NUOPT/SIMPLE チュートリアル

株式会社 数理システム
Phone: 03-3358-1701
Fax: 03-3358-1727
Email: nuopt-support@msi.co.jp

2006/01/16 更新

目次

1. はじめに

NUOPT は数理計画問題を解くための汎用ソルバであり，SIMPLE は数理計画問題を記述するモデリング言語です．本稿は NUOPT/SIMPLE の基本的な機能に関するチュートリアルです．本稿を一読していただければ，NUOPT/SIMPLE の基本的な利用方法がご理解いただけると思います．

NUOPT/SIMPLE を用いて数理計画問題を解くためには，まず SIMPLE にて問題を記述し，その問題に NUOPT を適用して解くことになります．本稿ではその流れに従って，まず，具体的な問題例を通じて SIMPLE の基本的な機能を解説します．その後，その問題に NUOPT を適用する方法を紹介します．

最後の章では NUOPT/SIMPLE を用いて資源制約付きスケジューリング問題を解く方法について具体的に解説しています．NUOPT には資源制約付きスケジューリング問題専用のアルゴリズムである rcpsp という方法が組み込まれており，対応する SIMPLE による問題記述には特殊なクラス（Activity,ResourceRequire,ResourceCapacity）を用います．ここでは SIMPLE による資源制約付きスケジューリング問題の記述から rcpsp の起動，rcpsp の機能の利用方法まで順を追って解説します．

2. 数理計画問題を記述する（SIMPLE チュートリアル）

2.1 目的関数・変数・制約

本稿では次のような生産計画問題を考えることにします.

2つの油田 X,Y が存在し, それぞれ一日あたり重油･ガスをそれぞれ次の量だけ生産する.

生産量/日

	重油	ガス
X	6t	4t
Y	1t	6t

また, 重油･ガスの週あたりの生産ノルマが, 次のように定められている.

	ノルマ/週
重油	12t
ガス	24t

油田 X,Y の日あたりの運転コストは, 次のとおりである.

	運転コスト/日
X	180
Y	160

それぞれの油田は最大で週 5 日まで運転可能である. ノルマを満たしながら運転コストを最小化するためには, それぞれの油田を週あたりそれぞれ何日運転すれば良いだろうか？

この問題を定式化すると，以下のようになります.

変数	x	**油田Ｘの運転日数/週**
	y	**油田Ｙの運転日数/週**
目的関数(最小化)	$180x+160y$	**運転コスト/週**
制約条件	$6x+y\geq 12$	**重油ノルマ/週**
	$4x+6y\geq 24$	**ガスノルマ/週**
	$0\leq x\leq 5$	**油田Ｘの週あたりの運転日数制約**
	$0\leq y\leq 5$	**油田Ｙの週あたりの運転日数制約**

それでは，この問題を SIMPLE で記述した例を見てみましょう．

[UNIX 版]

```
#include “simple.h”
void ufun()
{
    // 油田Ｘの運転日数/週（変数）
    Variable x(name="油田Ｘの運転日数");
    Variable y(name="油田Ｙの運転日数");

    // 運転コスト(目的関数)
    Objective cost(name="全運転コスト", type=minimize);
    cost = 180*x + 160*y;

    // 製品ノルマ
    6*x + y >= 12;     // 油田ノルマ/週
    4*x + 6*y >= 24;   // ガスノルマ/週

    // 各油田の日数制約
    0 <= x <= 5;       // 油田Ｘの週あたりの運転日数制約
    0 <= y <= 5;       // 油田Ｙの週あたりの運転日数制約

    // 求解
    solve();

    // 結果出力
    x.val.print();
    y.val.print();
    cost.val.print();
}
```

[Windows 版]

```
// 油田Ｘの運転日数/週（変数）
Variable x(name="油田Ｘの運転日数");
Variable y(name="油田Ｙの運転日数");

// 運転コスト(目的関数)
Objective cost(name="全運転コスト", type=minimize);
cost = 180*x + 160*y;

// 製品ノルマ
6*x + y >= 12;     // 油田ノルマ/週
4*x + 6*y >= 24;   // ガスノルマ/週

// 各油田の日数制約
0 <= x <= 5;       // 油田Ｘの週あたりの運転日数制約
0 <= y <= 5;       // 油田Ｙの週あたりの運転日数制約

// 求解
solve();

// 結果出力
x.val.print();
y.val.print();
cost.val.print();
```

（UNIX 版/Windows 版に関わらず）問題の定式化と SIMPLE の記述が，ほぼ１対１に対応していることがわかります．なお，UNIX 版/Windows 版の差については次ページをご覧下さい．

それでは，この SIMPLE による記述を上から順に見ていきましょう．

```
#include "simple.h"
void ufun()
{
    …
}
```

この部分の記述の有無が UNIX 版と Windows 版の差になります．UNIX 版にはこの記述があり，Windows 版にはありません．これは UNIX 版 SIMPLE で数理計画問題を記述するときの基本形式で，この通りに記述する必要があります．実際の問題定義は…の部分に記述していきます．

以下の説明では，この部分の記述は省略した形を用います．UNIX 版をお使いの方は，モデル記述に上記の記述が必要になることを忘れないで下さい．

```
// 油田 X の運転日数/週（変数）
Variable x(name="油田 X の運転日数");
Variable y(name="油田 Y の運転日数");
```

この部分は変数（油田の運転日数）の宣言です．モデル中で使用する変数は，使用する前に宣言する必要があります．name="…" の部分には変数の名前を指定します．name="…"は省略可能ですが，出力などで使用されますので，なるべく記述した方が良いでしょう．

"//" から行の終わりまではコメントです．

```
// 運転コスト(目的関数)
Objective cost(name="全運転コスト", type=minimize);
```

この部分は目的関数（運転コスト）の宣言です．目的関数の内容を定義する前に，宣言する必要があります．name="…" の部分には目的関数の名前を指定します．変数の宣言同様，name="…"は省略可能ですが，出力などに利用されますので，なるべく記述した方が良いでしょう．type=minimize で目的関数が最小化されるべきことを指示します．type=maximize とすれば，目的関数を最大化します．

```
cost = 180*x + 160*y;
```

この部分は目的関数（運転コスト）の内容定義です．= の左辺に目的関数を，右辺に目的関数の内容を記述します．* は積，+ は和を表す演算子です．SIMPLE では四則演算や初等関数(exp(),sin()…)などを式の記述に用いることができます．

```
// 製品ノルマ
    6*x + y >= 12;      // 油田ノルマ/週
    4*x + 6*y >= 24;    // ガスノルマ/週
```

この部分では制約式（生産ノルマ）を定義しています．関係演算子 >= の左辺，右辺には，任意の式を記述できます．目的関数の内容定義の際と同様に，任意の式の中に演算子や初等関数を記述できます．左辺と右辺の関係を表す関係演算子には，以下のものを指定できます．

SIMPLE の関係演算子	定式化時の記述
>=	≥
<=	≤
==	＝

```
// 各油田の日数制約
    0 <= x <= 5;        // 油田 X の週あたりの運転日数制約
    0 <= y <= 5;        // 油田 Y の週あたりの運転日数制約
```

この部分は制約式（運転日数の上下限）を定義しています．ここでは変数の上下限を指定していますが，SIMPLE では一般の制約式と変数の上下限制約を区別しませんので，x,y の部分に任意の式を書くことが可能です．

以上で，問題の定義の記述は完了です．
次に，これまでに定義した問題の最適解を求め，結果を出力する部分を記述します．

```
// 求解
    solve();
```

solve() は，定義したモデルについて最適解の計算を行う関数です．solve() は，必ずモデル記述の後に記述する必要があります．

```
// 結果出力
    x.val.print();
    y.val.print();
    cost.val.print();
```

この部分は, 最適化計算結果の出力を指定しています. 最適化計算後の値を出力するためには, 最適化計算 solve() の後に記述する必要があります.

以上でこのモデルについての SIMPLE の記述は終了です.

次にこのモデルを実行してみます（実行方法については，（3 数理計画問題を解く（NUOPT チュートリアル））を参照してください）．すると，数理計画モデルを解く経過が，以下のように出力されます．

```
SIMPLE x.x.x, Copyright (C) 1994-200x Mathematical Systems Inc.
<system code file name: sample.cc>
Expanding objective  (1/5 sample.cc:10 name="全運転コスト")
Expanding constraint (2/5 sample.cc:13)
Expanding constraint (3/5 sample.cc:14)
Expanding constraint (4/5 sample.cc:17)
Expanding constraint (5/5 sample.cc:18)
NUOPT x.x.x, Copyright (C) 1991-200x Mathematical Systems Inc.
PROBLEM_NAME                                      sample
NUMBER_OF_VARIABLES                                    2
NUMBER_OF_FUNCTIONS                                    3
PROBLEM_TYPE                                MINIMIZATION
METHOD                                     HIGHER_ORDER
<preprocess begin>.....<preprocess end>
<iteration begin>
   res=4.0e+01 .... 2.5e-06  1.2e-09
<iteration end>
STATUS                                         OPTIMAL
VALUE_OF_OBJECTIVE                                 750
ITERATION_COUNT                                      6
FUNC_EVAL_COUNT                                      9
FACTORIZATION_COUNT                                  7
RESIDUAL                                1.238460294e-09
ELAPSED_TIME(sec.)                                0.00
SOLUTION_FILE                                sample.sol
```

最後に結果出力に対応する結果が以下のように出力されます．

```
油田Xの運転日数=1.5
油田Yの運転日数=3
全運転コスト=750
```

= の左辺は指定した変数と目的関数の名前で，name="…"に記述したものが出力されます．右辺には変数と目的関数の値が出力されています．ここでは，結果の出力関数に print() を使

用しましたが，SIMPLE にはこの他にも様々な出力関数が用意されています．

他の出力関数については，**2.8 結果出力関数**で解説します．

2.2 パラメータ

現在は，モデル中に油田運転コストの値を直接記述しています．これを変更し，外部から任意の値を与えてみましょう．まず，定式化を以下のように変更します．

目的関数	$costX \cdot x + costY \cdot y$	運転コスト/週
パラメータ	costX	油田Xの運転コスト/日
	costY	油田Yの運転コスト/日

costX，costYはそれぞれ油田X,Yの運転コスト/日を表すパラメータです．SIMPLEの記述では，以上のようなパラメータを使用した記述が可能です．

ここでは，パラメータを用いて，運転コストを以下のように変更します．

```
cost = 180*x + 160*y;
```

↓

```
Parameter costX(name="油田Xの運転コスト/日");
Parameter costY(name="油田Yの運転コスト/日");
cost = costX*x + costY*y;
```

まず，Parameterで，パラメータを宣言します．モデル中で使用するパラメータは，使用する前に宣言する必要があります．パラメータの値は，モデル中で定義せず外部からデータファイルで与えます．変数，目的関数の宣言と同様に，name="…"には，パラメータ名を指定します．パラメータ名は，データファイル中のデータとの対応付けに使用されます．

次にモデルにパラメータを与えるために，以下のデータファイルを作成します．

```
"油田Xの運転コスト/日" = 180;
"油田Yの運転コスト/日" = 160;
```

=の左辺には，宣言時のname="…"で与えたパラメータ名を記述します．右辺には，パラメータ値を記述します．セミコロン；がパラメータデータの区切りになります．データファイル

中の "…" 内にないスペース，改行，タブは無視されます．

では，上記データファイルを入力として，実行してみます（実行方法については **3 数理計画問題を解く（NUOPT チュートリアル）**を参照してください）．

最適化経過の出力の後，次のような実行結果が得られます．

```
油田Ｘの運転日数=1.5
油田Ｙの運転日数=3
全運転コスト=750
```

前回と同じ結果が得られています．以上のように，Parameter を使用することで，データファイルの変更のみで違う問題を解くことができます．

では，データファイルを変更して実行してみましょう．以下のようにデータファイルを変更します．

```
"油田Ｘの運転コスト/日" = 100;
"油田Ｙの運転コスト/日" = 170;
```

実行すると，以下の結果が得られます．

```
油田Ｘの運転日数/週=5
油田Ｙの運転日数/週=0.666667
全運転コスト=613.333
```

2.3 集合・添字

実は，ここまでのモデルでは，次のように各油田について同じ日数制約を定義しているので，冗長な記述になっていると言えます．

$0 \le x \le 5$	油田Xの週あたりの運転日数制約
$0 \le y \le 5$	油田Yの週あたりの運転日数制約

そこで油田運転日数を一般的に記述することを考えてみましょう．まず油田運転日数x，yをそれぞれ x_0，x_1 と変更し，定式化を次のように変更します．

集合	$OilField = \{0,1\}$	油田集合
変数	x_i ， $i \in OilField$	油田iの運転日数/週
パラメータ	$costX$	油田0の運転コスト/日
	$costY$	油田1の運転コスト/日
目的関数(最小化)	$costX \cdot x_0 + costY \cdot x_1$	運転コスト/週
制約条件	$6x_0 + x_1 \ge 12$	重油ノルマ/週
	$4x_0 + 6x_1 \ge 24$	ガスノルマ/週
	$0 \le x_i \le 5$ $i \in OilField$	油田iの週あたりの運転日数制約

運転日数の制約を一行で書き表すことができました．

対応する SIMPLE の記述は，次のようになります．

```
// 油田集合と添字の定義
Set OilField(name="油田集合");
OilField = "0 1";
Element i(set=OilField);

// 油田 i の運転日数/週
Variable x(name="油田の運転日数/週", index=i);

// 油田運転コスト/日
Parameter costX(name="油田Xの運転コスト/日");
Parameter costY(name="油田Yの運転コスト/日");

// 運転コスト/週(目的関数)
Objective cost(name="全運転コスト", type=minimize);
cost = costX*x[0] + costY*x[1];

// 製品ノルマ
6*x[0] + x[1] >= 12;     // 重油ノルマ
4*x[0] + 6*x[1] >= 24;   // ガスノルマ

// 油田 i の週あたりの日数制約
0 <= x[i] <= 5;

// 求解
solve();

// 結果出力
x[i].val.print();
cost.val.print();
```

定式化と同様，日数制約を一行で書き表しています．

それでは，SIMPLE の記述の変更･追加点について，上から順に見ていきます．

```
Set OilField(name="油田集合");
```

ここでは集合（油田の集合）を宣言しています．SIMPLE で添字を使用する場合は，まず添字の属する集合を宣言する必要があります．変数，目的関数，パラメータと同様に，name="…" の部分には集合名を指定します．name="…"は省略可能ですが，内容を出力する際などで使用されますので，記述したほうが良いでしょう．

```
OilField = "0 1";
```

ここでは油田集合の内容を定義しています．先の定式化の添字範囲が{0, 1}なので，0, 1 を集合の要素とします．

```
Element i(set=OilField);
```

ここでは添字を宣言しています．set=… で添字が属する集合を定義します．

```
Variable x(name="油田の運転日数/週", index=i);
```

ここでは油田の運転日数を，添字付き変数として宣言しています．index=i で添字を指定します．

```
cost = costX*x[0] + costY*x[1];
```

ここでは運転コストを定義しています．添字付けは，x[添字] と記述します．

```
// 製品ノルマ
    6*x[0] + x[1] >= 12;
    4*x[0] + 6*x[1] >= 24;
```

ここではノルマを定義しています．以前に x, y と書いた変数部分を x[0],x[1] と置き換えただけです．

```
// 日数制約
    0 <= x[i] <= 5;
```

ここでは日数制約を定義します．添字に i と指定することで，全ての i ∈OilField について日数制約を，定義することができます．

```
// 結果出力
    x[i].val.print();
```

結果出力も上記日数制約と同様に，添字に i と指定することで，全ての i ∈OilField について x[i]の値が出力されます．

次に実行してみます（実行方法については **3 数理計画問題を解く（NUOPT チュートリアル）**を参照してください）．最適化経過が出力されたあと，x[i].val.print() に対応した，以下の出力が得られます．

```
油田の運転日数/週[0]=1.5
油田の運転日数/週[1]=3
```

