$C_j = s_j + p_j$
L_j - zpoždění úlohy (lateness)
$L_j = C_j - d_j$



Obr. 3.1

Navíc se rozvrhovací algoritmy snaží minimalizovat požadovanou cílovou funkci. Nejčastějším kritériem je délka rozvrhu značená $C_{max} = \max(C_j)$. Dalším častým kritériem je maximální zpoždění $L_{max} = \max(L_j)$.

3.2. Rozvrhovací problém s relativními omezeními

Uvažujme, že cílem je rozvrhnout úlohy τ na jeden procesor (stroj). Tento model uvažuje kladná a záporná relativní omezení. Kladná omezení (positive-lags) vyjadřují nejkratší možnou časovou vzdálenost mezi začátky vykonávání dvou úloh. Naproti tomu záporná omezení (negative-lags) vyjadřují nejdelší možnou časovou vzdálenost mezi začátky vykonávání dvou úloh. Cílem je nalézt takový přípustný rozvrh, který má minimální kritérium C_{max}.

Označme velikost kladného a záporného omezení l_{ij} , potom lze oba typy omezení vyjádřit jednou nerovnicí.

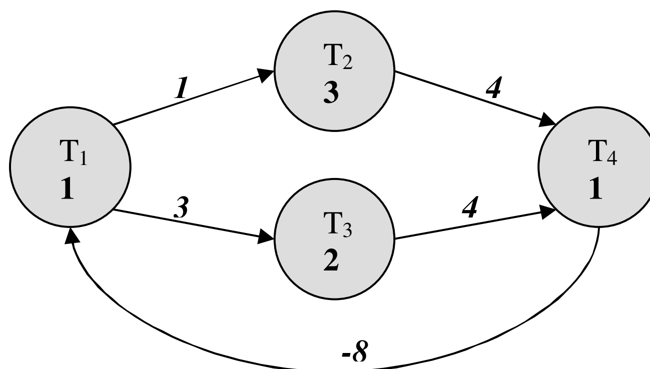
$$s_i + l_{ij} \leq s_j \qquad (2)$$

Pokud je $l_{ij} \leq 0$, potom l_{ij} vyjadřuje záporné relativní omezení. Naopak pro $l_{ij} > 0$ je vyjádřeno kladné relativní omezení. Tento problém je možno reprezentovat pomocí orientovaného grafu G.
Graf obsahuje:

- uzly (node) - reprezentují jednotlivé úlohy
- kladné váhy hran (forward edge) - reprezentují kladná omezení ($l_{ij} \leq 0$)
- záporné váhy hran (beckward edge) - reprezentují záporná omezení ($l_{ij} > 0$)

Tento graf může být dále reprezentován maticí **W** a vektorem **p**. Vektor **p** obsahuje doby vykonávání p_i úloh $T_i \in \tau$. Matice **W** definuje časová omezení mezi jednotlivými úlohami T_i, tj. obsahuje prvky l_{ij}. Index řádků matice značíme i a index sloupců matice j. Hrana, která není definována bude mít v matici **W** hodnotu mínus nekonečno ($-\infty$). Dvě úlohy, mezi kterými není definována hrana, se nazývají *disjunktní pár*.

Uvedeme zde příklad rozvrhovacího problému pro 4 úlohy
$\tau = \{T_1, T_2, T_3, T_4\}$. Relativní omezení jsou zadána grafem na obr. 3.2.



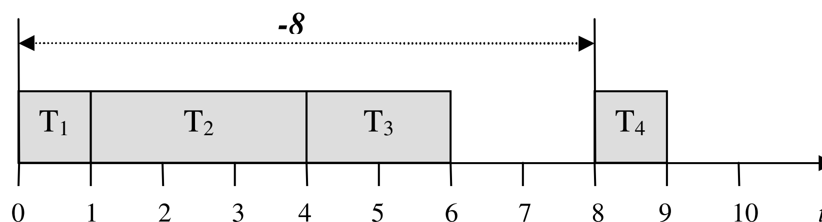
Obr. 3.2

Odpovídající matice **W** a vektor **p** jsou znázorněny na obr. 3.3. Z grafu je patrné, že graf má čtyři kladná relativní omezení a jedno záporné. Záporné omezení nám říká, že úloha T_4 musí začít nejpozději v $s_1 + 8$ strojových jednotek času.

$$\mathbf{W} = \begin{pmatrix} 0 & 1 & 3 & -\infty \\ -\infty & 0 & -\infty & 4 \\ -\infty & -\infty & 0 & 4 \\ -8 & -\infty & -\infty & 0 \end{pmatrix} \quad (\text{sloupce } T_1\ T_2\ T_3\ T_4) \qquad \mathbf{p} = (1,3,2,1)$$

Obr. 3.3

Uvažujeme-li jako cíl optimalizace kritérium C_{max}, pak optimální rozvrh příkladu z obrázku 3.3 je znázorněn na obrázku 3.4.



Obr. 3.4

4. Algoritmus

4.1. Struktura algoritmu

Rozvrhovací algoritmus se skládá ze dvou hlavních částí. Z alfa procedury a beta procedury. Alfa procedura je základem celého algoritmu a je založena na metodě *větví a mezí* (branch and bound). Metoda větví a mezí je nejběžnější optimalizační technika založená na ořezávání větví, ve kterých se nenachází požadované řešení. Prohledávaný stavový prostor je nejčastěji reprezentován jako binární strom, a proto časová složitost této metody roste exponenciálně s časem.

Profesor Brucker [1] se ve svém algoritmu snažil co nejvíce prohledávání binárního stromu urychlit. A to samotným způsobem, jakým se binární strom prohledává. Například některé větve jsou vybrány dříve, než ostatní. Dále pak použitím funkce *okamžitého výběru* (Immediate selection), která snižuje hloubku prohledávacího stromu.

Samotná alfa procedura prohledává stavový prostor (metodou větví a mezí), dokud nenalezne první přípustný rozvrh S, který má hodnotu kritéria $C_{max}(S)<UB$, kde UB je horní mez kritéria C_{max}. Alfa proceduru lze nepatrnou modifikací upravit tak, aby byla schopna sama nalézt optimální rozvrh. Beta procedura využívá alfa proceduru takovým způsobem, aby co nejvíce zefektivnila prohledávání binárního stromu a nalezla optimální řešení S^* vzhledem ke kritériu C_{max}.

Podle experimentů profesora Bruckera dosahuje varianta s beta procedurou výrazně lepších výsledků, a tak umožňuje řešit instance řádově do velikosti 100 úloh. Podrobně budou obě procedury vysvětleny v dalších podkapitolách.

4.2. Alfa procedura

4.2.1. Alfa procedura - inicializace

Alfa procedura dostává jako vstup matici **W'**, vektoru **p'** a *UB* - *horní mez* kritéria C_{max}. Výpočet kritéria *UB* je uveden v kapitole 4.4. Parametr **W'** nazvěme *rozšířenou maticí* a vektoru **p'** *rozšířeným vektorem.*

UB se bude během výpočtu postupně zmenšovat dokud nedosáhneme optima (C_{max}). Alfa procedura je spouštěna beta procedurou tak dlouho, dokud není nalezeno optimální C_{max}. Důležité je, aby se v prohledávaném stavovém prostoru neobjevovala řešení, která mají C_{max} horší nebo stejné, než nejlepší dosud nalezené. To umožní přidání dalšího omezení (viz Příloha 1). Aby bylo možné přidat toto omezení, je potřeba nejprve rozšířit graf *G* o dva uzly (úlohy) T_0 a T_{n+1}. T_0 je úloha, která je předchůdcem všech úloh $T_i \in \tau$ a její doba vykonávání je p_0=0. Naopak úloha T_{n+1} je následníkem všech úloh $T_i \in \tau$ a její doba vykonávání je p_{n+1}=0. Tím vytvoříme novou množinu úloh $\tau' = \{T_0, T_1, \ldots, T_n, T_{n+1}\}$. Potom kritérium C_{max} lze vyjádřit jako $C_{max} = s_{n+1} - s_0$ a lze jej omezit podmínkou,

