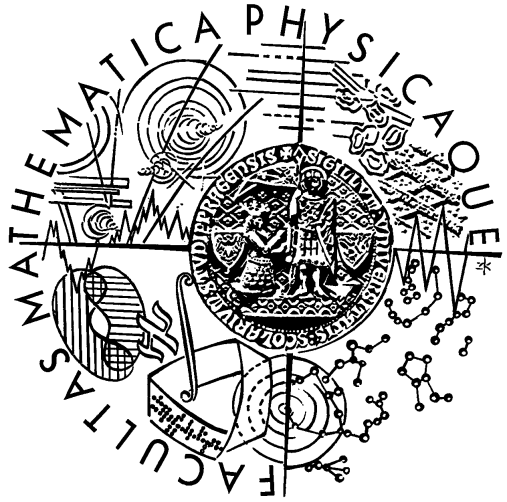
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Petr Vilím

Globální podmínky

Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce: **RNDr. Roman Barták, Ph.D.**

Studijní program: **Informatika**
Verze: 16. srpna 2001

Praha 2001

Poděkování

Na tomto místě bych chtěl poděkovat svému vedoucímu diplomové práce RNDr. Romanu Bartákovi, Ph.D. za řadu podnětných nápadů, pomoc při odstraňování chyb, za náměty pro zdokonalování práce a mnoho cenných rad.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 16. srpna 2001 Petr Vilím

Obsah

1 Úvod **3**

2 Programování s omezujícími podmínkami **4**
2.1 Úvod 4
2.2 Globální podmínky 5

3 Existující globální podmínky **7**
3.1 Alldifferent 7
3.2 Symetrická podmínka alldifferent 11
3.3 Podmínka kardinality 11
3.4 Hamiltonovská kružnice 13
3.5 Rozvrhování 15
3.6 Edge-Finding 17
3.6.1 Pravidla 18
3.6.2 Volba množiny Ω 19
3.6.3 Algoritmus 20
3.7 Not-first/not-last 21
3.8 Intervaly úkolů 25
3.8.1 Splnitelnost intervalu úkolů 25
3.8.2 Pravidla not-first a not-last 26
3.8.3 Pravidlo vyloučení 27
3.8.4 Udržování intervalů 28
3.8.5 Porovnání s edge-finding a not-first/not-last 29
3.9 Energetic reasoning 29
3.9.1 Pravidlo splnitelnosti 31
3.9.2 Graf funkce $W(i, t_1, t_2)$ 31
3.9.3 Relevantní intervaly $\langle t_1, t_2 \rangle$ 34
3.9.4 Algoritmus 37

4 Dynamické globální podmínky **39**
4.1 Motivace 39
4.2 Dynamizace globální podmínky 40
4.3 Dynamická podmínka alldifferent 41
4.3.1 Výsledek 42

5 Dávkové zpracování **44**
5.1 Definice 44
5.2 Délka přechodů 46
5.3 Délka zpracování 47
5.4 Algoritmus edge-finding 47
5.5 Současné zpracování úkolů 54
5.5.1 Spojování úkolů 54
5.5.2 Rozdělení úkolů 54
5.6 Algoritmus not-before/not-after 56
5.7 Algoritmus not-first/not-last 57
5.8 Spojování posloupností 60
5.9 Výsledek 65

6 Závěr **69**

Kapitola 1

Úvod

Cílem této práce je seznámit čtenáře s použitím globálních podmínek v programování s omezujícími podmínkami a navrhnout nové podmínky a jejich propagační algoritmy.

Práce nepředpokládá, že už je čtenář s programováním s omezujícími podmínkami seznámen. Proto je následující kapitola věnována stručnému úvodu do programování s omezujícími podmínkami – hlavně řešení problému pomocí propagačních technik.

V další kapitole jsou popsány některé existující globální podmínky a jejich propagační algoritmy. Protože globální podmínky se nejčastěji používají v rozvrhovacích problémech, je jim věnována větší část kapitoly. V kapitole edge-finding přináším důkaz ekvivalence dvou existujících algoritmů, v kapitole energetic reasoning pak nové – užší vymezení intervalů, pro které je třeba podmínku testovat.

V další kapitole vysvětlím myšlenku dynamických podmínek navrženou v [1], konkrétně ukážu dynamickou podmínku alldifferent a předvedu výhodnost její použití v praxi.

V kapitole dávkové rozvrhování pak upravím dva existující algoritmy a navrhnu dva zcela nové propagační algoritmy pro dávkové rozvrhování. Na konci kapitoly je pak měření výsledného algoritmu pro konkrétní testovací příklady.

Některé nové podmínky jsem naiplmentoval pro SICStus prolog. Zdrojové texty jsou na přiloženém CD.

Kapitola 2

Programování s omezujícími podmínkami

2.1 Úvod

Programování s omezujícími podmínkami (*constraint programming – CP*) se ukázalo jako metoda vhodná pro řešení mnoha praktických problémů jako je rozvrhování a plánování výroby, řízení letištního provozu apod.

Základní myšlenkou CP je popsat problém deklarativně pomocí podmínek (*constraints*) nad proměnnými (*variables*). Řešením je potom takové ohodnocení proměnných, které splňuje všechny zadané podmínky. Každá proměnná má přiřazenu množinu hodnot, kterých může nabývat. Tuto množinu budeme nazývat doména (*domain*). V našem případě bude doména vždy konečná (*finite domain constraint programming*) a většinou půjde o množinu celých čísel. Podmínka vyjadřuje souvislosti mezi podmnožinou proměnných – určuje, které kombinace hodnot proměnných jsou a nejsou přípustné. Podmínka může být zadána explicitně jako výčet všech správných kombinací hodnot, nebo implicitně jako matematické vyjádření nějaké relace (např. $x > 8$).

Problémy s omezujícími podmínkami se nejčastěji řeší propagací podmínek (*constraint propagation*), Propagace je metoda jak jednotlivé podmínky "spolupracují" při hledání řešení. Každá podmínka se snaží co nejvíce omezit domény svých proměnných (*domain filtering*) tak, že z domén odebírá hodnoty, kterých proměnná nemůže nabývat v žádném řešení této podmínky (tzv. nekonzistentní hodnoty). Které hodnoty to jsou a jak je nalézt závisí na konkrétní podmínce. Kdykoliv se změní doména některé z proměnných, všechny podmínky, které s touto proměnnou pracují, se "probudí" a snaží se dále omezit domény dalších proměnných. Podle toho, jestli umíme nalézt pro danou podmínku všechny nekonzistentní hodnoty mluvíme o *úplné* případně

neúplné propagaci (pro některé podmínky je nalezení všech nekonzistentních hodnot NP-těžký problém).

Např. pro proměnné $x, y \in \{1 \dots 15\}$ a podmínky $x < y$ a $x > 8$ propagace vyvodí, že doména proměnné y je $\{10 \dots 15\}$.

Propagací podmínek, jak jsme ji právě popsali, se dosáhne tzv. hranové konzistence (*arc consitency*). Existují i vyšší formy konzistence, jako *path consitency* nebo *k-consistency*. Ty jsou ale mnohem více časově náročné, proto se většinou nevyplatí.

Samotná propagace podmínek samozřejmě většinou nenajde řešení a musí být kombinovaná s nějakou metodou prohledávání prostoru řešení (např. *backtracking*, *branch and bound*). V každém kroku prohledávání vybereme jednu z proměnných a postupně jí zkoušíme přiřadit všechny hodnoty z její domény. Po každém takovém přiřazení se opět spustí propagace (nastala totiž změna domény na jednoprvkovou množinu), která dále omezí prostor řešení, nebo dokonce odhalí chybu (některá z domén je prázdná množina). Rychlost s jakou řešení nalezneme závisí samozřejmě i na tom, v jakém pořadí přiřazujeme hodnoty proměnným.

V této práci se budu zabývat algoritmy vyhledávání nekonzistentních hodnot pro některé globální podmínky.

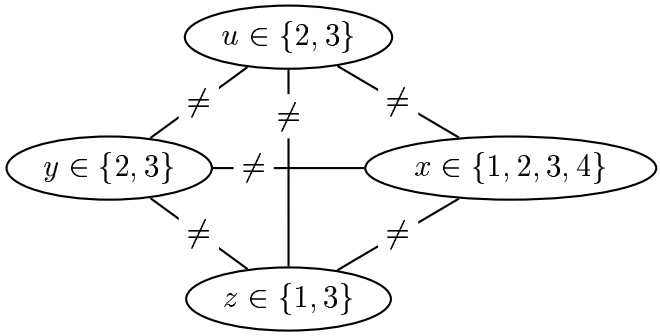
2.2 Globální podmínky

Globální podmínka je nahrazení sady jednoduchých podmínek jednou větší podmínkou. Často je globální podmínka tak komplexní, že rozepsat ji do více jednoduchých podmínek je poměrně komplikované. Globální podmínka "řeší podproblém" celého problému. Na rozdíl od jednoduchých podmínek se globální podmínka na podproblém dívá jako celek a využívá sémantiky podmínky, což většinou vede k lepší propagaci (tj. nalezneme a z domén odstraníme více nekonzistentních hodnot). Stejně dobré propagace bychom mohli dosáhnout i pomocí sady jednoduchých podmínek, ale museli bychom proto použít některou z pokročilejších (a časově mnohem náročnějších) konzistenčních technik. V neposlední řadě se podproblém snadněji popisuje jednou globální podmínkou, než sadou podmínek jednoduchých.

Většina jednoduchých podmínek má pevně danou aritu, např. už zmiňovaná podmínka $x < y$ je binární, $x > 8$ je unární. Naproti tomu globální podmínky většinou pevně danou aritu nemají, takže můžeme stejnou podmínku použít pro různý počet proměnných.

Často uváděný příklad globální podmínky je *alldifferent*. Tato podmínka říká, že každá z proměnných v dané množině $\{x_1, x_2, \dots x_n\}$ musí mít přiřazenu jinou hodnotu.

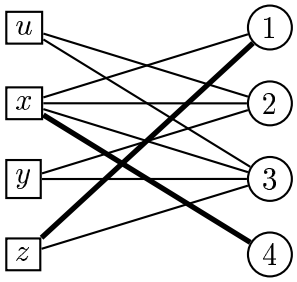
Uvažujme čtyři proměnné $u \in \{2,3\}$, $x \in \{1,2,3,4\}$, $y \in \{2,3\}$, $z \in \{1,3\}$. Podmínku alldifferent$\{u,x,y,z\}$ můžeme nahradit sadou šesti binárních podmínek $u \neq x$, $u \neq y$ atd, jak vidíme na obrázku:



Obrázek 2.1: Nahrazení alldifferent$\{u,x,y,z\}$ binárními podmínkami

Propagace přes binární podmínky neodhalí žádné nekonzistence. Vezměme si např. podmínku $u \neq z$. Proměnná u může nabývat hodnoty 2 protože je to v pořádku pro $z = 3$. Stejně tak může být $u = 3$, protože pak můžeme zvolit $z = 1$. Podobně z může být 1 i 3.

Naproti tomu alldifferent$\{u,x,y,z\}$ problém může chápat jako párování v bipartitním grafu:



Obrázek 2.2: Bipartitní graf alldifferent$\{u,x,y,z\}$

Vidíme, že $u \in \{2,3\}$ i $y \in \{2,3\}$, proto ani proměnná x ani proměnná z nemůže nabývat hodnoty 2 ani 3. A proto nutně $x = 4$ a $z = 1$.

Kapitola 3

Existující globální podmínky

3.1 Alldifferent

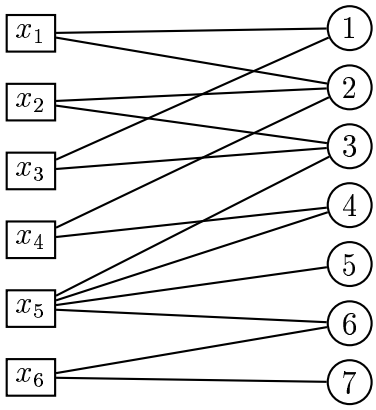
Podmínka alldifferent je jedna z nejpoužívanějších globálních podmínek. Např. při tvorbě školního rozvrhu chceme, aby každá třída měla ve stejnou dobu hodinu v jiné třídě a učil ji jiný učitel.

Podmínka alldifferent na množině $\{x_1, x_2, \ldots x_n\}$ říká, že každá z proměnných $x_1, x_2, \ldots x_n$ musí mít jinou hodnotu. Algoritmus pro propagaci této podmínky popsal Régin v [6], další algoritmy pro alldifferent můžeme najít v [17].

Definice 1 *Nechť $X = \{x_1, x_2, \ldots x_n\}$ je množina proměnných s doménami $D_1, D_2, \ldots D_n$. Sjednocení všech těchto domén označme D.* **Graf podmínky** alldifferent(**X**) *je bipartitní graf $G = (X, D, E)$, kde $E = \{(x_i, a) \mid a \in D_i\}$.*

Tedy vrcholy grafu jsou proměnné a hodnoty; hrany vedou mezi proměnnými a hodnotami, kterých může proměnná nabývat. Např:

$X = \{x_1, x_2, \ldots x_6\}$
$D_1 = \{1, 2\}$
$D_2 = \{2, 3\}$
$D_3 = \{1, 3\}$
$D_4 = \{2, 4\}$
$D_5 = \{3, 4, 5, 6\}$
$D_6 = \{6, 7\}$



Obrázek 3.1: Graf podmínky alldifferent

Pro naše účely ještě budeme potřebovat několik definicí z teorie grafů:

Definice 2 *Podmnožina hran M grafu G se nazývá* **párování**, *pokud žádné dvě hrany z této množiny nemají společný vrchol.* **Maximální párování** *je párování s největším počtem hran.* **Párování pokrývá množinu** *vrcholů Y pokud každý vrchol z Y je koncovým vrcholem jedné z hran v M.*

Každé přípustné ohodnocení proměnných z množiny X pak odpovídá párování pokrývající množinu X. Vzhledem k tomu, že se jedná o bipartitní graf, je toto párování také maximální (žádné párování nemůže být větší než $|X|$). A naopak, každé maximální párování velikosti $|X|$ pokrývá množinu X. Pokud je maximální párování menší než $|X|$, pak žádné přípustné ohodnocení neexistuje.

Našim úkolem je vyřadit z domén proměnných ty hodnoty, které nepoužívá ani jedno přípustné řešení, tj. nalézt všechny hrany, které nejsou ani v jednom maximálním párování velikosti $|X|$. K tomu nám pomůže následující věta:

Definice 3 *Nechť M je párování na grafu G. Všechny hrany z M se nazývají* **párovací**, *zbylé hrany grafu G jsou* **volné**. *Podobně vrchol grafu G je* **spárovaný**, *jestliže leží na některé z hran M, jinak je* **volný**. *Cesta nebo kružnice v grafu G se nazývá* **střídavá**, *jestliže se na ní pravidelně střídají párovací a volné hrany.*

Věta 1 *(Berge 1970) Hrana leží v některém ale ne ve všech maximálních párováních právě tehdy, když pro libovolné maximální párování M tato hrana leží na střídavé cestě sudé délky začínající ve volném vrcholu, nebo na střídavé kružnici sudé délky.*

Nejprve tedy musíme najít jedno maximální párování M, abychom větu vůbec mohli použít. Pak musíme o každé hraně rozhodnout, jestli leží na nějaké sudé střídavé cestě začínající ve volném vrcholu nebo na sudé střídavé kružnici.

Protože se jedná o bipartitní graf, je každá kružnice sudé délky. Navíc každá střídavá cesta liché délky z volného vrcholu musí začínat v nespárované hodnotě (proměnné jsou všechny spárované) a končit volnou hranou do spárované proměnné. Tuto cestu tudíž můžeme prodloužit o párovací hranu a dostaneme tak střídavou cestu sudé délky. O paritu střídavých cest a kružnic se tedy nemusíme starat.

Aby se nám tyto cesty a kružnice lépe hledaly, vytvoříme graf $\overrightarrow{G}$ zorientováním hran grafu G – párovací hrany mají směr z proměnné do hodnoty, volné opačný. Každá orientovaná cesta nebo kružnice v $\overrightarrow{G}$ odpovídá střídavé cestě nebo kružnici v původním grafu.

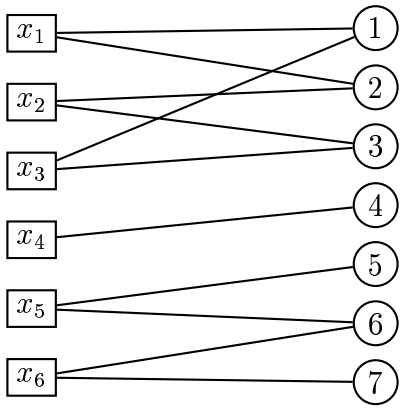
Hrana leží na nějaké střídavé cestě z volného vrcholu právě tehdy, když je v $\overrightarrow{G}$ dosažitelná z některého volného vrcholu. Všechny takové hrany tedy můžeme najít jedním průchodem grafu $\overrightarrow{G}$ do šířky z volných vrcholů.

Hrana leží na střídavé kružnici grafu G právě tehdy, když leží na kružnici v orientovaném grafu $\overrightarrow{G}$. Taková hrana má oba vrcholy ve stejné silně souvislé komponentě. Stačí tedy najít silně souvislé komponenty grafu $\overrightarrow{G}$.

Samotný algoritmus tedy je:

1. Vytvoř graf G.
2. Nalezni maximální párování M.
3. Pokud $|M| < |X|$, pak vrať *fail* a skonči.
4. Všechny hrany z $|M|$ označ jako použité
5. Vytvoř graf $\overrightarrow{G}$ z grafu G podle párování M.
6. Prohledej graf $\overrightarrow{G}$ do šířky ze všech volných vrcholů. Všechny dosažené hrany označ jako použitelné.
7. Najdi silně souvislé komponenty grafu $\overrightarrow{G}$. Všechny hrany, které mají vrcholy ve stejné komponentě souvislosti, označ jako použité.
8. Odstraň z domén všechny hodnoty, které odpovídají nepoužitým hranám.

$X = \{x_1, x_2, \ldots x_6\}$
$D_1 = \{1, 2\}$
$D_2 = \{2, 3\}$
$D_3 = \{1, 3\}$
$D_4 = \{4\}$
$D_5 = \{5, 6\}$
$D_6 = \{6, 7\}$



Obrázek 3.2: Graf po smazání zbytečných hran

Označme počet vrcholů n a počet hran m, tedy $n = |X| + |D|$ a $m = |D_1| + |D_2| + \cdots + |D_n|$. Kroky 1, 6 a 7 mají časovou složitost $O(n + m)$, kroky 4, 5 a 8 mají časovou složitost $O(m)$, krok 3 má složitost $O(1)$. Krok 2 má časovou složitost $O(\sqrt{|X|}m)$. Celková složitost propagačního algoritmu tedy je $O(\sqrt{|X|}m + n)$.

Právě jsme popsali první propagaci podmínky alldifferent. Až nějaká jiná podmínka zmenší doménu jedné z proměnných $x_1, x_2, \ldots, x_n$, podmínka alldifferent se znovu "probudí", aby opět zmenšila domény svých proměnných. Můžeme použít stejný algoritmus, můžeme si ale také pamatovat maximální párování "z minula", tedy z předchozí propagace. K tomu slouží v jazycích pro programování s omezujícími podmínkami *stav* podmínky. Jedná se o proměnnou, jejíž hodnota se při backtrackingu vrací zpět (je to tedy vlastně zásobník, aktuální hodnota je na vrcholu zásobníku, o ukládání vyzvedávání stavu se stará systém).

Když budeme mít štěstí, bude toto minulé párování maximální párování i nyní, nebo do maximálního párování bude chybět jen k hran (k hran z grafu odebraly jiné podmínky při propagaci). Pak můžeme maximální párování najít v čase $O(\sqrt{k}m)$.

Celý algoritmus má tedy složitost $O(\sqrt{|X|}m + n)$ při prvním zavolání a $O(\sqrt{k}m + m + n)$ v dalších propagacích.

Protože věta 1 je ve tvaru ekvivalence, zůstanou v grafu pouze hrany, které jsou v některém přípustném řešení. Propagace podmínky alldifferent je tedy úplná.

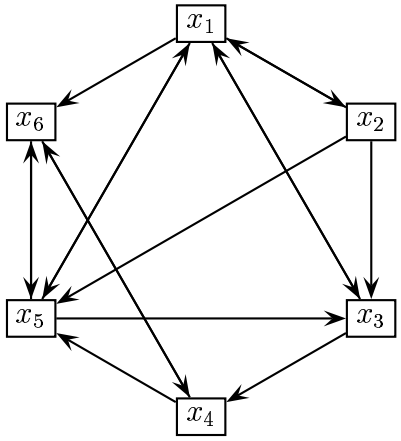
3.2 Symetrická podmínka alldifferent

Mějme množinu proměnných $X = \{x_1, x_2, \ldots, x_n\}$ jejichž domény jsou množiny proměnných, tj. $D_1, D_2, \ldots, D_n \subseteq X$. Symetrická podmínka alldifferent symalldiff(X) říká, že hodnoty $x_1, x_2, \ldots, x_n$ musí být různé, a navíc, když hodnota proměnné x_i je x_j, pak také hodnota proměnné x_j musí být x_i. Jedná se tedy o párování proměnných.

Tato podmínka je užitečná např. při rozvrhování sportovních utkání, kdy určuje, které týmy hrají proti sobě.