変数名が添字つきで出力されているのが確認できます．

ここまでの記述の変更で，油田集合 OilField を導入し，各油田の運転日数を x[i] と簡略化することができました．次に，油田運転コスト costX，costY も添字 i を用いて簡略化してみます．運転コストを添字付けし，以下のように表すことにします．

パラメータ	$costX_i$，i∈OilField	油田 i の運転コスト/日

$costX_0$，$costX_1$ はそれぞれ以前の costX， costY に対応するパラメータです．SIMPLE でも同様にパラメータの添字つけを用いて，以下のように修正します．

```
Parameter costX(name="油田Ｘの運転コスト/日");
Parameter costY(name="油田Ｙの運転コスト/日");
cost = costX*x[0] + costY*x[1];
```

↓

```
Parameter costX(name="油田運転コスト/日", index=i);
cost = costX[0]*x[0] + costX[1]*x[1];
```

パラメータの添字付けは，変数添字付けと同様に index=i と指定します．上記変更に合わせて，データファイルの内容を以下のように修正します．

```
"油田運転コスト/日" = [0] 180 [1] 160;
```

添字付きのパラメータ値を指定する右辺は，

［添字］値［添字］値 …

と記述します．

では，実行してみましょう(実行方法については**３数理計画問題を解く（NUOPT チュートリアル）**を参照ください)．最適化経過が出力された後，以下のように以前と同様の結果が得られます．

```
油田の運転日数/週[0]=1.5
油田の運転日数/週[1]=3
全運転コスト=750
```

ここで，油田集合とその要素について考えます．上記のデータファイル中には，運転コストの添字として 0，1 が記述されています．そして SIMPLE の記述中で，運転コストの添字は油田集合の要素であると明示しています．SIMPLE はこのような油田集合の要素は 0，1 であると推定することができますので，実は，以下の油田集合を与える記述は不要となります．

```
OilField = "0 1";
```

この記述を削除して実行してみますと，前回と同様の結果が得られるのが確認できます．

このように SIMPLE では，添字と集合の関係から集合の内容を自動的に推定する機能があり

ます．この機能を利用すれば集合の要素を SIMPLE で陽に記述する必要がなくなり，汎用的なモデル記述が可能となります．

次に，重油とガスの生産ノルマの値を外部から与えることを考えます．定式化において製品集合を導入して製品ノルマを以下のように記述します．

集合	Product = {重油, ガス}	製品集合
パラメータ	norma_j, $j \in \mathrm{Product}$	製品 j のノルマ/週

SIMPLE の記述においても同様にパラメータの添字付けを用いて表現し，ノルマに関する制約式を以下のように変更します．

```
6*x[0] + x[1] >= 12;
4*x[0] + 6*x[1] >= 24;
```

↓

```
Set Product(name="製品集合");
Element j(set=Product);
Parameter norma(name="製品ノルマ/週", index=j);
6*x[0] + x[1] >= norma["重油"];
4*x[0] + 6*x[1] >= norma["ガス"];
```

新たに製品集合を宣言し，ノルマを製品添字付きのパラメータにします．上記のように文字列を添字に使用する場合は，文字列を "…" の中に記述する必要があります．次に，データファイルにノルマを与えるデータを追加しましょう．データファイルは以下のようになります．

```
"油田運転コスト/日" = [0] 180 [1] 160;
"製品ノルマ/週" = ["重油"] 12 ["ガス"] 24;
```

SIMPLE の記述中で，製品ノルマの添字は製品集合の要素であると明示しています．このことから，SIMPLE は製品集合の要素は "重油", "ガス" であると推定することができます．ゆえに，SIMPLE の記述中に製品集合の要素を書く必要はありません．このことは，油田集合の要素の推定と同様です．実行させると以前と同様の結果が得られます．

ここまでの変更をまとめて，集合，変数，パラメータ，条件制約，目的関数を分類し整理すると，定式化と SIMPLE の記述は次のようになります.

集合	$\mathrm{OilField} = \{0,1\}$	**油田集合**
	$\mathrm{Product} = \{\textbf{重油}, \textbf{ガス}\}$	**製品集合**
パラメータ	$\mathrm{costX}_i,\ i \in \mathrm{OilField}$	**油田 i の運転コスト/日**
	$\mathrm{norma}_j,\ j \in \mathrm{Product}$	**製品 j のノルマ/週**
変数	$x_i,\ i \in \mathrm{OilField}$	**油田 i の運転日数/週**
目的関数(最小化)	$\mathrm{costX}_0 \cdot x_0 + \mathrm{costX}_1 \cdot x_1$	**運転コスト/週**
制約条件	$6x_0 + x_1 \geq \mathrm{norma}_{\textbf{重油}}$	**重油ノルマ/週**
	$4x_0 + 6x_1 \geq \mathrm{norma}_{\textbf{ガス}}$	**ガスノルマ/週**
	$0 \leq x_i \leq 5,\ i \in \mathrm{OilField}$	**油田 i の週あたりの運転日数制約**

```
// 油田集合
Set OilField(name="油田集合");
Element i(set=OilField);

// 製品集合
Set Product(name="製品集合");
Element j(set=Product);

// 油田 i の運転コスト/日
Parameter costX(name="油田運転コスト/日", index=i);

// 製品 j のノルマ/週
Parameter norma(name="製品ノルマ/週", index=j);

// 油田 i の運転日数/週（変数）
Variable x(name="油田の運転日数/週", index=i);

// 運転コスト/週（目的関数）
Objective cost(name="全運転コスト", type=minimize);
cost = costX[0]*x[0]+costX[1]*x[1];

// 製品ノルマ
6*x[0] + x[1] >= norma["重油"];    // 重油ノルマ/週
4*x[0] + 6*x[1] >= norma["ガス"]; // ガスノルマ/週

// 油田Ｉ の週当りの運転日数制約
0 <= x[i] <= 5;

// 求解
solve();
// 結果出力
x[i].val.print();
cost.val.print();
```

2.4 集約・複数の添字

コスト定義式

```
cost = costX[0]*x[0] + costX[1]*x[1];
```

は，すべての油田について運転コストの和をとるという意味なので，これを一般的に記述すると，以下のようになります．

$$\mathrm{cost} = \sum_{i} \mathrm{costX}_i \cdot \mathrm{x}_i$$

対応する SIMPLE の記述は，以下のようになります．

```
cost = sum(costX[i]*x[i], i);
```

sum()は $\sum$ に対応する関数で，

sum(和をとる式，添字)

の書式を持ちます．

次にノルマ制約についても，sum()を適用したいと考えますが，旧記述では，

```
6*x[0] + x[1] >= norma["重油"];
4*x[0] + 6*x[1] >= norma["ガス"];
```

と各油田の生産量/日が直接数値で記述されているので，一般化できません．そこで，定式化においてパラメータ $\mathrm{prodX}_{i,j}$ を導入し，制約式を次のように記述します．

制約条件	$\sum_i \mathrm{prodX}_{i,j} \cdot x_i \geq \mathrm{norma}_j$, $i \in \mathrm{OilField}$, $j \in \mathrm{Product}$	**製品 j のノルマ制約式/週**
パラメータ	$\mathrm{prodX}_{i,j}$, $i \in \mathrm{OilField}$, $j \in \mathrm{Product}$	**油田 i の製品 j 生産量/日**
	norma_j , $j \in \mathrm{Product}$	**製品 j のノルマ/週**

対応する SIMPLE の記述は，以下のようになります.

```
Parameter prodX(name="油田の生産量/日", index=(i,j));
sum(prodX[i,j]*x[i], i) >= norma[j];
```

複数の添字に依存するパラメータを宣言する際には，index=(i,j,..) と指定します. 上記 sum() は指定した添字 i のみの和をとります. i,j について和をとる場合は，sum(任意の式,(i,j))，と記述します.

次に油田の生産量/日の値を追加した以下のデータファイルを作成します.

```
"油田運転コスト/日" = [0] 180 [1] 160;
"製品ノルマ/週" = ["重油"] 12 ["ガス"] 24;
"油田の生産量/日" =
[0, "重油"] 6  [1, "重油"] 1
[0, "ガス"] 4  [1, "ガス"] 6
;
```

データファイル中の""に囲まれていない，スペース，タブ，改行は無視されますので，上記のように上記の〝油田の生産量/日〞のように値を複数の行にわたって記述することができます. 以上で, 変更可能性のある全ての数値データをデータファイルから入力することができました. 実行結果は以前と同様になります.

ここまでの変更をまとめて，集合，変数，パラメータ，条件制約，目的関数を分類し整理すると，定式化と SIMPLE の記述は次のようになります.

集合	$\mathrm{OilField} = \{0,1\}$	**油田集合**
	$\mathrm{Product} = \{\textbf{重油}, \textbf{ガス}\}$	**製品集合**
パラメータ	costX_i ， $i \in \mathrm{OilField}$	**油田 i の運転コスト/日**
	norma_j ， $j \in \mathrm{Product}$	**製品 j のノルマ/週**
	$\mathrm{prodX}_{i,j}$ ， $i \in \mathrm{OilField}$ ， $j \in \mathrm{Product}$	**油田 i の製品 j 生産量/日**
変数	x_i ， $i \in \mathrm{OilField}$	**油田 i の運転日数/週**
目的関数(最小化)	$\mathrm{costX}_0 \cdot x_0 + \mathrm{costY}_1 \cdot x_1$	**運転コスト/週**
制約条件	$\sum_i \mathrm{prodX}_{i,j} \cdot x_i \geq \mathrm{norma}_j$ ， $i \in \mathrm{OilField}$ ， $j \in \mathrm{Product}$	**製品 j のノルマ制約式/週**
	$0 \leq x_i \leq 5$ $i \in \mathrm{OilField}$	**油田 i の週あたりの運転日数制約**

```
// 油田集合
Set OilField(name="油田集合");
Element i(set=OilField);

// 製品集合
Set Product(name="製品集合");
Element j(set=Product);

// 油田 i の運転コスト/日
Parameter costX(name="油田運転コスト/日", index=i);

// 製品 j のノルマ/週
Parameter norma(name="製品ノルマ/週", index=j);

// 油田 i の製品 j 生産量
Parameter prodX(name="油田の生産量/日", index=(i,j));

// 油田 i の運転日数/週（変数）
Variable x(name="油田の運転日数/週", index=i);

// 運転コスト/週（目的関数）
Objective cost(name="全運転コスト", type=minimize);
cost = sum(costX[i]*x[i], i);

// 製品ｊのノルマ制約式/週
sum(prodX[i,j]*x[i], i) >= norma[j];

// 油田ｉの週当りの運転日数制約
0 <= x[i] <= 5;

// 求解
solve();

// 結果出力
x[i].val.print();
cost.val.print();
```

2.5 式

ここでは，これまでの結果出力(油田運転日数/週，全運転コスト)に加えて，各製品の生産量/週も出力してみます．

生産量/週は一般的に以下のように記述できます．

$$\mathrm{prod}_j = \sum_i \mathrm{prodX}_{i,j} \cdot x_i \ , i \in \mathrm{OilField} \ , \quad j \in \mathrm{Product}$$

製品 j の生産量/週

この式に対応する SIMPLE の記述は，以下のようになります．

```
Expression prod(name="製品の生産量/週", index=j); // 式の宣言
prod[j] = sum(prodX[i,j]*x[i], i); // 式の定義
```

まず，Expression で式を宣言します．name，index の指定は，変数宣言時（Variable）と同様に，name で名前を指定し，index で添字を指定します．prod[j] = …で式の内容を定義します．Expression は，任意の変数を含む式に名前を付けるためのもので，数理計画問題の変数の数が増加することはありません．

次に生産ノルマの記述を見てみます．

```
sum(prodX[i,j]*x[i], i) >= norma[j];
```

左辺は先ほど定義した prod[j]と全く同じ内容ですので，以下のように左辺を prod[j]に置き換えることができます．

```
prod[j] >= norma[j];
```

次に結果出力部分に以下のように prod[j]を追加します．

```
prod[j].val.print();
```

これで，製品の生産量/週が出力されるようになりました．生産量の出力結果は，以下のようになります．

```
製品の生産量/週[重油]=12
製品の生産量/週[ガス]=24
```

2.6 可変パラメータ

それでは，製品の生産ノルマを 1.5 倍，2.0 倍と変化するとき，最適解がどのように変化するかを観測してみましょう．まず倍率 1.5 倍, 2.0 倍を表現するために，以下のように可変パラメータを宣言します．

```
VariableParameter normaR(name="ノルマ倍率");
```

可変パラメータは，最適化計算 solve()の後でも，随時値を変更することができます（パラメータ（Parameter）は変更できません）．

次にノルマ制約式を，以下のようにノルマの倍率を乗じた制約式に変更します．

```
prod[j] >= norma[j] ;
```

↓

```
prod[j] >= norma[j] * normaR;
```

以上で，問題の記述部分の変更は完了です．以下は，ノルマ倍率を 1.0 倍, 1.5 倍, 2.0 倍として求解，結果出力する部分の記述です．

```
    // 求解
    normaR = 1.0;
    solve();
    // 結果出力
    prod[j].val.print();
    x[i].val.print();
    cost.val.print();

    // 求解
    normaR = 1.5;
    solve();
    // 結果出力
    prod[j].val.print();
    x[i].val.print();
    cost.val.print();

    // 求解
    normaR = 2.0;
    solve();
    // 結果出力
    prod[j].val.print();
    x[i].val.print();
    cost.val.print();
```

normaR = …で倍率を指定し，solve()で求解しています．実行結果出力は次のようになります．

```
（1 回目 solve()の実行経過出力）

製品の生産量/週[重油]=12
製品の生産量/週[ガス]=24
油田の運転日数/週[0]=1.5
油田の運転日数/週[1]=3
全運転コスト=750

（2 回目 solve()の実行経過出力）

製品の生産量/週[重油]=18
製品の生産量/週[ガス]=36
油田の運転日数/週[0]=2.25
油田の運転日数/週[1]=4.5
全運転コスト=1125

（3 回目 solve()の実行経過出力）

製品の生産量/週[重油]=32
製品の生産量/週[ガス]=48
油田の運転日数/週[0]=4.5
油田の運転日数/週[1]=5
全運転コスト=1610
```

上から順にノルマ 1.0 倍，1.5 倍，2.0 倍で最適化計算しています．1.0 倍と比べて，1.5 倍の解はすべての値が 1.5 倍になっていますが，2.0 倍では，油田１の運転日数が上限の５日に達しており，運転コストが 2.0 倍以上になっていることが確認できます．

次に，以上と同じ処理を行う別の記述方法を示します．SIMPLE では，C++言語の機能がそのまま使えます．C++言語の制御文 for 文を利用して，求解部分を以下のように書き換えます．

```
for(double nr = 1.0; nr <= 2.0; nr = nr + 0.5) {
        normaR = nr;
        solve();
}
```

実行すると同様の結果が得られます．C++言語の詳細については C++言語の参考書などを参照ください．

2.7 整数変数

ここまでは，運転日数を連続変数とみなして解いてきました．しかし実際には油田は 1 日単位でしか運転できません．そこで，運転日数を一日単位の整数変数とした，整数計画問題を解くことを考えます．そのために，変数（運転日数）の宣言を以下のように変更します．

```
Variable x(name="油田の運転日数/週", index=i);
```

↓

```
IntegerVariable x(name="油田の運転日数/週", index=i);
```

IntegerVariable で整数変数を宣言します．整数変数として宣言された変数は，値として整数のみを取ります．以上で変更完了です．

実行すると，以下の結果が得られます．

```
（1回目 solve()の最適化経過出力）
製品の生産量/週[重油]=15
製品の生産量/週[ガス]=26
油田の運転日数/週[0]=2
油田の運転日数/週[1]=3
全運転コスト=840
…
```

運転日数が整数になっているのが確認できます．このように変数を IntegerVariable で宣言するだけで，整数計画問題を記述することができます．

2.8 結果出力関数

ここまでは，結果の出力には print() を使用してきましたが，SIMPLE は他にも以下のような出力機能を持っています．

演算子 << で標準出力･ファイル(C++の ostream クラスオブジェクト)へと出力する
書式指定出力関数 simple_printf() を使用する
配列へ出力する

以下，それぞれの機能を簡単に紹介します．なお，説明中に C++言語の機能にふれる記述があります．C++言語については，C++言語の参考書等を参照してください．

2.8.1 書式指定出力

simple_printf()は書式を細かく指定できる出力関数です．
結果の確認程度の用途ならば print() で十分ですが，出力書式を細かく指定したい場合には simple_printf()を使用すると便利です．ここでは，運転日数の出力部を以下のように変更してみます．

```
x[i].val.print();
```

↓

```
simple_printf("油田 %d の最適運転日数 = %d¥n", i, x[i]);
```

対応する実行結果出力は以下のようになります．

```
油田 0 の最適運転日数 = 2
油田 1 の最適運転日数 = 3
```

関数 simple_printf()の書式指定は，

simple_printf(出力書式指定，出力対象 1，出力対象 2，…)

となります．
出力対象には，変数，式，パラメータ，可変パラメータ，目的関数，添字，など集合以外の任意のものを任意の個数だけ指定できます．出力書式指定の指定方法は，C++言語の標準関数

printf()の書式指定と同様のものが指定できます.