$$s_{n+1} \leq s_0 + UB - 1 \qquad (3)$$

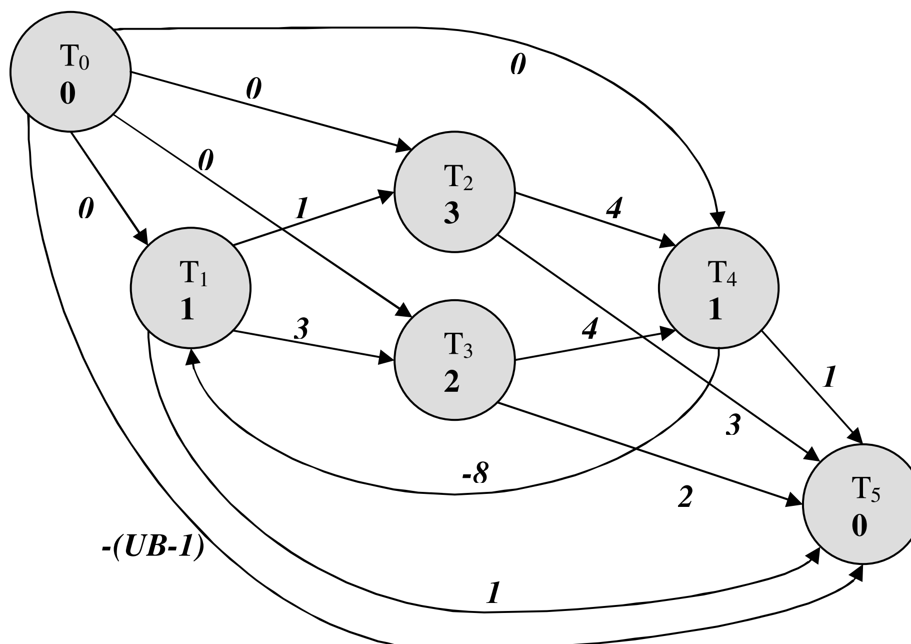
kterou lze zařadit mezi omezení (2). Tato podmínka zneplatní všechna řešení *S*, která mají $C_{max}(S) \geq UB$.

Odpovídající rozšířenou matici **W** budeme značit **W'** a rozšířený vektor **p** označíme **p'**. Do **W'** tedy zavádíme omezení $l_{n+1,0} = -(UB-1)$.
V orientovaném grafu *G* se projeví rozšíření tak, že do každého uzlu z množiny $\tau = \{T_1, \ldots, T_n\}$ vstupuje jedna hrana z uzlu T_0 a z každého uzlu z τ vstupuje jedna hrana do uzlu T_{n+1}. Všechny tyto přidané hrany mají kladný smysl. Vektor **p'** potom bude $\mathbf{p}' = (0, p_1, \ldots, p_n, 0)$ a matice **W'** je definována na obr. 4.1.

$$\mathbf{W}' = \begin{pmatrix} 0 & 0\ldots & \ldots 0 \ldots & \ldots 0 & 0 \\ -\infty & W_{11}\ldots & \ldots & \ldots W_{1n} & p_1 \\ -\infty & \vdots & \ldots W_{ij} \ldots & \vdots & p_i \\ -\infty & W_{n1}\ldots & \ldots & \ldots W_{nn} & p_n \\ -(UB-1) & -\infty\ldots & \ldots -\infty \ldots & \ldots -\infty & 0 \end{pmatrix}$$

Obr. 4.1

Graf *G* z příkladu z kapitoly 3.2 je rozšířen na *G*' v obrázku 4.2.



Obr. 4.2

Rozšíření **W** a **p** je ukázáno v obr. 4.3.

$$\mathbf{W}' = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ -\infty & 0 & 1 & 3 & -\infty & 1 \\ -\infty & -\infty & 0 & -\infty & 4 & 3 \\ -\infty & -\infty & -\infty & 0 & 4 & 2 \\ -\infty & -8 & -\infty & -\infty & 0 & 1 \\ -(UB-1) & -\infty & -\infty & -\infty & -\infty & 0 \end{pmatrix}$$

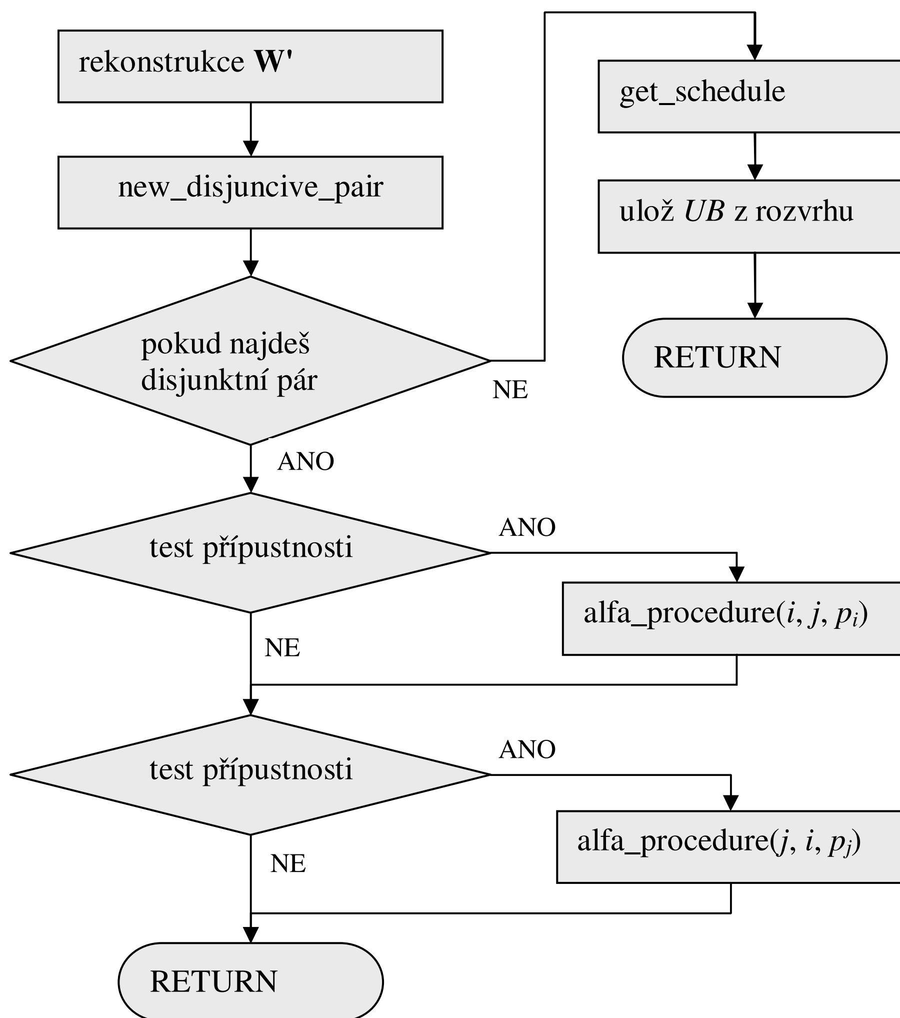
$$\mathbf{p}' = (0,1,3,2,1,0)$$

obr 4.3

Na takto rozšířené **W** a **p** můžeme aplikovat funkci okamžitého výběru, která bude popsána v podkapitole 4.3. Teprve nyní lze zavolat funkci, ve které je implementována alfa procedura.

4.2.2. Alfa procedura - metoda větví a mezí

Alfa procedura je implementována jako rekurzivní funkce *alfa_procedure*(i, j, l_{ij}), kde i a j jsou indexy matice **W'** a l_{ij} je váha hrany přidávaná na pozici i a j. K větvení dochází tím, že hrana je přidána z T_i do T_j a z T_j do T_i. Na obr. 4.4 je vývojovým diagramem naznačeno tělo funkce. Metody této funkce budou popsány v dalších podkapitolách.



obr. 4.4.

Na samém začátku je nutná rekonstrukce **W'**. Alfa procedura používá matici $\mathbf{W}_\alpha$, ve které je udržován aktuální stav procházeného stavového prostoru. Do $\mathbf{W}_\alpha$ je zkopírováno **W'** a modifikováno podle seznamu doposud přidaných hran (i, j, l_{ij}). Metoda *new_disjuncive_pair* hledá nové disjunktní páry a bude vysvětlena v samostatné podkapitole. Pokud nebude žádný pár nalezen, znamená to, že větvení binárního stromu je na nejnižší úrovni. Rozvrh sestavíme metodou *get_schedule* z matice $\mathbf{W}_\alpha$. Výstupem je $s = (s_0, s_1, ..., s_n, s_{n+1})$. Potom $UB = s_{n+1}$. Pokud nalezneme nějaký disjunktní pár, pak provedeme test přípustnosti. Ten se skládá ze dvou *testů neproveditelnosti* (infeasibility tests).

Provádíme dva testy:

1. Test na kladný cyklus
2. Test rozvrhnutelnosti

Prvním testem pomocí metody *not_positive_cycle* zjistíme, jestli přidáním hrany mezi dvojici uzlů jakožto disjunktního páru nevznikne v G' kladný cyklus. Druhým testem zjistíme, jestli rozvrh, do kterého přidáme hranu s váhou l_{ij} nebo l_{ji}, bude přípustný. To zjistíme tak, že problém převedeme na několik problémů $1|\,r_j\,;pmtn\,|\,L_{max}$ (viz Příloha 1), který lze vyřešit pomocí *Hornova algoritmu* v polynomiálním čase [7]. Testy provádíme pro obě větve binárního stromu. Nakonec příkazem *alfa_procedure*(i, j, p_i) nebo *alfa_procedure*(j, i, p_j) rekurzivně zavolá metoda sebe samu.