Každé přípustné ohodnocení proměnných je opět párování na grafu podmínky symalldiff, ten ale vypadá jinak než v případě podmínky alldifferent:

$X = \{x_1, x_2, \ldots x_6\}$
$D_1 = \{x_2, x_3, x_5, x_6\}$
$D_2 = \{x_1, x_3, x_5\}$
$D_3 = \{x_1, x_4\}$
$D_4 = \{x_5, x_6\}$
$D_5 = \{x_1, x_3, x_6\}$
$D_6 = \{x_4, x_5\}$



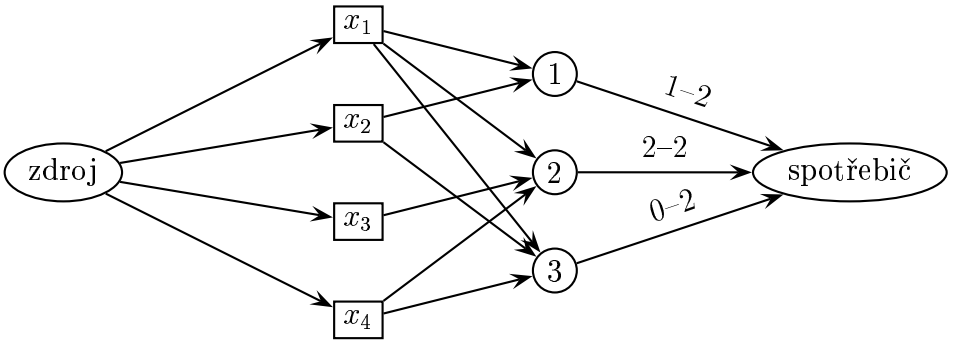
Obrázek 3.3: Graf podmínky symalldiff

Můžeme opět použít větu 1. Graf ale tentokrát není bipartitní, což hodně komplikuje situaci. V [8] jsou popsány dva algoritmy pro propagaci podmínky symalldiff. Prvý z nich je úplný a má časovou složitost $O(mn)$ díky problematickému hledání střídavých kružnic sudé délky. Druhý propagační algoritmus není úplný, ale má časovou složitost $O(m)$ na jednu smazanou hranu.

3.3 Podmínka kardinality

Mějme množinu proměnných $X = \{x_1, x_2, \ldots, x_n\}$. Sjednocení jejich domén $D_1, D_2, \ldots, D_n$ označme $D = \{v_1, v_2, \ldots v_d\}$. Podmínka kardinality požaduje, aby pro každé $i = 1, 2, \ldots d$ byl počet proměnných $x_1, x_2, \ldots, x_n$, které mají hodnotu v_i, v intervalu $\langle l_i, c_i \rangle$.

Algoritmus pro propagaci podmínky kardinality založený na tocích v sítích popsal Régin v [7].



$X = \{x_1, x_2, x_3, x_4\}$ $\quad l_1 = 1\ c_1 = 2$
$D_1 = \{1, 2, 3\}$ $\quad l_2 = 2\ c_2 = 2$
$D_2 = \{1, 3\}$ $\quad l_3 = 0\ c_3 = 2$
$D_3 = \{3\}$
$D_4 = \{2, 3\}$

Obrázek 3.4: Graf podmínky kardinality

Nejprve postavíme graf podmínky kardinality G. Vrcholy grafu jsou zdroj, spotřebič, proměnné a hodnoty. Ze zdroje vedou orientované hrany do všech proměnných, z každé hodnoty vede hrana do spotřebiče. Z proměnné x_i vede hrana do hodnoty v_j právě tehdy, když $v_j \in D_i$. Hrana z hodnoty v_i do spotřebiče má kapacitu c_i a dolní mez l_i. Zbylé hrany mají kapacitu 1 a dolní mez 0.

Každé přípustné ohodnocení proměnných $x_1, x_2, \ldots x_n$ odpovídá maximálnímu toku velikosti $|X|$ v grafu G a naopak (tok každou hranou musí být v intervalu dolní mez – kapacita). Potřebujeme vědět, které hodnoty nejsou použity ani v jednom přípustném řešení, tj. které hrany neleží na žádném maximálním toku velikosti $|X|$.

Věta 2 *Nechť f je maximální tok grafu G, (u, v) hrana G. Pro všechny maximální toky f' platí $f'(u, v) = f(u, v)$ právě tehdy, když hrana (u, v) ani (v, u) neleží na cyklu o alespoň třech vrcholech v modifikované [1] síti po toku f.*

Důkaz viz [7].

[1] Modifikovaná síť je termín používaný v algoritmech pro toky v sítích, např. v Dinicově algoritmu

My hledáme takové hrany (x_i, v_j), že pro každý maximální tok f' velikosti $|X|$ je $f'(x_i, v_j) = 0$. Najdeme tedy jeden maximální tok f, kandidáti budou ty hrany, pro které $f(x_i, v_j) = 0$. Všimněme si, že pro takové x_i a v_j je v modifikované síti po toku f jen hrana (x_i, v_j) a ne hrana (v_j, x_i), protože tok hranou (x_i, v_j) nelze zvětšit. Leží-li tedy hrana (x_i, v_j) na nějakém cyklu v modifikované síti po toku f, má tento cyklus alespoň 3 vrcholy. Hrana (v_j, x_i) na cyklu ležet nemůže, protože v modifikované síti ani není.

Hrana (x_i, v_j) leží na cyklu právě tehdy, když vrcholy x_i a v_j leží ve stejné komponentě silné souvislosti. Ty najdeme v čase $O(m+n)$ (n je počet vrcholů, m počet hran grafu G).

Nalezení maximálního toku Dinicovým algoritmem zabere čas $O(mn)$. Víme totiž, že maximální tok má velikost nanejvýš $|X| < n$, takže zlepšujících cest bude také nanejvýš $|X|$.

Při hledání maximálního toku v další propagaci můžeme opět využít maximální tok "z minula", kterému do maximálního toku chybí jen k. Nalezení maximálního toku pak bude trvat $O(km)$.

Tento propagační algoritmus je úplný.

3.4 Hamiltonovská kružnice

Mějme množinu proměnných $X = \{x_1, x_2, \ldots, x_n\}$ jejichž domény jsou množiny proměnných, tj. $D_1, D_2, \ldots D_n \subseteq X$. Stejně jako u symetrické podmínky alldifferent vytvoříme graf G. Podmínka hamilton(X) říká, že hrany x_i–hodnota x_i tvoří hamiltonovskou kružnici v G.

Nalézt všechny nekonzistentní hodnoty je v tomto případě NP-těžký problém. Kdybychom totiž uměli nalézt v polynomiálním čase všechny hrany, které neleží na hamiltonovské kružnici, uměli bychom také poznat, jestli daný graf obsahuje nějakou hamiltonovskou kružnici. A postupným odebíráním hran z grafu by nám pak zůstala samotná hamiltonovská kružnice.

Úplný propagační algoritmus proto nepřipadá v úvahu, protože v kombinaci s prohledáváním prostoru řešení nemá smysl použít propagační algoritmus s exponenciální časovou složitostí.

Popíšeme propagační algoritmus navržený v [9].

Hodnota x_i znamená následníka v hamiltonovské kružnici. Toho musí mít každý vrchol jiný, takže můžeme hned přidat podmínku alldifferent(X), jejíž časová složitost je $O(\sqrt{X}m)$.

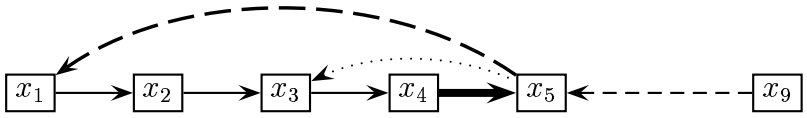
Pokud $|X| > 1$, tak jistě hodnota x_i nemůže být x_i. Tyto hodnoty můžeme z domény odebrat hned na začátku.

Dále se budeme snažit co nejdříve poznat, že hamiltonovská kružnice neexistuje. Když v grafu existuje hamiltonovská kružnice, musí být graf silně souvislý. Proto prohledávání prostoru řešení zastavíme, jakmile zjistíme, že graf není silně souvislý. Kontrola silné souvislosti zabere čas $O(m+n)$.

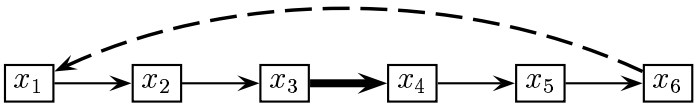
Můžeme také hledat hrany, jejichž odebráním by se silná souvislost porušila. Ty najdeme za $O(mn)$. Pro každou takovou hranu (u, v) můžeme odebrat všechny hrany (x, v), kde $x \neq u$, a hrany (u, y), kde $y \neq v$. Hledání těchto hran se ale většinou nevyplatí (zmenšení prostoru prohledávání je příliš malé na vynaložený čas).

Definice 4 *Cestu R v grafu G nazveme* **řetěz**, *pokud pro každou hranu $(u, v) \in R$ neexistuje v G hrana (u, x), kde $x \neq v$. Řetěz je tedy cesta, ze které se "nedá odbočit".*

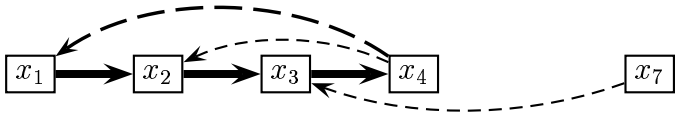
Pro každý řetěz R z vrcholu u do vrcholu v můžeme odebrat všechny hrany z v do všech vrcholů na řetězu R. Všechny tyto hrany až na hranu (v, u) už ale odebere podmínka alldifferent. Budeme si proto udržovat seznam všech řetězů a pokaždé, když vznikne nový řetěz, dva řetězy se spojí nebo se řetěz prodlouží, odebereme nadbytečné hrany. To zabere čas $O(n)$. Tento algoritmus je ale třeba iterovat tak dlouho, dokud se žádné další řetězy nespojí, celková složitost je tedy $O(kn)$, kde k je počet spojení (součet všech k v jedné větvi backtrackingu je menší nebo roven n).



Obrázek 3.5: Řetěz $(x1, x2, x3, x4)$ se prodloužil o hranu $(x4, x5)$. Můžeme odebrat hranu (x_5, x_1). Hranu (x_9, x_5) odebere podmínka alldifferent. Hranu (x_5, x_3) už podmínka alldifferent odebrala dříve.



Obrázek 3.6: Hrana (x_3, x_4) spojila řetězy (x_1, x_2, x_3) a (x_4, x_5, x_6). Můžeme odebrat hranu (x_6, x_1).



Obrázek 3.7: Vznikl nový řetěz (x_1, x_2, x_3). Můžeme odebrat hranu $(x4, x1)$. Hrany (x_4, x_2) a (x_7, x_3) odebere podmínka alldifferent.

3.5 Rozvrhování

Máme množinu úloh *(tasks)* T, každá úloha i má dánu dobu p_i jakou trvá její zpracování *(processing time)*, čas r_i kdy nejdříve může zpracování začít *(release time)* a čas d_i kdy musí nejpozději skončit *(due time)*. Máme k dispozici jeden zdroj *(resource)*, který je schopen úlohy zpracovávat. Podle zdroje rozlišujeme:

- **disjunktivní rozvrhování** *(Disjunctive Scheduling)* – zdroj nemůže zpracovávat více než jeden úkol zároveň.
- **kumulativní rozvrhování** *(Cumulative Scheduling)* – zdroj má kapacitu C. Každý úkol $i \in T$ má dánu potřebnou kapacitu c_i, kterou potřebuje po celou dobu zpracování *(required capacity)*. Součet kapacit c_i všech úkolů zpracovávaných v jeden okamžik nesmí přesáhnout kapacitu zdroje C.

Podle druhu úloh rozlišujeme:

- **preemptivní rozvrhování** *(Preemptive Scheduling)* – úkol nemusí být zpracován celý najednou, zpracování lze přerušovat. Většinou ne zcela libovolně, ale jsou dány nějaké další podmínky, kdy je dovoleno zpracování přerušit (např. minimální délka nepřerušitelného úseku, maximální počet přerušení, maximální doba přerušení apod.).
- **nepreemptivní rozvrhování** *(Non-Preemptive Scheduling)* – zpracování úkolu musí bez přerušení trvat celou dobu p_i. My se budeme zabývat výhradně tímto druhem rozvrhování.

Ve většině praktických problémů je zdrojů více než jeden a polotovar v daném pořadí musí projít zpracováním na několika strojích. Úkolem je rozvrhnout výrobu co nejlevněji, co nejrychleji skončit s celou výrobou *(minimum makespan)* apod. Příkladem takových problémů je *Job-Shop Scheduling Problem* ([11], [13], [5], [2]), *Resource-Constrained Project Scheduling Problem*

(RCPSP) ([13], [4]) nebo *Cumulative Scheduling Problem (CuSP)* ([13], [12], [2]).

My se budeme zabývat propagací globálních podmínek pouze pro jeden zdroj. Pokud je zdrojů více, je třeba také použít více podmínek.

Bylo vyvinuto několik algoritmů pro ověřování platnosti a splnitelnosti podmínek v nepreemptivním rozvrhování:

- **Edge-Finding** Pomocí tohoto algoritmu J. Carlier a E. Pinson jako první vyřešili některé benchmarkové problémy. Dnes se spíše používá ekvivalentní ale jednodušší algoritmus z [11]. Edge-Finding byl původně navržen pro disjunktivní rozvrhování, ale dá se přizpůsobit i pro kumulativní rozvrhování.
- **Intervaly úkolů** (*Tasks Intervals*). Tento algoritmus pro disjunktivní i kumulativní rozvrhování navrhli Y. Caseau a F. Laburthe v [2] a dále myšlenku rozvíjeli v [3], [4] a [5]. Kombinace Edge-Finding a Not-First/Not-Last tento přístup v podstatě překonává.
- **Not-First/Not-Last**. Další algoritmus pro disjunktivní rozvrhování představený v [12], dobře se doplňuje s Edge-Finding.
- **Elastic Relaxations** jsou dva algoritmy *Fully Elastic Relaxation of the CuSP* a *Partialy Elastic Relaxation of the CuSP* pro kumulativní rozvrhování založené na zjednodušení rozvrhovacích pravidel tak, že všechny platné rozvrhy podle původních pravidel jsou také platné rozvrhy podle nové verze pravidel. Vyřadíme-li tedy jisté hodnoty jako nemožné podle nových pravidel, jsou nemožné i podle původních pravidel. Podrobnosti viz [10] a [13].
- **Energetic Reasoning** nebo také **Left-Shift/Right-Shift** je vhodný pro kumulativní rozvrhování. Je pomalejší než Edge-Finding a Not-First/Not-Last, ale propaguje více změn a také dříve rozpozná nesplnitelnost podmínky.
- **Shaving** není samostatný algoritmus, ale metoda jak použít nějakou jinou podmínku a pomocí ní "ořezávat" okraje intervalu $\langle r_i, d_i \rangle$. Pro každou úlohu $i \in T$ zkusíme zafixovat provádění úkolu i od času r_i. Pokud podmínka signalizuje nesplnitelnost, tak můžeme r_i o jedna zvětšit a dále r_i "ořezávat". Podobně pro d_i.
  Shaving nad algoritmem Edge-Finding je popsán v [11].

Žádný z těchto algoritmů nedělá úplnou propagaci – úplná propagace je NP-těžký problém.

Ve všech algoritmech se budeme snažit zmenšovat interval $\langle r_i, d_i \rangle$ *(time window)*. Doménu D_i proměnné $t_i = \langle r_i, d_i - p_i \rangle$, která říká, kdy může začít zpracování úkolu i, tedy vždy budeme chápat jako interval. Zvětšení r_i nebo

zmenšení d_i bude odpovídat zmenšení domény D_i. Další propagace se spustí pouze tehdy, když se ještě více zvětší r_i nebo zmenší d_i. Když totiž nějaká jiná podmínka odebere z domény D_i hodnotu různou od $\min(D_i)$ a $\max(D_i)$, nemá cenu pouštět propagační algoritmus znovu, protože ten stejně hledí pouze na r_i a d_i.

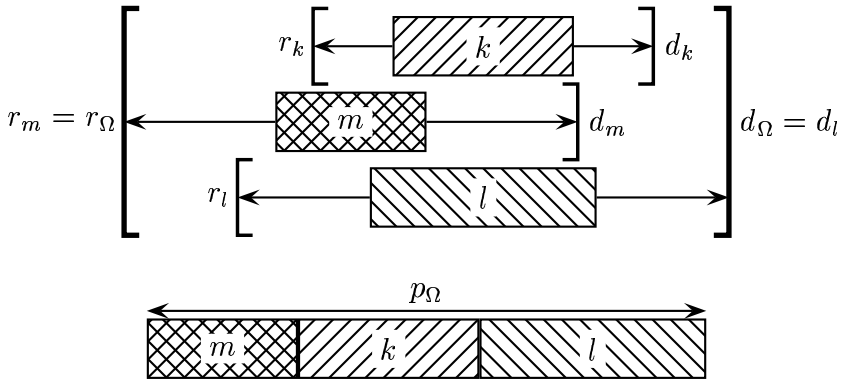
Dále už budeme mluvit jen o $\langle r_i, d_i \rangle$ místo o D_i.

3.6 Edge-Finding

V této kapitole popíšeme algoritmus edge-finding pro disjunktivní rozvrhování podle [11] a [12].

Nejprve si zavedeme další značení. Pro množinu úkolů $\Omega \subseteq T$ je r_Ω čas, kdy může začít zpracování prvního úkolu z Ω, d_Ω je čas, kdy nejpozději musí skončit všechny úkoly z Ω a p_Ω je celkový čas potřebný pro zpracování všech úloh z Ω:

$$
\begin{aligned}
r_\Omega &= \min\{r_i \mid i \in \Omega\} \\
d_\Omega &= \max\{d_i \mid i \in \Omega\} \\
p_\Omega &= \sum_{i \in \Omega} p_i
\end{aligned}
$$



Obrázek 3.8: Příklad množiny $\Omega = \{k, l, m\}$.

Fakt, že úkol i musí být zpracován před množinou úkolů Ω, budeme zapisovat $i \ll \Omega$. Podobně $\Omega \ll i$ znamená, že úkol i musí být zpracován až po všech úkolech z Ω.

3.6.1 Pravidla

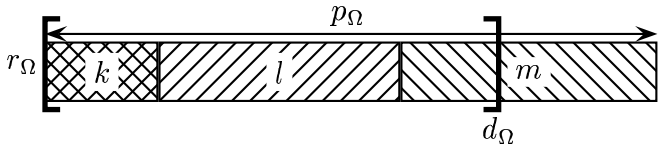
Edge-Finding je založen na následující úvaze:

$$\forall \Omega \subseteq T: \; d_\Omega - r_\Omega < p_\Omega \Rightarrow \text{fail} \tag{3.1}$$

$$\forall \Omega \subseteq T, \forall i \in T - \Omega: \; d_{\Omega\cup\{i\}} - r_\Omega < p_\Omega + p_i \Rightarrow i \ll \Omega \tag{3.2}$$

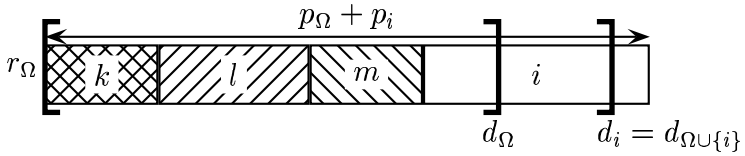
$$\forall \Omega \subseteq T, \forall i \in T - \Omega: \; d_\Omega - r_{\Omega\cup\{i\}} < p_\Omega + p_i \Rightarrow \Omega \ll i \tag{3.3}$$

První implikace říká, že pro každou množinu úloh Ω musí být v intervalu $\langle r_\Omega, d_\Omega\rangle$ dost času pro zpracování všech úkolů z Ω. Pokud je času méně, signalizujeme nesplnitelnost podmínky:



Obrázek 3.9: Příklad nesplnitelné množiny $\Omega = \{k, l, m\}$.

Druhá implikace pro libovolnou množinu Ω a úkol i, který v ní neleží, bere v úvahu interval $\langle r_\Omega, d_{\Omega\cup\{i\}}\rangle \supseteq \langle r_\Omega, d_\Omega\rangle$. Pokud se do tohoto intervalu úlohy z Ω a úloha i společně nevejdou, musí být úkol i zpracován už před Ω:



Obrázek 3.10: Úkol i nemůže být zpracován za množinou $\Omega = \{k, l, m\}$ ani v průběhu množiny Ω. Musí tedy být zpracován před množinou Ω. (Případ $d_i > d_\Omega$)

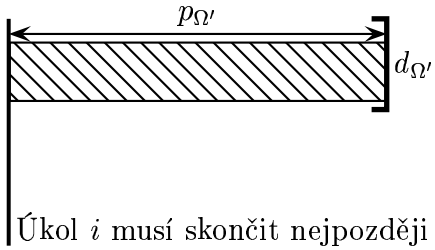
Třetí implikace je symetrická k druhé. Pokud se do intervalu $\langle r_{\Omega\cup\{i\}}, d_\Omega\rangle$ všechny úlohy z Ω dohromady s úlohou i nevejdou, musí být úkol i zpracován za všemi úkoly z množiny Ω.

Ze znalosti $i \ll \Omega$ nebo $\Omega \ll i$ můžeme vyvodit zmenšení intervalu $\langle r_i, d_i\rangle$:

$$i \ll \Omega \Rightarrow d_i \leq \min\{d_{\Omega'} - p_{\Omega'} \mid \Omega' \subseteq \Omega\} \tag{3.4}$$

$$\Omega \ll i \Rightarrow r_i \geq \max\{r_{\Omega'} + p_{\Omega'} \mid \Omega' \subseteq \Omega\} \tag{3.5}$$

Pokud totiž úkol i musí být vykonán před všemi úkoly z Ω, pak pro libovolnou množinu $\Omega' \subseteq \Omega$ musí úkol i skončit před $d_{\Omega'} - p_{\Omega'}$. Druhé pravidlo je symetrické.



Obrázek 3.11: Vysvětlení k pravidlu 3.4.

Na pravidlech 3.1–3.5 je založen algoritmus v článku [12]. V [11] je bez důkazu řečeno, že níže popsaný algoritmus dává stejné výsledky. My přinášíme důkaz tohoto tvrzení tím, že algoritmus přímo založíme na pravidlech 3.1–3.5.

3.6.2 Volba množiny Ω

Všech množin $\Omega \subseteq T$ je příliš hodně na to, abychom pro všechny testovali 3.1–3.3. Naštěstí není potřeba zkoušet všechny podmnožiny množiny T. Ukážeme, že stačí za Ω volit množiny ve tvaru:

$$[l, m] = \{k \mid k \in T \ \wedge \ r_k \geq r_l \ \wedge \ d_k \leq d_m\}$$

kde l a m jsou libovolné úkoly z množiny T (mohou být i stejné). V [2] se pro tyto množiny zavádí název *interval úkolů*. Takových množin je $O(n^2)$, kde n je počet úkolů.

Vezměme si libovolnou $\Omega \subseteq T$ a k ní vytvoříme množinu Ψ:

$$\Psi = \{k \mid k \in T \ \wedge \ r_k \geq r_\Omega \ \wedge \ d_k \leq d_\Omega\}$$

Množina Ψ má tvar intervalu úloh $[l, m]$, stačí zvolit za l, m ty úkoly z Ω, pro které $r_\Omega = r_l$ a $d_\Omega = d_m$.