2.8.2 ストリームへの出力

SIMPLE では，演算子 << を用いて変数，式，パラメータ，可変パラメータなど，集合以外のオブジェクトを標準出力･ファイル(C++の ostream クラスオブジェクト)へと出力することができます.

ここでは，運転日数の出力部を以下のように変更してみます.

```
x[i].val.print();
```

↓

```
cout << "油田の最適運転日数 = " << x[i] << "¥n";
```

対応する結果出力は以下のようになります.

```
油田の最適運転日数 = (2 3)
```

右辺のカッコの中に，x[0], x[1] の値が並べて出力されています.

cout の部分に，任意のファイル stream クラスオブジェクトを記述することで，任意のファイルに結果を出力させることができます.

2.8.3 配列への出力

dump() は，オブジェクトの値を配列に出力する関数です.

運転日数の出力部を以下のように変更してみます.

```
x[i].val.print();
```

↓

```
int len;
int* idx;
double* valueAry;
x[i].val.dump(len, idx, valueAry);
```

こうすることによって，x のインデクスが int 型の配列である idx に，内容が double 型の配列である valueAry に，出力されます．int, double, は，C++言語の組込み型です．この結果を以下のコードによって確認してみましょう．

```
int k;
for ( k = 0 ; k < len ; ++k ) {
    printf("idx[%d] = %d, valueAry[%d] = %5.1f¥n"
                              ,k,idx[k],k,valueAry[k]);
}

idx[0] = 0, valueAry[0] =   2.0
idx[1] = 1, valueAry[1] =   3.0
```

次のように出力され，idx,valueAry に正しく x の内容が出力されていることがわかります．
行列のように，整数の添字を二つ持つオブジェクトに関しては

```
int len;
int* idx1;
int* idx2;
double* valueAry;
Matrix[i,j].val.dump(len, idx1, idx2, valueAry);
```

のように，インデクスを格納する配列，idx1,idx2 を二つ与えます．添字が文字列を含む場合には

```
int len;
char** idx;
double* valueAry;
prodX[i,j].val.dump(len, idx, valueAry);
```

のように，文字列型のポインタ(char*)の配列を渡して，添字を文字列で受け取ります．

2.9 デバッグ出力関数

数理計画モデルが複雑になるほど，些細な記述ミスでも発見が困難になっていきます．そのようなミスを修正するための支援関数として showSystem()があります．showSystem()は，目的関数・制約式を実際のモデル内容に展開して出力します．

以下のように，showSystem()を最適化計算 solve()の直前に挿入してみます．

```
showSystem();
solve();
```

上記の位置に記述すれば，最適化計算を行うモデルの内容が出力できます．これを実行すると，showSystem()に対応した出力が以下のように得られます．

```
1-1 :  -6*油田の運転日数/週[0]-油田の運転日数/週[1]+12*ノルマ倍率 <= 0
1-2 :  -4*油田の運転日数/週[0]-6*油田の運転日数/週[1]+24*ノルマ倍率 <= 0

2-1 :  油田の運転日数/週[0]>= 0, <= 5
2-2 :  油田の運転日数/週[1]>= 0, <= 5

全運転コスト<objective>: 180*油田の運転日数/週[0]+160*油田の運転日数/週
[1] (minimize)
```

1-1，1-2 は次のノルマ制約式に対応しています．

```
prod[j] >= norma[j] * normaR;
```

2-1，2-2 は次の日数制約式に対応しています．

```
0 <= x[i] <= 5;
```

全運転コスト<objective>:は，次のコスト定義式に対応しています．

```
cost = sum(costX[i]*x[i], i);
```

このように，showSystem() を使用することによって，パラメータ値，添字等を実際の値に

置き換えた後の目的関数・制約式を確認することができます．この機能を利用すれば，意図しない記述ミスを簡単に発見することができ，効率の良いモデル記述が可能になります．

3. 数理計画問題を解く（NUOPT チュートリアル）

UNIX 版/Windows 版に関わらず，NUOPT を用いて最適化計算を行うには，次の手順が必要となります.

1. SIMPLE モデル記述ファイルを作成する
2. データファイルを作成する
3. 最適化計算を行う

以下，UNIX 版/Windows 版別に，上記の項目について，最適化計算の一連の流れを解説します．なお，詳細については「NUOPT/SIMPLE マニュアル」（UNIX 版/Windows 版），GUI 付属のヘルプファイル（Windows 版）をご覧下さい.

3.1 UNIX 版

1. SIMPLE モデル記述ファイルを作成する

適当なテキストエディタを用いて，SIMPLE でモデル記述し，拡張子が .cc となる適当なファイル名でセーブします．ここでは，以下のようなモデルを記述し，ファイル名 foo.cc にセーブします．

```
#include “simple.h”
void ufun()
{
    // 集合
    Set S, T;
    Element i(set=S), j(set=T);

    // パラメータ
    Parameter c(name="c", index=j);
    Parameter cu(name="cu", index=i);
    Parameter cl(name="cl", index=i);
    Parameter A(name="A", index=(i,j));
    Parameter bu(name="bu", index=j);
    Parameter bl(name="bl", index=j);

    // 変数
    Variable x(name="x", index=j);

    // 最小化
    Objective f(name="目的関数", type=minimize);
    f = sum(c[j] * x[j], j);

    // 条件
    cu[i] >= sum(A[i,j] * x[j], j) >= cl[i];
    bu[j] >= x[j] >= bl[j];
}
```

2.1 節でも説明しましたが，UNIX 版におけるモデル記述においては

```
#include “simple.h”
void ufun()
{
    …
}
```

という記述が必要となります．注意してください．

2. データファイルを作成する

モデルに与えるデータファイルを作成します．データファイルの拡張子は，.dat とします．ここでは，以下のデータファイル foo.dat を作成しました．

```
c = [1] -3 [2] 1;
cu = [1] 1000 [2] 1000 [3] 1000;
cl = [1] -1 [2] -2 [3] 2;
A =
[1,1] -1   [1,2] 0.1
[2,1] -0.2 [2,2] -1
[3,1] 2    [3,2] 1
;
bu = [1] 1 [2] 2;
bl = [1] 0 [2] 0;
```

3. 最適化計算を行う

それでは，準備したモデル記述ファイル foo.cc ，データファイル foo.dat を使用した場合の最適化計算を説明します．まず最適化計算実行モジュール foo を作成します．シェル上で以下のように入力します．

```
prompt% mknuopt foo.cc
```

次に最適化計算を実行します．以下のように入力します．

```
prompt% foo foo.dat
```

そうしますと，実行経過と実行結果が以下のように出力されます．

```
SIMPLE x.x.x, Copyright (C) 1994-200x Mathematical Systems Inc.
<system code file name: foo.cc>
<reading data_file: foo.dat>
Expanding  (1/3)(2/3)(3/3)ok!
NUOPT x.x.x, Copyright (C) 1991-200x Mathematical Systems Inc.
PROBLEM_NAME                                        foo
NUMBER_OF_VARIABLES                                   2
NUMBER_OF_FUNCTIONS                                   4
PROBLEM_TYPE                                 MINIMIZATION
METHOD                                      HIGHER_ORDER
<preprocess begin>.........<preprocess end>
<iteration begin>
   res=5.7e-01 .... 1.4e-04 .... 4.8e-10
<iteration end>
STATUS                                          OPTIMAL
VALUE_OF_OBJECTIVE                           -2.999998016
ITERATION_COUNT                                      10
FUNC_EVAL_COUNT                                      12
FACTORIZATION_COUNT                                  11
RESIDUAL                                4.751236142e-10
ELAPSED_TIME(sec.)                                 0.01
SOLUTION_FILE                                   foo.sol
```

3.2 Windows 版

Windows 版 NUOPT/SIMPLE を用いて最適化計算をするためには, GUI を用いる方法とコマンドプロンプトを用いる方法の二通りの方法があります. ここでは, これらについて順番に紹介します.

3.2.1 GUI を用いる方法

1. SIMPLE モデル記述ファイルを作成する

適当なテキストエディタを用いて, SIMPLE でモデル記述し, 拡張子が.smp となる適当なファイル名でセーブします. ここでは, 以下のようなモデルを記述し, ファイル名 foo.smp にセーブします.

```
// 集合
    Set S, T;
    Element i(set=S), j(set=T);

    // パラメータ
    Parameter c(name="c", index=j);
    Parameter cu(name="cu", index=i);
    Parameter cl(name="cl", index=i);
    Parameter A(name="A", index=(i,j));
    Parameter bu(name="bu", index=j);
    Parameter bl(name="bl", index=j);

    // 変数
    Variable x(name="x", index=j);

    // 最小化
    Objective f(name="目的関数", type=minimize);
    f = sum(c[j] * x[j], j);

    // 条件
    cu[i] >= sum(A[i,j] * x[j], j) >= cl[i];
    bu[j] >= x[j] >= bl[j];
```

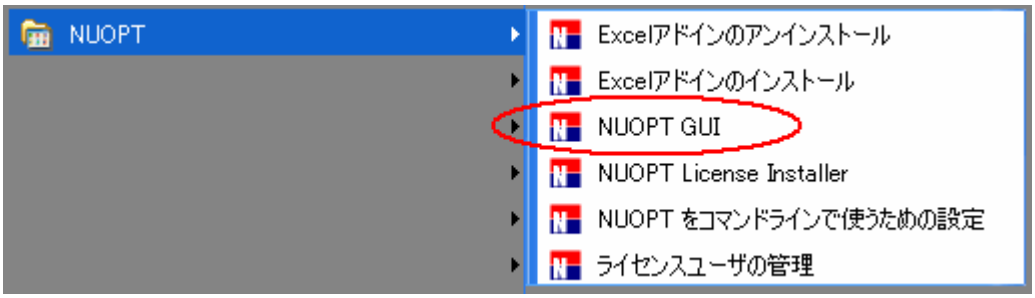
2. データファイルを作成する

モデルに与えるデータファイルを作成します．データファイルの拡張子は，.dat とします．ここでは，以下のデータファイル foo.dat を作成しました．

```
c = [1] -3 [2] 1;
cu = [1] 1000 [2] 1000 [3] 1000;
cl = [1] -1 [2] -2 [3] 2;
A =
[1,1] -1   [1,2] 0.1
[2,1] -0.2 [2,2] -1
[3,1] 2    [3,2] 1
;
bu = [1] 1 [2] 2;
bl = [1] 0 [2] 0;
```