4.2.3. Alfa procedura - hledání disjunktních párů

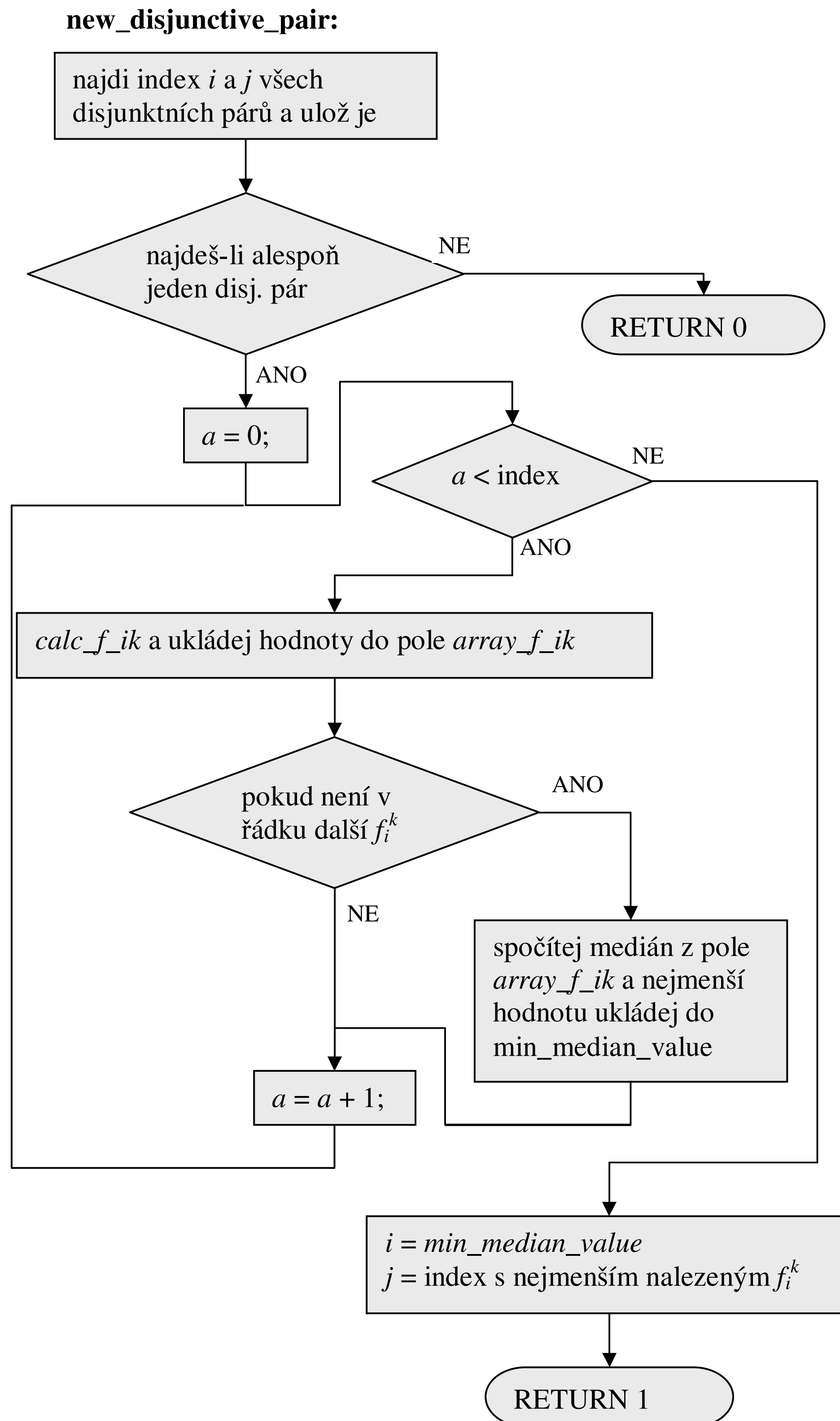
K hledání disjunktních párů slouží metoda, jejíž vstupem je $\mathbf{W}_\alpha$ a výstupem vybraný disjunktní pár (i, j). Její název je *new_disjunctive_pair*. Tato metoda po spuštění vyhledá všechny disjunktní páry $\{i, j\}$. To jsou takové, pro které neplatí $l_{ij} \geq p_i$ ani $l_{ji} \geq p_j$. Jeden z nich musíme vybrat pro následné větvení. Pro výběr disjunktního páru použijeme časové okno, které vyjadřuje, jak je začátek vykonávání úlohy T_i omezen vzhledem k ostatním úlohám. Úloha T_i ve vztahu k úloze T_k může být vykonávána v časovém oknu $[s_k + r_i^k, s_k + d_i^k]$, kde r_i^k = l_{ki} a d_i^k = p_i - l_{ik}. Pokud upevníme kteroukoli úlohu T_k, budou mít všechny úlohy časové okno, ve kterém budou vykonány. Zavedeme tedy časové okno [r_i^k , d_i^k].

Při výběru disjunktního páru analyzujeme pro každý disjunktní pár, jak jsou začátky vykonávání odpovídajících úloh omezeny časovým oknem [1]. Vypočteme číslo f_i^k , které udává počet možných celočíselných začátků vykonávání s_i úlohy T_i v časovém intervalu [r_i^k , d_i^k] normované součtem (p_i+p_k). Číslo f_i^k tedy spočteme pomocí vzorce (4), kde i jsou řádky $\mathbf{W}_\alpha$ a k jsou sloupce $\mathbf{W}_\alpha$.

$$f_i^k = \frac{-l_{ik} - l_{ki} - p_i - p_k + 2}{p_i + p_k} \qquad (4)$$

Pro větvení vybereme takový disjunktní pár, jehož úlohy nejsou příliš omezené a také nejsou příliš neomezené. Toho docílíme tak, že spočteme pro každou úlohu T_i medián ze všech f_i^k hodnot a vybereme takovou úlohu i, která má nejmenší medián. Z úloh, které mají shodný index i z předchozího výběru vybereme takovou úlohu j, která má nejmenší f_i^j hodnotu. Index i a j jsou souřadnice disjunktního páru, který bude následně větven. Implementace je na obr. 4.5.

new_disjunctive_pair:



Obr. 4.5

Proměnná *index* obsahuje počet nalezených disjunktních párů. Algoritmus nejprve vyhledá všechny indexy (i , k). Funkce *calc_f_ik* spočítá z $\mathbf{W}_\alpha$, **p** a indexů (i , k) číslo f_i^k . Pro všechny disjunktní páry jsou spočteny hodnoty f_i^k . Představme si, že bychom je uspořádali podle (i , k) do matice **F**. Potom bychom pro řádky **F** spočítali jejich mediány a vybrali takový řádek i, pro který je medián nejmenší. Dále vybereme takový sloupec j, pro který je f_i^j hodnota nejmenší a zároveň se vyskytuje v řádku i. Hodnoty f_i^j počítáme stejným způsobem jako f_i^k , akorát s tím rozdílem, že k ve vzorci (4) nahradíme j. Vybraný disjunktní pár je potom při větvení zafixován hranou z T_i do T_j a z T_j do T_i.

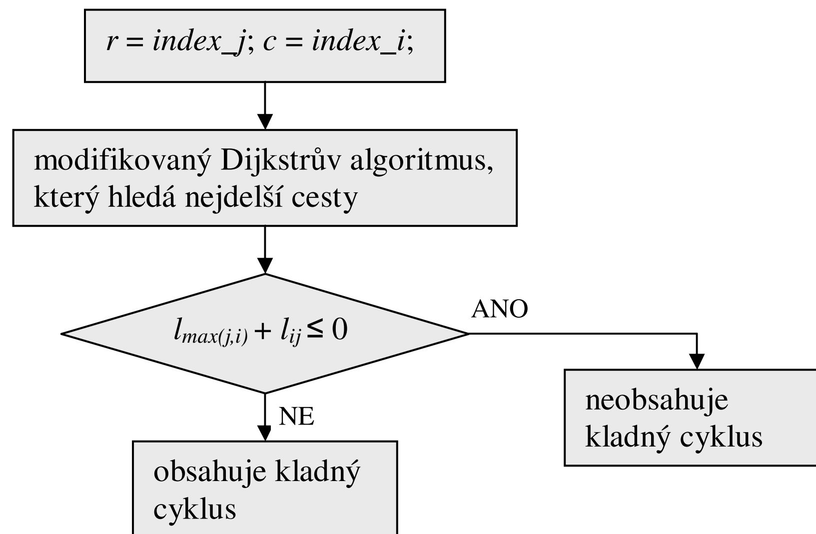
4.2.4. Alfa procedura - test na kladný cyklus