Pro množinu Ψ platí:

$$\begin{aligned} \Psi &\supseteq \Omega \\ r_\Psi &= r_\Omega \\ d_\Psi &= d_\Omega \\ p_\Psi &\geq p_\Omega \end{aligned}$$

Zvolíme-li tedy v implikaci 3.1 místo množiny Ω množinu Ψ, tak pokud Ω způsobila fail, tak Ψ ho způsobí také.

Pokud v implikaci 3.2 zvolíme místo Ω množinu Ψ, v závislosti na volbě úkolu i mohou nastat nastat dvě možnosti:

- $i \notin \Psi$. Pokud implikace 3.2 vyvodí $i \ll \Omega$, tak pak stejná implikace ale pro množinu Ψ a úkol i vyvodí $i \ll \Psi$, což je ještě lepší výsledek, jak je vidět z pravidla 3.4.
- $i \in \Psi$ a přitom $i \notin \Omega$. Pak předpoklad implikace 3.2 neplatí, nebo podmínka 3.1 pro množinu Ψ vyvodí fail (když $i \in \Psi$ a $i \notin \Omega$, tak $p_\Psi \geq p_\Omega + p_i$). Pokud tedy nejdříve otestujeme všechny relevantní podmnožiny T na podmínku 3.1, nemůže předpoklad implikace 3.2 pro $i \in \Psi$ nikdy platit (fail znamená okamžité ukončení propagačního algoritmu).

3.6.3 Algoritmus

Ukážeme, jak v čase $O(n^2)$ ověřit platnost podmínky 3.1 a najít všechny zmenšení intervalů $\langle r_i, d_i \rangle$ vyplývajících z pravidla 3.3. Zmenšení intervalů vyplývající z pravidla 3.2 je symetrické k 3.3.

Zvolíme si pevně úkol $j \in T$. Položme $\Omega_0 = [j, j]$. Množina Ω_1 bude nejmenší interval úkolů, který je rozšířením Ω_0 doleva. Najdeme tedy úkol i s hodnotou $r_i = \max\{r_k \mid k \in T - \Omega_0 \wedge d_k \leq d_j\}$ a položíme $\Omega_1 = [k, j]$. Takto pokračujeme dál až vytvoříme posloupnost množin $\Omega_0 \subsetneq \Omega_1 \subsetneq \cdots \subsetneq \Omega_p$.

Když si úkoly dopředu setřídíme podle hodnoty r_i, budeme schopni vytvořit posloupnost $\Omega_0, \Omega_1, \ldots \Omega_p$ v čase $O(n)$ a ještě pro každou tuto množinu Ω_l spočítat hodnotu p_{Ω_l}. Ověření podmínky 3.1 tedy zabere čas $O(n^2)$ (máme n možností jak zvolit j).

Pro každou množinu Ω_l si ještě potřebujeme dopředu spočítat hodnotu $\max\{r_{\Omega'} + p_{\Omega'} \mid \Omega' \subseteq \Omega_l\}$ pro pravidlo 3.5. Uvažujme, jak se hodnota změní $\max\{r_{\Omega'} + p_{\Omega'} \mid \Omega' \subseteq \Omega_l\}$ oproti $\max\{r_{\Omega'} + p_{\Omega'} \mid \Omega' \subseteq \Omega_{l-1}\}$. V množině Ω_l přibyly pouze úkoly k takové, že $r_k = r_{\Omega_l} < r_{\Omega_{l-1}}$. Podmnožiny množiny Ω_l, které nejsou podmnožinou množiny Ω_{l-1}, jsou proto právě takové množiny $\Omega' \subseteq \Omega$, že $r_{\Omega'} = r_{\Omega_l}$. A abychom z těchto nových podmnožin dosáhli co největšího $r_{\Omega'} + p_{\Omega'}$, musíme zvolit $\Omega' = \Omega_l$. Dostáváme tedy rekurzivní výpočet:

$$\max\{r_{\Omega'} + p_{\Omega'} \mid \Omega' \subseteq \Omega_l\} = \max\{\max\{r_{\Omega'} + p_{\Omega'} \mid \Omega' \subseteq \Omega_{l-1}\}, r_{\Omega_l} + p_{\Omega_l}\} \tag{3.6}$$

Spočítat tyto hodnoty nám zabere čas $O(n)$.

V pravidle 3.3 je zbytečné volit i a Ω tak, že $d_\Omega \geq d_i$. Pokud totiž platí předpoklad implikace 3.3, tak pro množinu $\Omega \cup \{i\}$ vyvodí pravidlo 3.1 fail.

Množina úkolů $U \subseteq T$, pro které má cenu testovat pravidlo 3.3 pro množiny $[j, j] = \Omega_0, \Omega_1, \ldots, \Omega_p$ tedy je: $U = \{i \mid i \in T \wedge d_i \geq d_j\}$. Úkoly z množiny U setříděné vzestupně podle p_i označme $u_0, u_1, \ldots, u_q$.

Předpoklad implikace v pravidle 3.3 můžeme rozepsat jako konjunkci dvou výrazů:

$$d_\Omega - r_\Omega < p_\Omega + p_i \tag{3.7}$$
$$d_\Omega - r_i < p_\Omega + p_i \tag{3.8}$$

Uvažme nyní dvojici Ω_p a u_q:

- Pokud pro dvojici Ω_p, u_q neplatí 3.7, pak 3.7 neplatí ani pro Ω_p a $u_0, u_1, \ldots, u_{q-1}$, protože $p_{u_0} \leq p_{u_1} \leq \cdots \leq p_{u_q}$. Můžeme tedy množinu Ω_p odebrat z posloupnosti $\Omega_0, \Omega_1, \ldots, \Omega_p$.
- Pokud pro dvojici Ω_p, u_q neplatí 3.8, pak 3.8 neplatí ani pro úkol u_q a množiny $\Omega_0, \Omega_1, \ldots, \Omega_{p-1}$, protože $p_{\Omega_0} \leq p_{\Omega_1} \cdots \leq p_{\Omega_p}$ a $d_{\Omega_0} = d_{\Omega_1} = \cdots = d_{\Omega_p}$. Můžeme proto odebrat u_q z množiny U.
- Pokud pro dvojici Ω_p, u_q platí 3.7 i 3.8, můžeme podle 3.5 opravit r_{u_q}. I kdyby pro nějakou další $\Omega_e \subsetneq \Omega_p$ pravidlo 3.3 také vyvodilo $\Omega_e \ll u_q$, hodnotu r_{u_q} už to nezmění (to vyplývá z tvaru pravidla 3.5. Můžeme proto odebrat úkol u_q z množiny U.

Množin v posloupnosti $\Omega_0, \Omega_1, \ldots, \Omega_p$ je $O(n)$, úkolů v množině U také $O(n)$. V každém kroku (který trvá $O(1)$) odebereme buď jeden úkol z množiny U, nebo jednu množinu z posloupnosti $\Omega_0, \Omega_1, \ldots, \Omega_p$. Pro dané j tedy vyvodíme všechny důsledky z pravidla 3.3 v čase $O(n)$, pro všechny j to pak v čase $O(n^2)$.

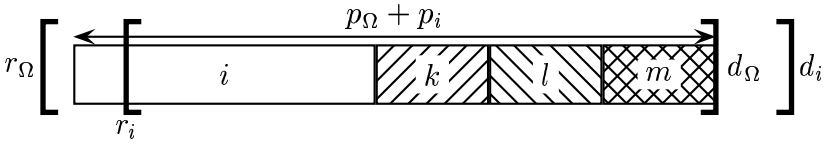
Jeden běh algoritmu edge-finding tedy trvá $O(n^2)$. Změny, které edge-finding nalezne, ale mohou způsobit, že další běh edge-finding nalezne ještě další změny. Proto se doporučuje edge-finding iterovat, dokud nevyvodí žádné další změny.

3.7 Not-first/not-last

Algoritmus not-first/not-last (viz. [12]) je založen na pravidlech:

$$\forall \Omega \subset T, \forall i \notin \Omega : \; d_\Omega - r_i < p_\Omega + p_i \Rightarrow i \not\ll \Omega \tag{3.9}$$
$$\forall \Omega \subset T, \forall i \notin \Omega : \; d_i - r_\Omega < p_\Omega + p_i \Rightarrow \Omega \not\ll i \tag{3.10}$$

Prvé pravidlo *not-first* říká, že pokud není v intervalu r_i – d_Ω dost času pro zpracování všech úkolů z Ω společně s úkolem i, pak úkol i nemůže být zpracován před všemi úkoly z Ω ($i \not\ll \Omega$). Podobně druhé pravidlo *not-last* rozpozná, kdy úkol i nemůže být vykonán až po všech úkolech z Ω ($\Omega \not\ll i$).



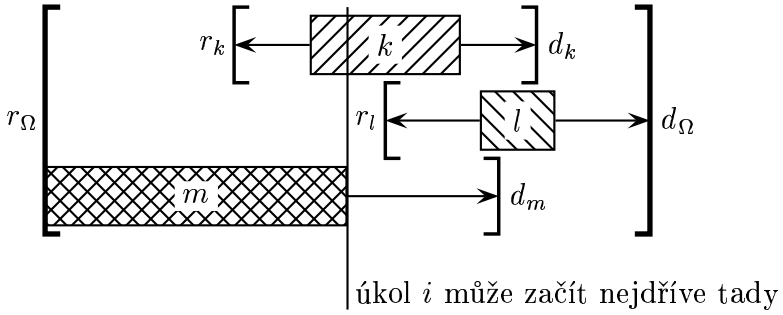
Obrázek 3.12: Úkol i nemůže být zpracován před množinou $\Omega = \{k, l, m\}$.

Z vlastností $i \not\ll \Omega$ a $\Omega \not\ll i$ můžeme odvodit zvětšení hodnoty r_i případně zmenšení hodnoty d_i:

$$i \not\ll \Omega \Rightarrow r_i \geq min\{r_j + p_j \mid j \in \Omega\} \tag{3.11}$$

$$\Omega \not\ll i \Rightarrow d_i \leq max\{d_j - p_j \mid j \in \Omega\} \tag{3.12}$$

Změna r_i podle 3.11 vychází z toho, že před úkolem i musí být vykonán alespoň jeden úkol $j \in \Omega$. Podobně 3.12 říká, že po skončení úkolu i musí být ještě dost času pro zpracování alespoň jednoho úkolu j z množiny Ω.



Obrázek 3.13: Změna r_i podle pravidla not-first 3.11

Ukážeme algoritmus z [12] pro pravidlo not-first. Tento algoritmus s časovou složitostí $O(n^2)$ a prostorovou složitostí $O(n)$ najde všechny změny, které plynou z pravidel 3.9 a 3.11.

Podobně jako u algoritmu edge-finding ukážeme, že v pravidle 3.9 není potřeba volit všechny kombinace množin Ω a úkolu i abychom z pravidel 3.9 a 3.11 vytěžili maximum.

Dále budeme předpokládat, že úkoly jsou seřazeny vzestupně podle d_i, tj. $i \leq j \Leftrightarrow d_i \leq d_j$. Pro dané úkoly $j, k \in T$ takové, že $j \leq k$ a $r_j + p_j \leq r_k + p_k$, označme $\Omega(jk)$ množinu:

$$\Omega(jk) = \{m \mid m \in T \ \wedge \ m \leq k \ \wedge \ r_j + p_j \leq r_m + p_m\}$$

Pokud j a k nesplňují $j \leq k$ nebo $r_j + p_j \leq r_k + p_k$, tak položíme $\Omega(jk) = \emptyset$ a $d_{\Omega(jk)} = \infty$. Pro libovolný úkol $i \in T$ budeme pod $\Omega(ijk)$ myslet množinu:

$$\Omega(ijk) = \Omega(jk) \setminus \{i\}$$

Pokud tedy $i \notin \Omega(jk)$, tak $\Omega(ijk) = \Omega(jk)$. Množina $\Omega(ijk)$ je největší množina taková, že $d_{\Omega(jk)} = d_k$ a pravidlo 3.11 by úkolu i posunulo hodnotu r_i na $r_j + p_j$ (pokud by platilo 3.9).

Věta 3 *Pokud pravidla 3.9 a 3.11 pro úkol $i \in T$ a množinu $\Omega \subset T$ změní hodnotu r_i na $r_j + p_j$, pak existuje úkol $k \in T$ takový, že $k \neq i$, $k \geq j$ a pravidla 3.9, 3.11 pro úkol i a množinu $\Omega(ijk)$ také změní hodnotu r_i na $r_j + p_j$.*

Důkaz: Zvolme $k = \max\{l \mid l \in \Omega\}$. Je zřejmé, že $k \geq j$, protože $j \in \Omega$. Dále $k \neq i$, protože $i \notin \Omega$, zatímco $k \in \Omega$. Z volby k vyplývá, že $\Omega \subseteq \Omega(ijk)$, $d_{\Omega(ijk)} = d_\Omega$ a $p_{\Omega(ijk)} \geq p_\Omega$. Protože pravidlo 3.9 platí pro Ω, platí i pro $\Omega(ijk)$ a pravidlo 3.11 pro $\Omega(ijk)$ tedy odvodí $r_i \geq r_j + p_j$. □

Označme:

$$\delta_{j,k} = \min\{d_{\Omega(jl)} - p_{\Omega(jl)} \mid l \in \{1, 2, \ldots k\}\}$$

Všechny hodnoty $\delta_{j,k}$ pro dané j můžeme spočítat v čase $O(n)$ ze vztahu:

$$\delta_{j,k} = \min\{d_{\Omega(jk)} - p_{\Omega(jk)},\ \delta_{j,k-1}\}$$

Věta 4 *Nechť i a j jsou dva různé úkoly. Pravidla 3.9 a 3.11 dovolí upravit r_i podle $r_i \geq r_j + p_j$ právě tehdy, když platí alespoň jedna z možností:*

1. $r_i + p_i < r_j + p_j$ *a* $\delta_{j,n} < r_i + p_i$
2. $r_i + p_i \geq r_j + p_j$ *a* $\delta_{j,n} < r_i$ *nebo* $\delta_{j,i-1} < r_i + p_i$

Důkaz: Podle věty 3 je možné hodnotu r_i opravit na $r_i \geq r_j + p_j$ právě tehdy, když existuje úkol k takový, že platí 3.9 pro množinu $\Omega(ijk)$ a úkol i. Rozlišíme dva případy:

1. $r_i + p_i < r_j + p_j$. Pak i nemůže ležet v žádné množině $\Omega(jk)$(z definice množiny $\Omega(jk)$). Proto $\Omega(ijk) = \Omega(jk)$. Opravit r_i v tomto případě můžeme právě tehdy, když existuje úkol k takový, že (upravená nerovnost z 3.9):

   $$d_{\Omega(jk)} - p_{\Omega(jk)} < p_i + r_i$$

   A takový úkol k existuje právě tehdy, když:

   $$\min\{d_{\Omega(jl)} - p_{\Omega(jl)} \mid l \in \{1, 2, \ldots, n\}\} < r_i + p_i$$

   Což je přesně $\delta_{j,n} < r_i + p_i$.

2. $r_i + p_i \geq r_j + p_j$. Rozlišíme ještě další tři případy:

   - $i = k$. Podle věty 3 není třeba tento případ testovat.
   - $i > k$. Ani tentokrát není úkol i v množině $\Omega(jk)$ a proto $\Omega(ijk) = \Omega(jk)$. Můžeme tedy opravit hodnotu r_i právě tehdy, když existuje nějaké $k < i$ takové, že:

     $$d_{\Omega(jk)} - p_{\Omega(jk)} < r_i + p_i$$

     Což je právě tehdy když:

     $$\delta_{j,i-1} = \min\{d_{\Omega(jl)} - p_{\Omega(jl)} \mid l \in \{1, 2, \ldots, i-1\}\} < r_i + p_i$$

   - $i < k$. Pak úkol i leží v $\Omega(jk)$ a proto $p_{\Omega(ijk)} = p_{\Omega(jk)} - p_i$. Protože navíc $i \neq k$, tak $d_{\Omega(ijk)} = d_{\Omega(jk)}$. Proto můžeme v tomto případě nerovnost 3.9 upravit na:

     $$d_{\Omega(ijk)} - p_{\Omega(ijk)} < r_i + p_i$$
     $$d_{\Omega(jk)} - p_{\Omega(jk)} < r_i$$

     Hodnotu r_i tedy můžeme opravit právě tehdy, když:

     $$\min\{d_{\Omega(jl)} - p_{\Omega(jl)} \mid l \in \{i, i+1, \ldots, n\}\} < r_i \tag{3.13}$$

     Pokud už uspěl předchozí případ, tj. $\delta_{j,i-1} < r_i + p_i$, tak už na platnosti nerovnosti $\delta_{j,n} < r_i$ nezáleží.

     Pokud ovšem předchozí případ neuspěl, tak můžeme v nerovnosti 3.13 volit $l \in \{1, 2, \ldots, n\}$ místo $l \in \{i, i+1, \ldots, n\}$, protože tím minimum nemůžeme snížit pod r_i. Tedy r_i můžeme posunout právě tehdy, když $\delta_{j,n} < r_i$. □

Z této věty pak vyplývá následující algoritmus, který v čase $O(n^2)$ provede všechny změny vyplývající z 3.9 a 3.11. Algoritmus pro 3.10 a 3.12 je symetrický.

```
for j:=1 to n do begin
  spočti δ_{j,1}, δ_{j,2}, ..., δ_{j,n};
  for i:=1 to n do
    if r_i + p_i < r_j + p_j then begin
      if r_i + p_i > δ_{j,n} then
        r_i := max(r_i, r_j + p_j);
    end else begin
      if r_i + p_i > δ_{j,i-1} OR r_i > δ_{j,n} then
        r_i := max(r_i, r_j + p_j);
    end;
end;
```

3.8 Intervaly úkolů

Ukážeme algoritmus intervalů úkolů z [2], [5] a [3]. Rozebereme jednotlivá pravidla vzhledem k pravidlům edge-finding a not-first/not-last a také porovnáme celý algoritmus s kombinací algoritmu not-first/not-last (pokud je mi známo, takto porovnávány tyto algoritmy zatím nikde nebyly).

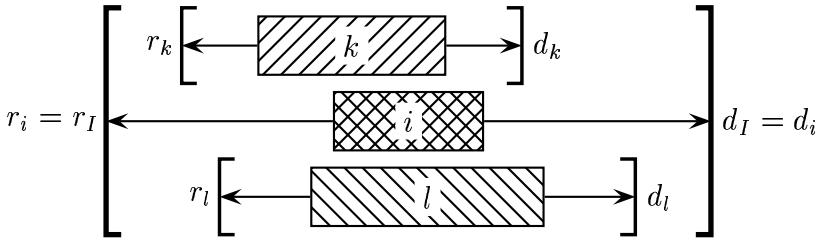
Intervaly úkolů *(task intervals)* byly poprvé definovány v [2]. My použijeme definici z [3]:

Definice 5 *Nechť $i, j \in T$ jsou dva úkoly (nemusí být nutně různé), pro které platí $r_i \leq r_j \ \wedge \ d_i \leq d_j$, pak* **interval úkolů** *je množina:*

$$I \ = [i, j] = \{t \mid r_i \leq r_t \ \wedge \ d_t \leq d_j\}$$

Takový interval úkolů nazýváme **aktivní**.

Pokud i a j nesplňují $r_i \leq r_j \ \wedge \ d_i \leq d_j$, pak $I = [i, j] = \emptyset$.



Obrázek 3.14: Příklad intervalu úloh $I = [i, i] = \{i, k, l\}$

Ve stavu podmínky si budeme udržovat seznam všech intervalů (udržujeme aktivní i neaktivní intervaly), těch je $O(n^2)$ (n je počet úloh $|T|$). Pro každý z těchto intervalů $I = [i, j]$ udržujeme tyto hodnoty:

- Množinu I. Autoři [2] doporučují bitové pole.
- Čas p_I potřebný pro vykonání všech úkolů z I.
- Jestli je interval I aktivní, neboli $I \neq \emptyset$.

Všechny tyto hodnoty se během procesu řešení mění.

Intervaly budeme ukládat do matice tak, abychom pro daný úkol i mohli projít všechny intervaly ve tvaru $[i, j]$ a $[j, i]$.

3.8.1 Splnitelnost intervalu úkolů

První z podmínek nad intervalem úkolů kontroluje, jestli je vůbec v intervalu dost času na splnění všech úkolů které v něm jsou.

Pro aktivní interval I:

$$p_I > d_I - r_I \Rightarrow \text{fail}$$

Toto pravidlo je ekvivalentní s pravidlem 3.1 algoritmu edge-finding.

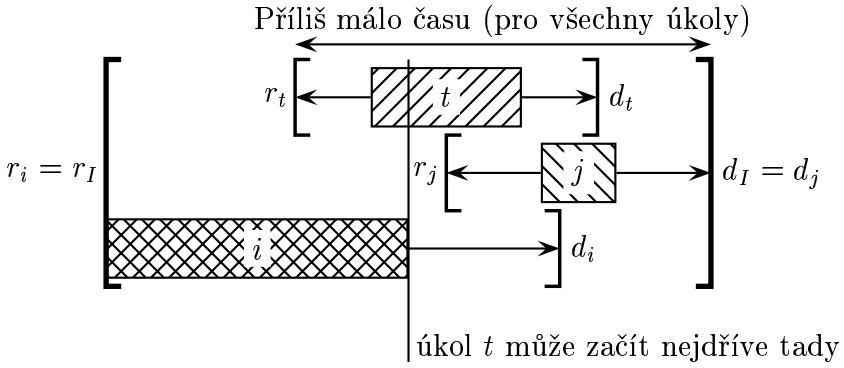
3.8.2 Pravidla not-first a not-last

Autoři [2] tato pravidla nazvali *edge-finding*, ale jak uvidíme, jedná se o pravidla *not-first* a *not-last*.

Uvažujme, jestli daný úkol $t \in I$ může být v rámci I vykonán jako první. Jestliže by provádění intervalu I začalo úkolem t, všechny úkoly z I (včetně t) by musely být provedeny v době $\langle d_t, r_I \rangle$. Úkol t tedy lze vykonat jako první pouze pokud v intervalu $\langle r_t, d_I \rangle$ je dost času na vykonání všech úkolů z I.

