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

2007/01/28 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．

最終章には SIMPLE を用いた一般的な数理計画問題のモデル化の例を掲載しています．モデルを作成する際の参考にして下さい．

1.1 数理計画問題とは

数理計画問題とは,「与えられた条件の下で, 望ましさの尺度を表す何らかの関数の最小値（最大値）を求め，さらにその最小値（最大値）を与える不特定要素の値を決定する」という問題です．

上記における,「与えられた条件」は制約条件,「望ましさの尺度を表す関数」は目的関数,「不特定要素」は変数, と一般に呼ばれています. この用語を用いて書き直すと, 数理計画問題とは,「制約条件を満たす範囲における目的関数の最小値（最大値），及びその最小値（最大値）を与える変数を求める問題」といえます．

例えば，$x \geq 0$ において $3x+2$ の最小値を求める問題は，数理計画問題です．この場合，制約条件は $x \geq 0$，目的関数は $3x+2$，変数は x となります．

この問題は数理計画の世界では次のように書かれます：

- 目的関数： $3x+2 \rightarrow$ 最小化
- 制約条件： $x \geq 0$

考える間もなく，上記の数理計画問題の最もよい目的関数値は 2 （$x=0$ のとき）となります．このときの変数の値を最適解と呼びます．

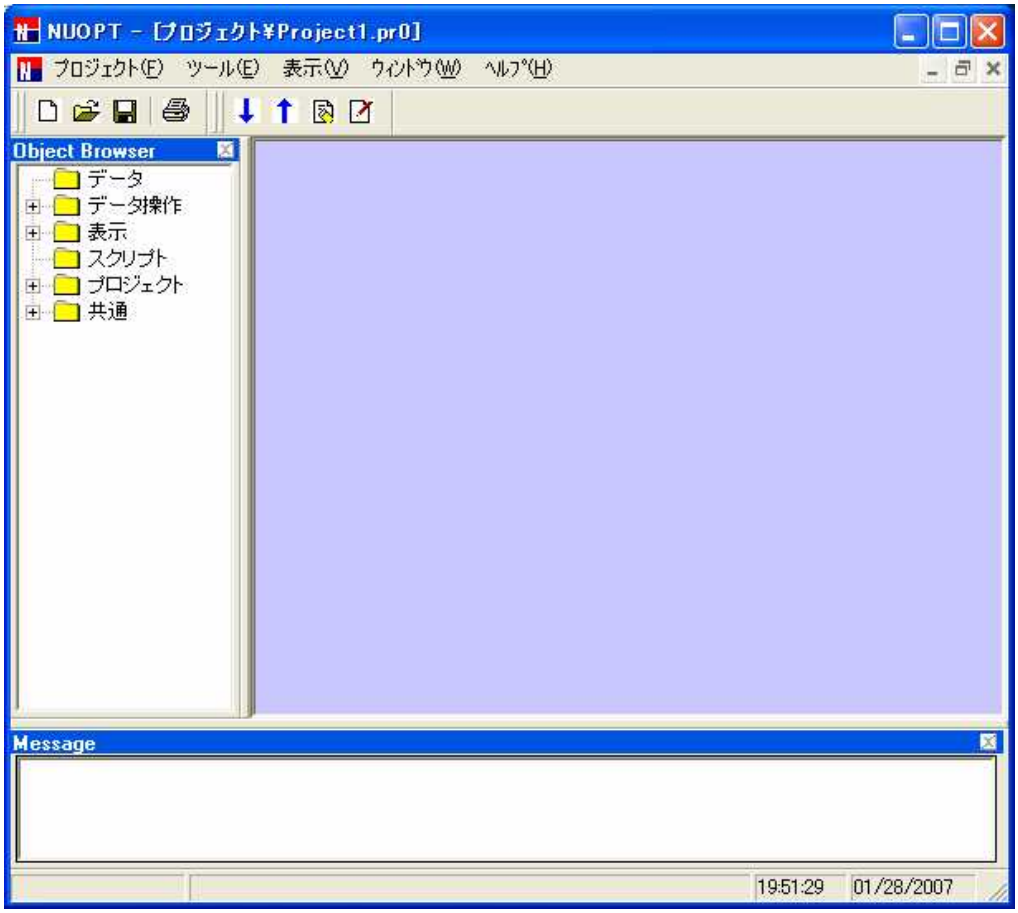
最適解を求めることを,「数理計画問題を解く」あるいは「最適化する」といいます．

1.2 NUOPT の起動

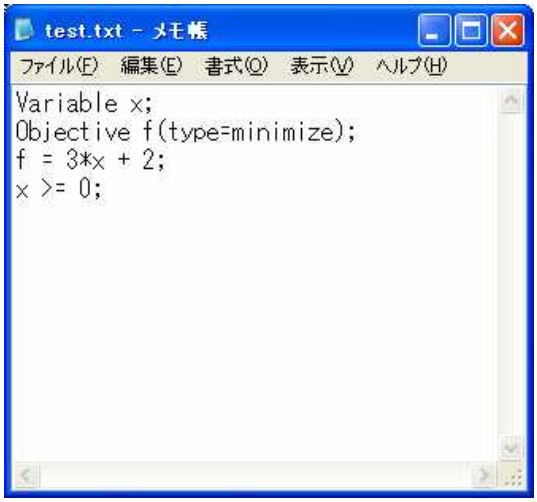
早速 windows 版 NUOPT を起動させてみましょう，まずは左下のスタート画面から