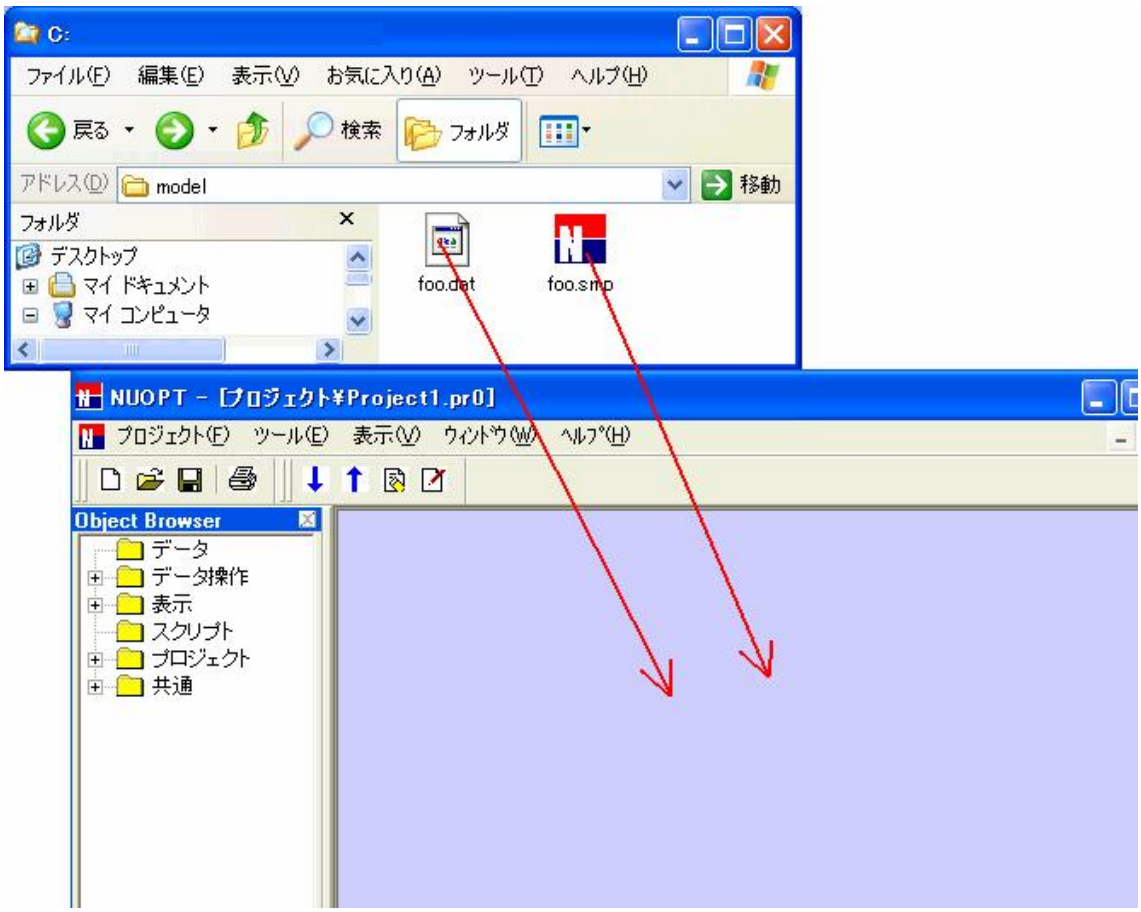
3. 最適化計算を行う

まず，NUOPT GUI を起動します．



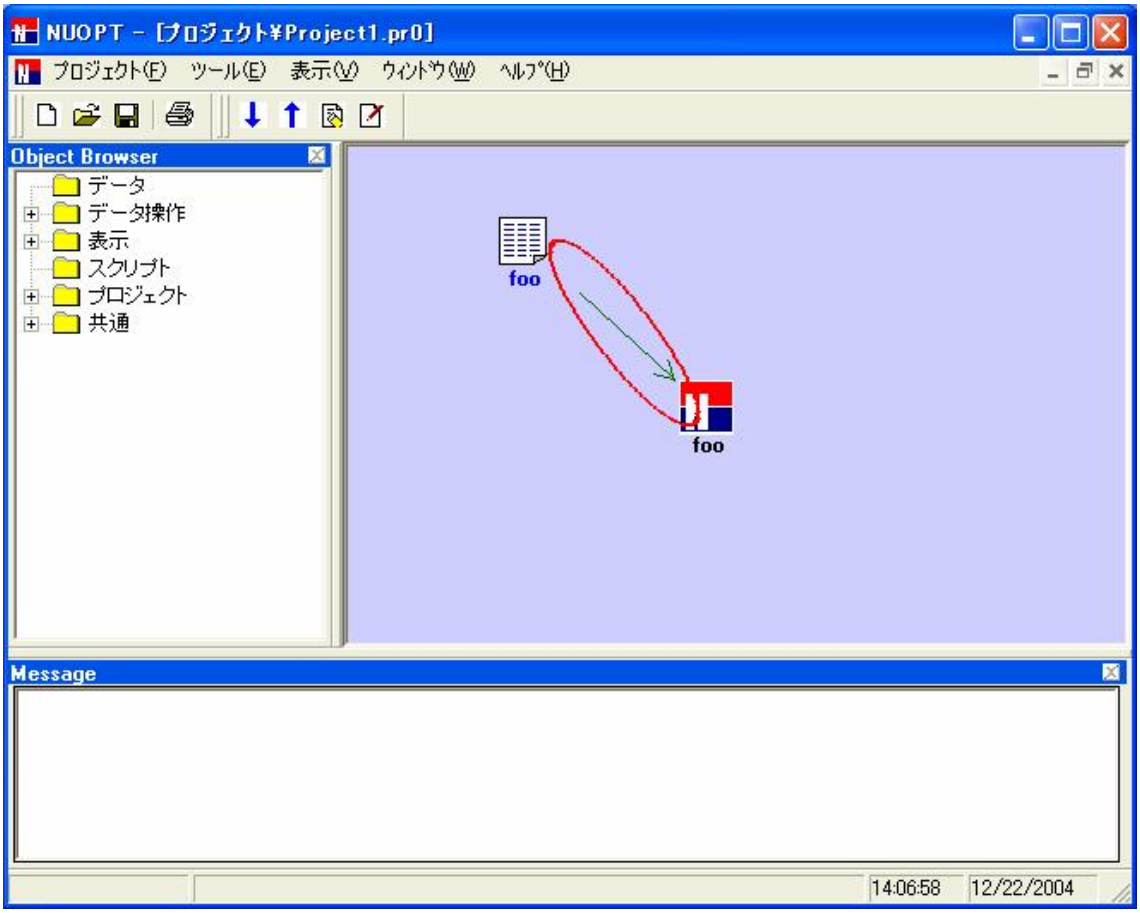
すると，次のような画面が立ち上がります．

次に，ファイルエクスプローラからプロジェクトボードに，モデルファイル `foo.smp` とデータファイル `foo.dat` をドラッグ&ドロップします.

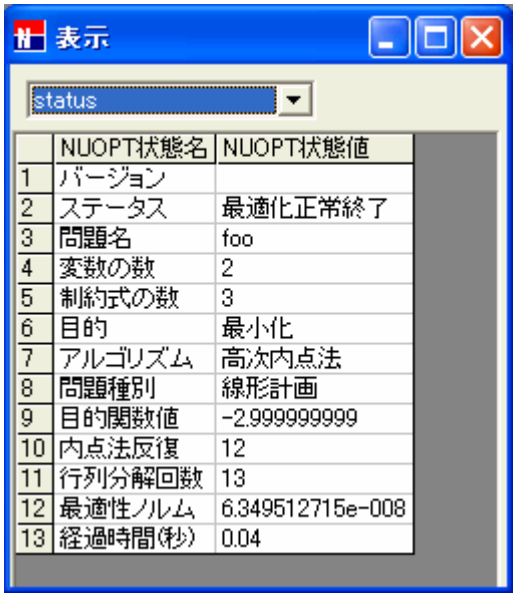


次に, プロジェクトボードに表示されたデータアイコンからモデルアイコンに矢印を繋げます. アイコンを矢印によって連結するには以下の操作を行ってください.

(1) マウスポインタを始点となるアイコン上に置きます.
(2) 右クリックをしたまま左クリックします. あるいは 3 ボタンマウスであれば真ん中のボタンをクリックします.
(3) そのままの状態で, マウスポインタを終点のアイコンの上まで移動させ, ボタンを離します.



この状態で，モデルアイコンをダブルクリックすると，最適化計算が実行され，結果ウインドウが表示されます．



	NUOPT状態名	NUOPT状態値
1	バージョン	
2	ステータス	最適化正常終了
3	問題名	foo
4	変数の数	2
5	制約式の数	3
6	目的	最小化
7	アルゴリズム	高次内点法
8	問題種別	線形計画
9	目的関数値	-2.999999999
10	内点法反復	12
11	行列分解回数	13
12	最適性ノルム	6.349512715e-008
13	経過時間(秒)	0.04

また，メッセージ表示ウインドウには，以下のような実行経過と実行結果が表示されます.

```
<reading data_file: foo.dat>
展開中 目的関数 (1/3 foo.smp:18 name="目的関数")
展開中 制約式   (2/3 foo.smp:21)
展開中 制約式   (3/3 foo.smp:22)
NUOPT x.x.x, Copyright (C) 1991-200x Mathematical Systems Inc.
PROBLEM_NAME                                        foo
NUMBER_OF_VARIABLES                                   2
NUMBER_OF_FUNCTIONS                                   4
PROBLEM_TYPE                                 MINIMIZATION
METHOD                                      HIGHER_ORDER
<preprocess begin>.........<preprocess end>
<iteration begin>
   res=1.4e+005 .... 4.5e+002 .... 5.4e-004 . 6.3e-008
<iteration end>
STATUS                                            OPTIMAL
VALUE_OF_OBJECTIVE                           -2.999999999
ITERATION_COUNT                                      12
FUNC_EVAL_COUNT                                      15
FACTORIZATION_COUNT                                   13
RESIDUAL                                 6.349512715e-008
ELAPSED_TIME(sec.)                                 0.00
SOLUTION_FILE                                    foo.sol
```

3.2.2 コマンドプロンプトを用いる方法

1. SIMPLE モデル記述ファイルを作成する
2. データファイルを作成する

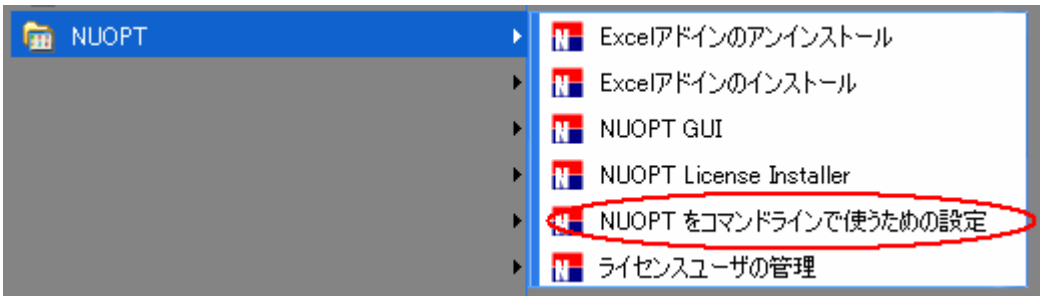
これらの操作については「**3.2.1GUI を用いる方法**」と同様の作業を行ってください.

3. 最適化計算を行う

それでは，準備したモデル記述ファイル foo.smp ，データファイル foo.dat を使用した

場合の最適化計算を説明します．まず最適化計算実行モジュール foo を作成します．

そのための準備として，「スタート」メニュー→「NUOPT」から選択できる「NUOPT をコマンドラインで使うための設定」を実行してください．



この作業により，コマンドプロンプトにて最適化計算実行モジュールを作成できるようになります．

この作業の後，コマンドプロンプトを立ち上げ，次のように入力します．

```
> mknuopt foo.cc
```

次に最適化計算を実行します．以下のように入力します．

```
> foo foo.dat
```

そうしますと，「**3.2.1GUI を用いる方法**」にてメッセージ表示ウインドウに表示されていた実行経過・実行結果と同じものがコマンドプロンプトウインドウに出力されます．

4. 資源制約付きスケジューリング問題の記述と rcpsp による求解

NUOPT には資源制約付きスケジューリング問題のための専用アルゴリズムである rcpsp が組み込まれております. 例えば, タスクスケジューリング, ジョブショップスケジューリング, 人員スケジューリングなど, 典型的なスケジューリング問題 (時間を含む意思決定) の記述と求解が可能です.

資源制約付きスケジューリング問題は NUOPT に組み込まれている混合整数計画法にて, 定式化し分枝限定法を適用する事によっても解くことができますが, 一般に非常に難しい組み合わせ問題に分類されるため特に大規模問題に対しての速度はあまり期待できません.

したがって, もし, あなたが NUOPT をある種のスケジューリング, すなわち様々な行為を行うタイミングを決定するために用いようとしているのであれば, この章をご一読いただき, 関心のある問題が rcpsp の守備範囲かどうかを確認することはおそらく非常に有益です. なぜなら, rcpsp は問題の形に特化したメタヒューリスティクス (タブーサーチ) を用いることによって, 資源制約付きスケジューリング問題に対して高速に実用上問題のない範囲の精度の解を与えることが確認されているからです. 以下では, 人員スケジューリングの例を取り, ステップを追って rcpsp の機能について具体的に説明します. ここで説明した rcpsp の機能が別種のスケジューリング問題にどのように用いられるのかも適宜説明を加えます.

4.1 問題の定式化（アクティビティ・資源・モードの定義）

まず次のような人員スケジュール問題を考えます．この問題は非常に基本的なものですが，資源制約付きスケジューリング問題の要素をすべて含み，今後の説明の核となるものです．

（例題１）
6 つの仕事（1,..,6）を A,B,C の 3 人に割り振ろうとしている．各人は同時に二つ以上の仕事はできず，A,B,C の習熟度により，各人が仕事の完成に必要な日数は異なっている．6 つの仕事それぞれは均質であるので，すべての仕事について各人の所要時間は以下となると考えてよい．

仕事１－６の所要時間

	所要時間
A	6 日
B	8 日
C	11 日

この時，すべての仕事が完成するまで最短で何日程度所要するか，その際の A,B，C への仕事の割り当てはどのようにすればよいか．

rcpsp でこの問題を定式化するには，次の手順で考えます．

1. アクティビティの定義

アクティビティとは実施のタイミングを問われている活動で，必ず行わねばならないものを言います．この問題では，「6 つの仕事」に対応します．

2. モードの定義

モードとは各アクティビティを実施するやり方（複数）で，一般にアクティビティ毎に異なりますが，この問題ではすべての仕事は｛A に任せる，B に任せる，C に任せる｝の三種類のモードで実施されると考えます．

3. 資源の定義

資源とはアクティビティを各モードで実行する際に消費する物です．各モードの各時間ステップ(この問題では日)で消費する資源の量を設定し，さらに各資源に利用可能な資源の量を与えておけば，rcpsp はいずれの時刻においても資源が利用可能な資源の量の最大を超えないようなスケジュールを決定しますので，資源の定義によってアクティビティの実行時間とモードに対する制約を表現することができます．

この問題では「各人は同時に二つ以上の仕事はできない」という制約を表現するため，各人 A,B,C が資源だとして，利用可能量をすべての時間ステップにおいて「1」と定義しておきます．そうして，各仕事のモード「A に任せる」を「資源 A を 6 日間 1 単位消費する」，「B に任せる」

を「資源Bを 8 日間 1 単位消費する」,「C に任せる」を「資源 C を 11 日間 1 単位消費する」としておきます．こうすれば，各資源，すなわち人がいかなる時間ステップにおいても 2 以上利用されない，すなわち「各人は同時に二つ以上の仕事はできない」という制約を定義したことになります．

上記を整理すると次のようになります．

（例題 1 の rcpsp による表現のための整理）

1. アクティビティ

各仕事 j(j=1..6)に対応してアクティビティが存在し，各仕事の作業モードと開始時刻，終了時刻を決定したい．

2. モード

各仕事 j には以下の 3 つのモードが対応付けられる．

仕事 1—6 に対応するモード

モード種別	所要時間	消費資源
A_does	6 日	A を各日について 1
B_does	8 日	B を各日について 1
C_does	11 日	C を各日について 1

3. 資源

各人に対応する A,B,C があり，次の量が利用可能である．

資源	利用可能量
A	全時間ステップで 1
B	全時間ステップで 1
C	全時間ステップで 1

rcpsp のための定式化とは，問題を分析して，このように互いに連関する三要素（アクティビティ，モード，資源）を設定することに他なりません．

この定式化には若干直感と反する部分があると感じられるかもしれません．アクティビティを「活動」と字義通りに捉えると仕事を行うのはあくまで「人」ですので,「各人が各仕事を行う」のをアクティビティと考えるのが直感的には妥当と言えます．

しかし，rcpsp におけるアクティビティとは

実施のタイミングを問われている活動で，必ず行わねばならないもの

とする，というところに注意ください．

実はこの問題の場合，各人が（例えば A）が仕事（例えば仕事 2）を行うかどうかはスケジュールの結果次第で変化します．仕事 2 はスケジューリングの結果として A よりも B が適当であるのならば「A が仕事 2 を行う」という活動は起りえません．一方 rcpsp では，アクティビテ

ィとして定義されたものを実施しない，という解はあり得ません．あらゆる活動にかならず開始と終了を定義することが rcpsp の目的です．すなわち，このケースにおいて，「各人が各仕事を行う」というのをアクティビティであると定義することは不適切である，ということになります．

この問題の場合，必ず行われるのは「仕事 1—6 が成される」という活動であり，仕事は必ず実施しなければならないので，アクティビティとして定義されるのが適当です．その仕事が誰によって成されるか，というのを仕事の成されるやり方であると考え，人を資源として考えるのが適当です．

ここで人を「機械」と見立てればいわゆるジョブショップスケジューリングの定式化とそのまま等価です．

4.2 rcpsp モデル記述の実際

定式化が完了したら，スケジューリング問題定義を行うためのクラス（Activity, ResourceRequire,ResourceCapacity）のオブジェクトと目的関数の種別（すべてのアクティビティの完了時間の最小化）を定義することによって具体的なスケジューリングを実施することができます．次は Windows 版[1]におけるこの問題に対応するモデル記述です．

```
//
//  例題１（人員スケジュール問題基本形）
//
Set M = "A_does B_does C_does"; // モード
Element m(set=M);
Set R = "A B C";   // 資源
Element r(set=R);
Set D = "1 .. 11"; // 各モードの作業時間の最大
Element d(set=D);
// モードと資源消費の連関
ResourceRequire  req(mode=M, resource=R, duration=D);
req["A_does,A",d] = 1, 1 <= d <= 6;
req["B_does,B",d] = 1, 1 <= d <= 8;
req["C_does,C",d] = 1, 1 <= d <= 11;
// アクティビティ
Set J = "1 .. 6";
Element j(set=J);
Activity act(name="act",index=j,mode=M); // 作業(j=1,..6)
// 利用可能な資源の定義
Set T = "0 .. 40"; // スケジューリング全体の時間(日単位)
Element t(set=T);
ResourceCapacity cap(resource=R,timeStep=T);
cap[r,t] = 1;
Objective f(type=minimize);
f = completionTime;
options.maxtim = 2;
solve();
// 解の表示
simple_printf("job=%d %s %2d %2d %2d¥n",j,act[j],act[j].startTim
 e,act[j].endTime,act[j].processTime);
```

[1] UNIX 版ではこの問題記述全体を ufun という関数の中に記述します．詳細は本マニュアルの８ページをご覧ください．以下本マニュアルでは Windows 版に従って説明します．

この問題記述は上記で完結しており，このモデルに対してデータファイルは不要です．実行してみると[2]次のような出力が得られます．

```
job=1 "A_does"  6 12  6
job=2 "B_does"  8 16  8
job=3 "A_does" 12 18  6
job=4 "C_does"  0 11 11
job=5 "A_does"  0  6  6
job=6 "B_does"  0  8  8
```