Pokud úloha t nemůže být zpracována v intervalu I jako první, tak před úlohou t musí být provedena nějaká jiná úloha z I, takže úloha t nemůže začít dříve než skončí alespoň jeden z úkolů v intervalu I:

$$\forall I \;\; \forall t \in I : d_I - r_t < p_I \Rightarrow r_t \geq \min\{r_k + p_k, k \in I \;\wedge\; k \neq t\}$$



Obrázek 3.15: Příklad použití pravidla *not-first*

Často se může stát, že se toto pravidlo spustí (tj. platí předpoklad implikace), ale nezvětší hodnotu r_t, protože už je větší než počítané minimum. Abychom počet takových zbytečných výpočtů zmenšili, pozměníme pravidlo:

$$\begin{aligned} \forall I = [i, j]\, \forall t \in I : d_I - r_t < p_I \;\wedge\; r_t < r_i + p_i \\ \Rightarrow r_t \geq \min\{r_k + p_k,\; k \in I \;\wedge\; k \neq t\} \end{aligned}$$

Tato verze pravidla je stejně silná jako ta předchozí, ale nebude potřeba tak často počítat minimum.

Vidíme, že toto pravidlo odpovídá pravidlům not-first 3.9 a 3.11 pro množinu $\Omega = I - t$ a úkol t.

Symetrické pravidlo *not-last* pak rozhoduje, který úkol z intervalu I může být vykonán jako poslední:

$$\forall I = [i,j]\ \forall t \in I : d_t - r_I < p_I \ \wedge\ d_t > d_j - p_j \\ \Rightarrow d_t \leq \max\{d_k - p_k,\ k \in I \ \wedge\ k \neq t\}$$

3.8.3 Pravidlo vyloučení

V článku [2] je toto pravidlo nazváno *exclusion*. Jedná se o kombinaci edge-finding a not-first/not-last.

Budeme zkoumat pořadí mezi intervalem I úlohou $t \notin I$. Pokud $d_I - r_t < p_I + p_t$, pak úkol t nemůže být vykonán před intervalem I. Pokud navíc $d_I - r_I < p_I + p_t$, tak úkol t nelze vykonat ani během intervalu I (tj. t by byl mezi dvěmi úkoly z I). Pro každý aktivní interval I a úkol $t \notin I$ tedy máme pravidlo:

Require $d_I - r_t < p_I + p_t$;
  if $d_I - r_I < p_I + p_t$ **then begin**
    $r_t \geq r_I + p_I$;
    $\forall k \in I : d_k \leq d_t - p_t$;
  end else
    $r_t \geq \min\{r_k + p_k \mid k \in I\}$;

Podobně jako v předchozích pravidlech not-first a not-last se můžeme někdy vyhnout zbytečnému počítání výrazu $\min\{r_k + p_k \mid k \in I\}$ a budeme ho počítat pouze pokud $r_t < r_i + p_i$, kde $I = [i,j]$.

Prvá úprava $r_t \geq r_I + p_I$ odpovídá pravidlu 3.3 edge-finding, ale hodnota r_t by šla zvětšovat více, viz pravidlo 3.5. Takto bychom mohli algoritmus intervalů úkolů vylepšit, aniž bychom zvětšili jeho časovou náročnost (pro každý interval Ω bychom udržovali ještě hodnotu $\max\{r_{\Omega'} + p_{\Omega'} \mid \Omega' \subseteq \Omega$, který bychom při změně intervalu mohli opravit podle 3.6).

Stejný, ne-li lepší výsledek než druhá úprava ($d_k \leq d_t - p_t$), dá pravidlo not-last (viz. 3.10 a 3.12) pro množinu $I \cup \{t\} - \{k\}$ a úkol k.

Třetí pravidlo zase odpovídá pravidlu not-first (viz. 3.9 a 3.11).

Symetrické pravidlo not-last pak reaguje na to, že úkol $t \notin I$ nemůže být vykonán za intervalem I:

Require $d_t - r_I < p_I + p_t$;
  if $d_I - r_I < p_I + p_t$ **then begin**

$d_t \leq d_I - p_I$;
$\forall k \in I : r_k \geq r_t + p_t$;
else
$d_t \leq \max\{d_k - p_k \mid k \in I\}$;

3.8.4 Udržování intervalů

Jak už bylo řečeno výše, většina atributů intervalu se během procesu řešení mění. Musíme tedy být schopni rychle počítat změny intervalů – množina I se může zvětšovat a zmenšovat, aktivní interval se může stát neaktivním a naopak. Potřebujeme tedy reagovat na změny hodnot r_t a d_t; hodnota r_t se může při propagaci pouze zvětšovat a hodnota d_t zase pouze zmenšovat.

Při zvětšení hodnoty r_t z v na w musíme:

1. Deaktivovat intervaly ve tvaru $[t, j]$, když $w > d_j$.
2. Vyřadit úkoly k z aktivního intervalu $[t, j]$, pro které $r_k < w$.
3. Přidat úkol t do aktivních intervalů $[i, j]$ takových, že $v < r_i \leq w \ \wedge \ d_t \leq d_j$.
4. Aktivovat intervaly $[i, t]$ pro i, které splňuje $v < r_i \leq w \ \wedge \ d_i \leq d_t$.

Algoritmus pro zvětšení hodnoty r_t na w

$v := r_t$;
for active $I = [t, j]$ **do begin**
  if $r_j < w \ \wedge \ j \neq t$ **then begin**
    $I := \emptyset$;
    active(I) := **false** ;
  end ;
  for $k \in I$ **do**
    if $r_k < w$ **then begin**
      $I := I \setminus \{k\}$;
      $p_I := p_I - p_k$;
    end ;
end ;
$r_t := w$;
for $i \in T$ such as $t \neq i \ \wedge \ v < r_i \leq w$ **do**
  for $I = [i, j]$ such as $d_t \leq d_j \ \wedge \ t \notin I$ **do begin**
    $I := I \cup \{t\}$;
    $p_I := p_I + p_t$;
  end ;
for $I = [i, t]$ such as $\neg$ active$(I) \ \wedge \ r_i \leq w \ \wedge \ d_i \leq d_t$ **do begin**

$I := \{k \mid r_i \leq r_k \ \wedge \ d_k \leq d_t\}$;
$p_I := \sum_{k \in I} p_k$;
active(I) := **true**;
end;

Algoritmus pro zmenšení hodnoty d_t je symetrický.

Vidíme, že samotné zvýšení hodnoty r_t má časovou složitost $O(n^2)$. Tím se navíc může změnit až $O(n^2)$ intervalů, pro které je třeba znovu testovat pravidla. A otestování pravidla not-first, not-last a edge-finding pro jeden interval a všechny úkoly trvá $O(n)$. Pokud tedy algoritmus zúží nějakou doménu, bude časová složitost minimálně $O(n^3)$.

3.8.5 Porovnání s edge-finding a not-first/not-last

Ukázali jsme, že algoritmus intervalů úloh by šel snadno vylepšit, takže by byl stejně silný jako edge-finding. V algoritmu jsou použita i pravidla not-first a not-last, nejsou ale testována pro všechny potřebné dvojice množin a úkolů, nevyvodí se tedy všechny změny, na které příjde algoritmus not-first/not-last. K tomu bychom museli kromě intervalů úkolů udržovat ještě všechny množiny $\Omega(jk)$ podle věty 3 (str. 23).

Těžko odhadnout, jak dlouho bude algoritmus intervalů úloh běžet, dokud žádné z pravidel nebude moci propagovat další změnu. Stejně to ale nemůžeme odhadnout ani u algoritmu edge-finding a not-first/not-last (jedna iterace trvá $O(n^2)$, ale oba algoritmy je třeba iterovat).

Pro intervaly úloh mluví to, že se testují pouze ty intervaly, ve kterých skutečně došlo ke změně, zatímco edge-finding a not-first/not-last pořád dokola zbytečně testují i nezměněné intervaly.

Algoritmy edge-finding a not-first/not-last však nepotřebují stav podmínky a mají paměťovou náročnost $O(n)$. Intervaly úloh potřebují na uložení stavu prostor $O(n^2)$ a tento stav musí být zpětně rekonstruovatelný při návratu v algoritmu prohledávajícím prostor řešení. Tím se prostorová náročnost ještě více zvětší.

V neposlední řadě se algoritmy not-first/not-last a edge-finding mnohem jednodušeji implementují (není potřeba udržovat stav).

To vše jsou důvody, proč se intervaly úkolů už dnes příliš nepoužívají.

3.9 Energetic reasoning

Popíšeme a mírně vylepšíme algoritmus podmínky splnitelnosti pro energetic reasoning neboli *left-shift/right-shift* podle [10] a [13]. Další pravidla

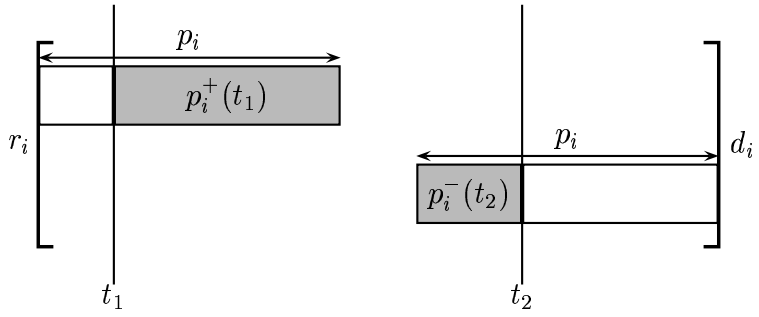
pro energetic reasoning zde uvádět nebudeme.

Energetic reasoning je algoritmus pro kumulativní nepreemptivní rozvrhování, dá se nicméně použít i pro disjunktivní rozvrhování (disjunktivní rozvrhování je speciální případ kumulativního rozvrhování pro kapacitu zdroje $C = 1$ a požadavky na kapacitu $c_1 = c_2 = \cdots = c_n = 1$).

Zvolme si libovolný časový interval $\langle t_1, t_2 \rangle$ ($t_1, t_2 \in \mathbb{R}$). Budeme zkoumat, kolik kapacity (energie) se v tomto intervalu nutně musí spotřebovat. Označme:

$$p_i^+(t_1) = \max(0, p_i - \max(0, t_1 - r_i))$$
$$p_i^-(t_2) = \max(0, p_i - \max(0, d_i - t_2))$$

Hodnota $p_i^+(t_1)$ je minimální doba, po kterou musí úkol i probíhat po okamžiku t_1, hodnota $p_i^-(t_2)$ zase minimální doba, po kterou musí úkol i probíhat před okamžikem t_2:



Obrázek 3.16: Ukázka $p_i^+(t_1)$ a $p_i^-(t_2)$

Minimální kapacita spotřebovaná úkolem i v intervalu $\langle t_1, t_2 \rangle$ pak je:

$$W(i, t_1, t_2) = c_i * \min(p_i^+(t_1), p_i^-(t_2), t_2 - t_1)$$

Celkově všechny úkoly spotřebují v intervalu $\langle t_1, t_2 \rangle$ kapacitu:

$$W(t_1, t_2) = \sum_{i \in T} W(i, t_1, t_2)$$

V intervalu $\langle t_1, t_2 \rangle$ tedy ještě zbývá volná kapacita:

$$S(t_1, t_2) = C * (t_2 - t_1) - W(t_1, t_2)$$

3.9.1 Pravidlo splnitelnosti

Pravidlo splnitelnosti kontroluje, jestli v některém intervalu není spotřebováno více kapacity než je možné:

$$\forall t_1, t_2 \in \mathbb{R}, t_1 < t_2 : \; S(t_1, t_2) < 0 \Rightarrow \text{fail} \tag{3.14}$$

V [10] a [13] jsou pak další pravidla pro zmenšování intervalů $\langle r_i, d_i \rangle$. Časová složitost algoritmů na nich založených je $O(n^3)$. My se budeme dále zabývat pouze pravidlem 3.14, proto tato pravidla neuvádíme.

V [10] a [13] je dokázána následující věta, my ji uvedeme bez důkazu, protože později dokážeme novou, ještě silnější verzi.

Věta 5 *Nechť O_1, O_2 a $O(t)$ jsou množiny:*

$$\begin{aligned} O_1 &= \{r_i \mid 1 \leq i \leq n\} \cup \{d_i - p_i \mid 1 \leq i \leq n\} \cup \{r_i + p_i \mid 1 \leq i \leq n\} \\ O_2 &= \{d_i \mid 1 \leq i \leq n\} \cup \{r_i + p_i \mid 1 \leq i \leq n\} \cup \{d_i - p_i \mid 1 \leq i \leq n\} \\ O(t) &= \{r_i + d_i - t \mid 1 \leq i \leq n\} \end{aligned}$$

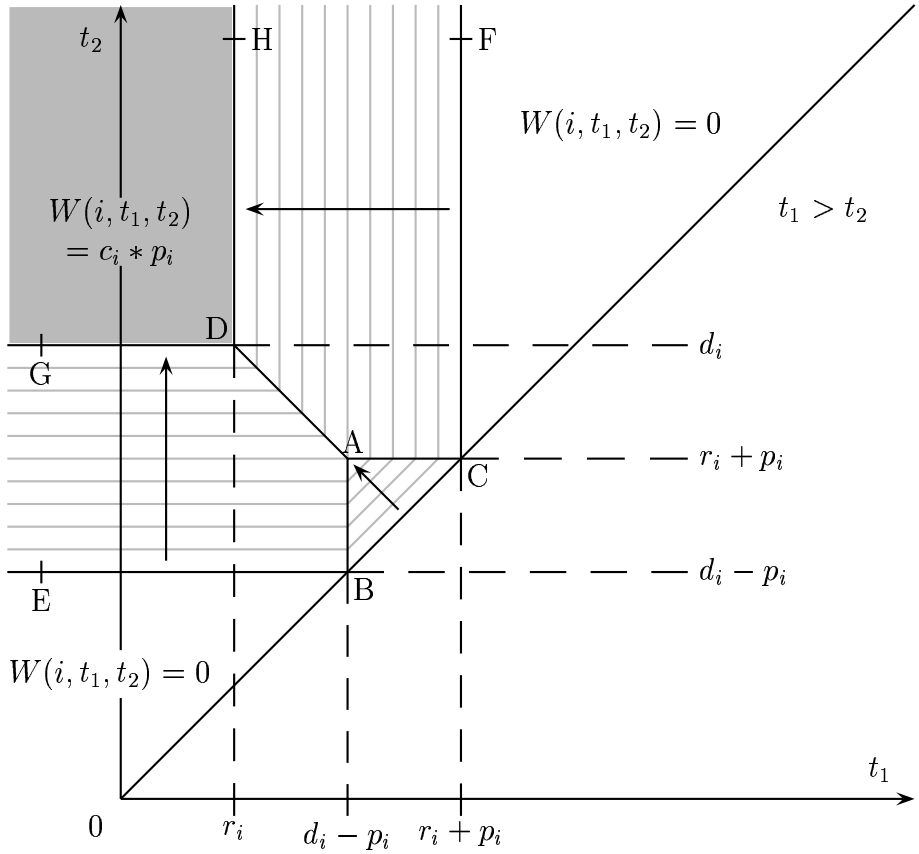
Pak $\forall t_1, t_2 \in \mathbb{R}, t_1 \leq t_2 : S(t_1, t_2) \geq 0$ platí právě tehdy, když platí $S(s, e) \geq 0$ pro všechny s a e takové, že $(s \in O_1 \wedge e \in O_2 \cup O(s)) \vee (e \in O_2 \wedge s \in O(e))$.

Tato věta nám tedy dovoluje volit jen $O(n^2)$ intervalů a pro ně testovat podmínku 3.14.

V [13] je vyslovena domněnka, že počet testovaných intervalů by mohl jít ještě zmenšit. My takové zmenšení ukážeme.

3.9.2 Graf funkce $W(i, t_1, t_2)$

Rozebereme graf funkce $W(i, t_1, t_2)$ v závislosti na t_1 a t_2. Jde o trojrozměrný graf, pohled se shora vidíme na následujícím obrázku:

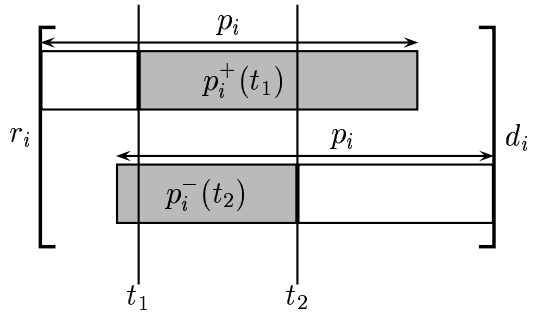


Obrázek 3.17: Graf funkce $W(i,t_1,t_2)$ pro případ $d_i - r_i < 2p_i$

Graf se skládá z částí rovin, nakloněné roviny jsou šrafované a šipkou je naznačeno kterým směrem rostou. Rozebereme jednotlivé části grafu:

- $t_1 \geq r_i + p_i$. Pak $p_i^+(t_1) = 0$ a proto také $W(i,t_1,t_2) = 0$.
- $t_2 \leq d_i - p_i$. Pak $p_i^-(t_2) = 0$ a proto také $W(i,t_1,t_2) = 0$.
- $t_1 \leq r_i$ a $t_2 \geq d_i$. Pak je interval $\langle r_i, d_i \rangle$ uvnitř intervalu $\langle t_1, t_2 \rangle$ a tudíž $W(i,t_1,t_2) = c_i * p_i$.
- Zjistíme, pro které t_1 a t_2 je $\min(t_2 - t_1, p_i^+(t_1), p_i^-(t_2)) = t_2 - t_1$.

  Nerovnost $t_2 - t_1 \leq p_i^+(t_1)$ znamená, že i když posuneme úkol i co nejvíce doleva, stále bude po celý interval $\langle t_1, t_2 \rangle$ probíhat zpracování úkolu i:



Proto musí být $t_2 \leq r_i + p_i$. Podobně když posuneme úkol co nejvíce doprava, stále bude zpracování probíhat po celý interval $\langle t_1, t_2 \rangle$ a proto $t_1 \geq d_i - p_i$. Navíc musí být $t_1 \leq t_2$.

V trojúhelníku ABC tedy $W(i, t_1, t_2) = c_i * (t_2 - t_1)$.

V případě, že $d_i - r_i > 2 * p_i$ tento případ vůbec nenastává (pokud pro t_1 a t_2 platí $t_1 \geq d_i - p_i$ a $t_2 \leq r_i + p_i$, nemůže už platit $t_1 \leq t_2$).

- Najdeme, pro které t_1 a t_2 platí $\min(t_2 - t_1, p_i^+(t_1), p_i^-(t_2)) = p_i^+(t_1)$, přičemž se omezíme jen na ty dvojice t_1 a t_2, pro které zatím nevíme kolik $W(i, t_1, t_2)$ je, tedy:

$$t_1 < r_i + p_i$$
$$t_2 > d_i - p_i$$

Stačí zjistit kdy $p_i^+(t_1) < p_i^-(t_2)$, protože případ $t_2 - t_1 \leq p_i^+(t_1)$ jsme již probrali.

Aby mohlo být $p_i^+(t_1) < p_i^-(t_2)$, musí být $p_i^+(t_1) < p_i$, protože $p_i^-(t_2) \leq p_i$. Proto $t_1 > r_i$. Pomocí těchto nerovností pro t_1 dostaneme:

$$\begin{aligned} p_i^+(t_1) &= \max(0, p_i - max(0, t_1 - r_i)) = \\ &= \max(0, p_i - t_1 + r_i) = \\ &= p_i + r_i - t_1 \end{aligned}$$

Dále rozlišíme dva případy podle t_2:

- $t_2 \geq d_i$. Pak $p_i^-(t_2) = p_i$ a tedy $p_i^+(t_1) \leq p_i^-(t_2)$.
- $t_2 < d_i$. Pak z nerovností pro t_2 dostáváme:

$$\begin{aligned} p_i^-(t_2) &= \max(0, p_i - \max(0, d_i - t_2)) = \\ &= \max(0, p_i - d_i + t_2)) = \\ &= p_i - d_i + t_2 \end{aligned}$$

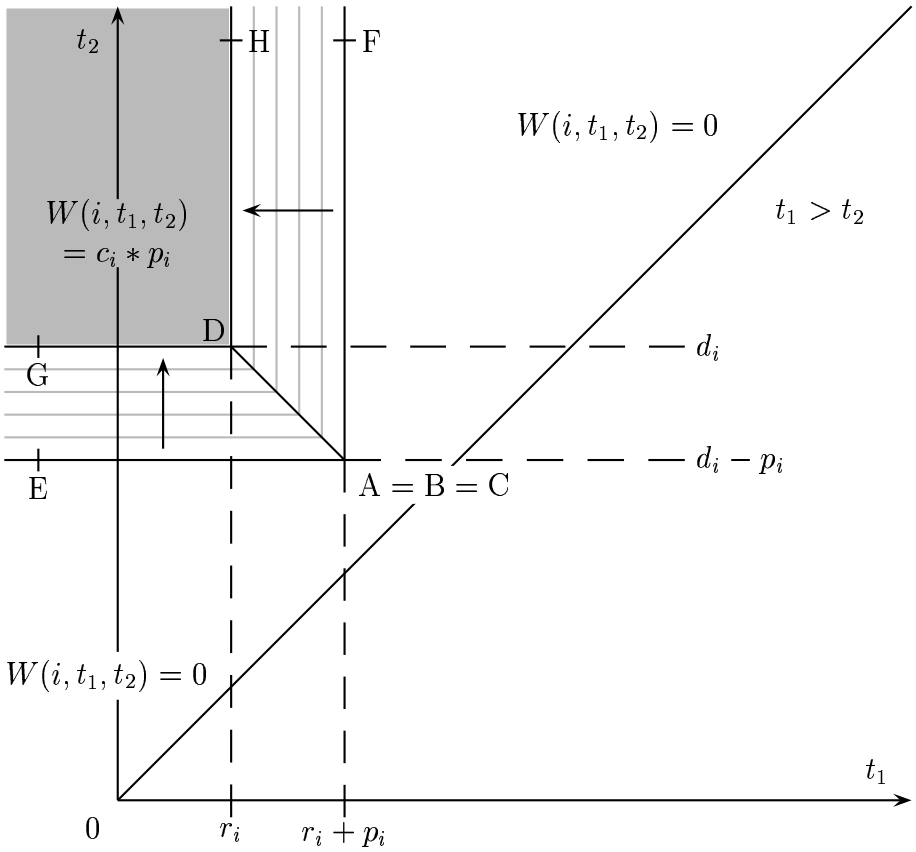
Tedy $p_i^+(t_1) \leq p_i^-(t_2)$ právě tehdy, když:

$$p_i + r_i - t_1 \leq p_i - d_i + t_2$$
$$r_i + d_i \leq t_1 + t_2$$

V oblasti FCADH tedy $W(i, t_1, t_2) = p_i^+(t_1) = p_i + r_i - t_1$.