全てのプログラム -> NUOPT -> NUOPT GUI

を選択して下さい．以下のような画面が立ち上がります．

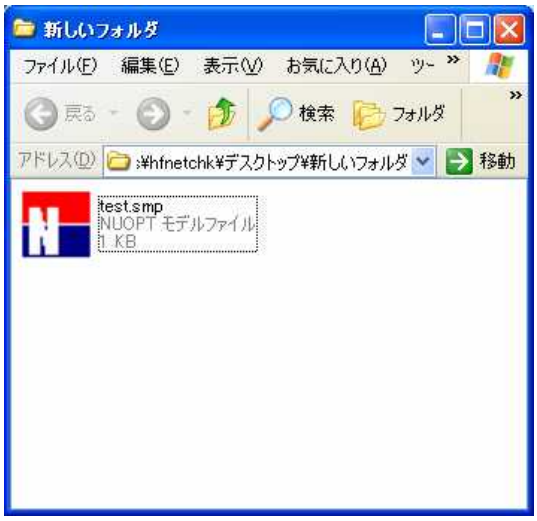


次に，どこでもいいので新規にテキストファイルを作成して下さい．例えば `test.txt` を作成したとしましょう．その中に，次のように書いてください．

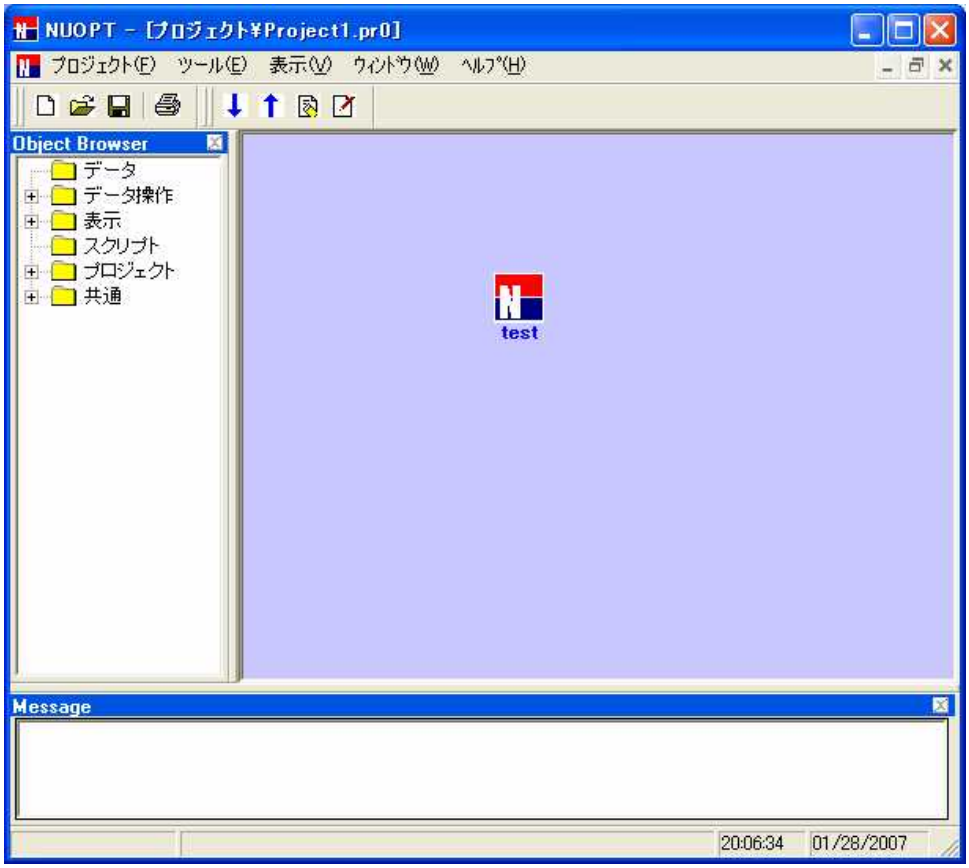


次に，このファイルの拡張子を `.txt` から `.smp` に変更して下さい．

「拡張子を変えるとファイルが使えなくなる可能性があります. 変更しますか？」というメッセージが表示されますが, 無視して下さい. 拡張子を変えると, 次のようなファイルができます.



このファイル `test.smp` をドラッグ&ドロップで `GUI` の中に移動させます.