Pro detekci kladného cyklu je použit modifikovaný *Dijkstrův algoritmus*, který je popsán v [6, str. 95]. Ten se používá na hledání nejkratších cest v grafu z jednoho vrcholu r do ostatních vrcholů. Může být upraven i tak, aby hledal nejdelší cesty v grafu. Tento algoritmus vrací vektor **U**, který obsahuje nejdelší vzdálenosti z vrcholu r do všech ostatních uzlů. Pro nás je důležitá vzdálenost do vrcholu c. Maximální vzdálenost z r do c označme $l_{max(j,i)}$. Test je implementován jako funkce *not_positive_cycle* jejíž vstupní parametry jsou:

- $\mathbf{W}_\alpha$ (matice, ve které chceme zjistit, zda vznikne kladný cyklus pokud do ní přidáme hranu o váze l_{ij})
- i
- j
- l_{ij} (váha přidávané hrany)

Výstupem funkce je logická hodnota. Pokud je výstup log. 1, potom graf kladný cyklus po přidání hrany neobsahuje. Tato funkce má tu nevýhodu, že $\mathbf{W}_\alpha$ nesmí žádné kladné cykly obsahovat. V opačném případě by došlo v programu k nekonečné smyčce. V našem případě $\mathbf{W}_\alpha$ nemůže obsahovat kladný cyklus, jelikož každé přidání hrany je kontrolováno. Na obrázku 4.6 je uveden jednoduchý vývojový diagram, který ukáže návaznost modifikovaného Dijkstrova algoritmu, který hledá nejdelší cesty a funkcí *not_positive_cycle*.

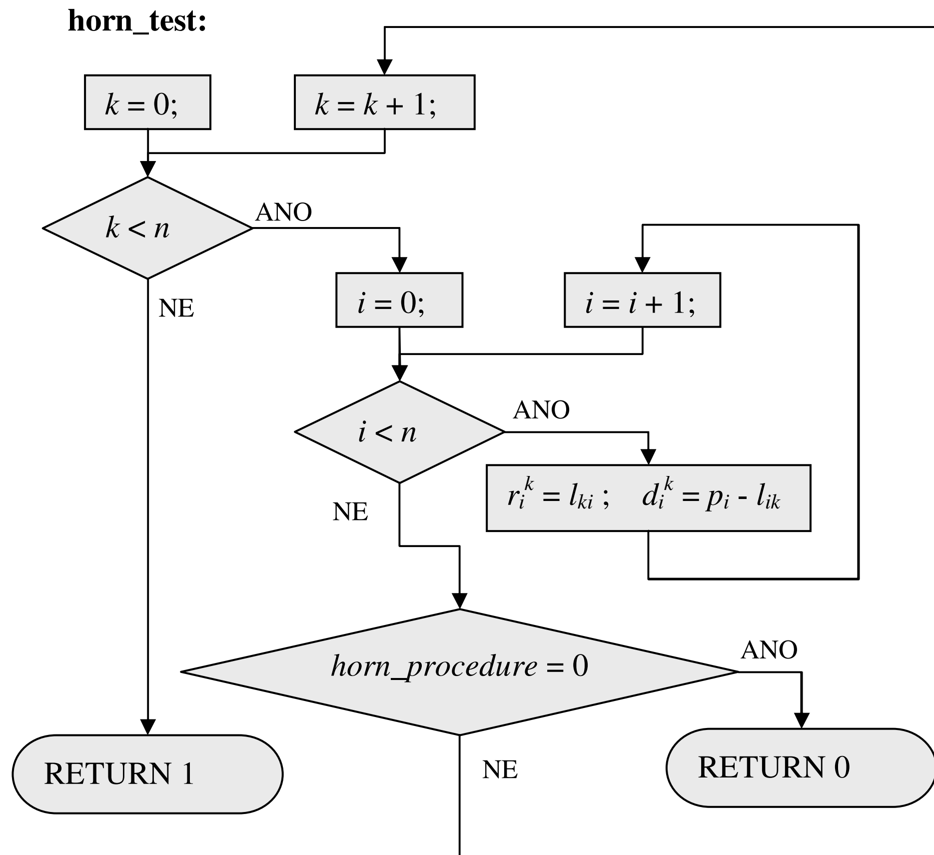
not_positive_cycle:



Obr. 4.6

4.2.5. Alfa procedura - test rozvrhnutelnosti

Pro testování rozvrhnutelnosti se používá n-krát (kde n je počet úloh) Hornův test. Je to řešení problému $1|\, r_j\, ;\, pmtn \,|\, L_{max}$ (viz Příloha 1). Pro každou úlohu k, kde k označuje sloupce $\mathbf{W}_\alpha$ uděláme následující. Spočítáme časové okno [r_i^k, d_i^k] pro všechny úlohy T_i ve vztahu k úloze T_k. Časové okno se skládá z (release date) $r_i^k = l_{ki}$ a (dead line) $d_i^k = p_i - l_{ik}$. Problém je nerozvrhnutelný, když L_{max} je kladné. Jinými slovy nerozvrhnutelnost nastane právě tehdy, když alespoň jedna úloha překročí čas svého d_i^k. Tato metoda se nazývá *horn_test*. Jejím vstupem je $\mathbf{W}_\alpha$ a **p**. Výstupem je pak log. 1, pokud byl test úspěšný, a naopak log. 0, pokud byl test neúspěšný. Na obrázku 4.7 je pomocí vývojového diagramu metoda *horn_test* znázorněna.



Obr. 4.7

Funkce *horn_procedure* hledá optimální řešení problému $1| r_j ; pmtn | L_{max}$. Funkce *horn_procedure* vrací také logickou hodnotu. Pokud funkce vrátí log. 0, znamená to, že Hornův test i celá metoda je neúspěšná. Hornův algoritmus je popsán v literatuře [7]. Popišme nyní Hornův algoritmus řešící problém $1| r_j ; pmtn | L_{max}$.

Proměnná ρ_1 udává časový okamžik r_j všech úloh, které mají nejmenší r_j. Proměnná ρ_2 udává časový okamžik následujícího r_j. E je množina všech úloh, které mají r_j v čase ρ_1.

Hornův algoritmus:

begin
repeat

$\rho_1 := \min_{T_j \in \tau} \{r_j\};$

if všechny úlohy jsou k dispozici v čase ρ_1

then $\rho_2 := \infty$

else $\rho_2 := \min\{r_j \mid r_j \neq \rho_1\};$

$E := \{T_j \mid r_j = \rho_1\};$

Vyber $T_k \in E$ takové, že $d_k = \min_{T_j \in E} \{d_j\};$

$l := \min\{p_k, \rho_2 - \rho_1\};$

Přiřaď T_k do intervalu $[\rho_1, \rho_1 + l)$;

if $p_k \leq l$

then $\tau := \tau - \{T_k\}$

else $p_k := p_k - l$;

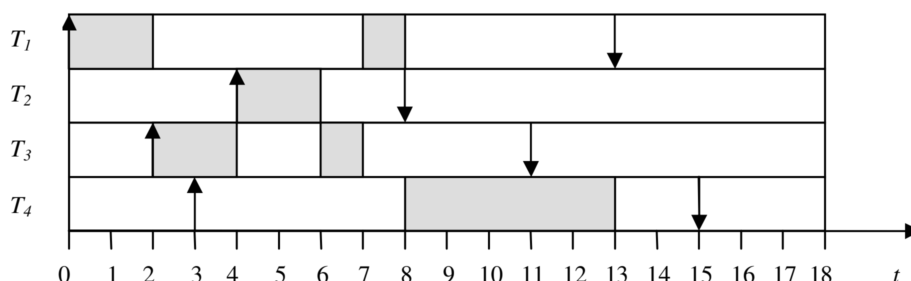
for all $T_j \in E$ **do** $r_j := \rho_1 + l$;

until $\tau = \emptyset$;
end;

Hornův algoritmus funguje tak, že v daný čas ρ_l je rozvržena úloha T_k, která je připravena ($r_k \leq \rho_l$) a má nejmenší d_k. Pokud by nastal případ, kdy existuje úloha T_i, která je připravena ($r_i \leq \rho_l$) a úloha T_k ještě nebude připravena, začne být vykonávána úloha T_i, která může být přerušena úlohou T_k, jestliže v čase přerušení je T_k již připravena. Nejlépe bude princip algoritmu zřetelný z příkladu na obrázku 4.8. Vektor **r** označuje množinu časů r_i a vektor **d** množinu časů d_i.

Z příkladu je zjevné, že nejmenší r_i má úloha T_1. V čase 2 se algoritmus rozhodne rozvrhovat úlohu T_3, protože $d_3 < d_1$. Úloha T_3 je přerušena úlohou T_2 ze stejného důvodu jako úloha T_1. Jakmile je T_2 rozvrhnuta algoritmus dokončuje úlohy v pořadí tak, aby úlohy s menším d_i byly rozvrhnuty dříve.