- Podobně v oblasti EBADG je $W(i, t_1, t_2) = p_i^-(t_2) = t_2 + p_i - d_i$.

Tím jsme ukázali, že graf funkce $W(i, t_1, t_2)$ pro pevné i vypadá tak, jak je naznačeno na obrázku 3.17 pro $d_i - r_i < 2p_i$, nebo pro $d_i - r_i \geq 2p_i$ na obrázku 3.18. V obou případech je graf spojitý.



Obrázek 3.18: Graf funkce $W(i, t_1, t_2)$ pro případ $d_i - r_i \geq p_i$

3.9.3 Relevantní intervaly $\langle t_1, t_2 \rangle$

Vzhledem k tomu, že grafy funkcí $W(i, t_1, t_2)$ jsou spojité a skládají se z částí rovin, je funkce $S(t_1, t_2) = C*(t_2 - t_1) - \sum_{i \in T} W(i, t_1, t_2)$ také spojitá a

skládá se z částí rovin. Při ověřování podmínky 3.14 stačí zkontrolovat lokální minima funkce $S(t_1, t_2)$. Ta ale nemusí být ostrá, tj. minima se nenabývá v jednom bodě, ale na celé úsečce nebo ploše. Z takových neostrých minim stačí když otestujeme jeden bod na okraji, vnitřní body úsečky případně plochy testovat nemusíme.

Definice 6 *Funkce S má v bodě $[x, y]$* **zajímavé minimum**, *pokud má v bodě $[x, y]$ lokální minimum a bod $[x, y]$ neleží uvnitř nějaké úsečky, na které se lokální minimum nabývá.*

Stačí tedy projít zajímavá minima, protože to jsou právě ta minima, která neleží na okraji úsečky nebo plochy, na které se minimum nabývá.

Lemma 1 *Pokud na nějakém ϵ-okolí bodu $[x, y]$ je funkce $S(t_1, t_2)$ lineární, tak v bodě $[x, y]$ není zajímavé minimum.*

Důkaz je zřejmý.

Z předchozího lemmatu plyne, že body $[x, y]$, které musíme testovat, musí pro některý úkol i ležet na jedné z úseček AB, AC, BC, AD nebo polopřímek $\overrightarrow{BE}$, $\overrightarrow{DG}$, $\overrightarrow{DH}$, $\overrightarrow{CF}$. Jinak by totiž na nějakém ϵ-okolí bodu $[t_1, t_2]$ byla funkce S jen součtem lineárních funkcí a byla by tedy také lineární.

Lemma 2 *Pokud na nějakém ϵ-okolí bodu $[x, y]$ je funkce $F(t) = S(x + at, y + bt)$ lineární, tak v bodě $[x, y]$ není zajímavé minimum.*

Důkaz: Když je funkce $F(t)$ lineární, tak řez grafem funkce S bodem $[x, y]$ ve směru $\overrightarrow{(a, b)}$ je lineární funkce a bod $[x, y]$ tedy buď není lokální minimum, nebo je uvnitř úsečky kde funkce S lokální minimum nabývá. □

Toto lemma říká, že pokud bod $[x, y]$ leží pouze na jediné úsečce nebo polopřímce, tak ho není potřeba testovat. Testovat jsou potřeba pouze průsečíky úseček/polopřímek různých směrů.

Lemma 3 *Nechť bod $[x, y]$ leží na jedné z polopřímek $\overrightarrow{\mathrm{CF}}$, $\overrightarrow{\mathrm{BE}}$ nebo úsečce* BC *úkolu i. Pak funkce S má v bodě $[x, y]$ zajímavé minimum pouze tehdy, když v bodě $[x, y]$ má i funkce $S(t_1, t_2) + W(i, t_1, t_2)$ zajímavé minimum.*

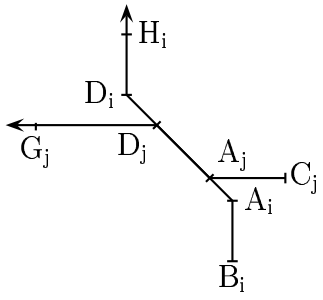
Důkaz: Nechť $[x, y]$ leží na polopřímce $\overrightarrow{\mathrm{CF}}$, $\overrightarrow{\mathrm{BE}}$ nebo úsečce BC úkolu i. Pak $W(i, x, y) = 0$. Rozlišíme dvě možnosti:

- Funkce $S(t_1, t_2) + W(i, t_1, t_2)$ nemá v bodě $[x, y]$ lokální minimum. Pak existuje ϵ-okolí bodu $[x, y]$ takové, že pro každou hodnotu $0 < \delta < \epsilon$ existuje bod $[u, v]$ takový, že vzdálenost bodů $[x, y]$ a $[u, v]$ je menší než δ a $S(u, v) + W(i, u, v) < S(x, y) + W(i, x, y) = S(x, y)$. Funkce W je nezáporná, takže $W(i, u, v) \geq 0$ a proto $S(u, v) < S(x, y)$. Funkce S tedy nemá v bodě $[x, y]$ lokální minimum.

- Funkce $S(t_1, t_2) + W(i, t_1, t_2)$ má v bodě $[x, y]$ lokální minimum, ale bod $[x, y]$ leží uvnitř nějaké úsečky, na které se toto lokální minimum nabývá. Vezměme si bod $[u, v]$ na této úsečce. Pak $S(u, v)+W(i, u, v) = S(x, y) + W(i, x, y) = S(x, y)$. Protože funkce W je nezáporná, tak $W(i, u, v) \geq 0$ a tudíž $S(u, v) \leq S(x, y)$. Funkce S tedy buď v bodě $[x, y]$ lokální minimum nemá, nebo to není jeho krajní bod. □.

Z tohoto lemmatu plyne, že pokud bod $[x, y]$ leží na průsečíku pouze dvou úseček/polopřímek a jedna z nich je $\overrightarrow{CF}$, $\overrightarrow{BE}$ nebo BC, tak bod $[x, y]$ nemusíme testovat. Kdybychom totiž úkol i odebrali, měla by "nová" funkce $S(t_1, t_2)$ tvar $S(t_1, t_2) + W(i, t_1, t_2)$. Bod $[x, y]$ by pro tuto novou funkci $S(t_1, t_2)$ ležel pouze na jedné úsečce/polopřímce, což podle předposledního lemmatu znamená, že funkce $S(t_1, t_2) + W(i, t_1, t_2)$ nemá v bodě $[x, y]$ zajímavé minimum, a tedy ani funkce $S(t_1, t_2)$ nemá v bodě $[x, y]$ zajímavé minimum.

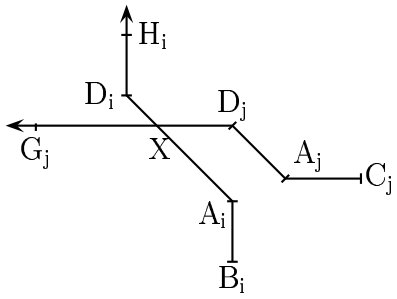
Pro každou dvojici úkolů i, j tedy musíme otestovat průsečíky lomených čar $B_iA_iD_iH_i$ a $C_jA_jD_jG_j$, přičemž společné vnitřní body úseček A_iD_i a A_jD_j se nepočítají. Takové průsečíky mohou být dva, jeden nebo žádný, jak vidíme na příkladech na obrázcích 3.19, 3.20.



Obrázek 3.19: Příklad dvou průsečíků A_j a D_j čar $B_iA_iD_iH_i$ a $C_jA_jD_jG_j$

.

Musíme tedy testovat maximálně $2n(n-1)$ intervalů oproti $15n^2$ podle věty 5. Samotný algoritmus se však bohužel zrychlí pouze o třetinu, jak uvidíme dále.



Obrázek 3.20: Příklad jednoho průsečíků X čar $B_iA_iD_iH_i$ a $C_jA_jD_jG_j$

.

Věta 6 *Označme:*

$$M_1(i) = \{[d_i - p_i, t] \mid d_i - p_i \leq t \leq r_i + p_i\} \cup \{[r_i, t] \mid t \geq d_i\}$$
$$M_2(j) = \{[t, r_i + p_i] \mid d_i - p_i \leq t \leq r_i + p_i\} \cup \{[t, d_i] \mid t \leq r_i\}$$
$$M_3(i) = \{[r_i + t, d_i - t] \mid 0 \leq t \leq p_i - max(r_i + 2p_i - d_i, 0\}$$

Pokud v bodě $[x, y]$ je zajímavé minimum funkce S, tak $[x, y]$ leží v množině:

$$\bigcup_{i \in T} \bigcup_{j \in T} \left((M_1(i) \cap M_2(j)) \cup (M_1(i) \cap M_3(j)) \cup (M_3(i) \cap M_2(j)) \right)$$

Důkaz: Množina M_1 odpovídá úsečce AB a polopřímce $\overrightarrow{DH}$. Množina M_2 zase odpovídá úsečce AC a polopřímce $\overrightarrow{DG}$. A množina M_3 je úsečka AD. □

3.9.4 Algoritmus

Zvolme si pevně $x \in \mathbb{R}$ tak, že pro nějaké $y \in \mathbb{R}$ a $i \in T$ je $[x, y] \in M_1(i)$. Takových x je $2n$.

Funkce $F(t_2) = S(x, t_2)$ má tvar lomené čáry. Spočítáme její hodnoty ve všech zlomech a ověříme v nich podmínku 3.14.

Vidíme, že:

$$F(t_2) = (t_2 - x) * C - \sum_{j \in T} W(j, x, t_2)$$

Označme:

$$F_j(t_2) = W(j, x, t_2)$$

Tato funkce má tvar lomené čáry s nejvýše dvěma zlomy (viz. grafy 3.17 a 3.18). Jsou jen 4 druhy zlomů: úsečky AC, AD a polopřímky $\overrightarrow{DG}$ a $\overrightarrow{BE}$.

Úkoly můžeme setřídit podle souřadnice y, kdy bod $[x, y]$ leží na přímce $\overleftrightarrow{AC}$. Stejně tak můžeme úkoly setřídit podle dalších tří kritérií: souřadnice y,

kdy bod $[x, y]$ leží na přímce $\overleftrightarrow{AD}$, $\overleftrightarrow{DG}$ resp. $\overleftrightarrow{BE}$. Toto pořadí nezávisí na volbě x, protože přímky stejného druhu jsou pro různé úkoly vždy rovnoběžné.

Při počítání funkce $F(t_2) = S(x, t_2)$ začneme s $t_2 = x$ a tedy $F(x) = S(x, x) = 0$. Souřadnici t_2 budeme postupně zvyšovat od jednoho průsečíku s jednou ze přímek $\overleftrightarrow{AC}$, $\overleftrightarrow{AD}$, $\overleftrightarrow{DG}$ nebo $\overleftrightarrow{BE}$ k dalšímu průsečíku. V těchto průsečících může být zlomový bod funkce F (pokud $[x, t_2]$ leží na AC, AD, $\overrightarrow{DG}$ nebo $\overrightarrow{BE}$). Mezi zlomovými body je funkce F lineární, pro vypočtení hodnoty funkce F v dalším zlomovém bodu proto stačí znát směrnici a hodnotu v minulém zlomovém bodu. Pokud je tedy průsečík skutečně zlomový bod, vypočteme hodnotu funkce F, ověříme podmínku 3.14 a spočítáme novou směrnici grafu funkce F.

Takto postupnou volbou x určitě ověříme podmínku 3.14 ve všech bodech z množiny

$$\bigcup_{i \in T} \bigcup_{j \in T} \big(\, (M_1(i) \cap M_2(j)) \, \cup \, (M_1(i) \cap M_3(j)) \, \big)$$

Abychom ještě ověřili body z množiny

$$\bigcup_{i \in T} \bigcup_{j \in T} \big(\, M_3(i) \cap M_2(j) \, \big)$$

provedeme ještě symetrický algoritmus, kde si pevně volíme $y \in \mathbb{R}$ takové, že pro nějaké $x\mathbb{R}$ a $j \in T$ je $[x, y] \in M_2(j)$.

Celkově tedy ověření podmínky 3.14 bude trvat $O(n^2)$. Oproti větě 5 jsme ušetřili volbu x na $\overrightarrow{CF}$ a volbu y na $\overrightarrow{BE}$, tedy třetinu času původního algoritmu.

Kapitola 4

Dynamické globální podmínky

Ve většině jazyků pro programování s omezujícími podmínkami můžeme v průběhu řešení přidávat nové proměnné a nové podmínky (při návratu v prohledávání prostoru řešení se tyto přidané proměnné a podmínky zase odeberou). Nelze však do už existující globální podmínky přidat novou proměnnou. Přitom právě tato vlastnost by byla velmi užitečná v řadě praktických aplikací (v [1] se takové problémy zavádí název vysoce dynamické (*highly dynamic problems*)).

V této kapitole budeme mluvit o dynamických globálních podmínkách, tedy o globálních podmínkách, do kterých lze přidat v průběhu řešení další proměnné.

4.1 Motivace

Vysoce dynamické problémy se nejčastěji objevují v plánování a rozvrhování. Jeden z příkladů uváděných v [1] je přihřátí *(re-heating)*: pokud polotovar čeká na zpracování na dalším stroji příliš dlouho, je nutné ho nejdříve znovu ohřát. Je tedy třeba přidat novou aktivitu a zařadit ji do rozvrhu.

Standardně se vysoce dynamické problémy řeší pomocí "zbytečných" proměnných *(dummy variables)*, tedy proměnných, které se hned na začátku zařadí do globálních podmínek "do zásoby". Teprve když vyvstane potřeba zařadit do globální podmínky novou proměnnou, použije se některá z těchto "zbytečných" proměnných.

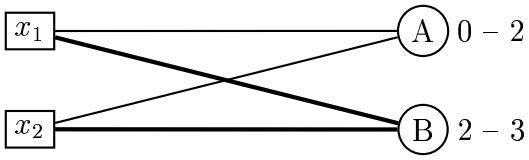
Tento přístup má několik nevýhod:

- Musíme dopředu dobře odhadnout, kolik "zbytečných" proměnných máme přidat, jinak v průběhu řešení zjistíme, že jich máme málo a nemůžeme proto pokračovat v hledání řešení. Z tohoto důvodu bývá počet "zbytečných" proměnných dost nadsazený.

- Globální podmínka pracuje s více proměnnými (a často také většími doménami – přidání "zbytečných" proměnných si může vynutit také přidání "zbytečných" hodnot) a tím se propagační algoritmus zbytečně prodlužuje. Pokud navíc globální podmínka používá stav, zvětšují se také paměťové nároky.
- Takový přístup je neintuitivní a nepřehledný.

4.2 Dynamizace globální podmínky

Dynamická globální podmínka je globální podmínka, do které můžeme v průběhu řešení přidat novou proměnnou. Ne z každé globální podmínky lze udělat dynamickou globální podmínku. Původní podmínka bez přidaných proměnných totiž mohla při propagaci z domén odstranit některé hodnoty, které rozeznala jako nepřípustné. Po přidání proměnných by ale takové hodnoty mohly být opět přípustné, jenže z domén už byly jednou vyřazeny. Příklad takové situace pro podmínku kardinality je na obrázcích 4.1 a 4.2.



Obrázek 4.1: Proměnné x_1 i x_2 musí mít hodnotu B, protože hodnotu B musí nabývat 2 až 3 proměnné. Hodnota A se proto odstraní z domén x_1 i x_2

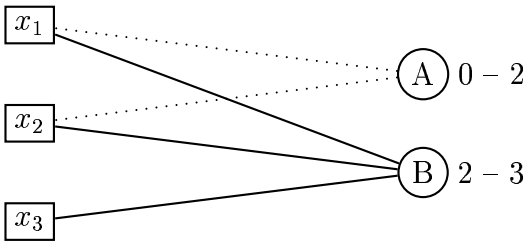
.

Definice 7 *Nechť G je globální podmínka nad množinou proměnných X. Označme* $\mathrm{Sol}(G(X))$ *množinu všech přípustných ohodnocení proměnných z X pro podmínku G.*

Podmínka G se nazývá **monotónní**, *jestliže pro každé množiny proměnných X, Y platí* $\mathrm{Sol}(G(X \cup Y)) \downarrow X \subseteq \mathrm{Sol}(G(X))$ [1].

Tato vlastnost nám zaručuje, že i když přidáme nové proměnné, všechna omezení domén, které jsme udělali dříve, zůstávají v platnosti. Dynamizovat tedy lze právě monotónní globální podmínky.

[1] Operátor $\downarrow$ značí zúžení na ohodnocení pouze množiny X



Obrázek 4.2: Po přidání proměnné x_3 by opět mohlo být $x_1 = \mathrm{A}$ i $x_2 = \mathrm{A}$. Hodnota A však už byla z domén x_1 a x_2 odstraněna.

Z podmínek v kapitole 3 není monotónní symetrická podmínka alldifferent (kapitola 3.2), podmínka kardinality (kapitola 3.3) a hamiltonovská kružnice (kapitola 3.4).

V [1] je navržen obecný algoritmus dynamizace globální podmínky:

Algoritmus pro přidání proměnné x_{k+1} do podmínky $G(x_1, x_2, \ldots x_k)$

1. Deaktivuj $G(x_1, x_2, \ldots x_k)$. Tj. ulož interní data podmínky (stav) na zásobník a zastav propagaci této podmínky.
2. Přidej podmínku $G(x_1, x_2, \ldots x_k, x_{k+1})$.

Pro podmínky, které nepoužívají stav, už lepší způsob dynamizace neexistuje. Z kapitoly 3 se jedná o algoritmy edge-finding (kapitola 3.6), not-first/not-last (kapitola 3.7) a energetic reasoning (kapitola 3.9).

Bohužel např. v SICStus prologu z důvodu closed-source distribuce není možné takto zdynamizovat existující dynamické podmínky.

V případě, že podmínka stav používá, musí si ho v bodě 2 vytvořit úplně znovu. S pomocí stavu původní podmínky by ovšem mohl být vytvořen rychleji, to už ale závisí na konkrétní podmínce. Jako příklad ukážeme podmínku alldifferent.

4.3 Dynamická podmínka alldifferent

Rozšíříme podmínku alldifferent z kapitoly 3.1 na dynamickou verzi.

Propagační algoritmus zůstane stejný, jde pouze o to, jak po přidání nové proměnné vytvořit z původního stavu podmínky stav nový.

Stav podmínky alldifferent je maximální párování v grafu podmínky nalezené při poslední propagaci. Při další propagaci se z tohoto párování odstraní už neexistující hrany a výsledek slouží jako výchozí párování při hledání nového maximálního párování.

Není tedy důležité, že ve stavu je maximální párování. Ve stavu je jednoduše největší známé párování. Pokud tedy algoritmus pro propagaci podmínky alldifferent bude připraven na to, že párování, které má uloženo ve stavu nemusí pokrývat všechny vrcholy, nemusíme stav při přidání proměnné nijak měnit.

4.3.1 Výsledek

Dynamickou podmínku alldifferent jsem naprogramoval v SICStus prologu. Tento prolog už podmínku alldifferent obsahuje, ovšem ne dynamickou verzi.

Abych mohl porovnat dynamický alldifferent s řešením pomocí "zbytečných" proměnných, vytvořil jsem následující příklad A7:

$$
\begin{aligned}
x_1, x_2, \ldots, x_n &\in \{1, 2, \ldots, 7\} \\
\text{alldifferent}(x_1, x_2, \ldots, x_n) & \\
x_1 \leq 4 &\Rightarrow \text{přidej proměnnou } y_1 \in \{5, 6, \ldots 15\} \\
x_2 \leq 5 &\Rightarrow \text{přidej proměnnou } y_2 \in \{6, 7, 8\} \\
x_3 \leq 5 &\Rightarrow \text{přidej proměnnou } y_3 \in \{5, 6, 8\} \\
x_4 \leq 5 &\Rightarrow \text{přidej proměnnou } y_4 \in \{5, 6, 8\} \\
x_5 \leq 2 &\Rightarrow \text{přidej proměnnou } y_5 \in \{4, 5, \ldots 9\} \\
x_6 \leq 2 &\Rightarrow \text{přidej proměnnou } y_6 \in \{4, 5, \ldots 9\} \\
x_7 \leq 2 &\Rightarrow \text{přidej proměnnou } y_7 \in \{5, 6, \ldots 11\}
\end{aligned}
$$

Další dvě varianty A6 a A5 vznikly odebráním proměnné x_7 a odebráním proměnných x_6, x_7. Příklady B7, B6 a B5 vzniknou z A7, A6 a A5 změnou pravidel pro přidávání proměnných $y_1, y_2, \ldots, y_7$:

$$
\begin{aligned}
x_1 \leq 1 &\Rightarrow \text{přidej proměnnou } y_1 \in \{5, 6, \ldots 10\} \\
x_2 \leq 1 &\Rightarrow \text{přidej proměnnou } y_2 \in \{6, 7, 8\} \\
x_3 \leq 1 &\Rightarrow \text{přidej proměnnou } y_3 \in \{5, 6, 8\} \\
x_4 \leq 1 &\Rightarrow \text{přidej proměnnou } y_4 \in \{5, 6, 8\} \\
x_5 \leq 2 &\Rightarrow \text{přidej proměnnou } y_5 \in \{4, 5, \ldots 9\} \\
x_6 \leq 2 &\Rightarrow \text{přidej proměnnou } y_6 \in \{4, 5, \ldots 9\} \\
x_7 \leq 2 &\Rightarrow \text{přidej proměnnou } y_7 \in \{5, 6, \ldots 9\}
\end{aligned}
$$

Pro každý z těchto případů jsem změřil čas potřebný pro nalezení všech řešení[2]. Výsledky ukazuje následující tabulka:

[2]Měření probíhala na počítači Intel Celeron 333Mhz

příklad	počet řešení	SICStus	"zbytečné" proměnné	dynamicky
A7	5280	1.9s	6.23s	4.31s
A6	12216	3.66s	13.06s	9.20s
A5	16908	4.44s	15.72s	11.69s
B7	9000	4.13s	13.97s	8.89s
B6	12600	5.15s	16.85s	9.14s
B5	6390	2.34s	6.94s	3.78s

V sloupečku SICStus je doba výpočtu pomocí "zbytečných" proměnných a podmínky alldifferent dodávané společně se SICStus prologem. Sloupeček zbytečné proměnné udává délku výpočtu pomocí mé implementace podmínky alldifferent a "zbytečných" proměnných. Konečně ve sloupečku dynamicky je doba řešení pomocí dynamické podmínky alldifferent.