この出力は，最適化の実行（直前の solve()呼び出し）が終わった後の

```
// 解の表示
simple_printf("job=%d %s %2d %2d %2d¥n",j,act[j],act[j].startTime
,act[j].endTime,act[j].processTime[j]);
```

に対応するもので，rcpsp が求めた各仕事についてのモード，作業開始時刻，終了時刻，作業所要時間が表示されています．ここから，例えば仕事 1 は A に実施させ（A_does というモードを適用），作業開始は 6 日目，終了は 12 日目で，作業所要時間は 6 という解となっていることがわかります．細かな点ですが，仕事 1 の場合，作業開始は 6 日目のスタートで，作業終了は 12 日目が始まる直前（すなわち 11 日目一杯まで）と解釈してください．このように解釈すると，作業所要時間は作業終了時刻から作業開始時刻を引いたものになります．

次からは上記モデルの記述について詳細に見ていきます．

モデルの柱となるのは rcpsp 用に設けられている次の 3 つのクラスのオブジェクトの定義で，これらの 3 つのクラスのオブジェクトの定義が必ず必要となります．

表 1 rcpsp に用いられるクラス

クラス名	必須な添字	意味
ResourceRequire	モード，資源，時刻刻み	各モードと資源の消費
Activity	モード	アクティビティとモードの対応
ResouceCapacity	資源，時刻刻み	資源の利用可能量とスケジューリング全体時間の対応

「必須な添字」は集合（Set）もしくは要素（Element）で，これらのクラスのオブジェク

[2] このモデルファイル(assign1.smp)を含むプロジェクトは Windows 版では NUOPT¥samples¥rcpspTutorial.prj，UNIX 版では nuopt/examples にあります．

トが意味上必ず持たなければならない添え字です．ただし，これらの引数に与える集合は１次元である必要があります．このほかに各クラスは name（名前，データファイルとの対応付けを示す[省略可]）や Activity については一般の添え字（同種のものが複数ある場合）を定義することができます．次から各クラスについて順に見てみます．

4.2.1 クラス ResourceRequire

各作業モードが，その作業中の各時点において消費するリソースの量を示します．前項の定式化において

表 2 消費リソース量とモードの関係

モード種別	所要時間	消費資源
A_does	6 日	A を各日について１
B_does	8 日	B を各日について１
C_does	11 日	C を各日について１

という表の情報を与えるためのものです．

表 3 ResourceRequire に必須な添え字

書式	意味
mode=集合または要素	作業モード
resource=集合または要素	所要資源
duration=集合または要素	経過時間

必須な添え字は上記の通りで（mode,resource,duration は予約語）です．SIMPLE の他のオブジェクトの添え字同様に，値としては Set（集合）あるいは Element（集合の要素）を用います．
次にこのモデルにおける宣言部分を抜き出しますが，ResourceRequire の宣言にはモード，資源

```
Set M = "A_does B_does C_does"; // モード
Element m(set=M);
Set R = "A B C";   // 資源
Element r(set=R);
Set D = "1 .. 11"; // 各モードの作業時間の最大
Element d(set=D);
// モードと資源消費の連関
ResourceRequire  req(mode=M, resource=R, duration=D);
```

という二つの重要な概念が両方ともに現れていることがわかります．集合 D は各モードで作業を行った際の所要時間の添え字が入る便宜的なもので，ここではもっとも時間のかかるモード（C_does）の所要時間となっています．

宣言に引き続く

```
req["A_does,A",d] = 1, 1 <= d <= 6;
req["B_does,B",d] = 1, 1 <= d <= 8;
req["C_does,C",d] = 1, 1 <= d <= 11;
```

という部分が資源消費量を具体的に定義しています．モード，資源，時間の添え字の並び順は宣言の並び順に対応しています（必要があって添え字の順番を変更したい場合には宣言における mode,resource,duration の並びを変更してください）．

ResourceRequire の値の設定は，SIMPLE の Parameter クラスに対するやり方と全く同一[3]で，集合の要素（ここでは，集合 D の要素 d）を用いて，条件式を用いた代入を行って設定を簡便にしています．本来ここは，

```
req["A_does","A",d] = 1, 1 <= d <= 6;
```

のように書きたいところですが，こうすると，モデルの解釈上の問題（文字列が"，"で連結されたものは最後の部分のみしか解釈されない）[4]でエラーが起きるので

```
req["A_does,A",d] = 1, 1 <= d <= 6;
```

のように"A_does,A"をまとめて""で括っています．この文は

```
req["A_does,A,1"] = 1;
req["A_does,A,2"] = 1;
req["A_does,A,3"] = 1;
req["A_does,A,4"] = 1;
req["A_does,A,5"] = 1;
req["A_does,A,6"] = 1;
```

という意味で，A_does というモードで実施されるとモードの開始時点から 6 日目まで，A という資源（実は人）を 1 だけ消費することを述べています．資源所要量は経過時間に依存して与えることができます．ここでは，経過時間に依らずに消費資源は 1 ですが，

```
req["A_does,A",d] = 1, d == 1 || d == 6;
```

のように定義して，作業開始と終わりのみに資源を所要する，といった設定も可能です（この場合，A は，割り当てられたこの作業の進行中にも他の仕事も並行して行うことができます）．ここを様々に設定することによって，最初だけ忙しくだんだん負荷が減る，などの作業の波を定

[3] C++での扱いは Parameter を継承して定義されておりますので，Parameter に対する操作（代入，表示など）をすべて受け付けることができます．

[4] SIMPLE の解釈には C++の処理系を用いていることから来る制約です．

義することができます.

作業に所要する時間は消費資源が与えられた最大経過時刻であるものとして定義されます.

```
req["A_does,A,3"] = 1;
req["A_does,A,6"] = 1;
```

たとえば上記のように定義されたら，作業所要時間は 6 で，経過時間 3,6 でのみ資源を 1 所要するものと解釈されます.

また，作業時間は 10 かかるものの，経過時間 7 以降は資源を消費しない，という作業モードを定義するには,

```
req["A_does,A,10"] = 0;
```

のようにして，明示的に最大経過時刻 10 を定義するのみで可能です．この時，時刻 7－9 の定義されていない値は通常の Paramter と同様 0 で初期化されますが,

```
ResourceRequire req(mode=M, resource=R, duration=D, defaultval = 1;
```

のように引数 defaultval に値を設定する事により, 初期化される値を変更する事が出来ます.

表 4 ResourceRequire 特有の引数

書式	意味
defalutval=パラメータ	初期化されるデフォルト値

ただし，defaultval は添え字を用いる事が出来ませんのでご注意下さい.

ResourceRequire に対して同一の設定を行うのにデータファイルを用いることもできます. ResourceRequire のオブジェクト req に対して，通常の SIMPLE の Parameter と全く同様に行います．ただ，データファイルには一般の添え字が使えませんので，記述は次のように冗長になります．データファイルから与える場合，二つ以上の添え字を""で括ると意味が変わりますので，""は取ってください.

```
req=
[ A_does, A, 1] 1 [ B_does, B, 1] 1 [ C_does, C, 1] 1
[ A_does, A, 2] 1 [ B_does, B, 2] 1 [ C_does, C, 2] 1
[ A_does, A, 3] 1 [ B_does, B, 3] 1 [ C_does, C, 3] 1
[ A_does, A, 4] 1 [ B_does, B, 4] 1 [ C_does, C, 4] 1
[ A_does, A, 5] 1 [ B_does, B, 5] 1 [ C_does, C, 5] 1
[ A_does, A, 6] 1 [ B_does, B, 6] 1 [ C_does, C, 6] 1
                  [ B_does, B, 7] 1 [ C_does, C, 7] 1
                  [ B_does, B, 8] 1 [ C_does, C, 8] 1
                                    [ C_does, C, 9] 1
                                    [ C_does, C, 10] 1
                                    [ C_does, C, 11] 1
;
```

データファイルから ResourceRequire の内容を与えるメリットとしては，SIMPLE の自動代入機能により，モードや資源集合の値を陽に与えることを省略できる，ということです．このようにデータファイルから与える場合，ResourceRequire の添え字に現れている集合の内容の設定

```
Set M = "A_does B_does C_does"; // モード
Set R = "A B C";   // 資源
Set D = "1 .. 11"; // 各モードの作業時間の最大
```

は不要で，

```
Set M; // モード
Set R;   // 資源
Set D; // 各モードの作業時間の最大
```

としておけば，ResourceRequire のデータに含まれる添え字から，自動的に設定されます．

モデルにおいて ResourceRequire の宣言が最初に書かれているのは，この自動代入の結果による設定内容を使って効率よくモデル定義を行うことができるためです．

自動代入は特に大規模問題の場合，定義するデータ量を削減できるのでかなり便利ですが，データファイルの記述ミスから思わぬ誤動作を招くことがあります．

用心する場合には集合の lock 機能を用いて，自動代入される添え字が妥当なものかどうかを自動的にチェックさせる方法が有効です．

具体的には，モード集合などの値を与えて，集合に対して，自動代入による追加を禁止する lock()という手続きを呼びます．

```
Set M = "A_does B_does C_does"; // モード
M.lock();
Set R = "A B C";   // 資源
R.lock();
Set D = "1 .. 11"; // 各モードの作業時間の最大
D.lock();
```

こうすると，req の添え字がここで与えている集合の範囲外となったときに，（自動代入が禁じられているので）次のようなエラー（次は資源の添え字を P とミスタイプした場合の表示）となります．

```
<<SIMPLE 207>> データ "req" の設定の際に、
        自動代入で lock されている集合 "R" に要素が追加されようとしました.
        新たに加えようとした要素: {P}
```

最後に，rcpsp において扱えるのは**すべて整数値**ということにご注意ください．rcpsp において，時間ステップや資源所要量はすべて整数として扱われます．

4.2.2 クラス Activity

次にこの問題の「変数」に相当する Activity の定義です．

```
Set J = "1 .. 6";
Element j(set=J);
Activity act(index=j,mode=M); // 作業(j=1,..6)
```

Activity に必須な引数は次の mode のみです．

表 5 Activity に必須な引数

書式	意味
mode=集合または要素	作業に適用可能な作業モード

作業が 6 つあるので，ここでは j という添え字が付加しています．
Activity の宣言の「mode=..」という部分で，その Activity が実行される作業モードの集合を与えます．この場合，6 つの作業は，等質で誰にでも割り振ることができるので，すべての作業についてモード集合は同一，すなわちモード集合 M を与えています．しかし，一般には各作業を担当できる人員や所要時間は作業毎に異なることが十分考えられます．そのような状況に対応することができるよう mode の右辺の集合は index の右辺の添え字と同一の添え字を与えることができます（このようなケースは以降の例題 3 で現れます）．

各作業に納期が定義されており，納期からの遅れを最小化するには Activity に納期を示す duedate を与えます. 各作業に別々の納期を設定するために, duedate には Activity と同一の index を与えることができます（具体例は以降の例題 2 で示します）.

表 6 Activity 特有の引数

書式	意味
duedate=パラメータ	納期

Activity そのものは変数（Variable）の一般化[5]ですので，表示などの操作は通常の Variable と同じように可能です．Activity は宣言時にモードとして与えられた集合の要素のいずれかを取り，その作業はどの作業モードで処理されたかを表します．例えば，このケースで各作業が，解においてどのモードで実施されるのか（誰に割り振られたのか）は，例えば

```
act.val.print(); // 標準出力する．すべてを表示する場合，添え字 j は省略可能
act.val.dump();  // Excel 連携するのならば
```

として表示することができます．さらに Activity オブジェクトは

表 7 Activity のメンバ

メンバ名	意味
startTime	作業の開始時刻
endTime	作業の終了時刻
processTime	作業の所要時間

という Variable をメンバーとして持っています．作業モードの所要時間は決定していますので，上記メンバーの一つのみを決定すれば後は自動的に決定しますが，便宜上 3 つが定義されています.

このサンプルでは,

```
// 解の表示
simple_printf("job=%d %s %2d %2d %2d¥n",j,act[j],act[j].startTime
,act[j].endTime,act[j].processTime[j]);
```

のように，simple_printf を用いて仕事 j のアクティビティすべてについて作業モードと開始時刻，終了時刻，所要時間を表示しています.

また，アクティビティを定義すると，全アクティビティに先行する sourceActivity，全アクティビティに後続する sinkActivity が自動的に定義されます．sourceActivity，

[5] クラス Variable の継承クラスとして定義されています.

sinkActivity は，モデル中に定義されている全ての資源を消費する処理時間 0 のモード DummyMode を唯一持つアクティビティです．

その為，モデル中でこれらのアクティビティ，モードを用いる事も出来ますが，**sourceActivity，sinkActivity，DummyMode は予約語になります．**

4.2.3 クラス ResourceCapacity

最後は各モードが所要する資源の利用可能な消費量を定義するクラス ResourceCapacity です．

```
Set T = "0 .. 40"; // スケジューリング全体の時間
Element t(set=T);
ResourceCapacity cap(resource=R,timeStep=T);
```

クラス ResourceCapacity に必須な添え字は以下の通りです．

表 8 ResourceCapacity に必須な添え字

書式	意味
resource=集合または要素	利用可能量を定義する資源
timeStep=集合または要素	全体時刻の定義

アクティビティの各モードが所要している各資源が，各時間ステップにおいてどの程度の量利用可能かを示します．副次的な効果として重要なのはスケジューリングを考慮する全体時間の定義はクラス ResourceCapacity の定義を通じて行われるということです．

ここでは，T をモデル中で与えて，

```
// 利用可能な資源の定義
cap[r,t] = 1;
```

と与えることにより，全時刻において各人に対応する資源の供給量が１である（同一の人は二つ以上の仕事を同時に行うことはできない）ことを示します．この定義に続けて

```
cap[“A,3”] = 0; // A は３日目に休暇
```

とすれば，一部の人がある時間において部分的に作業ができないことを示すことができます．

また，通常，この ResourceCapacity で与えられた資源の利用可能な消費量を必ず超えない(ハード制約)ようにスケジュールは構成されるのですが，これらの利用可能量に対して重みを与える事により，なるべく超えないという(考慮制約)スケジュールにする事も可能です．この重みは，ResourceCapacity の宣言時に引数 weight に Parameter 値を与える事で実現

できます.

```
Set T = "0 .. 40"; // スケジューリング全体の時間
Element t(set=T);
ResourceCapacity cap(resource=R,timeStep=T,weight=100);
```

上記の場合, 資源 R の全時刻において利用可能量に対する重み 100 が与えられます. また, weight に与える Parameter は資源，時刻で添え字付ける事も可能です.

```
Set T = "0 .. 40"; // スケジューリング全体の時間
Element r(set=R);
Element t(set=T);
Parameter w(index=(r,t));
ResourceCapacity cap(resource=R,timeStep=T,weight=w[r,t]);
```

```
Set T = "0 .. 40"; // スケジューリング全体の時間
Element r(set=R);
Element t(set=T);
Parameter w(index=r);
ResourceCapacity cap(resource=R,timeStep=T,weight=w[r]); // w[t] も同様
```

表 9 ResourceCapacity 特有の引数

書式	意味
weight=パラメータ	利用可能量に対する重み

weight に与えられる値は，正の整数値である事に注意して下さい（ただし，明示的にハード制約にしたい場合には-1 を与えます）また，納期遅れ最小化時には重みを設定する事はできません.

4.2.4 目的関数の設定

rcpsp で扱うことのできる目的関数は「最後の作業の完了時刻，納期遅れ」のみ[6]で，このモデル記述において行っているようにモデル記述において次のように指定します.