Máme zadány tři vektory **p**, **r**, **d**.
p = (3 2 3 5)
r = (0 4 2 3)
d = (13 8 11 15)



Obr 4.8

4.3. Funkce okamžitého výběru

Účelem této funkce je přiřadit hrany mezi takové disjunktní páry, které by bylo zbytečné větvit v alfa proceduře. Jsou to takové disjunktní páry, které mají jednoznačné pořadí. Funkce okamžitého výběru se sestává ze dvou částí.

První část funkce zafixuje hrany, u kterých by bylo na první pohled z grafu *G'* vidět jejich pořadí. Budeme hledat nejdelší cesty v grafu *G*. K tomu použijeme upravený *Floydův algoritmus*, jehož výstupem je matice **L** nejdelších cest v grafu *G*. Označme vzdálenost těchto cest $L(i, j)$. A nyní vybereme ze všech disjunktních párů takové, které splňují podmínku $L(\mathrm{i}, \mathrm{j}) > -p_j$ v matici **L**. Pro tyto T_i a T_j potom nahradíme v matici **W'** hrany s váhou $l_{ij} = p_i$.

Další způsoby okamžitého výběru jsou vysvětleny v literatuře [1]. Nyní se podíváme, jak pracuje Floydův algoritmus. V literatuře [6] je popsán Floydův algoritmus, který hledá nejkratší cesty mezi jednotlivými uzly grafu *G'*. Zde bude tento algoritmus nepatrně modifikován tak, aby hledal nejdelší cesty mezi T_i a T_j.

Algoritmus lze popsat takto:

Vstup: Matice **L** , která obsahuje délky hran.
Výstup: Je tvořen touž maticí **L** , která obsahuje vzdálenosti.
Algoritmus:

pro k := 1,2,3,...,n proveď:
 pro i := 1,2,3,...,n proveď:
  pro j := 1,2,3,...,n proveď:
   pokud $L(i,j) < L(i,k) + L(k,j)$,
    pak proveď $L(i,j) := L(i,k) + L(k,j)$

Časové složitost: $O(n^3)$.

4.4. Beta procedura

Alfa procedura může být vylepšena beta procedurou, která se skládá ze tří fází.

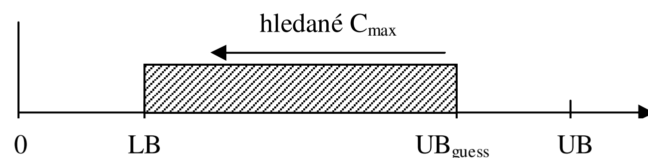
V první fázi se pokoušíme zlepšit spodní odhad hodnoty kritéria $C_{max} \in [LB, UB_{guess}]$. Pro inicializaci vypočteme spodní a horní odhad LB (lower bound) a UB (upper bound). UB je délka nejhoršího možného rozvrhu , který se počítá podle vzorce (5).

$$UB = \sum_{i=1}^{n} \max \left\{ p_i , \max_{(i,j)\in R} l_{ij} \right\} \qquad (5)$$

Výpočet LB lze nalézt v literatuře [1]. Vypočteme UB_{guess} podle vzorce (6),

$$UB_{guess} = \mu(LB - UB) + UB \qquad (6)$$

kde $0 < \mu < 1$. Typické hodnoty pro μ jsou $\mu = 1/3$ nebo $\mu = 1/4$. Experimenty dokazují, že je zbytečně vynaloženo výpočetní úsilí pro hledání přípustného řešení v intervalu $[UB_{guess}, UB]$. Proto zúžíme prostor hledání C_{max}, jak je vidět z obrázku 4.9.



Obr. 4.9

Abychom se vyvarovali dlouhým výpočtům, modifikujeme alfa proceduru tak, že její větvení provádíme pouze do hloubky *d*. Typické hodnoty pro *d* jsou 0, 1 nebo 2. Možné výsledky takto modifikované alfa procedury jsou dva:
1. Je dokázána nepřípustnost.
2. Není dokázána nepřípustnost.

V 1. případě to znamená, že $UB_{guess} \leq C^*$ a v takovém případě musíme zvýšit *LB* na hodnotu UB_{guess}. Potom budeme stejnou proceduru aplikovat na interval [*LB*, *UB*]. V 2. případě aplikujeme stejnou proceduru na interval [LB, UB_{guess}]. První fáze je zakončena tak, že zvažovaný interval obsahuje pouze jeden prvek. Jako výsledek první fáze je zlepšený *LB*.

<u>Ve druhé fázi</u> využijeme vylepšený *LB* a vypočteme $UB_{guess} = LB + delta$, kde *delta* je celé číslo. Na začátku je *delta* = 1. Spustíme alfa proceduru, která již větví celý stavový prostor s parametrem UB_{guess} jako *UB*. Pokud je nalezeno přípustné řešení *S*, optimální řešení hledáme již pouze v relativně malém intervalu [LB, $C_{max}(S)$]. V třetí fázi beta procedury. Pokud alfa procedura detekuje nepřípustnost, potom bude $LB = UB_{guess}$ a přidáme novou *delta* hodnotu k *LB*. Nová hodnota proměnné *delta* se určí jako nejmenší délka kladného cyklu, který odhalila funkce *not_positive_cycle* během provádění předchozí iterace alfa procedury. Dále opakujeme fázi 2. Další možnosti výpočtu *delta* je popsáno v literatuře [1].

<u>Ve třetí fázi</u> opakovaně aplikujeme alfa proceduru s $UB = C_{max}(S)$. Mohou nastat dvě situace. Alfa procedura nalezne přípustné řešení *S*. V takovém případě provedeme aktualizaci *UB* jako $UB := C_{max}(S)$. Poté znovu opakujeme třetí fázi. Druhá situace nastane, když alfa procedura nenalezne přípustné řešení *S* pro $UB-1$. Potom optimálním řešením S^* je poslední nalezené řešení *S*. Algoritmus popisující beta proceduru by mohl vypadat následně.

beta procedura:

0) ***Inicializace***

Výpočet *UB* (horní mez C_{max}) a *LB* (dolní mez C_{max}) :

$LB \leq C_{max} \leq UB$.

UB1=*UB*; *LB1*=*LB*;

delta=1;

*S** = [];

1) ***Zlepšení LB***

$UB_{guess} = \lceil \mu.LB1 + (1-\mu).UB1 \rceil$

S = alfa_procedure(*LB1*, UB_{guess}, 1)

if(*S* je přípustný)

$UB1 = UB_{guess}$;

else

$LB1 = UB_{guess}$;

end

if(*UB1*-*LB1* <= 1)

LB = *LB1*;

goto 2;

else

goto 1;

end

2) ***Zlepšení UB***

$UB_{guess} = LB + delta$;

S = alfa_procedura(*LB*, *UBguess*, inf)

if(*S* je přípustný)

$UB = C_{max}(S) - 1$;

*S** = *S*;

goto 3;

else

$LB = UB_{guess}$;

delta = min. délka kladného cyklu nalezená v *not_positive_cycle*

goto 2;

end

3) ***Nalezení řešení***

S = alfa_procedura(*LB*, UB_{guess}, inf)

if(*S* je přípustný)

$UB = C_{max}(S) - 1$;

*S** = *S*;

goto 3;

else

return *S**;

end

5. Implementace programu

5.1. Obsluha programu

Program je možno spustit dvěma způsoby. První způsob je pomocí programu MATLAB, kde se přímo ze zdrojových souborů vytvoří *dll* knihovna. Funkcí *BruckerBaB*, která se napíše v prostředí MATLAB, se spustí výpočet naprogramovaný v jazyce C. Parametry funkce jsou (**p**,**W**,0), kde poslední parametr zatím nemá význam. Po spuštění příkazu je zobrazen vektor **s**, jehož velikost je stejná, jako u vektoru **p**. Z **p** a **s** lze sestavit rozvrh, například pomocí grafického prostředí již implementovaného v TORSCHE Scheduling Toolbox pro Matlab. Druhý způsob spuštění je pomocí přeloženého souboru **BruckerBaB.exe** . Vstupní data musí být v souboru **data.txt** a soubor musí být ve stejném adresáři jako je program. Struktura textového souboru je na obrázku 5.1. Příklad zadání dat z příkladu v kapitole 3.2 je na obrázku 5.2.

počet uzlů grafu G
*vektor **p***
*matice **W***

Obr. 5.1

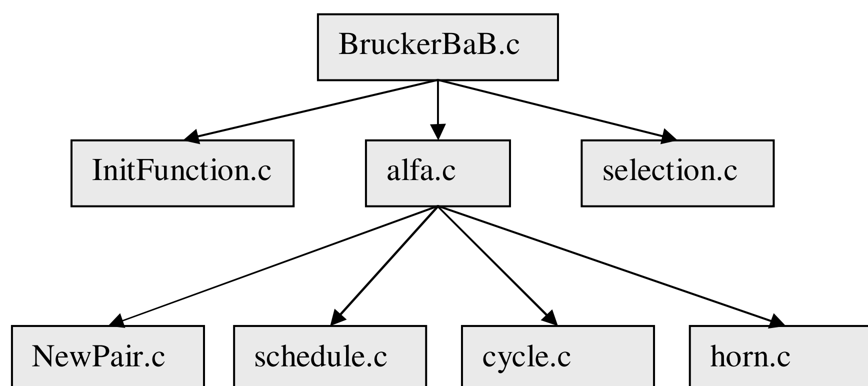
$$\begin{array}{cccc} 4 & & & \\ 1 & 3 & 2 & 1 \\ 0 & 1 & 3 & -I \\ -I & 0 & -I & 4 \\ -I & -I & 0 & 4 \\ -8 & -I & -I & 0 \end{array}$$

Obr. 5.2

Program je ošetřen proti zadání neplatných dat. Jakmile proběhne výpočet, výsledek **s** se zapíše na konec souboru **data.txt**. Pokud ze zadaných dat nelze sestavit přípustný rozvrh, program namísto vektoru **s** zapíše *rozvrh neexistuje* (Schedule doesn’t exist).