最後に，この test アイコンをダブルクリックします．すると NUOPT が計算を開始し，次のような結果が表示されます．

	NUOPT状態名	NUOPT状態値
1	バージョン	8.1.3
2	ステータス	最適化正常終了
3	問題名	test
4	変数の数	1
5	制約式の数	0
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
10	内点法反復	5
11	行列分解回数	6
12	最適性ノルム	8.333333583e-011
13	経過時間(秒)	0.09

この一連の操作で，あなたは NUOPT を使って次の数理計画問題を解いたことになります．

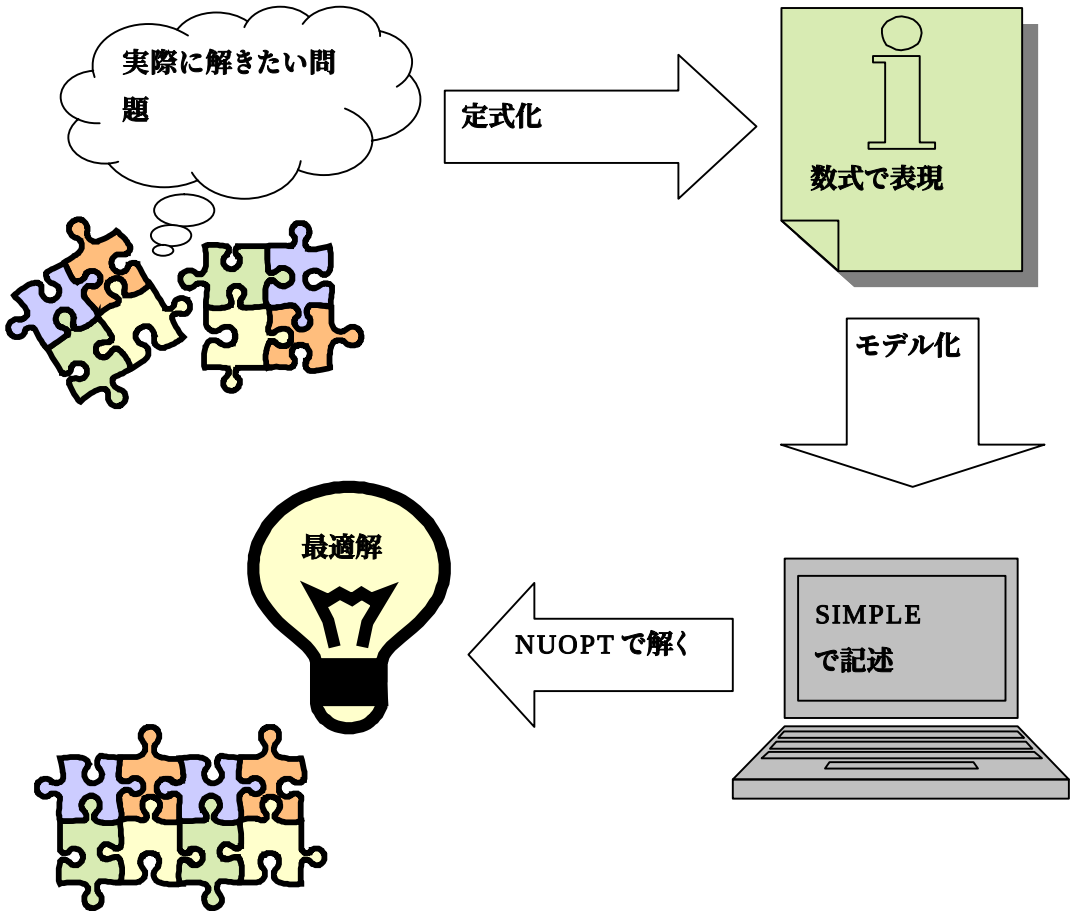
- **目的関数：** $3x+2$ → **最小化**
- **制約条件：** $x \geq 0$