[6] ただし，以降に述べる一般の考慮制約を用いればその他の評価項目を設定する事は可能です.

```
// 最後の作業の完了時刻の最小化
Objective f(type=minimize);
f = completionTime;
```

一般の考慮制約（4.6 で具体的に説明します）が存在するとき，目的関数と考慮制約はそれぞれ重み付けされ，その考慮制約の違反量と目的関数値の和が最小化されますが，その場合の目的関数の重みを定義するのがパラメータ：

```
options.defaultObjectiveWeight
```

です．defaultObjectiveWeight には整数値を与えます．

納期遅れの最小化を行う場合には，Actvity に duedate を設定した後，

```
// 納期遅れ最小化
Objective f(type=minimize);
f = tardiness;
```

とします．具体的な例は以降の例題 2 で説明します．ただし，納期遅れ最小化の場合には重みを与える事が出来ません．

4.3 ガントチャートの出力

スケジューリングの結果を直観的に見るにはガントチャートがもっとも適しています. NUOPT には簡便な汎用ガントチャート表示ツールが付属していますので，その利用方法を紹介します.

まず，モデル中で次のようにしてガントチャート用のデータの生成を行います．例題 1 のモデルの結果については solve()の後に以下の文を追加します.

```
Gantt g;  // ガントチャート用のデータ(宣言)
g.add(act[j], j); // アクティビティ j についてガントチャートの行に加える
g.dump();  // 出力（カレントディレクトリに出力される）
```

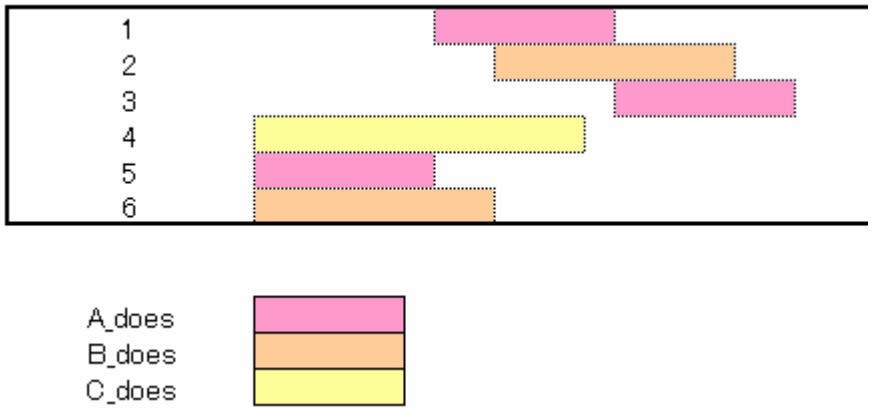
また，アクティビティ 1 .. 3 までをガントチャートに加えたい場合には,

```
g.add(act[j], (j, 1 <=j<=3)); //アクティビティ 1..3 をガントチャートに加え
る
```

のようにして添え字を条件付ける事も可能です.

実行方法は通常の Excel 連携と同じで，Excel のメニューバーの NUOPT メニューから実行をクリックします．そうすると，Excel 上のシート(NGanttChartSheet[7])にガントチャートが出力されます．Excel 連携の実行方法の詳細は,「Excel 連係マニュアル，Excel 連係チュートリアル」をご参照下さい.

図 1 例題１の解の出力例



各行が仕事に対応しており，帯によって仕事の開始と終了が，帯の色によってモードが示されています．A が仕事 5，1，3 をこの順に担当し，B は 6，2 を，C は 4 のみを担当していること，

[7] NGanttChartSheet, NGanttChartColorIndexSheet は予約語で，シート名に用いる事は出来ません.

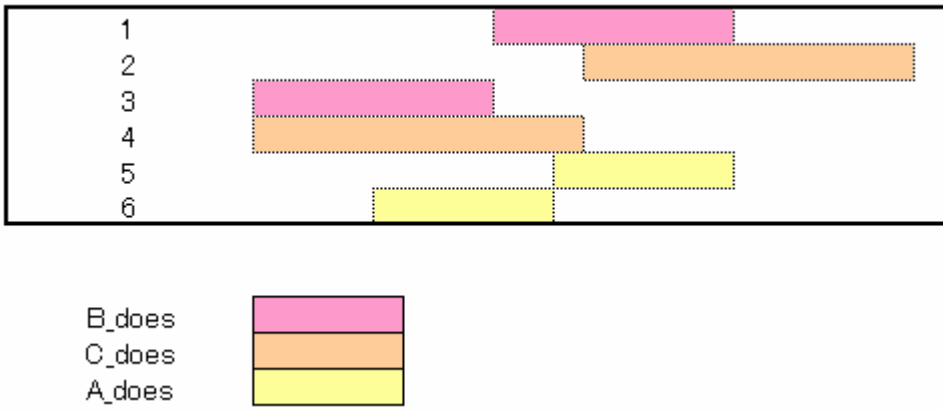
最後の作業の完了時刻の最小化という観点で，A，B，C の順に担当する仕事が多いことより直観的に妥当な結果であることが判断できます．

では ResourceCapacity の定義に以下の一行を加えて実行してみましょう．

```
cap[r,t] = 1;
cap[“A,3”] = 0; // A は３日目に休暇
```

得られた結果は次にようになります．A が 3 日目に休暇を取ることにより，A が 1 日目から稼動することはできないので，最も遅い C も含めてそれぞれ仕事を 2 つずつ担当することになり，全体の完了時刻は遅れています．

図 2 例題１で A が３日目に休む場合のスケジュール



以降，例題 3 で説明するように，いくつかの Activity がまとまって一つながりの作業の各工程を示すようにするモデル化を行った場合には，それらをまとめて一つの帯として見せた方が好ましいケースがあります．

その場合には，まとめたい添え字と条件式をさらにカンマで結合し記述します．

```
Gantt g;
g.add(act[j,s],j,s); // 仕事 j の各ステップ s を一本の帯にまとめる
g.dump();
```

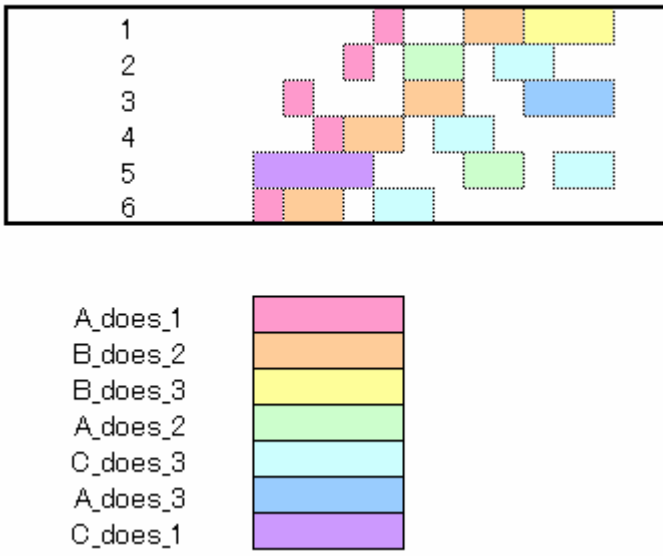
上記のように記述すると，添え字 s で１つの帯にまとまります．また，s についても条件付ける事も出来ます．

```
g.add(act[j,s],j,(s,1<=s<=3));
```

次はこのようにして出力されたガントチャートの例です．これは，例題 3 のモデルの結果の

出力です.

図 3　帯をまとめた場合の出力（例題 3 の結果）



4.4 納期遅れ最小化

rcpsp においては最後の作業の完了時刻の最小化のみを意図していましたが，実際のプロジェクトスケジューリングにおいては，以下のように各仕事に納期が設定されており，納期遅れを最小化したいケースも多く存在します．

（例題２　納期遅れ最小化）
例題１の状況において，各仕事について次のような納期が設定されているとき，納期遅れを最小化するようなスケジュールを出力せよ．

仕事	納期
1	10 日
2	10 日
3	10 日
4	17 日
5	17 日
6	6 日

これは Activity の定義に納期情報を設定し，目的関数に納期遅れ最小化を定義することによって可能です．

```
Set J = "1 .. 6";
Element j(set=J);
Parameter due(index=j);
Activity act(index=j,mode=M,duedate=due[j]);
```

下線部が納期情報に対応する引数です，アクティビティと同一の添え字を持つパラメータを与えます．納期を示すパラメータ due はデータファイルから次のように与えます．

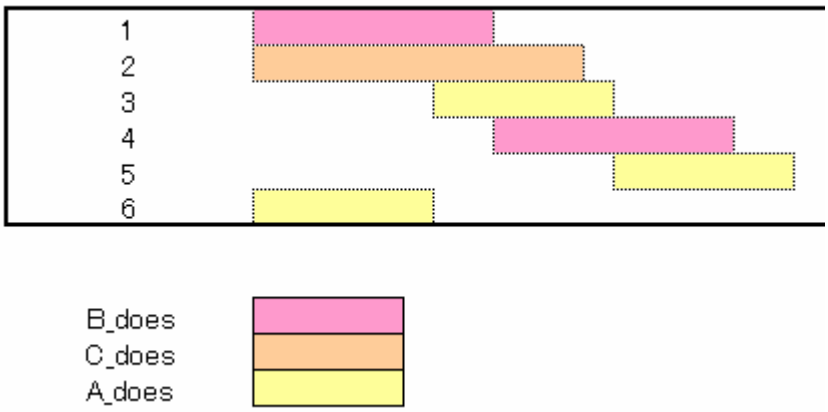
```
due = [1] 10 [2] 10 [3] 10
      [4] 17 [5] 17 [6] 6;
```

目的関数には次のようにして納期遅れを設定します．

```
// 納期遅れ最小化
Objective f(type=minimize);
f = tardiness;
```

このようにすると，納期遅れを最小化する解を求めます．この結果のガントチャート出力は以下のようになります．

図 4　例題２（納期遅れ最小化）の例



４.５作業の工程への分割

例題１で，Ａが３日目に休みを取って作業ができないという設定にすると，全体納期はかなりの遅れとなりました．以下，並べて表示します．

図 ５　元のスケジュール（最終完了時刻 18）

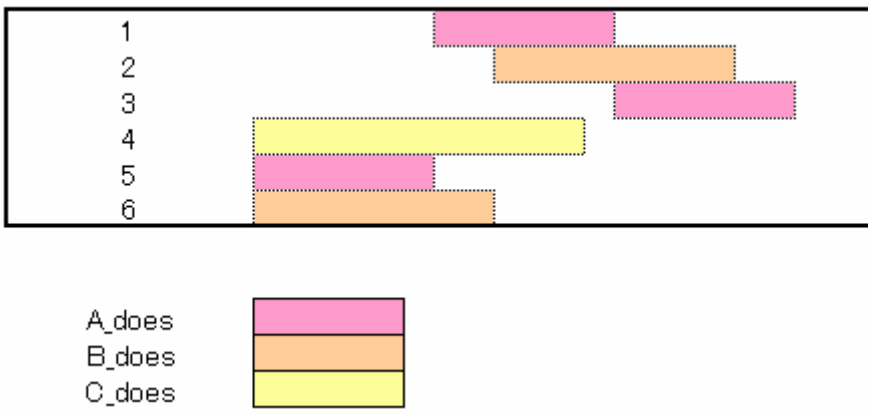
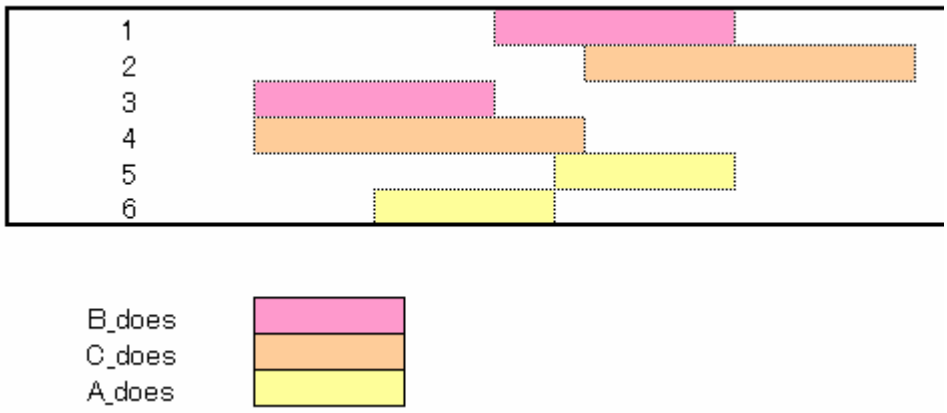


図 ６　Ａが時刻３で休む場合のスケジュール（最終完了時刻 22）



これは各作業の途中中断や担当替えができないためであると考えることができます．しかし，実際には厳密な意味でこのようなケースはむしろ少なく，仕事はいくつかの工程に分けられると考えるのが自然な状況です．次の例題３のような問題設定を考えましょう．

(例題３)

各仕事は３工程に分割できる．A,B,C の作業員の各工程に対する所要時間を調べたところ，以下のようなばらつきがあり，各人によって各工程の得手不得手が存在することが明らかになった．

担当員	工程１所要	工程２所要	工程３所要	合計
A_does	１日	２日	３日	６日
B_does	３日	２日	３日	８日
C_does	４日	５日	２日	11 日

作業工程別の担当替えや作業の工程別の中断が可能なものとして，完了時間の最小化を行え．

問題のそのほかの設定は例題１と同一とする．

これまでは，各仕事が一つのアクティビティでしたが，この問題設定では，各仕事の各工程が別の人員によって（異なるモードで）担当されることから，アクティビティを，各作業を分割した「工程」であるものとして定義する必要があります．

作業モードについても細分化が必要です．各作業の工程１はもし A が担当するなら１日，しかし工程 2，３は同じ A が担当してもそれぞれ２日，３日かかりますから，「だれが担当するか」のみでは作業モードは確定できません．作業モードは「どの工程をだれが担当するか」という組み合わせ分（9 通り）だけ存在します．整理しましょう．

（例題３の rcpsp による表現のための整理）

1. アクティビティ

各仕事 j(j=1..6)の各工程（s=1,2,3）に対応して 18 個のアクティビティ act[j,s] が存在する．各アクティビティの作業モードと開始時刻，終了時刻を決定したい．

2. モード

各アクティビティ j，s （j=1..6，s=1,2,3）には以下の３つのモードが対応付けられる．

各モードと対応するアクティビティ

モード種別	所要時間	消費資源	対応するアクティビティ
A_does_1	１日	A を各日について１	act[j,1]
A_does_2	２日	A を各日について１	act[j,2]
A_does_3	３日	A を各日について１	act[j,3]
B_does_1	３日	B を各日について１	act[j,1]
B_does_2	２日	B を各日について１	act[j,2]
B_does_3	３日	B を各日について１	act[j,3]
C_does_1	４日	C を各日について１	act[j,1]
C_does_2	５日	C を各日について１	act[j,2]
C_does_3	２日	C を各日について１	act[j,3]

3. 資源

各人に対応する A,B,C があり，次の量が利用可能である．

資源	利用可能量
A	全期間について１
B	全期間について１
C	全期間について１

次が工程分割を含むスケジューリング問題記述の全体です．

```
//
//  例題３（人員スケジュール問題，工程分割）
//
Set M;
Element m(set=M); // モード
Set R;
Element r(set=R); // 資源
Set D;
Element d(set=D); // 各モードの作業時間の最大
// モードと資源消費の連関
ResourceRequire  req(mode=M, resource=R, duration=D);
Set T = "0 .. 40"; // スケジューリング全体期間
Element t(set=T);
ResourceCapacity cap(name="cap",resource=R,timeStep=T);
cap[r,t] = 1;

Set J = "1 .. 6"; // 仕事
Set S = "1 2 3";  // 工程
Element j(set=J);
Element s(set=S);
Set stepToMode(index=s);
Activity act(index=(j,s),mode= stepToMode[s]);
// 先行制約
act[j,s-1] < act[j,s], s > 1 ;

// 最後の作業の完了時刻の最小化
Objective f(type=minimize);
f = completionTime;
options.maxtim = 10;
solve();
// ガントチャート出力
Gantt g;
g.add(act[j,s],j,s);
g.dump();
```

例題１のモデルとのモデル記述の違いは下線部です．主に Activity の定義に関するが異なっていることがわかります．上で見るようにモードも異なるのですが，この記述では，モードの