5.2. implementační poznámky

Pokud budeme program používat prvním nebo druhým způsobem, vždy budou překládány jen ty části kódu, které daný způsob vyžaduje. Implementace algoritmu byla rozvržena do osmi souborů s příponou *c*. Hlavní soubor, ve kterém je umístěna hlavní funkce programu *main* se jmenuje **BruckerBaB.c**. V tomto souboru je kód funkce *mexFunction*, která je volána programem MATLAB. Ve funkci *main* je implementována beta procedura, která v příslušných místech programu volá funkci *alfa_procedure*. Hned po spuštění *main* proběhne inicializacační funkce *init_data* a sekvence funkcí pro alokaci paměti. Každý soubor s příponou *c* má pro své dynamické proměnné alokováno místo pomocí funkcí, které jsou implementovány ve stejnojmenném souboru a volány právě z funkce *main*. Všechny alokace paměti dynamických polí, o kterých víme, jaká bude jejich velikost, jsou ošetřeny proti nedostatku paměti. Dynamická pole mají velikost závislou na počtu zadaných úloh. Dále funkce *main* spustí *immediate_selection* a následně kód beta procedury. Nakonec proběhnou funkce, které uvolňují paměť. Struktura celého programu rozloženého na jednotlivé soubory je na obrázku 5.3.



Obr. 5.3

Všechny soubory mají také hlavičkový soubor s příponou *h*.
V následujících odstavcích popíšeme úlohu jednotlivých zdrojových souborů.

Soubor **InitFunction.c** obsahuje načítání dat ze souboru a ukládání výsledků do souboru **data.txt** (platí pouze pro 2. způsob spouštění), výpočet *UB*, výpočet *LB*, rozšíření matice **W** na **W'** a **p** na **p'**. Dále pak pomocné výpisy na obrazovku pro ladění, které se zapínají nadefinováním konstanty *PRINT*. Při ladění je také užitečné zakázat rozšíření **W** a **p** pomocí konstanty *SMALL*. Volání funkcí z tohoto souboru naplní globální proměnné:

- *n* - počet uzlů rozšířeného grafu *G*'
- *W_matrix* - dvourozměrné pole matice **W'**
- *processing_time* - jednorozměrné pole vektoru **p'**

Soubor **alfa.c** je jádrem celého programu. Zde je implementována metoda větví a mezí (branch and bound algorithm), která využívá strukturu *disjPairList* $\{i, j, l_{ij}\}$. Struktura je na začátku prázdná a v průběhu větvení se naplňuje a vyprazdňuje. Obsahuje metodu *reconstruct_actual_W*, která zkopíruje **W'** do proměnné $\mathbf{W}_\alpha$ (zde je implementována jako dvourozměrné pole *W_reconstruct*) a při každém zavolání modifikuje $\mathbf{W}_\alpha$ přesně podle struktury disjPairList. Vstupem této metody jsou *n*, *W_matrix*, *processing_time*. Výstupem je **s'**, což je sestavený optimální rozvrh o délce *n* v jednorozměrném poli *schedule_min*.

Soubor **selection.c**, jak naznačuje název, obsahuje funkci *immediate_selection*. Výhodou této funkce je kromě popisovaného v kapitole 4.3 ještě detekce kladných cyklů ve vstupním grafu G'. Detekci provádí právě Floydův algoritmus, který je implementován jako samostatná funkce, a je pro tento účel nepatrně modifikován [6].

Soubor **NewPair.c** obsahuje funkci *new_disjunctive_pair*, která se skládá ze dvou hlavních částí:

- nalezení všech disjunktních párů
- nalezení nejvhodnějšího disjunktního páru

Dále pak obsahuje pomocné funkce *calc_f_ik*, *median*, *min_value*. U první zmiňované funkce je třeba zmínit ošetření při odečítání $-\infty$. Jako $+\infty$ u reálných čísel je používána konstanta *FLT_MAX* a $-\infty$ nahrazuje konstanta -*FLT_MAX*. U celých čísel je vytvořeno takzvané symetrické nekonečno. Kladné nekonečno je nahrazeno *INT_MAX* a záporné reprezentujeme jako konstantu *INT_MIN* + 1. Konstanta je definována jako *MINUS_INF* ve všech souborech pomocí hlavičky **definitions.h**. Výhoda symetrického nekonečna je ta, že usnadní některé výpočty. Například pokud sčítáme *INT_MIN* a *MINUS_INF*, je výsledek 0. To je pro výpočty příznivé. Konstanty jsou definovány ve standardní knihovně jazyka ANSI C **limits.h** a **float.h**.

V Souboru **schedule.c** je umístěna funkce *get_schedule*, která z matice $\mathbf{W}_\alpha$ sestaví rozvrh. Podmínkou je, aby matice $\mathbf{W}_\alpha$ neobsahovala disjunktní páry, a tudíž bylo pořadí úloh jednoznačné. Funkce vrací rozvrh **s'** reprezentován jako jednorozměrné pole *schedule*.

Soubor **cycle.c** obsahuje test na kladný cyklus. Modifikovaný Dijkstrův algoritmus, který hledá nejdelší cesty je rozdělen do tří funkcí. Hlavní funkce je *not_positive_cycle*. Tato funkce volá ještě dvě pomocné funkce *remove_element* a *add_element*. Funkce *remove_element* odebírá prvek z množiny M a *add_element* prvek přidává. Proměnné M a U jsou jednorozměrná pole, která se indexují v tomto souboru od 1 do n+1. Dále je možné také použít naimplementovaný test *not_positive_cycle_floyd*. Jeho efektivita je sice ve většině případů nižší, než efektivita Dijkstrova algoritmu, ale pro ladící a experimentální účely je možné ho využít.

Poslední soubor **horn.c** obsahuje Hornův algoritmus, který je znázorněn pomocí vývojového diagramu na obrázku 4.7. Jako vstupní proměnné slouží matice *W_reconstruct* a vektor *processing_time*. Vektory *release_date* a *dead_line* jsou statické globální proměnné a jsou počítány při každém spuštění Hornova algoritmu. Vstupní data funkce *horn_test* jsou stejná jako u *not_positive_cycle*, ale zde přidávanou hranu s váhou l_{ij} zapíšeme přímo do *W_reconstruct* ($\mathbf{W}_\alpha$). Poté spustíme *horn_procedure*. Na konci testu je nutné původní hranu vrátit zpět do $\mathbf{W}_\alpha$.

6. Experimentální výsledky

Nejprve bylo třeba program dobře odladit a otestovat na chyby. Jakmile byl program připraven, mohl být testován na rychlost výpočtu. Program může být spuštěn v několika módech. V programu je možno pomocí podmíněných překladů odpojit některé funkce. Odpojit lze funkci *immediate_selection* a *horn_test*. Pokud je nepatrně upravena funkce *alfa_procedure*, lze jí volat přímo bez použití *beta procedury* z hlavní funkce *main*. Stav algoritmu je následující:

- beta procedura je kompletní
- alfa procedura má odpojen *horn_test*
- funkce okamžitého výběru odpojena

Hornův test není totiž pro rychlost programu dost efektivní. Jeho efektivnost je předpokládána ve spojení s *immediate_selection*. Druhá část *immediate_selection* není implementována.

Testy byli prováděny na PC Intel Pentium 4 2.4GHz, 786MB RAM. Jako testovací prostředí byl použit program Matlab 6.5 a program přeložen ve Visual C++ 6.0.

Náhodná data generujeme v určitém intervalu a s určitou pravděpodobností. Počet kladných hran označíme ne_+ , počet záporných hran potom bude ne_- . Hrany přidáváme do grafu náhodně, ale tak, aby nevznikl kladný cyklus. Uvedeme zde v jakých intervalech se generují proměnné p_i, l_{ij}, l_{ij}.