Počet "zbytečných" proměnných byl vždy 7. U příkladů B by sice stačily pouze 2 přidané proměnné, ale v praxi není tak jednoduché počet "zbytečných" proměnných odhadnout, proto je schválně takto nadsazený.

Jak je vidět, moje implementace je zhruba 3.5-krát pomalejší než ta dodávaná se SICStus prologem. Modul CLPFD pro programování s podmínkami je v SICStus prologu naprogramován v jazyce C. Rozhraní je však pouze v prologu, takže podmínka alldifferent, přestože je také naprogramovaná v C, musí neustále volat prolog, což je zdlouhavé. Naproti tomu podmínka alldifferent ze SICStus prologu může volat CLPFD přímo (je to patrné z části zdrojového kódu dodávaného se SICStus prologem).

U stejné implementace podmínky alldifferent je ale zřejmé, že dynamická verze se vyplatí, zvláště u příkladů B, kde je počet "zbytečných" proměnných více nadsazený.

Kapitola 5

Dávkové zpracování

V této kapitole se budeme zabývat rozvrhováním dávkové výroby s nastavovacím časem závislým na pořadí (*batch processing with sequence dependent setup times*). Podobnými problémy se zabývá operační výzkum ([14], [15]). V programování s omezujícími podmínkami se však tímto směrem výzkum zatím moc nevedl, jediná práce v tomto oboru kterou jsem našel, je [16] pojednávající o nastavovacích časech.

5.1 Definice

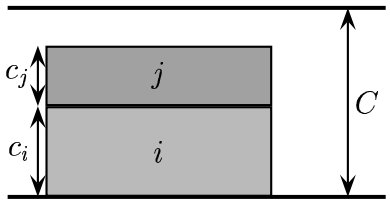
Máme k dispozici zdroj s kapacitou C. Na něm mají být zpracovány úkoly z množiny T. Každý úkol $i \in T$ má následující atributy:

- čas r_i (*release time*), ve kterém nejdříve může začít jeho zpracování.
- čas d_i (*due time*), kdy nejpozději musí zpracování skončit.
- kapacitu c_i (*capacity*), kterou během zpracování úloha spotřebovává ze zdroje.
- druh úkolu f_i (*family*)

Množinu všech typů označíme $F = \{f_i \mid i \in T\}$. Úkoly stejného typu $f \in F$ mají všechny stejný čas potřebný pro zpracování úkolu p_f (*processing time*).

Zdroj může současně zpracovávat pouze úkoly stejného druhu, které začaly ve stejný okamžik, navíc součet jejich požadovaných kapacit nesmí překročit kapacitu zdroje C.

Před zpracováním úkolů typu f je třeba zdroj přenastavit. Jak dlouho přenastavení trvá záleží také na typu g, který zdroj zpracovával těsně před ním (*sequence-dependent setup time*). Délku tohoto přenastavení budeme značit s_{gf}.



Obrázek 5.1: Pokud jsou úkoly i a j stejného typu (tj. $f_i = f_j$), tak mohou být zpracovány zároveň, tak jak je na obrázku.

Mezi dvěmi úkoly stejného typu není třeba zdroj přenastavovat, tedy:

$$\forall f \in F : \quad s_{ff} = 0$$

Dále budeme předpokládat, že přenastavování vyhovuje trojúhelníkové nerovnosti:

$$\forall f_1, f_2, f_3 \in F : \quad s_{f_1 f_3} \leq s_{f_1 f_2} + s_{f_2 f_3} \tag{5.1}$$

Tato nerovnost je klíčová v celém zbytku kapitoly, bez ní by byla situace mnohem komplikovanější. Stejný předpoklad je použit také v [16].

Není dáno, na který typ zpracování je zdroj nastaven před začátkem veškerého zpracování, tj. první úkol může začít bez jakéhokoli nastavování. Pokud bychom ale chtěli, aby zdroj byl na začátku nastaven na typ zpracování f, můžeme navíc přidat další úkol typu f, který bude mít pevně určenou dobu zpracování před všemi zbylými úkoly.

Značení zavedené pro typy rozšíříme také na úkoly, tj:

$$p_i = p_{f_i}$$
$$s_{ij} = s_{f_i f_j}$$

Počet úkolů budeme značit $n = |T|$, počet jejich typů $k = |F|$. Dále budeme předpokládat, že k je oproti n malé.

Ve zbytku kapitoly budeme hledat propagační algoritmy pro dávkové zpracování, tj. budeme zkoumat, kdy je zadání splnitelné (kdy lze aktivity rozvrhnout) a zmenšovat intervaly $\langle r_i, d_i \rangle$ odebíráním neplatných hodnot.

Upravím některé algoritmy pro rozvrhovací podmínky z kapitoly 3 a navrhnu dvě nové podmínky pro tento problém. Všechny následující algoritmy se navzájem doplňují. Volil jsem jen takové algoritmy, abych celkovou časovou složitost udržel $O(kn^2)$ (plus úvodní fáze s časovou složitostí $O(k^3 2^k)$ pro vybudování polí s délkami přechodů). Jako u všech rozvrhovacích podmínek není propagace úplná a je třeba ji iterovat dokud způsobuje další změny (ani poté není samozřejmě propagace úplná).

5.2 Délka přechodů

Pro algoritmy v dalších kapitolách budeme často potřebovat znát minimální dobu, kterou strávíme přenastavováním zdroje při výrobě úkolů typů $\phi \subseteq F$ (můžeme si zvolit, jak je zdroj na začátku nastaven). Tuto dobu budeme značit $s(\phi)$. Číslo $s(f, \phi)$ udává totéž, ale za podmínky, že zpracování začneme typem $f \in \phi$; podobně $s(\phi, f)$ za podmínky, že zpracování skončíme typem f.

V [16] se tyto hodnoty počítají teprve až když jsou skutečně potřeba. Přesný výpočet je časově náročný, proto se počítá pouze dolní odhad. My zvolíme jiný přístup: budeme předpokládat, že k je malé a hodnoty $s(\phi)$, $s(f, \phi)$ a $s(\phi, f)$ si spočteme dopředu a uložíme do pole. Ukážu, jak hodnoty s spočítat v čase $O(k^2 2^k)$.

Je zřejmé, že:

$$\forall \phi \subseteq F: \quad s(\phi) = \min\{s(f, \phi) \mid f \in \phi\}$$

Pro jednoprvkové množiny je čas potřebný k přenastavení 0:

$$\forall f \in F: \quad s(f, \{f\}) = 0$$

Vzhledem k tomu, že pro přenastavování zdroje platí trojúhelníková nerovnost 5.1, můžeme další hodnoty $s(f, \phi)$ spočítat rekurzivně:

$$\forall \phi \subset F, \forall f \in (F \setminus \phi): \; s(f, \{f\} \cup \phi) = \min\{s_{fg} + s(g, \phi) \mid g \in \phi\}$$

Hodnoty $s(\phi, f)$ spočítáme podobně.

Funkci $s(f, \phi)$ je podobná funkce $q(f, g, \phi)$. Také udává minimální délku přechodů při zpracování typů ϕ, když musíme začít typem f. Navíc však zpracování typu g musí být alespoň jednou přerušeno jiným typem. Funkci $q(f, g, \phi)$ můžeme spočítat rekurzivně v čase $O(k^3 2^k)$ podobně jako $s(f, \phi)$:

$$\begin{aligned} \forall f \in F: \quad & q(f, f, \{f\}) = \infty \\ \forall \phi \subset F, \phi \neq \emptyset, \forall f \in (F \setminus \phi), \forall g \in \phi: & \\ & q(f, f, \phi \cup \{f\}) = \min\{s_{fh} + s(h, \phi \cup \{f\}) \mid h \in \phi\} \\ & q(f, g, \phi \cup \{f\}) = \min\{s_{fh} + q(h, g, \phi) \mid h \in \phi\} \end{aligned}$$

Zavedeme ještě funkci $q(g, \phi)$:

$$\forall \phi \subset F, \forall g \in \phi: \quad q(g, \phi) = \min\{q(f, g, \phi) \mid f \in \phi\}$$

Všechny tyto hodnoty si spočítáme hned při vytvoření podmínky. Výsledky si ukládáme do pole indexovaného bitovou maskou množiny ϕ, tj. např. hodnotu $s(\{1, 3, 7\})$ najdeme v $s[2^1 + 2^3 + 2^7]$.

5.3 Délka zpracování

V minulé kapitole jsme pro danou množinu úloh zkoumali minimální dobu strávenou přechody mezi typy. V této kapitole budeme hledat minimální dobu, kterou strávíme skutečným zpracováním.

Uvažujme libovolnou množinu úkolů $\Omega \subseteq T$. Součet kapacit všech úkolů z Ω typu f označme $c(\Omega, f)$:

$$c(\Omega, f) = \sum_{\substack{i \in \Omega \\ f_i = f}} c_i$$

Zpracování úkolů typu f z množiny Ω tedy bude trvat minimálně čas:

$$u(\Omega, f) = \left\lceil \frac{c(\Omega, f)}{C} \right\rceil p_f$$

protože $\left\lceil \frac{c(\Omega,f)}{C} \right\rceil$ je minimální počet dávek délky p_f, které budou potřeba.

Minimální doba pro zpracování všech úkolů z Ω bez přenastavování zdroje pak je:

$$u(\Omega) = \sum_{f \in F} u(\Omega, f)$$

A s přenastavováním zdroje:

$$p(\Omega) = s(F_\Omega) + u(\Omega)$$

Kde F_Ω je množina všech typů úloh z množiny Ω, tj. $F_\Omega = \{f_i \mid i \in \Omega\}$.

Pokud musí zpracování začínat případně končit úkolem $j \in \Omega$, pak je minimální doba zpracování množiny Ω:

$$\begin{aligned} p(j, \Omega) &= s(f_j, F_\Omega) + u(\Omega) \\ p(\Omega, j) &= s(F_\Omega, f_j) + u(\Omega) \end{aligned}$$

Při zpracování úkolů Ω tedy zbývá časová rezerva:

$$m(\Omega) = d_\Omega - r_\Omega - p(\Omega)$$

5.4 Algoritmus edge-finding

Upravíme algoritmus edge-finding z kapitoly 3.6 pro dávkové zpracování.

Pravidlo splnitelnosti

Aby vůbec mohl existovat nějaký rozvrh, musí pro každou množinu Ω být dost času pro zpracování všech jejich úkolů (obdoba pravidla 3.1):

$$\forall \Omega \subseteq T: \quad m(\Omega) < 0 \Rightarrow \text{fail} \tag{5.2}$$

Pravidla edge-finding

Upravíme pravidla 3.2 – 3.5. Pro algoritmus edge-finding podle [11] jsme předpoklad pravidla 3.2 (které odvozuje $i \ll \Omega$) rozepsali do dvou nerovností 3.7 a 3.8. Při úpravě pro dávkové zpracování upravíme tyto nerovnosti zvlášť.

Vezměme si libovolnou množinu $\Omega \subset T$ a úkol i, který v ní neleží. Pokud v intervalu $\langle r_\Omega, d_\Omega \rangle$ není dost času pro zpracování úkolů $\Omega \cup \{i\}$, pak úkol i musí být zpracován buď před, nebo až po všech úkolech z Ω:

$$\forall \Omega \subset T, \forall i \in (T \setminus \Omega): \\ d_\Omega - r_\Omega < p(\Omega \cup \{i\}) \Rightarrow (\Omega \ll i \ \vee \ i \ll \Omega) \tag{5.3}$$

Je důležité, že přenastavování splňuje trojúhelníkovou nerovnost 5.1. Bez ní bychom totiž nemohli vyvodit, že úkol i nemůže být zpracován mezi úkoly Ω. Vložení dalšího úkolu mezi Ω by totiž mohlo zkrátit čas potřebný k přenastavování. Podobně důležitá je trojúhelníková nerovnost i u dalších pravidel v celé kapitole dávkové zpracování, ale nebudeme už na to více upozorňovat.

Potřebujeme ještě vyloučit jednu z možností $i \ll \Omega$ nebo $\Omega \ll i$. Když by úkol i byl zpracován před Ω, mohlo by zpracování úkolů $\Omega \cup \{i\}$ začít nejdříve v čase r_i právě úkolem i. Jeho zpracování bude trvat p_{f_i} a teprve poté mohou být zpracovány úkoly Ω. To bude bez přenastavování trvat minimálně $u(\Omega)$. Přenastavování bude trvat minimálně $s(f_i, F_\Omega \cup \{f_i\})$, dohromady proto zpracování skončí nejdříve v čase $r_i + p_{f_i} + u(\Omega) + s(f_i, F_\Omega \cup \{f_i\})$. Pokud je toto číslo větší než d_Ω, tak zpracování i před Ω není možné. Dostáváme tedy pravidlo 5.4 a symetrické pravidlo 5.5:

$$\forall \Omega \subset T, \forall i \in (T \setminus \Omega): \\ d_\Omega - r_i < p_{f_i} + u(\Omega) + s(f_i, F_\Omega \cup \{f_i\}) \Rightarrow i \not\ll \Omega \tag{5.4}$$

$$d_i - r_\Omega < u(\Omega) + p_{f_i} + s(F_\Omega \cup \{f_i\}, f_i) \Rightarrow \Omega \not\ll i \tag{5.5}$$

Jde o nová pravidla not-before/not-after (ne not-first/not-last), která ještě použijeme v kapitole 5.6.

Ze znalosti $\Omega \ll i$ nebo $i \ll \Omega$ můžeme odvodit změnu r_i případně d_i:

$$\Omega \ll i \Rightarrow r_i \geq \max\{r_{\Omega'} + u(\Omega') + s(F_{\Omega'} \cup \{f_i\}, f_i) \mid \Omega' \subseteq \Omega\} \tag{5.6}$$

$$i \ll \Omega \Rightarrow d_i \leq \min\{d_{\Omega'} - u(\Omega') - s(f_i, F_{\Omega'} \cup \{f_i\}) \mid \Omega' \subseteq \Omega\} \tag{5.7}$$

Algoritmus

Podobně jako u původního algoritmu edge-finding je jednoduché ukázat, že pokud místo množiny $\Omega \subseteq T$ zvolíme množinu $\{k \mid k \in T \ \wedge \ r_k \geq r_\Omega \ \wedge \ d_k \leq d_\Omega\}$, dostaneme ve všech pravidlech stejné nebo ještě lepší výsledky. Za množinu Ω tudíž stačí volit množiny ve tvaru intervalu úkolů $[i,j] = \{k \mid k \in T \ \wedge \ r_k \geq r_i \ \wedge \ d_k \leq d_j\}$.

Dopředu si setřídíme úkoly vzestupně podle r_i, to zabere čas $O(n \log n)$. Následující algoritmus pak v čase $O(n^2)$ ověří podmínku splnitelnosti pro všechny množiny Ω ve tvaru intervalu úloh. Jde o stejný algoritmus jako v kapitole 3.6, jen zde musíme jinak počítat p_Ω:

```
for j ∈ T do begin
  // Vytvoříme prázdnou množinu Ω = ∅
  mask := 0;
  p_Ω := 0;
  for f ∈ F do
    c(Ω, f) := 0;

  // Ω budeme postupně rozšiřovat doleva na intervaly úkolů [i, j].
  for i ∈ T v pořadí klesajícího r_i do begin
    if d_i > d_j then
      // [i, j] není interval úkolů
      continue;

    // Přidáme úkol i do množiny Ω a přepočteme p_Ω.
    if c(Ω, f_i) = 0 then begin
      // V množině Ω zatím není žádný úkol typu f_i.
      p_Ω := p_Ω − s[mask];
      mask := mask + 2^{f_i};
      p_Ω := p_Ω + s[mask];
    end;
    p_Ω := p_Ω − ⌈c(Ω, f_i)/C⌉p_{f_i};
    c(Ω, f_i) := c(Ω, f_i) + c_i;
    p_Ω := p_Ω + ⌈c(Ω, f_i)/C⌉p_{f_i};

    // Ověříme platnost podmínky splnitelnosti:
    if d_j − r_i < p_Ω then fail;
  end;
end;
```

Není tak úplně přesné, že každá množina Ω, pro kterou v algoritmu pod-

mínku splnitelnosti ověřujeme, je nějaký interval úkolů $[i, j]$. Pokud totiž dva různé úkoly i, l mají $r_i = r_l$ a $d_i = d_l$, tak $[i, j] = [l, j]$, v algoritmu však pro tyto intervaly úkolů pracujeme s různými množinami Ω. Každopádně však jednou skutečný interval $[i, j] = [l, j]$ otestujeme.

Algoritmus pro zbylé pravidla ukážu pouze pro případ $\Omega \ll i$, případ $i \ll \Omega$ je symetrický.

Zvolme si pevně úkol j a typ g, vyvodíme všechny změny vyplývající ze všech intervalů úloh $[i, j]$ pro úkoly typu g.

Algoritmus je založen na následující úvaze. Ze všech intervalů $[i, j]$ vytvoříme posloupnost $\Omega_0 \subsetneq \Omega_1 \subsetneq \cdots \subsetneq \Omega_x$, $d_{\Omega_0} = d_{\Omega_1} = \cdots = d_{\Omega_x} = d_j$. Úkoly typu g setřídíme vzestupně podle požadované kapacity c_i do posloupnosti $t_0, t_1, \ldots t_y$. Nyní uvažujme dvojici Ω_x, t_y, musí nastat jedna ze čtyř možností:

1. $d_{t_y} \leq d_{\Omega_x}$. Ukážeme, že tento případ není třeba řešit, protože kdyby platilo 5.3 a 5.4, vyvodí symetrický algoritmus edge-finding pro $i \ll \Omega$ společně s podmínkou splnitelnosti 5.2 fail. A protože celý algoritmus edge-finding se iteruje, fail se určitě vyvodí.

   Předpokládejme tedy, že pro t_y a Ω_x platí 5.3 a 5.4. Z $d_{t_y} \leq d_{\Omega_x}$ a 5.3 plyne:

   $$d_{t_y} - r_{\Omega_x} \leq d_{\Omega_x} - r_{\Omega_x} < p(\Omega_x \cup \{t_y\})$$

   Je zřejmé, že:

   $$p(\Omega_x \cup \{t_y\}) \leq u(\Omega_x) + p_g + s(F_{\Omega_x} \cup \{g\}, g)$$

   Z toho vyplývá, že pro t_y a Ω_x platí také 5.5 což společně s 5.3 vyvodí $i \ll \Omega$. Po opravě d_{t_y} podle 5.7 pak bude:

   $$d_{t_y} \leq d_{\Omega_x} - u(\Omega_x) - s(g, F_{\Omega_x} \cup \{g\})$$

   I nadále však pro t_y a Ω_x bude platit 5.4 a proto:

   $$r_{t_y} > d_{\Omega_x} - p_g - u(\Omega_x) - s(g, F_{\Omega_x} \cup \{g\})$$

   Nyní uvažujme množinu $\Psi = \{t_y\}$. Vidíme:

   $$\begin{aligned} d_\Psi - r_\Psi = d_{t_y} - r_{t_y} \leq \\ \leq [d_{\Omega_x} - u(\Omega_x) - s(g, F_{\Omega_x} \cup \{g\})] - r_{t_y} < \\ < [d_{\Omega_x} - u(\Omega_x) - s(g, F_{\Omega_x} \cup \{g\})] - \\ - [d_{\Omega_x} - p_g - u(\Omega_x) - s(g, F_{\Omega_x} \cup \{g\})] = \\ = p_g = p_\Psi \end{aligned}$$

Tedy $d_\Psi - r_\Psi < p_\Psi$ a podmínka splnitelnosti 5.2 vyvodí fail.

Zbývá ukázat, že symetrický algoritmus edge-finding pro $i \ll \Omega$ nevynechá případ t_y a Ω_x s podobným odůvodněním, tj. že ho vyřeší algoritmus pro $\Omega \ll i$ a podmínka splnitelnosti.

Aby algoritmus pro $i \ll \Omega$ tento případ vynechal, muselo by být $r_{t_y} \geq r_{\Omega_x}$, a tedy $t_y \in \Omega_x$. Pak ovšem pravidla 5.3 - 5.5 nelze na t_y a Ω_x vůbec aplikovat.

Ukázali jsme, že když $d_{t_y} \leq d_{\Omega_x}$, tak vůbec není třeba 5.3 a 5.4 aplikovat. Protože $d_{\Omega_0} = d_{\Omega_1} = \ldots d_{\Omega_x}$, tak není třeba 5.3 a 5.4 aplikovat pro t_y a žádnou z množin $\Omega_0, \Omega_1, \ldots, \Omega_x$.

V následujících případech už proto bude $d_{t_y} > d_{\Omega_x}$ a proto také $t_y \notin \Omega_x$.

2. Pro množinu Ω_x a úkol t_y neplatí podmínka 5.3, tj. úkol t_y lze vykonat společně s úkoly Ω_x. Pak lze ovšem společně s úkoly Ω_x vykonat i úkoly $t_0, t_1, \ldots, t_{y-1}$, protože ty potřebují stejně nebo ještě méně kapacity než úkol t_y. Množinu Ω_x tedy můžeme zahodit (tj. zkrátit posloupnost $\Omega_0, \Omega_1, \ldots, \Omega_x$), protože nevyvodí žádné změny.

3. Pro množinu Ω_x a úkol t_y neplatí podmínka 5.4, tj. úkol t_y může být zpracován ještě před množinou Ω_x. Pak ovšem může být úkol t_y zpracován i před množinami $\Omega_0, \Omega_1, \ldots, \Omega_{x-1}$. Úkol t_y tedy můžeme zahodit (tj. zkrátit posloupnost $t_0, t_1, \ldots t_y$).

4. Pro Ω_x a t_y platí 5.3 i 5.4. Můžeme tedy opravit hodnotu r_{t_y} podle pravidla 5.6. Tato hodnota r_{t_y} už nepůjde pomocí množin $\Omega_0, \Omega_1, \ldots, \Omega_{x-1}$ zlepšit (to vyplývá z tvaru pravidla 5.6). Můžeme tedy úkol t_y zahodit (zkrátit posloupnost $t_0, t_1, \ldots, t_y$).