内容をデータファイルから入力するようにしていますので，ResourceRequire の定義に関するモデルの変更はありません．このモデルに対応するデータファイルは以下の通りです．

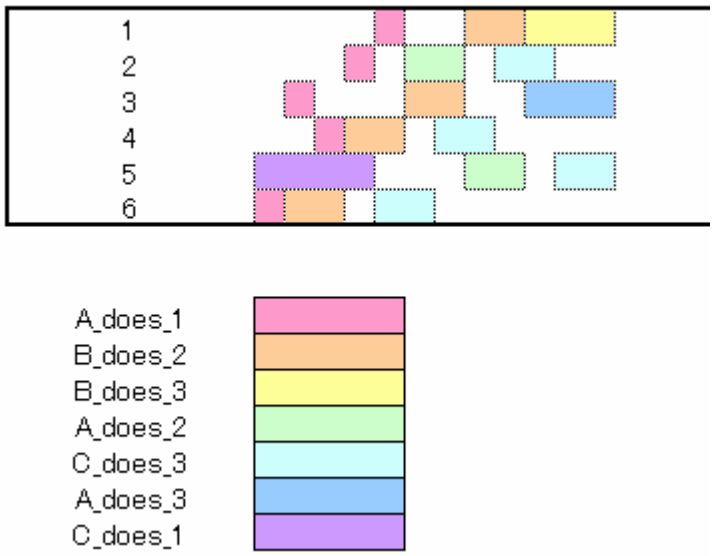
```
req=
[ A_does_1, A, 1] 1  [ B_does_1, B, 1] 1  [ C_does_1, C, 1] 1
                     [ B_does_1, B, 2] 1  [ C_does_1, C, 2] 1
[ A_does_2, A, 1] 1  [ B_does_1, B, 3] 1  [ C_does_1, C, 3] 1
[ A_does_2, A, 2] 1                       [ C_does_1, C, 4] 1
                    [ B_does_2, B, 1] 1
[ A_does_3, A, 1] 1  [ B_does_2, B, 2] 1  [ C_does_2, C, 1] 1
[ A_does_3, A, 2] 1                       [ C_does_2, C, 2] 1
[ A_does_3, A, 3] 1  [ B_does_3, B, 1] 1  [ C_does_2, C, 3] 1
                     [ B_does_3, B, 2] 1  [ C_does_2, C, 4] 1
                     [ B_does_3, B, 3] 1  [ C_does_2, C, 5] 1

                                          [ C_does_3, C, 1] 1
                                          [ C_does_3, C, 2] 1
;
stepToMode =
[1] A_does_1 B_does_1 C_does_1
[2] A_does_2 B_does_2 C_does_2
[3] A_does_3 B_does_3 C_does_3
;
```

モードが細分化されていることがわかります．stepToMode は各工程に関するモードと工程に関するアクティビティを結合するために定義しておきます．

次がこの問題に関する rcpsp の結果です．

図 7 例題3の解



全体の終了時刻は 12（例題１では 18）とかなり改善しています．一見して気づくのが，各人の得意（A は１の工程，B は２の工程，C は３の工程）が生かされたスケジュールとなっていることです．作業５について C が，苦手な（5 日もかかる）１の工程を担当していますが，これは A が他の仕事の１の工程を行うのに忙しく, B も２の工程を行うのに忙しいのでやむをえない結果と言えるでしょう．次から実際のモデルの記述について順に見てゆきます．

4.5.1 クラス Activity と Activity 毎のモードの定義

このモデル記述では Activity は仕事と工程で添え字付けられており，6（仕事）×3（工程）の 18 個のアクティビティを定義しています．各アクティビティには，その工程に対応した作業モードが対応付けられます．

```
Element j(set=J);
Element s(set=S);
Set stepToMode(index=s);
Activity act(index=(j,s),mode=stepToMode[s]);
```

上記のように, 作業モードに与えられる添え字がアクティビティの添え字の一部になっている場合，以下のように記述した事と同義になります．

```
Activity act(index=(j,1),mode=stepToMode[1]);
Activity act(index=(j,2),mode=stepToMode[2]);
Activity act(index=(j,3),mode=stepToMode[3]);
```

また,

```
Element j(set=J);
Element s(set=S);
Set stepToMode(index=(j,s));
Activity act(index=(j,s),mode=stepToMode[j,s]);
```

のように，同一の添え字を用いる事も出来ます.

4.5.2 先行制約

この問題で定義されている 18 個のアクティビティは取り得るモードによって意味づけられているものの, そのままでは全く独立したアクティビティとしてスケジューリングされてしまいます. 例えば act[1,1],act[1,2],act[1,3]が同一のジョブの 3 つの工程であることを意識しないので，act[1,2]⇒act[1,1]⇒act[1,3]のように工程の順序を考慮しないで処理されたり, 重なってしまうようにスケジューリングされる可能性もあります. そのようなことを排除し, 同一の仕事の工程が順に処理されるように制約するにはアクティビティ同士の先行制約を用います．先行制約（a<b）は式で書くと

```
a.endTime <= b.startTime
```

ということであり, 二つのアクティビティ同士の実行順序を規定するのみならず, 二つのジョブが重ならないことも保証します.

```
// 先行制約
act[j,s-1] < act[j,s], s > 1 ;
```

この記述は SIMPLE で通常行うように，添え字の条件式を用いて,

```
// 先行制約（j はすべての仕事）
act[j,1] < act[j,2]; // 工程1は2に先行
act[j,2] < act[j,3]；// 工程２は３に先行
```

であることを定義しています．また,

```
// 先行制約
act[j,s-1] < act[j,s], s > 1, 10
```

と記述する事によって，より一般的な先行制約，工程 s-1 は工程 s より 10 以上先に開始されなければならない，という事も表現できます.

a は b より，integer 以上先に開始されなければならないという表現は,

a.endTime + integer <= b.startTime

と同値であり，a<b,Parameter として記述できます．また Parameter 値は負の値でも可能です.

ただし，これらの先行制約は常に hard 制約（必ず満たさなければならない制約）として扱われます soft 制約(出来るだけ満たしたい制約)として扱いたい場合，以降で述べる一般の考慮制約を用いる必要があります.

4.5.3 直前先行制約

直前先行制約とは，ある同一のリソース R を消費する可能性のあるアクティビティ a,b について課される次のような制約です.

もし，a,b が両者ともに R を消費するモードが割り当てられたならば，a の直後に b が実施される.

言い換えると

このリソース R を消費するモードを取っており，かつ a の endTime の後に startTime を持つアクティビティのあらゆるもの中で b の startTime が最小である.

ということで，先行制約よりも，より限定のきつい制約といえます.

この例では A の：

もし工程１の仕事が来るのならば,仕事１の工程1⇒仕事２の工程1⇒仕事３の...

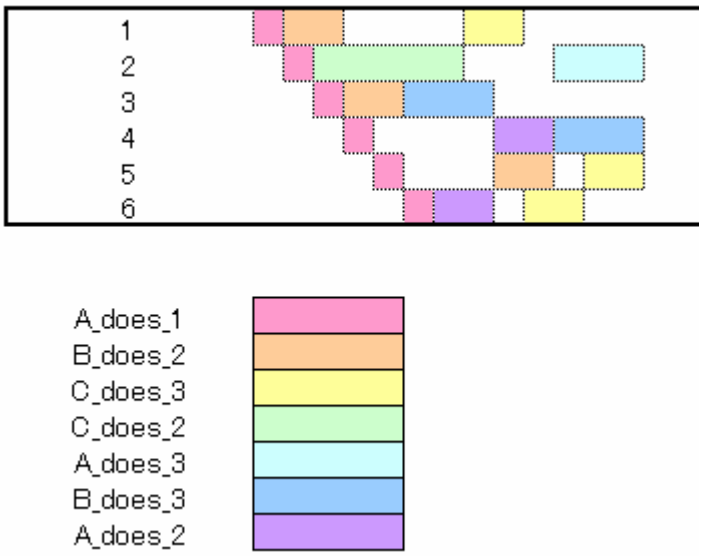
と順序良く来て欲しい,

という要請は，この直前先行制約を用いて叶えることができます．生産スケジューリング問題において，ジョブを遂行する工作機械の準備の都合を考慮するなど，広汎に利用することができます．A の要請を，SIMPLE を用いて記述すると次のようになります.

```
// 直前先行制約（A にとって１の工程は１，２，３... の順で来て欲しい）
act[j-1,1] << act[j,1], "A", j > 1;
```

この制約を追加して解くと次のようなスケジュールが得られます.

図 8　Aの工程１について直前先行制約導入後のスケジュール



Aの工程１はジョブ番号順に並ぶという結果が得られています．直前先行制約は rcpsp のみの機能なので，直前先行制約を定義した状態で納期遅れの最小化はできません．ご了承ください．

4.6 一般の考慮制約の導入

プロジェクト全体の予算の制約など，完了時刻のみならず，各アクティビティに適用した作業モードに依存した量を最小化する場合には一般の制約を記述します．一般の制約は，次の量

Boolean(Activity=="モード") // Activity が"モード"によって処理されるなら 1，そうでなければ 0

Activity.startTime

Activity.endTime

Activity.processTime

上記 Boolean(Activity==" モード ") と同一の Activity の startTime ,endTime ,processTime の積

を定数倍したものを足し合わせた全体に対する下限，上限，等式を設定することによって行います．係数となる定数には通常の SIMPLE による数理モデルと定義と同じく一般の数字や Parameter が利用可能です．

4.6.1 一般の考慮制約の導入の例

例えば次のような例題を考えます．

（例題４）

A,B,C の作業員の各工程について，割り当てた仕事を 2 日以下で行うことができた場合には，1000 円をベースとして日数短縮分に応じた報奨金を払うこととなった．現在各担当員の所要工数は次の通りであるので，

担当員	工程１所要	工程２所要	工程３所要	合計
A_does	１日	２日	３日	６日
B_does	３日	２日	３日	８日
C_does	４日	５日	２日	11 日

もし，各作業を担当員に割り当てた場合の報奨金は次の通りとなる（単位・円）．

担当員	工程１所要	工程２所要	工程３所要	合計
A_does	2000	1000	0	６日
B_does	0	1000	0	８日
C_does	0	0	1000	11 日

問題のそのほかの設定は例題３と同一とするとき，支払う報奨金と納期のトレードオフ関係を求めよ．

この表から，各アクティビティに対してある作業モードを取った際のコスト表が定義できます．

作業モードに応じたコスト表

モード種別	コスト(千円)
A_does_1	2
A_does_2	1
A_does_3	0
B_does_1	0
B_does_2	1
B_does_3	0
C_does_1	0
C_does_2	0
C_does_3	1

コスト表のデータをパラメータ（cost）として次のような考慮制約を定義します.

```
// コストの最小化
Parameter cost(index=m);
Expression costSum;
costSum = sum(Boolean(act[j,s] == m)*cost[m],(j,s,m,m<stepToMode[s]));
softConstraint(10); // 考慮制約の重み
costSum <= 0; // コストの最小化に対応する考慮制約
```

データ cost は次のように与えます.

```
cost =
[A_does_1] 2  [B_does_1]  0 [C_does_1]  0
[A_does_2] 1  [B_does_2]  1 [C_does_2]  0
[A_does_3] 0  [B_does_3]  0 [C_does_3]  1
;
```

コストが 0 を超過した分と，考慮制約の重みを掛けたものをペナルティと設定したことになりますので，コストの最小化を行う意味となります．rcpsp はこのペナルティに，最大完了時刻（これまでの最小は 12 でした）を足した全体を最小化します.

重みの設定には，制約充足ソルバ wcsp と同様 softConstraint() を用います.

```
softConstraint(10); // 考慮制約の重み
costSum <= 0; // コストの最小化に対応する考慮制約
```

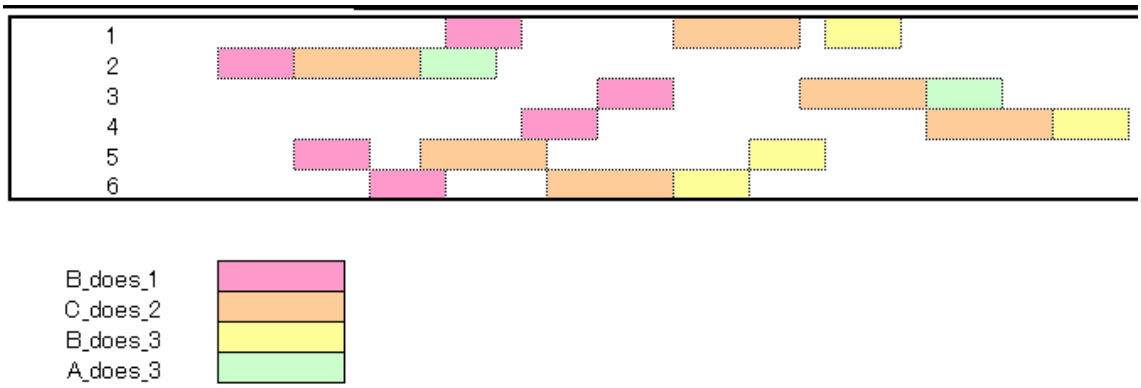
softConstraint() 以下にある制約式に対しては全てその重みが与えられます．sofrConstraint() が記述されていない場合には,

```
options.defaultConstraintWeight
```

の値に従います．また，重みには整数値のみが与えられます．

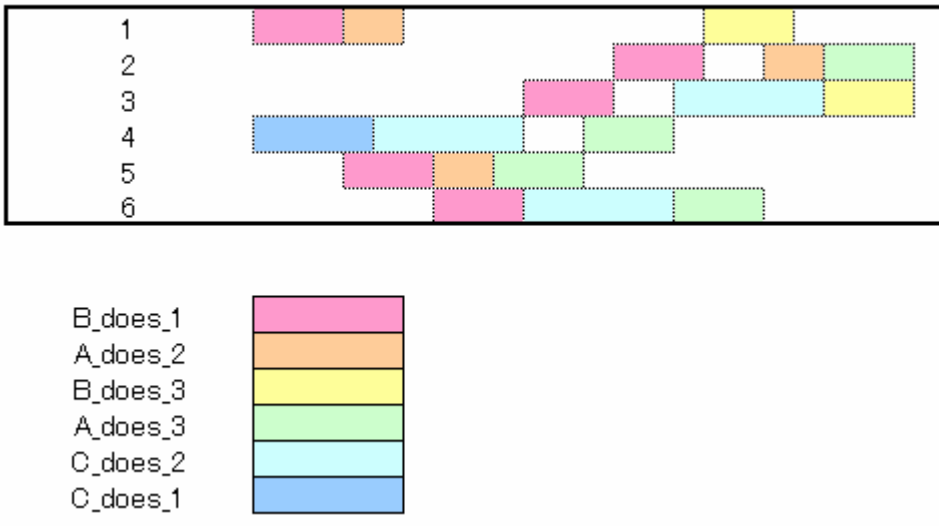
この状態では考慮制約の重みが大きいので次のような報奨金コスト 0（完了時刻 36）の解が出力されます．

図 9　コスト 0，完了時刻 36 の解



重みを 5 以下に緩めると，コストを所要するが完了時刻は短縮された解が出力されるようになります．例えば次は重みを 2 に設定した場合に出力されるコストが 3000 円の解（完了時刻 22）です．

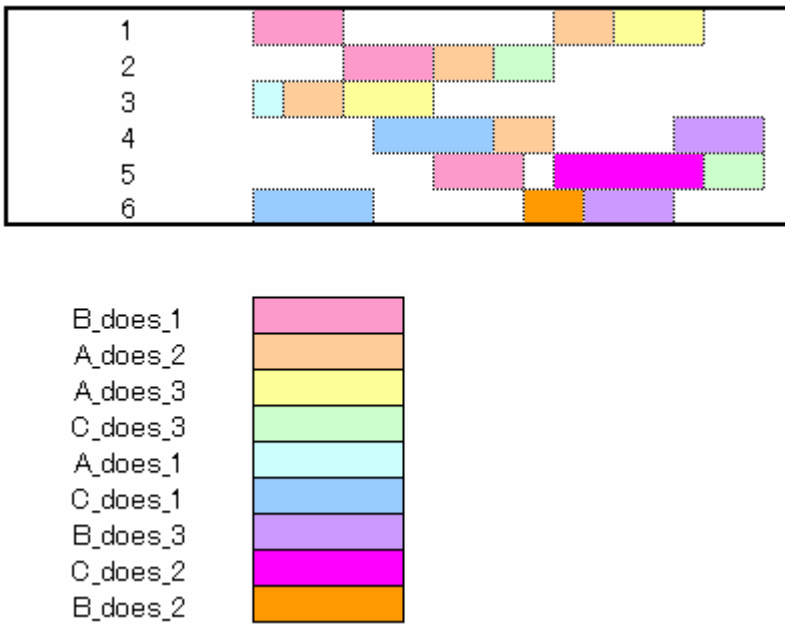
図 10　コスト 3000，完了時刻 22 の解



重みには整数値しか与えられませんので，考慮制約の重みを 1 と設定したら，より最終完了時刻の最小化を志向した解とするには options.defaultObjectiveWeight を大きく設定してゆきます．次は考慮制約の重みを 1，目的関数の重みを 2 とした際のコスト 9000 円，完了時

刻 17 の解です.