- doba vykonávání p_i byla vybrána z rovnoměrného diskrétního rozložení v intervalu <1,20>

- délka kladných hran s váhou l_{ij} byla vybrána z rovnoměrného diskrétního rozložení v intervalu <1,20>

- délka záporných hran s váhou l_{ij} byla vybrána z rovnoměrného diskrétního rozložení v intervalu <1,50>

Testovací algoritmus náhodných instancí popíšeme následujícím pseudokódem.

```
W_ij = -∞ , ∀i∈{1,…,n}, ∀i∈{1,…,n} a i≠j
W_ii = 0 , ∀i∈{1,…,n}
p_i = rand(<1,20>), ∀i∈{1,…,n}

počet_hran=0
while( počet_hran < ne_+ )
{
        i = rand(<1,n>);
        j = rand(<1,n>);
        if(i==j nebo W_ij ≠ -∞ ) continue;
        l_ij = rand(<1,20>);

        if( not_positive_cycle(W, i, j, l_ij) )
        {
                W_ii = l_ij ;
                počet_hran= počet_hran + 1;
        }
}

 počet_hran=0
while( počet_hran < ne_- )
{
        i = rand(<1,n>);
        j = rand(<1,n>);
        if(i==j nebo W_ij ≠ -∞ ) continue;
        l_ij = - rand(<1,50>);

        if( not_positive_cycle(W, i, j, l_ij) )
        {
                W_ii = l_ij ;
                počet_hran= počet_hran + 1;
        }
}
```

Experimenty:

Experimenty algoritmu Brucker budeme porovnávat s algoritmem ILP (Integer Linear Programming), který je popsán v literatuře [5].

1. experiment:

Průměrná doba rozvrhování $\bar{t}_{CPU}$ (500 instancí na jedno n). Počet kladných hran $ne_+ = (n^2 - n)/8$ (12,5% všech hran v G). Počet záporných hran $ne_- = (n^2 - n)/8$ (12,5% všech hran v G). Výsledky jsou v Tab. 1.

n [-]	8	10	12	14	16
$\bar{t}_{CPU}$ [s] (Brucker)	0.009498	0.06905	0.24386	0.75551	1.9928
$\bar{t}_{CPU}$ [s] (ILP)	0.0414	0.11216	0.089022	0.10521	0.13408

Tab. 1

Na obrázku 6.1 je porovnání algoritmů z Tab. 1, kde porovnáváme průměrné časy výpočtu v závislosti na n, kde n je počet úloh. Na obrázku 6.2 je závislost počtu rozvržených instancí na době výpočtu pro různá n. Počet instancí je udáván v [%], kde 100% je v našem případě 500 instancí.

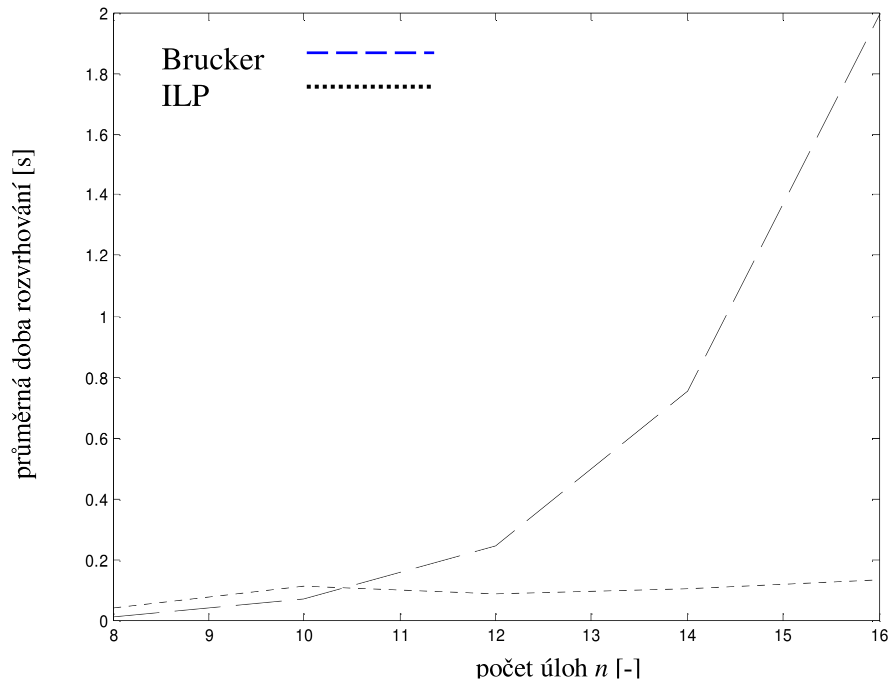
2. experiment:

Průměrná doba rozvrhování (500 instancí na jedno n). Počet kladných hran $ne_+ = 2 \cdot n$. Počet záporných hran $ne_- = n$. Výsledky jsou v Tab. 2.

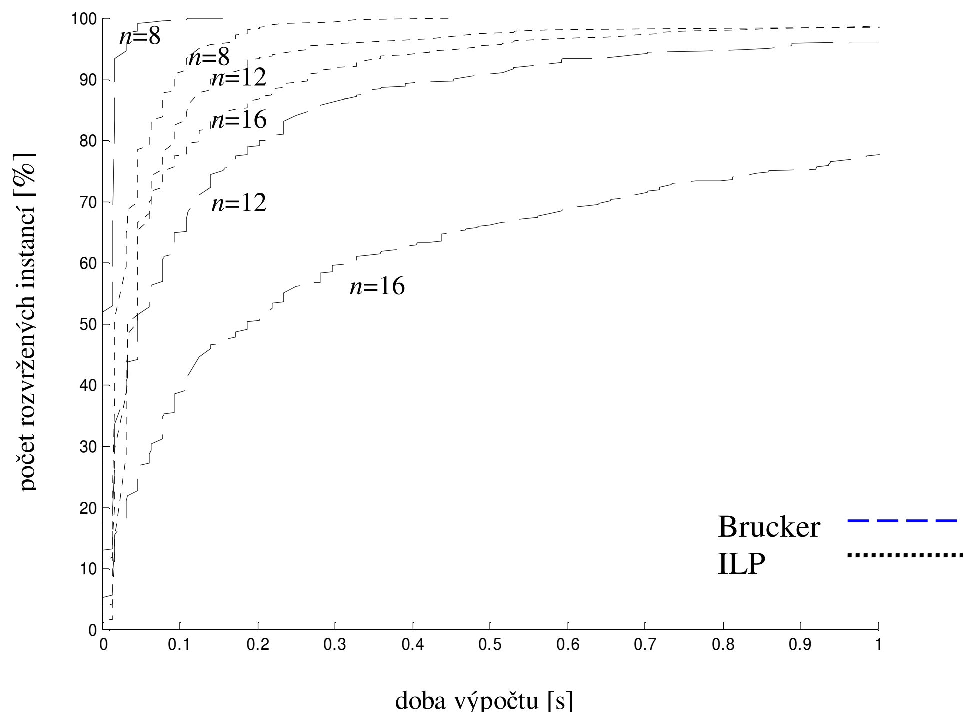
n [-]	8	10	12	14	16
$\bar{t}_{CPU}$ [s] (Brucker)	0.001616	0.005312	0.045386	0.23142	3.5956
$\bar{t}_{CPU}$ [s] (ILP)	0.00566	0.009646	0.019386	0.05492	0.22134

Tab. 2

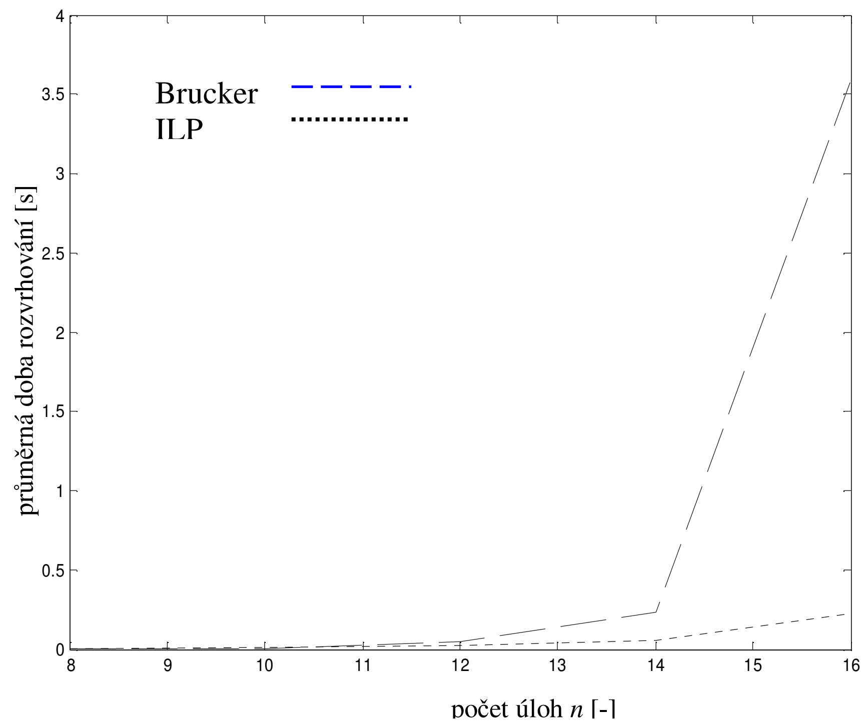
Na obrázku 6.3 je porovnání algoritmů z Tab. 2, kde porovnáváme průměrné časy výpočtu v závislosti na n. Na obrázku 6.4 je závislost počtu rozvržených instancí na době výpočtu pro různá n. Pokud hodnota v grafu dosáhne 100%, bylo vyřešeno všech (v našem případě) 500 instancí problému.