１.３ NUOPT で数理計画問題を解く手順

NUOPT を用いて数理計画問題を解く際，以下の手順を取ります．

1. 解きたい問題を数理計画問題として取り扱う（定式化）
2. 数理計画問題を， NUOPT が可読な形式で記述する（モデル化）
3. NUOPT で実行する

（1）の定式化に関しては，NUOPT とは独立でユーザ自身の手で行う必要があります．（2）のモデル化は，数式に近い表現で記述することができるモデリング言語 SIMPLE を使用します．（3）の実行は（2）で作成されたモデルファイルを NUOPT に渡し，最適化を行うということです．



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

次のような生産計画問題を考えます.

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．

[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・Linux 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/UNIX・Linux 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． windows 版/UNIX・Linux 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 定数

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
定数	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/日を表す定数です．SIMPLEの記述では，以上のような定数を使用した記述が可能です．

ここでは，定数を用いて，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");
Parameter costY(name="油田Yの運転コスト");
cost = costX*x + costY*y;
```

まず，Parameter で，定数を宣言します．モデル中で使用する定数は，使用する前に宣言する必要があります．定数の値は，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，定数名を指定します．定数名は，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．

次にモデルに定数を与えるために，以下の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 である必要があります．

```
"油田Xの運転コスト" = 180;
"油田Yの運転コスト" = 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，定数値

を記述します．セミコロン ； が定数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． Parameter とデータファイルを使用することで，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" = 100;
"油田Ｙの運転コスト" = 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=5
油田Ｙの運転日数=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
定数	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)	$\text{costX} \cdot x_0 + \text{costY} \cdot x_1$	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", index=i);

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

// 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，定数と同様に，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", 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[0]=1.5
油田の運転日数[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．

定数	$costX_i$，$i \in OilField$	油田 i の運転コスト/日

$costX_0$，$costX_1$ はそれぞれ以前の costX， costY に対応する定数です．SIMPLE でも同様に定数の添字付けを用いて，以下のように修正します．

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

↓

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

定数の添字付けは，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" = [0] 180 [1] 160;
```

添字付きの定数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[0]=1.5
油田の運転日数[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，以下の油田集合の具体的な要素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
定数	$norma_j,\ \ j \in Product$	製品 j のノルマ/週

SIMPLE の記述においても同様に定数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", 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定数にします．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" = [0] 180 [1] 160;
"製品ノルマ" = ["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，定数，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	$OilField = \{0,1\}$	油田集合
	$Product = \{\text{重油},\text{ガス}\}$	製品集合
定数	$\mathrm{cost}X_i,\ i \in OilField$	油田 i の運転コスト/日
	$norma_j,\ j \in Product$	製品 j のノルマ/週
変数	$x_i,\ \ i \in OilField$	油田 i の運転日数/週
目的関数(最小化)	$\mathrm{cost}X_0 \cdot x_0 + \mathrm{cost}X_1 \cdot x_1$	運転コスト/週
制約条件	$6x_0 + x_1 \geq norma_{\text{重油}}$	重油ノルマ/週
	$4x_0 + 6x_1 \geq norma_{\text{ガス}}$	ガスノルマ/週
	$0 \leq x_i \leq 5,\ \ i \in OilField$	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", index=i);

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

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

$$cost = \sum_i costX_i \cdot x_i$$

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，定式化において定数 $prodX_{i,j}$ を導入し，制約式を次のように記述します．

制約条件	$\sum_i prodX_{i,j} \cdot x_i \geq norma_j$, $i \in OilField$ ， $j \in Product$	製品 j のノルマ制約式/週
定数	$prodX_{i,j}$ ， $i \in OilField$ ， $j \in Product$	油田 i の製品 j 生産量/日
	$norma_j$ ， $j \in Product$	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", index=(i,j));
sum(prodX[i,j]*x[i], i) >= norma[j];
```