図 11　コスト 9000，完了時刻 17 の解



以下は重みの設定と，結果として得られたコストと最大完了時刻を表にしました．重みの設定に対して意図どおりの解が表れていることがわかります．

表 10　コストと最大完了時刻を考慮した場合の出力

考慮制約重み	目的関数重み	コスト(千円)	最大完了時刻
10	1	0	36
5	1	1	31
4	1	2	26
3	1	3	22
1	1	5	20
1	2	9	17
1	3	20	12

次はコストと最大完了時刻をプロットしたものです．トレードオフ関係が明確に現れています．

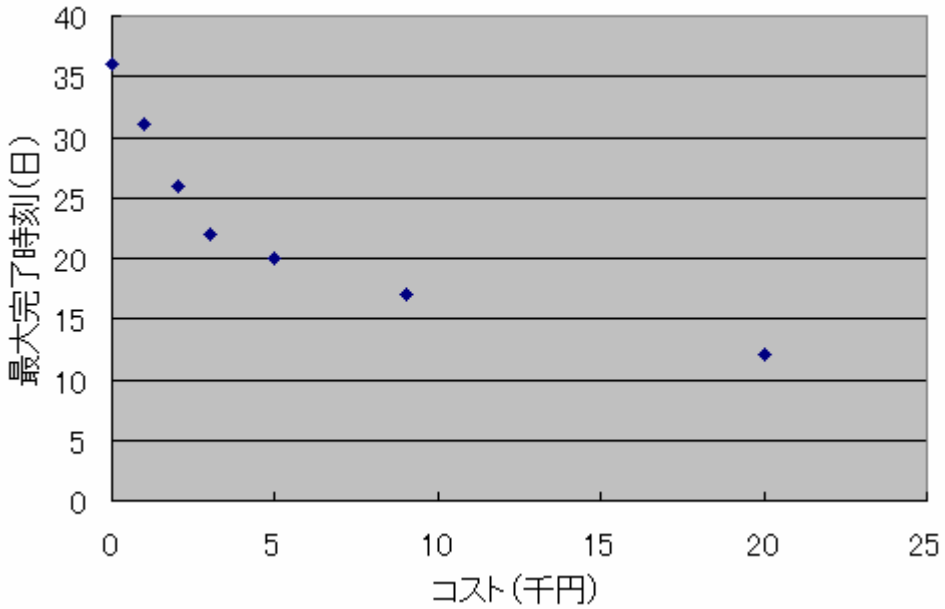


図 12 コストと最大完了時刻の関係

4.6.2 一般の考慮制約の導入に関する注意

考慮制約ではかなり自由な式を記述することができますが，実装の都合上，あらゆるパターンの式が必ず考慮されるとは限らないことにご注意ください．例えば，あるアクティビティの開始，あるいは終了時刻を直接制約するような

```
a.startTime >= 3;
b.startTime == 5;
```

のような制約は rcpsp の実装の都合上ほとんど考慮されません．アクティビティ同士の相対的な時間の順序の考慮

```
a.startTime >= b.startTime;
```

であればある程度考慮されます．前項で説明した Activity が取るモードに対する Boolean 関数を用いた式は比較的良く考慮されるものの例です．ただし，Boolean を用いた場合には，内部で ActMode オブジェクトが作成されるため，**ActMode は予約語**となっています．

また，納期最小化時には，考慮制約には重みを与える事はできません．さらに，一般の考慮制約の記述には制限があり，Boolean のみを用いた式，もしくは，2 つのアクティビティの先行関係を表す以下の式，

```
a.startTime(endTime) + Parameter <= b.startTime(endTime);
```

を満たす式のみの記述となります．

4.7 同時，オーバーラップの制約

アクティビティ同士の開始，終了時刻についての

あるアクティビティ群は同時に開始させたい

あるアクティビティ群は重なっているようにしたい

あるアクティビティが終了したら一定時間後に別のアクティビティが実行されるようにしたい

という制約は人工的な資源とアクティビティを定義し，直前先行制約を使うと表現することができます．

4.7.1 同時開始制約

例題3のモデルに，次の状況を織り込んでみます．

> （例題３a）
> 仕事１，2，３のいずれかの工程１がAに割り当てられるときにはCにも同時に仕事１，2，３の工程１が割り当てられるようにして技術を学ばせたい．その他の問題設定は例題３と同一．

すなわち仕事１，2，３のうちの一つの工程１がＡに，一つの工程がＣによって担当されるならば，その工程は同時に開始しなければならない．ということになります．例えば二つのアクティビティ a,b を同時に開始させるには，あと一つダミーのアクティビティ d を定義し，R は a,b,d が共通に消費する資源として

```
d << a, "R";
d << b, "R";
```

と定義すればよい，ということが知られています．なぜなら，これらの定義によって a,b はともに d の endTime 後に開始するアクティビティのうち，最も開始の速いものでなければならない（**4.5.2** 参照）ということになります．これを満たすには a,b は同時に開始しなければならないからです．

さて，例題 3a に向けてモデルを修正します．まず，同時に開始させたいアクティビティ群が適用されるモードが共通に，あるダミーの資源（dummyResource）を利用するようにします．この場合にはＡが工程１を行う，というモード（A_does_1）とＣが工程１を行う，というモード（C_does_1）の二つが dummyResource を利用するようにします．さらにダミーのアクティビティ（dummyActivity）を宣言し，それはダミーのモード（dummyMode）によって実行されるものとします．dummyMode[8]は dummyResource を利用するように定義します．これで，ターゲットとなるリソースがすべてダミーリソース dummyResource を利用するようにで

[8] 自動で定義される DummyMode と混同しないようご注意下さい．

きました．次に dummyActivity と仕事 1，2，3 の工程 1 に対応するアクティビティに次のような直前先行制約を課します．

```
dummy << act[j,1], "dummyResource", 1 <= j <= 3;
```

この直前先行制約は，アクティビティ act[ｊ，１]に dummyResource を使用するモードが割り当てられた場合にのみ，act[j,1]を同時に実行することを制約するものですので，（dummyResource を消費するように定義していない）B_does_1 モードが割り当てられている act[ｊ，1]は無関係です．これは A と C がかかわる場合のみ，という意図に合致しています．忘れがちなポイントですが，リソースを増やしたので，その容量を定義しておきます．ダミーの資源については容量制約を考慮させないために，次のように潤沢に利用可能量を定義しておきます．

```
cap["dummyResource",t] = 10; // ダミーは潤沢に定義する.
```

以下にモデルの全体を示します．

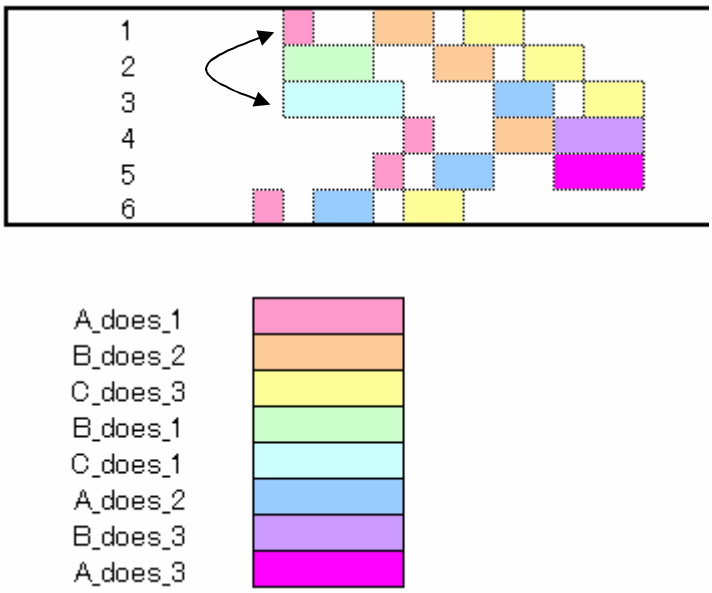
```
//
//  例題 3a（同時制約の紹介）
//
Set M;
Element m(set=M); // モード
      ..
ResourceCapacity cap(resource=R,timeStep=T);
cap[r,t] = 1;
cap["dummyResource",t] = 10; // ダミーは潤沢に定義する.
      ..
Set J = "1 .. 6"; // 仕事
      ..
// 追加される直前先行制約
Activity dummy(name="dummy",mode=Set("dummyMode"));
dummy << act[j,1], "dummyResource", 1 <= j <= 3;
      ..
```

データファイルには

```
req =
[ A_does_1, A, 1] 1  [ B_does_1, B, 1] 1  [ C_does_1, C, 1] 1
            ...
[ dummyMode, dummyResource, 1] 1
[ A_does_1,  dummyResource, 1] 1
[ C_does_1,  dummyResource, 1] 1
;
stepToMode = ...
```

と記述し, モードと資源を追加します. 上記のモデルによるスケジュール結果は次の通りです.

図 13　例題 3a の解



意図どおり，仕事 1，3 の工程 1 に A，C が割り当てられていますが，その実施時刻が揃っています.

4.7.2 オーバーラップ制約

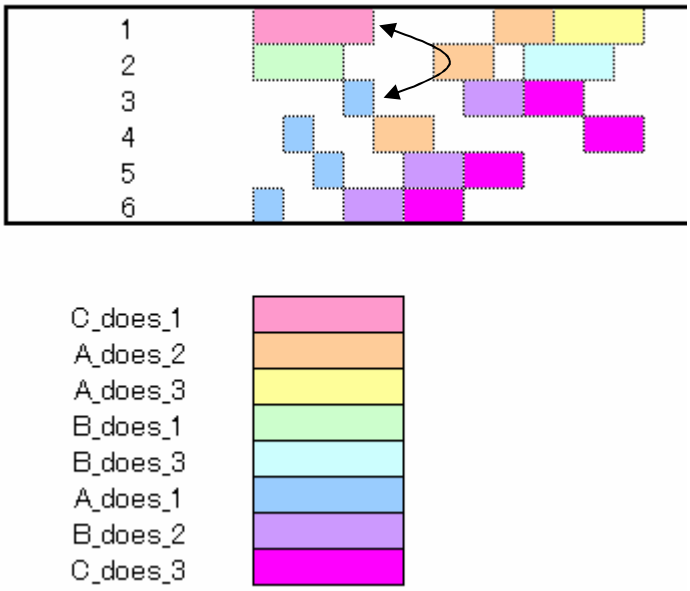
直前先行制約の定義において

```
act[j,1]<<dummy , "dummyResource", 1 <= j <= 3;
```

のように先行関係を逆にすると今度は，A，C による工程 1 を担当する実施時期が少なくとも重

なっているようにする（オーバーラップ制約），という意味となります．これは act[j,1]の endTime 後に開始する最初のアクティビティが dummy でなければならないことによります．これが言えるのは，act[j,1]が互いに重なっているときのみです．結果として次のようなスケジュールが得られ，仕事１，仕事３の工程１を A，C が担当している時期が一日ですが，重なっています．

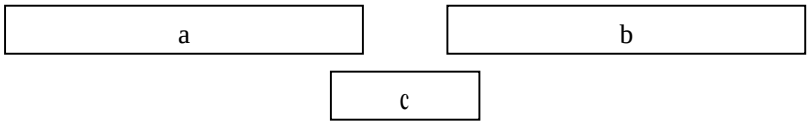
図 14 オーバーラップ制約を導入したモデルの解



4.7.3 一定時間後の開始制約

オーバーラップ制約を二つ用いることによって，a が終了してから一定時間 S 後に b が開始する，という制約を表現することができます．

図 15 開始間隔の制約概念図



上図で c は S＋2 の長さを持つアクティビティで，a と c，c と b に対して，上記の方法でオーバーラップ制約を掛けておきます．そうすると，c の長さが決まっているので，a，b の間隔は必ず S 以下となります．特に S の長さを 2 とすれば，a と b は連続します．

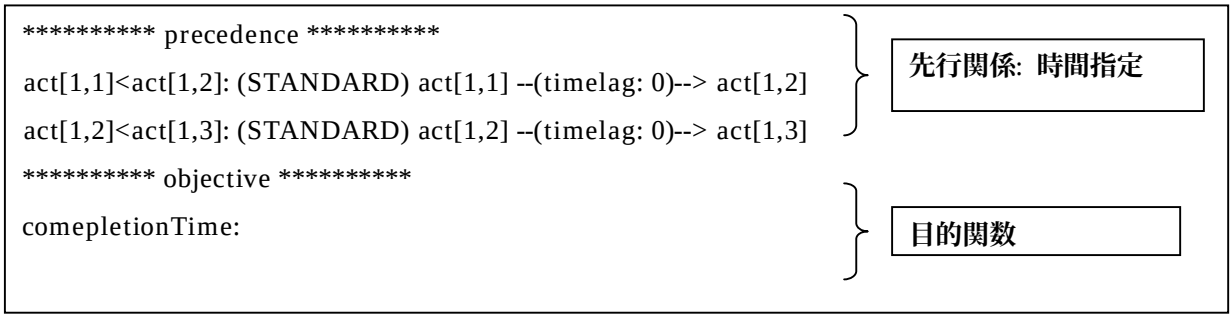
4.8 showSystem() の出力

ミスタイプ等の誤った記述により，期待した動作がなされない場合には，SIMPLE の機能である，showSystem() を用いて，rcpsp のモデルの内容を表示させると便利です．例題 3 (人員スケジュール問題，工程分割)のモデル内容を一部抜き出してみます．

```
********** activity **********
. . .
act[1,1]:
        duedate = inf                                   ← 納期 (inf は設定なし)
        "A_does_1": time = 1
                "A":
                        (  0,  1]...    1
        "B_does_1": time = 3                            ← 処理モード:処理時間
                "B":                                        消費される資源
                        (  0,  3]...    1                       時間ステップ
        "C_does_1": time = 4                                    毎の消費量
                "C":
                        (  0,  4]...    1

        1 predecessor(s):                               ← 先行するアクティビティ
                sourceActivity

        1 successor(s):                                 ← 後続するアクティビティ
                act[1,2]
. . .
********** resource **********
"A":                                                    ← 資源
        sourceActivity                                  消費される可能性のある
        act[1,1]                                        アクティビティ
        act[1,2]
        act[1,3]
. . .
        act[6,3]
        sinkActivity                                    時間ステップ毎の利用可能
        (  0, 41]...    1/hard                              消費量
```


********** precedence **********
act[1,1]<act[1,2]: (STANDARD) act[1,1] --(timelag: 0)--> act[1,2]
act[1,2]<act[1,3]: (STANDARD) act[1,2] --(timelag: 0)--> act[1,3]
先行関係: 時間指定
********** objective **********
comepletionTime:
目的関数