Abychom mohli upravit r_{t_y} podle pravidla 5.6, musíme znát hodnotu

$$z_x = \max\{r_{\Omega'} + u(\Omega') + s(F_{\Omega'} \cup \{g\}, g) \mid \Omega' \subseteq \Omega_x\}$$

Vezměme si libovolnou množinu $\Omega' \subseteq \Omega_x$ a k ní vytvořme množinu $\Omega'' = \{k \mid k \in \Omega_x \ \wedge \ r_k \geq r_{\Omega'}\}$. Je zřejmé, že pak:

$$r_{\Omega'} + u(\Omega') + s(F_{\Omega'} \cup \{g\}, g) \leq r_{\Omega''} + u(\Omega'') + s(F_{\Omega''} \cup \{g\}, g)$$

takže za množiny Ω' stačí volit pouze intervaly úkolů $[i, j]$ takové, že $d_j = d_\Omega$. To jsou ale přesně množiny $\Omega_0, \Omega_1, \ldots, \Omega_x$. Tedy:

$$z_y = \max\{r_{\Omega'} + u(\Omega') + s(F_{\Omega'} \cup \{g\}, g) \mid \Omega' \in \{\Omega_0, \Omega_1, \ldots, \Omega_x\}\}$$

A odtud dostáváme vzoreček pro rekurzivní výpočet z_y:

$$z_y = \max\{z_{y-1}, r_{\Omega_y} + u(\Omega_y) + s(F_{\Omega_y} \cup \{g\}, g)\}$$

Úlohy si předem setřídíme podle r_i a také c_i. Samotný algoritmus pak je:

```
for g ∈ F do begin
  x := 0; // Prozatímní počet množin v posloupnosti
  for j ∈ T do begin
    // Vytvoříme prázdnou množinu Ω = ∅
    mask := 0;
    u(Ω) := 0;
    for f ∈ F do
      c(Ω, f) := 0;

    // Ω budeme postupně rozšiřovat doleva na intervaly úkolů [i, j].
    for i ∈ T v pořadí klesajícího r_i do begin
      if d_i > d_j then
        // [i, j] není interval úkolů
        continue;

      // Přidáme úkol i do množiny Ω a přepočteme u_Ω.
      if c(Ω, f_i) = 0 then mask := mask + 2^{f_i};
      u(Ω) := u(Ω) − ⌈c(Ω, f_i)/C⌉ p_{f_i};
      c(Ω, f_i) := c(Ω, f_i) + c_i;
      u(Ω) := u(Ω) + ⌈c(Ω, f_i)/C⌉ p_{f_i};

      // Právě jsme vytvořili Ω_x, zapíšeme si hodnoty do pole:
      mask[x] := mask;
      u[x] := u(Ω);
      c[x, g] := c(Ω, g);
      r[x] := r_i;
      if x=0 then
        z[x] := r_i + u(Ω)+s[mask OR 2^g, g];
      else
        z[x] := max(z[x−1], r_i + u(Ω)+s[mask OR 2^g, g]);
        // V předchozím přiřazení je OR jako bitová operace
      x := x+1;
    end;

    y := úkol typu g s největší požadovanou kapacitou c_y;
    while (y ≥ 0 AND x ≥ 0) do begin
```

```
        if d_y ≤ d_j then
        begin
          // Podle bodu 1 tento případ není třeba testovat
          y := další úkol typu g s menším nebo stejným c_y, jinak -1;
          continue;
        end;

        // Otestujeme pravidlo 5.3
        // Spočteme p(Ω_x ∪ {y}):
        p := s[mask[x] OR 2^g]+u[x]-⌈c[x, g]/C⌉*p_g;
        p := p+⌈(c[x, g]+c_y)/C⌉*p_g;
        if d_j - r[x] ≥ p then
          // Podle bodu 2 můžeme zahodit množinu Ω_x
          x := x-1;
          continue;
        end;

        // Otestujeme pravidlo 5.4 pro Ω_x a y.
        // Spočteme s(g, F_{Ω_x} ∪ {g}) + u(Ω_x) + p_g:
        p := s[g, mask[x] OR 2^g]+u[x]+p_g;
        if d_j - r_y ≥ p then
          // Podle bodu 3 můžeme úkol y zahodit:
          y := další úkol typu g s menším nebo stejným c_y, jinak -1;
          continue;
        end;

        // Platí 5.3 i 5.4.
        // Podle bodu 4 opravíme r_y a úkol y zahodíme:
        r_y := max(r_y, z[x]);
        y := další úkol typu g s menším nebo stejným c_y, jinak -1;
      end;
    end;
end;
```

Jak vidíme, časová složitost druhé části algoritmu edge-finding se oproti disjunktivnímu rozvrhování prodloužila z $O(n^2)$ na $O(kn^2)$.

5.5 Současné zpracování úkolů

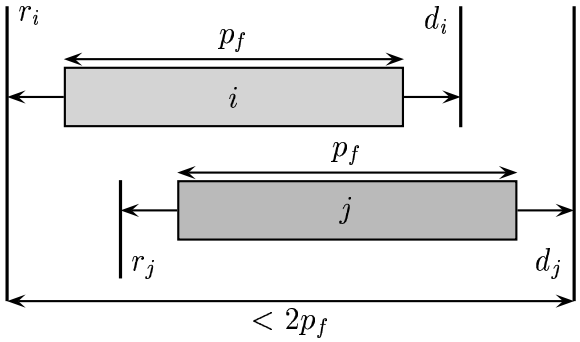
V dávkovém zpracování mohou být úkoly zpracovány současně pouze tehdy, když jsou stejného typu a jejich zpracování začne současně. V této kapitole budeme hledat, jaké zmenšení domén bychom z toho mohli vyvodit.

5.5.1 Spojování úkolů

Vezměme si dvě libovolné úlohy i, j stejného typu f. Když platí

$$\max\{d_i, d_j\} - \min\{r_i, r_j\} < 2p_f \tag{5.8}$$

tak tyto dvě úlohy musí být zpracovány současně:



Za těchto podmínek můžeme odvodit:

$$r_i := \max\{r_i, r_j\} \tag{5.9}$$
$$r_j := \max\{r_i, r_j\} \tag{5.10}$$
$$d_i := \min\{d_i, d_j\} \tag{5.11}$$
$$d_j := \min\{d_i, d_j\} \tag{5.12}$$

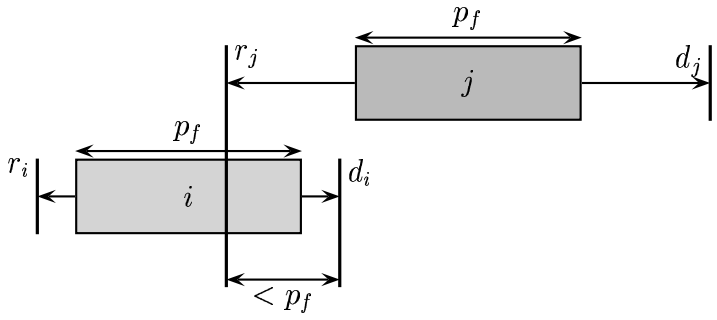
Toto pravidlo můžeme jednoduše aplikovat na všechny dvojice úloh stejného typu v čase $O(n^2)$. V další kapitole však navrhneme algoritmus not-before/not-after s časovou složitostí $O(kn^2)$, který je ještě silnější.

5.5.2 Rozdělení úkolů

Můžeme také uvažovat o tom, kdy dva úkoly naopak nemohou být zpracovány současně. Rozlišíme dva případy, podle toho jestli jsou nebo nejsou stejného typu.

Úkoly stejného typu

Uvažujme dva úkoly i a j, $r_i < r_j$, stejného typu f. Tyto úkoly musí být zpracovány odděleně, pokud je čas ve kterém mohou být zpracovány současně příliš krátký:



Tedy:

$$d_i - r_j < p_f \tag{5.13}$$

Pak musí být nejdříve zpracován úkol i a teprve potom úkol j, můžeme tedy odvodit:

$$d_i \leq d_j - p_f \tag{5.14}$$
$$r_j \geq r_i + p_f \tag{5.15}$$

Úkoly různého typu

Dva úkoly i, j různého typu nikdy nesmí být zpracovány zároveň. Když chceme nějak omezit jejich domény, musíme nejprve rozhodnout, v jakém pořadí se musí tyto úkoly zpracovat. Úkol i musí být před j, pokud opačné pořadí není možné, tj:

$$r_j + p_{f_j} + s_{f_j f_i} + p_{f_i} > d_i \tag{5.16}$$

A z toho můžeme vyvodit:

$$d_i \leq d_j - p_{f_j} - s_{f_i f_j} \tag{5.17}$$
$$r_j \geq r_i + p_{f_i} + s_{f_i f_j} \tag{5.18}$$

Pravidla pro rozdělení úkolů můžeme také snadno použít v čase $O(n^2)$, jak však ukážeme později, můžeme použít silnější algoritmus not-first/not-last s časovou složitostí $O(kn^2)$.

5.6 Algoritmus not-before/not-after

Vraťme se k pravidlům 5.4, 5.5 not-before/not-after z kapitoly 5.4. Když úkol i nemůže být zpracován před Ω (tj. $i \not\ll \Omega$), tak zpracování úkolu i může začít nejdříve společně s Ω. Tak dostáváme pravidlo 5.19 a symetrickou verzi 5.20.

$$i \not\ll \Omega \Rightarrow r_i \geq r_\Omega \tag{5.19}$$

$$\Omega \not\ll i \Rightarrow d_i \leq d_\Omega \tag{5.20}$$

Věta 7 *Pravidla 5.4, 5.5, 5.19, 5.20 jsou silnější (tj. nalezne více nekonzistencí), než pravidlo pro spojování úkolů 5.8 - 5.12.*

Důkaz: Vezměme si dva úkoly i, j typu f pro které platí $\max\{d_i, d_j\} - \min\{r_i, r_j\} < 2p_f$. Pak pravidla not-before a not-after pro úkol i a množinu $\Omega = \{j\}$ odvodí 5.9 a 5.11, pro úkol j a množinu $\Omega = \{i\}$ zase 5.10 a 5.12. □

Jako obvykle stačí za množiny Ω volit intervaly úkolů. Vezměme si libovolnou množinu Ω a k ní nejmenší interval úkolů, který ji obsahuje, tj. množinu $\Psi = \{j \mid j \in T \ \wedge \ r_j \geq r_\Omega \ \wedge \ d_j \leq d_\Omega\}$. Pro libovolný úkol i pak pravidla 5.4, 5.5, 5.19, 5.20 s množinou Ψ dají tytéž nebo ještě lepší výsledky než s množinou Ω.

Z toho plyne následující věta:

Věta 8 *Nechť i je úkol typu g, j libovolný jiný úkol. Pravidla 5.4 a 5.19 dovolí opravit hodnotu r_i na $r_i \geq r_j$ právě tehdy, když:*

$$r_i > \min\{d_\Omega - s(g, F_\Omega \cup \{g\}) - u(\Omega) - p(g) \mid \Omega = [j, k], k \in T\} \tag{5.21}$$

Důkaz: Nerovnost 5.21 platí právě tehdy, když existuje nějaký interval úkolů $[j, k]$, pro který platí 5.4. A jak jsme ukázali, za Ω stačí volit právě intervaly úkolů. □

Z této věty vyplývá následující algoritmus s časovou složitostí $O(kn^2)$ pro 5.4 a 5.19:

```
for g ∈ F do begin
  for j ∈ T do begin
    m := ∞;
    Ω := ∅;
    for k ∈ T v pořadí rostoucího d_k do begin
      if r_k < r_j then
        // [j, k] není interval úloh
```

```
        continue;
      Ω := Ω ∪ {k};
      m := min{d_Ω − s(g, F_Ω ∪ {g}) − u(Ω) − p(g), m};
    end;

    for i ∈ T, f_i = g do
      if r_i > m then
        r_i ≥ r_j;
  end;
end;
```

Algoritmus pro 5.5 a 5.20 je symetrický.

5.7 Algoritmus not-first/not-last

V této kapitole upravíme algoritmus not-first/not-last z kapitoly 3.7 pro dávkové zpracování. Ukážeme, že tento algoritmus je silnější než pravidla pro rozdělování úkolů z kapitoly 5.5.

Na rozdíl od not-before/not-after, nás budou zajímat takové úkoly i a množiny Ω, že úkol i nemůže začít před Ω, ale také nemůže začít současně s Ω. Tuto vlastnost budeme značit $i \not\ll \Omega$ (podobně $\Omega \not\ll i$):

$$\forall \Omega \subset T, \forall i \in (T \setminus \Omega): \quad d_\Omega - r_i < p(i, \Omega \cup \{i\}) \Rightarrow i \not\ll \Omega \tag{5.22}$$

$$\forall \Omega \subset T, \forall i \in (T \setminus \Omega): \quad d_i - r_\Omega < p(\Omega \cup \{i\}, i) \Rightarrow \Omega \not\ll i \tag{5.23}$$

Pokud úkol i nemůže začít před Ω ani současně s Ω, tak může začít teprve až skončí alespoň jeden úkol z Ω:

$$i \not\ll \Omega \Rightarrow r_i \geq min\{r_j + p_j + s_{f_j f_i} \mid j \in \Omega\} \tag{5.24}$$

$$\Omega \not\ll i \Rightarrow d_i \leq max\{d_j - p_j - s_{f_i f_j} \mid j \in \Omega\} \tag{5.25}$$

Pravidla 5.22 a 5.23 však vedou k příliš časově náročnému algoritmu. Proto místo nich navrhneme slabší pravidla, která však povedou k časové složitosti $O(kn^2)$. Odhadneme v nich $p(\Omega \cup \{i\})$ zezdola pomocí dolního odhadu kapacity c_i.

Nechť $f \in F$. Označme:

$$\underline{c}_f = \min\{c_i \mid i \in T \ \wedge \ f_i = f\}$$

$$t(f, \Omega) = s(f, F_\Omega \cup \{f\}) + u(\Omega) - \left\lceil \frac{c(\Omega, f)}{C} \right\rceil p_f + \left\lceil \frac{c(\Omega, f) + \underline{c}_f}{C} \right\rceil p_f$$

$$t(\Omega, f) = s(F_\Omega \cup \{f\}, f) + u(\Omega) - \left\lceil \frac{c(\Omega, f)}{C} \right\rceil p_f + \left\lceil \frac{c(\Omega, f) + \underline{c}_f}{C} \right\rceil p_f$$

tedy $t(\Omega, f)$ je nejmenší možné $p(i, \Omega \cup \{i\})$, kde i je úkol typu f. Slabší náhrada pravidel 5.22 a 5.23 je:

$$\forall \Omega \subset T, \forall i \in (T \setminus \Omega): \quad d_\Omega - r_i < t(f_i, \Omega) \Rightarrow i \not\prec \Omega \tag{5.26}$$
$$\forall \Omega \subset T, \forall i \in (T \setminus \Omega): \quad d_i - r_\Omega < t(\Omega, f_i) \Rightarrow \Omega \not\prec i \tag{5.27}$$

Pokud pro všechny úkoly i typu f bude $c_i = \underline{c}_f$, pak jsou pravidla 5.26 a 5.27 ekvivalentní s pravidly 5.22 a 5.23.

Věta 9 *Pravidla 5.24 - 5.27 jsou silnější než pravidla 5.13 - 5.18 pro rozdělování úkolů.*

Důkaz: Vezměme si úkoly i a j, pro které lze uplatnit 5.13 - 5.15 nebo 5.16 - 5.18. Pak pravidla not-first a not-last pro úkol i a množinu $\{j\}$ a pro úkol j a množinu $\{i\}$ odvodí totéž. □

Budeme se zabývat pouze pravidly not-first, tedy 5.22, 5.26 a 5.24; algoritmus pro pravidla not-last je symetrický.

Dále budeme předpokládat, že úkoly jsou seřazeny podle d_i, tj. $i \leq j \Leftrightarrow d_i \leq d_j$. Zvolme si úkol i typu f a další dva úkoly j, k (libovolného typu) takové, že:

$$j \leq k \tag{5.28}$$
$$r_j + p_{r_j} + s_{f_j f} \leq r_k + p_{r_k} + s_{f_k f} \tag{5.29}$$

Pak množinami $\Omega(fjk)$ a $\Omega(ijk)$ budeme myslet:

$$\Omega(fjk) = \{m \mid m \in T \ \wedge \ m \leq k \ \wedge \ r_m + p_{f_m} + s_{f_m f} \geq r_j + p_{f_j} + s_{f_j f}\}$$
$$\Omega(ijk) = \Omega(f_i jk) \setminus \{i\}$$

Pokud j a k nesplňují 5.28 nebo 5.29, tak položíme $\Omega(jk) = \emptyset$ a $d_{\Omega(fjk)} = \infty$.

Dokážeme obdobu věty 3:

Věta 10 *Pokud pravidla 5.26 a 5.24 pro úkol $i \in T$ a množinu $\Omega \subset T$ změní hodnotu r_i na $r_j + p_j + s_{f_j f_i}$, pak existuje úkol $k \in T$ takový, že $k \neq i$, $k \geq j$ a pravidla 5.26, 5.24 pro úkol i a množinu $\Omega(ijk)$ také změní hodnotu r_i na $r_j + p_j + s_{f_j f_i}$.*

Důkaz: Za k zvolme úkol $k = \max\{m \mid m \in \Omega\}$. Je zřejmé, že $j \leq k$, protože $j \in \Omega$. Z volby k vyplývá $\Omega \subseteq \Omega(ijk)$, $d_\Omega = d_{\Omega(ijk)}$. Protože 5.26 platí pro Ω, musí platit i pro $\Omega(ijk)$ a 5.24 pak odvodí $r_i \geq r_j + p_j + s_{f_j f_i}$. □

Označme:

$$\gamma_{j,k}(f) = \min\{d_{\Omega(fjl)} - s(f, F_{\Omega(fjl)}) - u(\Omega(fjl)) \mid l \in \{k, k+1, \ldots, n\}\}$$
$$\delta_{j,k}(f) = \min\{d_{\Omega(fjl)} - t(f, \Omega(fjl)) \mid l \in \{1, 2, \ldots, k\}\}$$

Pro dané j a f můžeme hodnoty $\gamma_{j,k}(f)$ a $\delta_{j,k}(f)$ spočítat v čase $O(n)$.

Ukážeme obdobu věty 4 pro dávkové zpracování:

Věta 11 *Nechť i je úkol typu f, j libovolný jiný úkol. Pokud platí alespoň jedna z možností:*

1. $r_i + p_i < r_j + p_j + s_{f_j f}$ *a* $\delta_{j,n}(f_i) < r_i$.
2. $r_i + p_i \geq r_j + p_j + s_{f_j f}$ *a* $\delta_{j,i-1}(f_i) < r_i$ *nebo* $\gamma_{j,i}(f_i) < r_i$.

tak pravidla 5.22, 5.24 dovolí upravit r_i podle $r_i \geq r_j + p_j + s_{f_j f}$.

Pokud neplatí ani 1. ani 2., není možné úpravy $r_i \geq r_j + p_j + s_{f_j f}$ dosáhnout použitím slabšího pravidla 5.26.

Důkaz: Rozlišíme dvě možnosti:

1. $r_i + p_i < r_j + p_j + s_{f_j f}$. Pak pro libovolný úkol k máme $i \notin \Omega(fjk)$ a tedy $\Omega(ijk) = \Omega(fjk)$. Pravidlo 5.26 nám podle věty 10 dovolí úpravu r_i na $r_i \geq r_j + p_j + s_{f_j f}$ právě tehdy, když existuje úkol l takový, že:

   $$d_{\Omega(fjl)} - t(f, \Omega(fjl)) < r_i$$

   A ten existuje právě tehdy, když:

   $$\min\{d_{\Omega(fjl)} - t(f, \Omega(fjl) \mid l \in T\} < r_i$$

   což je právě tehdy, když $\delta(j, n) < r_i$.
2. $r_i + p_i \geq r_j + p_j + s_{f_j f}$. Podle věty 10 je možné odvodit $r_i \geq r_j + p_j + s_{f_j f}$ právě tehdy, když existuje úkol l takový, že:

   $$d_{\Omega(ijl)} - t(f, \Omega(ijl)) < r_i$$

   Rozlišíme dva případy, podle toho jestli $i \in \Omega(fjl)$ nebo $i \notin \Omega(fjl)$:

   - $i \notin \Omega(fjl)$ a to je právě tehdy, když $i > l$. Pak $\Omega(ijl) = \Omega(fjl)$ a hledané l v tomto případě existuje právě tehdy, když:

     $$\min\{d_{\Omega(fjl)} - t(f, \Omega(fjl)) \mid l \in \{1, 2, \ldots, i-1\}\} < r_i$$

     A to je přesně:

     $$\delta_{j,i-1}(f) < r_i$$

- $i \in \Omega$, tedy musíme projít $l > i$ (případ $l = i$ není třeba podle věty 10 testovat). Tentokrát použijeme silnější pravidlo 5.22. Protože $i \in \Omega(fjl)$, tak:

$$p(i, \Omega(fjl) \cup \{i\}) = p(i, \Omega(fjl))$$

Hledané l v tomto případě existuje právě tehdy, když:

$$\min\{d_{\Omega(fjl)} - p(i, \Omega(fjl)) \mid l \in \{i, i+1, \ldots, n\}\}$$

A to je právě tehdy, když $\gamma_{j,i}(f) < r_i$. □

Z této věty pak vychází následující algoritmus, který v čase $O(kn^2)$ odvodí všechny změny vyplývající z 5.26, 5.24 a navíc některé vyplývající z 5.22:

```
for f ∈ F do begin
  for j ∈ T do begin
    spočti γ_{j,1}(f), γ_{j,2}(f), ... γ_{j,n}(f);
    spočti δ_{j,1}(f), δ_{j,2}(f), ... δ_{j,n}(f);
    for i ∈ T, f_i = f do
      if r_i + p_i < r_j + p_j + s_{f_j f} then begin
        if δ_{j,n}(f_i) < r_i then
          r_i ≥ r_j + p_j + s_{f_j f};
      end else begin
        if δ_{j,i-1}(f_i) < r_i OR γ_{j,i}(f_i) < r_i then
          r_i ≥ r_j + p_j + s_{f_j f};
      end;
  end;
end;
```

5.8 Spojování posloupností

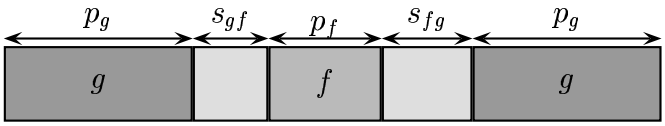
Vezměme si typ g a množinu Ω takovou, že obsahuje alespoň dvě úlohy typu g. Pokud bychom zpracování úkolů typu g z Ω přerušili zpracováním jiného úkolu z Ω, trvalo by nám zpracování množiny minimálně Ω čas:

$$v(\Omega, g) = \begin{cases} q(F_\Omega, g) + u(\Omega) & \text{když } u(\Omega, g) > p_g \\ q(F_\Omega, g) + u(\Omega) + p_g & \text{když } u(\Omega, g) = p_g \end{cases}$$

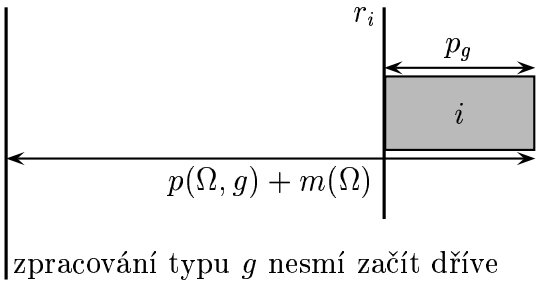
Může se stát, že v intervalu r_Ω – d_Ω tolik času není, tj:

$$d_\Omega - r_\Omega < v(\Omega, g) \tag{5.30}$$

Pak musí být v Ω všechny úkoly typu g zpracovány bez přerušení jiným úkolem z Ω. K přerušení přesto může dojít, ale úkolem, který v Ω není (a to ještě typu, který v Ω také není), jak ukazuje obrázek:



V každém případě však zpracování všech úkolů typu g z Ω (včetně případných vložených úkolů, které nejsou v Ω) může trvat maximálně $p(\Omega, g) + m(\Omega)$. Aby úkol $i \in \Omega$ typu g mohl být zpracován společně s ostatními úkoly typu g, nesmíme začít zpracovávat úkoly typu g dříve než v čase $r_i + p_g - p(\Omega, g) - m(\Omega)$:



Podobně musíme se zpracováním typu g skončit nejpozději v čase $d_i - p_g + p(\Omega, g) + m(\Omega)$.