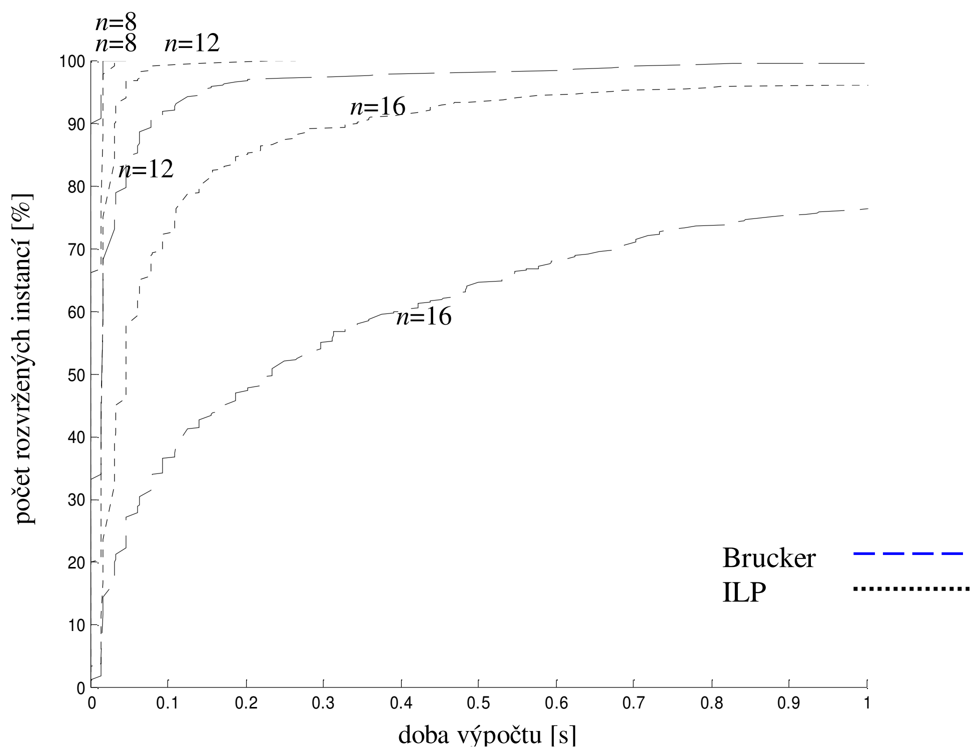
Obr. 6.1



Obr. 6.2



Obr. 6.3



Obr. 6.4

7. Závěr

V zimním semestru akademického roku 2005/2006 jsem si vybral toto téma jako Bakalářskou práci. V tomto semestru jsem se také teoreticky seznámil s modelem podle B. Roy [4], který využívá relativní omezení, a s některými algoritmy, které jsou na tomto modelu založeny. Začátkem letního semestru akademického roku 2005/2006 jsem začal algoritmus profesora Petera Bruckera [1] implementovat do jazyka ANSI C. Jako první část algoritmu byla naimplementována alfa procedura. Dále pak první část funkce okamžitého výběru. Druhá část funkce okamžitého výběru není v jazyce ANSI C implementována. Důvod je ten, že tato část je příliš složitá a Profesor Brucker ve své publikaci odkazuje do jiných článků, ze kterých se dá jen těžko zrekonstruovat přesné chování této funkce. Následně byl takto implementovaný algoritmus laděn a testován na chyby. Tyto testy byli prováděny v prostředí programu Matlab 6.5 pomocí náhodných sekvencí vstupních dat. Pro kontrolu správnosti výpočtu byl použit algoritmus ILP [5]. Po odladění všech chyb byla dodělána beta procedura. Když porovnáme výsledky efektivity výpočtů programů *BruckerBaB* a *ILP* zjistíme, že *BruckerBaB* je rychlejší než *ILP* v intervalu <1,10> úloh. A v intervalu <11,+∞> je program *ILP* rychlejší, než program *BruckerBaB*. Předpokládám, že pokud by se povedlo naprogramovat celou funkci okamžitého výběru, byl by program Brucker výrazně rychlejší. Tento algoritmus bude součástí TORSCHE Scheduling Toolbox pro Matlab a bude součástí příští verze tohoto nástroje.

Literatura

[1] P. Brucker, T. Hilbig, and J. Hurink. A branch and bound algorithm for a single-machine scheduling problem with positive and negative time-lags. Discrete Applied Mathematics, 94(1-3):77–99, May 1999.

[2] J. Hurink and J. Keuchel. Local search algorithms for a single-machine scheduling problem with positive and negative time-lags. Discrete Applied Mathematics, 112(1-3):179–197, 2001.

[3] A. M. Kordon. Minimizing makespan for a bipartite graph on a single processor with an integer precedence delay. Operations Research Letters, 32(6):557–564, November 2004.

[4] B. Roy. Contribution de la théorie des graphes à l'étude de certains problèmes linéaires. C. R. Acad. Sci. Paris, 248:2437–2439, 1959.

[5] P. Šůcha and Z. Hanzálek. Scheduling of Tasks with Precedence Delays and Relative Deadlines - Framework for Time-optimal Dynamic Reconfiguration of FPGAs. In WPDRTS'06 IEEE Workshop on Parallel and Distributed Real-Time Systems, April 2006.

[6] J. Demel. Grafy a jejich aplikace. ACADEMIA Praha, leden 2002.

[7] J. Błażewicz, K.H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz, Scheduling Computer and Manufacturing Processes. 2nd. Springer-Verlag New York, Inc. 2001

[8] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling theory: a survey. Ann. Discrete Math. 5. 287-326. 1979.

[9] J. Błażewicz, J. K. Lenstra, and A. H. Rinnooy Kan. Scheduling subject to resource constrains: classification and complexity. 5. Ann. Discrete Math. 11-24. 1933.

[10] E. D. Wikum, D. C. Llewellyn, and G. L. Nemhauser. One-machine generalized precedence constrained scheduling problems. Operations Research Letters, 16:87 - 89, 1994.

[11] P. Šůcha, M. Kutil, M. Sojka and Z. Hanzálek. TORSCHE Scheduling Toolbox for Matlab, IEEE International Symposium on Computer-Aided Control Systems Design, Munich, Germany, October, 2006.

Příloha 1 – Notace podle Graham-Błażewicz

Pro klasifikaci problémů v rozvrhování se používá klasifikace podle Graham-Błażewiczovi [8, 9] notace. Notace se skládá ze tří prvků $\alpha \mid \beta \mid \chi$. Symbol α popisuje procesory, symbol β charakterizuje rozvrhovací problém a symbol χ specifikuje optimalizační kritérium. Nejčastěji používané prvky jsou:

Prvek α:

Prvek	***Význam***
1	jeden procesor (single processor)
P	paralelní identické procesory (identical processors)
Q	různě rychlé procesory (uniform processors)
J	dedikované procesory – „dílna“ (job shop)

Prvek β:

Prvek	***Význam***
r_j	úlohy jsou omezeny termínem dostupnosti (release date)
d_j	úlohy jsou omezeny nejpozdější dobou dokončení (deadline)
$p_j = p$	všechny úlohy mají stejnou dobu vykonávání (processing time)
prec	úlohy jsou omezeny relacemi následností (precedence constraints)
pmtn	vykonávání úloh je možné přerušit (preemption)

Prvek χ:

Prvek	***Význam***
C_{max}	kritériem je délka rozvrhu (makespan)
ΣC_j	kritériem je minimalizace sumy časů dokončení jednotlivých úloh
L_{max}	kritériem je maximální zpoždění (maximum lateness)

Příklad:

$1 \mid r_j ; pmtn \mid L_{max}$ – Problém na jednom procesoru, kde úlohy jsou omezeny termínem dostupnosti a je možné je přerušit. Cílem je najít rozvrh s nejmenším kritériem $L_{max} = \max\{L_j\}$.