Zpracování úkolů typu g tedy může začít nejdříve v čase

$$t_1(\Omega) = \max\{r_i \mid i \in \Omega \ \wedge \ f_i = g\} + p_g - p(\Omega, g) - m(\Omega)$$

a skončit nejpozději v čase:

$$t_2(\Omega) = \min\{d_i \mid i \in \Omega \ \ \wedge \ f_i = g\} - p_g + p(\Omega, g) + m(\Omega)$$

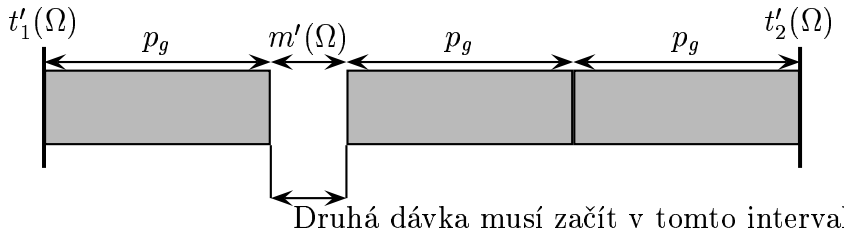
Může se stát, že $t_1(\Omega) < r_\Omega$ nebo $t_2(\Omega) > d_\Omega$). Proto ještě zavedeme hodnoty:

$$t'_1(\Omega) = \max\{t_1(\Omega), r_\Omega\}$$
$$t'_2(\Omega) = \min\{t_2(\Omega), d_\Omega\}$$

Všechny úkoly typu g z množiny Ω pak musí být zpracovány v intervalu $\langle t'_1(\Omega), t'_2(\Omega)\rangle$. Kdybychom v tomto intervalu zpracovávali pouze úkoly typu g z Ω, pak něm ještě zbývá čas:

$$m'(\Omega) = t'_2(\Omega) - t'_1(\Omega) - p(\Omega, g)$$

Prvá dávka zpracování typu g musí začít v intervalu $\langle t'_1(\Omega), t'_1(\Omega)+m'(\Omega)\rangle$, druhá dávka v intervalu $\langle t'_1(\Omega)+p_g, t'_1(\Omega)+p_g+m'(\Omega)\rangle$ atd.:



Z toho můžeme odvodit, jak pro každý úkol $j \in \Omega$ typu g posunout hodnotu r_j:

$$r_j \in \bigcup_{l \in \mathbb{N}_0} \langle t'_1(\Omega) + lp_g, t'_1(\Omega) + lp_g + m'(\Omega)\rangle \tag{5.31}$$

A podobně je to s d_j:

$$d_j \in \bigcup_{l \in \mathbb{N}_0} \langle t'_2(\Omega) - lp_g - m'(\Omega), t'_2(\Omega) - lp_g\rangle \tag{5.32}$$

Sjednocení množin v těchto pravidlech vlastně znamená, že r_i a d_i musí dávat zbytek po dělení p_g z jistého intervalu. Označme proto $\Lambda(a,b,c)$ množinu zbytků čísel $a, a+1, \ldots, b$ po dělení c, tj:

$$\Lambda(a,b,c) = \{a \bmod c, (a+1) \bmod c, \ldots, b \bmod c\}$$

Množina $\Lambda(a,b,c)$ je interval v tělese $\mathbb{N} \bmod c$. Můžeme ho tudíž reprezentovat dvěmi čísly - horním a dolním okrajem. Průnik dvou intervalů $\Lambda(a,b,c)$ a $\Lambda(d,e,c)$ je opět množina ve tvaru $\Lambda(x,y,c)$, její nalezení zabere čas $O(1)$.

Vlastnosti 5.31 a 5.32 nyní můžeme pomocí Λ přepsat na ekvivalentní verzi:

$$r_i \geq t_1(\Omega) \tag{5.33}$$
$$r_i \bmod p_g \in \Lambda(t'_1(\Omega), t'_1(\Omega) + m'(\Omega), p_g) \tag{5.34}$$
$$d_i \leq t_2(\Omega) \tag{5.35}$$
$$d_i \bmod p_g \in \Lambda(t'_2(\Omega) - m'(\Omega), t'_2(\Omega), p_g) \tag{5.36}$$

Věta 12 *Abychom našli všechny změny vyplývající z 5.30, 5.31 a 5.32 stačí volit množiny Ω ve tvaru intervalu úloh.*

Důkaz: Zvolme libovolnou množinu Ω. K ní vytvořme množinu:

$$\Psi = \{i \mid i \in T \ \wedge \ r_i \geq r_\Omega \ \wedge \ d_i \leq d_\Omega\}$$

Množina Ψ je interval úloh a ukážeme, že zmíněná pravidla vyvodí stejné, nebo ještě lepší změny r_i.

Z volby Ψ vyplývá:

$$\begin{aligned} \Psi &\supseteq \Omega \\ r_\Psi &= r_\Omega \\ d_\Psi &= d_\Omega \end{aligned}$$

Z toho plyne, že $v(\Omega, g) \leq v(\Psi, g)$ pro libovolný typ g. Protože 5.30 platí pro Ω, tím spíše pak musí platit pro Ψ.

Z předchozích tří vlastností můžeme dále vyvodit:

$$\begin{aligned} m(\Psi) &\leq m(\Omega) \\ p(\Psi, g) &\geq p(\Omega, g) \\ \max\{r_i \mid i \in \Psi \ \wedge \ f_i = g\} &\geq \max\{r_i \mid i \in \Omega \ \wedge \ f_i = g\} \\ \min\{d_i \mid i \in \Psi \ \wedge \ f_i = g\} &\leq \min\{d_i \mid i \in \Omega \ \wedge \ f_i = g\} \\ t_1(\Psi) &\geq t_1(\Omega) \\ t_2(\Psi) &\leq t_2(\Omega) \\ t'_1(\Psi) &\geq t'_1(\Omega) \\ t'_2(\Psi) &\leq t'_2(\Omega) \\ m'(\Psi) &\leq m'(\Omega) \end{aligned}$$

Dále $p(\Omega, g) + m(\Omega) \geq p(\Psi, g) + m(\Psi)$, protože $u(\Psi, g)$ nemohlo vzrůst oproti $u(\Omega, g)$ více, než kleslo $m(\Psi)$ oproti $m(\Omega)$.

Z toho je vidět že vymezení r_i a d_i podle 5.33 a 5.34 je pro Ψ stejné nebo ještě lepší než pro Ω. □

Kdybychom upravili r_i podle množiny Ω_1 a pravidla 5.34 a později znovu ale podle množiny Ω_2, mohli bychom dostat hodnotu r_i, která už by ale neodpovídala 5.34 pro množinu Ω_1. Proto nebudeme pravidla 5.34 a 5.36 uplatňovat okamžitě, ale jen si zapamatujeme v kterém intervalu Λ má $(r_i \bmod p_i)$ případně $(d_i \bmod p_i)$ ležet. Tyto intervaly budeme skládat a až úplně nakonec provedeme samotnou změnu r_i a d_i.

Zvolme si úkol j a typ g. Vytvoříme řadu všech intervalů úloh $\Omega_0 \subseteq \Omega_1 \subseteq \cdots \subseteq \Omega_x$ takových, že $d_j = d_{\Omega_0} = d_{\Omega_1} = \cdots = d_{\Omega_x}$, tj. posloupnost intervalů se stejným $d_\Omega = d_j$ postupně se rozšiřující doleva.

Označme:

$$
\begin{aligned}
n(\Omega, f) &= |\{k \mid k \in \Omega \ \wedge \ c_k = f\}| \\
M(i) &= \{k \mid k \in \{i, i+1, \ldots, x\} \ \wedge \ d_{\Omega_k} - r_{\Omega_k} < v(\Omega_k, g) \ \wedge \ n(\Omega_k, g) \geq 2\} \\
t_1(i) &= \max\{t_1(\Omega_k) \mid k \in M(i)\} \\
t_2(i) &= \min\{t_2(\Omega_k) \mid k \in M(i)\} \\
\Lambda_r(i) &= \bigcap_{k \in M(i)} \Lambda(t_1'(\Omega_k), t_1'(\Omega_k) + m'(\Omega_k), p_g) \\
\Lambda_d(i) &= \bigcap_{k \in M(i)} \Lambda(t_2'(\Omega_k) - m'(\Omega_k), t_2'(\Omega_k), p_g)
\end{aligned}
$$

Pro daný úkol j a typ g můžeme všechny hodnoty $t_1(i)$, $t_2(i)$, $\Lambda_r(i)$ a $\Lambda_d(i)$ spočítat v čase $O(n)$.

Nyní uvažujme úkol k typu g takový, že $k \in \Omega_i$ a zároveň $k \notin \Omega_{i-1}$. Pak pravidla 5.30, 5.31 a 5.32 pro úkol k a množiny Ω takové, že $d_\Omega = d_j$, odvodí právě:

$$
\begin{aligned}
r_i &\geq t_1(i) \\
r_i \bmod p_g &\in \Lambda_r(i) \\
d_i &\leq t_2(i) \\
d_i \bmod p_g &\in \Lambda_r(i)
\end{aligned}
$$

A na tom je založen následující algoritmus s časovou složitostí $O(kn^2)$ (předpokládá, že úkoly jsme dopředu setřídili podle r_i):

for $i \in T$ **do begin**
  // Nastavíme množinu povolených zbytků r_i a d_i po dělení p_{f_i}:
  lr[i] := $\{0, 1, \ldots, p_{f_i} - 1\}$;
  ld[i] := $\{0, 1, \ldots, p_{f_i} - 1\}$;
end;

for $g \in F$ **do begin**
  for $j \in T$ **do begin**
    Spočítej $t_1(i)$, $t_2(i)$, $\Lambda_r(i)$ a $\Lambda_d(i)$ pro $i = 0, 1, \ldots, x$;

    k := úkol s nejmenším r_i;
    i := x;
    while ($i \geq 0$ **AND** $k \in T$) **do begin**
      if ($f_k \neq g$) **OR** ($d_k > d_j$) **then**
        k := další úkol se stejným nebo větším r_k;

```
        continue;
      end;

      // Víme k je typu g, k ∈ Ω_i.
      if r_k < r_{Ω_{i-1}} then begin
        // k ∉ Ω_{i-1}
        r_k ≥ t_1(i);
        d_k ≤ t_2(i);
        lr[k] := lr[k] ∩ Λ_r(i);
        ld[k] := ld[k] ∩ Λ_d(i);
        k := další úkol se stejným nebo větším r_k;
      end else
        // Všechny úkoly k typu g takové, že
        // k ∈ Ω_i a k ∉ Ω_{i-1} už jsme probrali.
        // Pokračujeme Ω_{i-1}:
        i := i-1;
    end;
  end;
end;

for i ∈ T do begin
  if lr[i] = ∅ OR ld[i] = ∅ then fail;
  r_i := nejmenší číslo větší nebo rovné r_i, že r_i mod p_{f_i} ∈ lr[i];
  d_i := největší číslo menší nebo rovné d_i, že d_i mod p_{f_i} ∈ ld[i];
end;
```

5.9 Výsledek

Všechny algoritmy z této kapitoly jsem sdružil do jedné podmínky *batch*, která má následující propagační algoritmus:

```
repeat
  edge_finding;
  not_before_not_after;
  not_first_not_last;
  spojeni_posloupnosti;
until nenašly se žádné další změny;
```

Tuto podmínku jsem naimplementoval v C pro SICStus prolog a vyzkoušel na řadě příkladů[1]. Měřil jsem čas a počet návratů při hledání všech řešení.

Příklady byly vygenerovány následujícím způsobem. Nejprve byl náhodně vytvořen platný rozvrh, tedy jedno řešení. Pokud úkol i začínal v tomto rozvrhu v okamžik t_i, tak hodnoty r_i a d_i pak byly voleny:

$$r_i = t_i - \text{rand}(m)$$
$$d_i = t_i + \text{rand}(m) + p_i$$

Funkce rand(m) vrací celé náhodné číslo v rozsahu $0..(m-1)$. Čím větší zvolíme m, tím je zadání více nepřesné a má tedy i více řešení. Nalezení všech řešení pak trvá dost dlouho (např. v příkladě b). Většinou jsem volil příklady, které neměly příliš hodně řešení. Výsledky[2] udává tabulka 5.1.

V další tabulce 5.2 jsou doby řešení a počty návratů, když vynecháme algoritmus not-before/not-after (sloupeček bez before/after) nebo algoritmus spojování posloupností.

Vynechání algoritmu not-before/not-after značně zpomalí řešení některých příkladů (např i a r), často ale počet návratů nevzroste a doba řešení se tedy zmenší.

Když vynecháme algoritmus spojování posloupností, zmenší se počet návratů pouze u příkladů s, w a x. Celková doba řešení ovšem je vždy kratší. Algoritmus spojování posloupností se zřejmě vyplatí jen ve výjimečných případech.

Aby se zbytečně neiterovaly algoritmy, které změny nevyvozují, upravil jsem algoritmus podmínky batch takto:

```
repeat
  repeat
    repeat
      edge_finding;
      not_first_not_last;
    until nenašly se žádné další změny;
    not_before_not_after;
  until nenašly se žádné další změny;
  spojeni_posloupnosti;
until nenašly se žádné další změny;
```

Tento algoritmus má stejný počet návratů jako původní, je však rychlejší, jak je vidět z posledního sloupečku tabulky 5.2.

[1] Zdrojový kód společně s příklady je na přiloženém CD. Je zde také program pro vygenerování dalších příkladů.

[2] Měření probíhala na počítači Pentium Celeron 333Mhz.

příklad	úkolů	typů	počet řešení	návratů	čas
a	25	5	5	17	0.78s
b	25	5	56251	5064	41m 23s
c	25	5	72	146	6.57s
d	40	6	12	33	4.9s
e	40	6	1	0	0.42s
f	50	6	6	15	2.22s
g	150	5	1	1	2.76s
h	200	5	1	2	6.8s
i	50	5	48	110	17.08s
j	50	5	10	27	4.57s
k	50	7	9	0	2.72s
l	50	5	4	4	1.44s
m	30	5	18	2	1.22s
n	30	5	39	52	4.51s
o	50	5	32	32	9.35s
p	100	5	16	23	20.67s
q	50	7	228	102	58.97s
r	50	7	324	162	1m 21s
s	100	7	50	76	1m 31s
t	200	7	8	40	1m 53s
v	100	7	240	1811	18m 19s
w	50	2	24	47	4.27s
x	50	2	1368	2175	3m 10s

Tabulka 5.1: Základní algoritmus

	Základní alg.		bez before/after		bez spojování		nový alg.
	návratů	čas	návratů	čas	návratů	čas	čas
a	17	0.78s	17	0.78s	17	0.69s	0.74s
b	5064	41m 23s	94985	59m 16s	5064	31m 41s	39m20s
c	146	6.57s	146	6.16s	146	5.2s	5.52s
d	33	4.9s	34	4.85s	33	4.15s	4.89s
e	0	0.42s	0	0.42s	0	0.41s	0.40s
f	15	2.22s	15	2.22s	15	1.80s	2.00s
g	1	2.76s	1	2.71s	1	2.20s	2.43s
h	2	6.8s	2	6.56s	2	5.57s	5.39s
i	110	17.08s	1718	2m 57s	110	13.91s	14.79s
j	27	4.57s	44	6.23s	27	3.68s	3.99s
k	0	2.72s	23	4.51s	0	1.94s	2.34s
l	4	1.44s	7	1.72s	4	1.19s	1.29s
m	2	1.22s	2	1.22s	2	1.00s	1.16s
n	52	4.51s	79	5.03s	52	3.80s	3.80s
o	32	9.35s	32	8.87s	32	7.53s	7.74s
p	23	20.67s	23	19.87s	23	16.87s	17.29s
q	102	58.97s	102	55.94s	102	46.05s	49.50s
r	162	1m 21s	978	2m 40s	162	1m 4s	1m 14s
s	76	1m 31s	76	1m 27s	83	1m 16s	1m 15s
t	40	1m 53s	40	1m 53s	40	1m 35s	1m 49s
v	1811	18m 19s	1811	17m 12s	1811	15m 1s	15m 17s
w	47	4.27s	53	4.02s	53	3.91s	4.01s
x	2175	3m 10s	2175	3m 3s	2231	2m 37s	2m 42s

Tabulka 5.2: Upravené algoritmy

Kapitola 6

Závěr

Práce je navržena tak, aby byla pochopitelná i pro čtenáře, který není s programováním s omezujícími podmínkami seznámen.

V kapitole existující globální podmínky je popsána většina nejčastěji používaných globálních podmínek, ve většině případů i je předveden a dokázán propagační algoritmus. Pokud ne, je alespoň naznačen s odkazem na literaturu.

V práci jsou dva významné nové výsledky: užší vymezení testovaných intervalů v algoritmu energetic reasoning (kapitola 3.9) a čtyři nové algoritmy pro rozvrhování dávkové výroby. Dva z těchto algoritmů jsou upravené algoritmy pro disjunktivní rozvrhování, zbylé dva jsou navrženy výhradně pro problém dávkového zpracování.

Dále by se dalo pokračovat ve zkoumání rozvrhovacích problémů, zejména problémů s více zdroji a úkoly, které nemají pevně daný zdroj, na kterém musí být zpracovány. V tomto směru zatím mnoho algoritmů navrhnuto nebylo.

Literatura

[1] Roman Barták. *Dynamic Global Constraints in Constraint Logic Programming*,
Proceedings of CP-AI-OR 2001 Workshop, pp. 39–49, Wye College, April 2001

[2] Y. Caseau, F. Laburthe. *Improved CLP Scheduling with Task Intervals*, Proc. of the 11th International Conference on Logic Programming, The MIT Press, June 1994

[3] Y. Caseau, F. Laburthe. *Cumulative Scheduling with Task Intervals*, Logic Programming: proc. of JICSLP'96, M. Maher ed., The MIT Press, 1996.

[4] Y.Caseau, F. Laburthe. *A Constraint based approach to the RCPSP*, CP97 Workshop on Industrial Constraint Directed Scheduling, 1997.

[5] Y. Caseau, F. Laburthe. *Disjunctive Scheduling with Task Intervals*, LIENS Technical Report 95-25, Laboratoire d'Informatique de I'Ecole Normale Superieure.

[6] Jean-Charles Régin. *A filtering algorithm for constraints of difference*, in Proceedings AAAI-94, pages 362–367, 1994.

[7] Jean-Charles Régin. *Generalized Arc Consistency for Global Cardinality Constraint*,
in Proceedings AAAI-96, 1996.

[8] Jean-Charles Régin. *The Symmetric Alldiff Constraint*,
in Proceedings IJCAI'99, 1999.

[9] Y. Caseau, F. Laburthe. *Solving Small TCPs with Constraints*,
to appear in Proc. of the 14th International Conference on Logic Programming, The MIT Press, 1997.

[10] Philippe Baptiste, Claude Le Pape, Wim Nuijten. *Satisfiability Tests and Time-Bound Adjustments for Cumulative Scheduling Problems*, Working Paper, Bouygues, Direction Scientifique, 1997

[11] Paul Martin, David B. Shmoys. *A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem*, Proc. 5th International Conference on Integer Programming and Combinatorial Optimization.

[12] Philippe Baptiste and Claude Le Pape. *Edge-Finding Constraint Propagation Algorithms for Disjunctive and Cumulative Scheduling.*, Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group, Liverpool, United Kingdom, 1996.

[13] Phillippe Baptiste. *A Theoretical and Experimental Study of Resource Constraint Propagation*, Thése de doctorat, Université de Technologie de Compiegne, 1998.

[14] Phillippe Baptiste. *Batching Identical Jobs*, Technical Report, University of Technology of Compigne, 1999.

[15] Bilal Ayduran. *Single Machine Group Scheduling* http://citeseer.nj.nec.com/ayduran99single.html, 1999

[16] Brucker P., Thiele O. *A branch and bound method for the general-shop problem with sequence dependent setup-times* OR Spektrum, 18:1450-161, 1996.

[17] W.J. van Hoeve. *The* alldifferent *Constraint: A Survey* Proceedings of ERCIM Workshop of Constraints, Praha, June 2001