複数の添字に依存する定数を宣言する際には, 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" = [0] 180 [1] 160;
"製品ノルマ" = [“重油”] 12 [“ガス”] 24;
"油田の生産量" =
[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．従って，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，定数，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	$OilField = \{0,1\}$	**油田集合**
	$Product = \{$**重油,ガス**$\}$	**製品集合**
定数	$\mathrm{cost}X_i$ ， $i \in OilField$	**油田 i の運転コスト/日**
	$norma_j$ ， $j \in Product$	**製品 j のノルマ/週**
	$prodX_{i,j}$ ， $i \in OilField$ ， $j \in Product$	**油田 i の製品 j 生産量/日**
変数	x_i ， $i \in OilField$	**油田 i の運転日数/週**
目的関数(最小化)	$\mathrm{cost}X_0 \cdot x_0 + \mathrm{cost}Y_1 \cdot x_1$	**運転コスト/週**
制約条件	$\sum_i prodX_{i,j} \cdot x_i \geq norma_j$ ， $i \in OilField$ ， $j \in Product$	**製品 j のノルマ制約式/週**
	$0 \leq x_i \leq 5$ ， $i \in OilField$	**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", index=i);

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

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

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

$$prod_j = \sum_i prodX_{i,j} \cdot x_i \ , i \in OilField \ , \quad j \in Product$$ 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", 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[重油]=12
製品の生産量[ガス]=24
```

2.6 可変定数

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，以下のように可変定数を宣言します．

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

可変定数は，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（定数（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[重油]=12
製品の生産量[ガス]=24
油田の運転日数[0]=1.5
油田の運転日数[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[重油]=18
製品の生産量[ガス]=36
油田の運転日数[0]=2.25
油田の運転日数[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[重油]=32
製品の生産量[ガス]=48
油田の運転日数[0]=4.5
油田の運転日数[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", index=i);
```

↓

```
IntegerVariable x(name="油田の運転日数", 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[重油]=15
製品の生産量[ガス]=26
油田の運転日数[0]=2
油田の運転日数[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，定数，可変定数，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 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]);
}
```

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.

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

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

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

全運転コスト<objective>: 180*油田の運転日数[0]+160*油田の運転日数[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，定数値，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．

2.10 SIMPLE FAQ

モデリング言語 SIMPLE を用いる際に頻出する注意点を列挙します.

- 大文字と小文字は区別される
- 積演算子を省略してはならない
- 半角スペースは自由に入れてよい（等号/不等号の間は駄目）
- 改行は自由に入れてよい
- 文末には必ず半角セミコロン ; を入れる
- // はコメントを意味する
- name= 引数はダブルクォート " で囲む
- minimize や maximize はダブルクォート " で囲まない
- name や type のように複数の設定を行うときはカンマで区切る
- name や type を設定する順番は変えて良い
- 等式付不等号 <= と >= は使用できるが，等式なし不等号 < と > は使用できない
- = は代入， == は等価を表わす

3. 数理計画問題を解く（NUOPT チュートリアル）

windows 版/UNIX・Linux 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， windows 版/UNIX・Linux 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」，GUI 付属のヘルプファイル（windows 版）をご覧下さい.

3.1 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．

3.1.1 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];
```

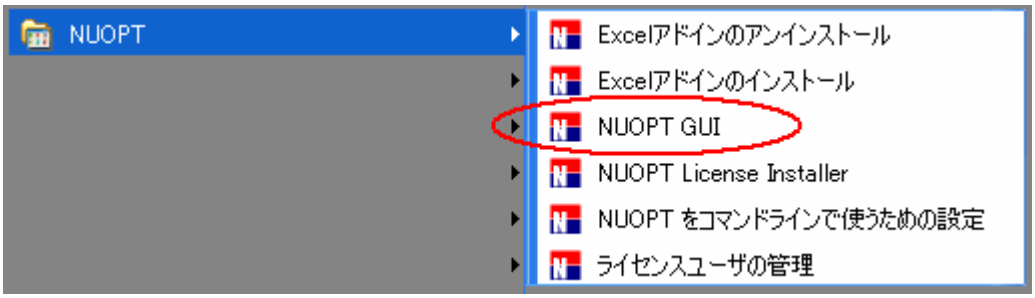
2. データファイルを作成する

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;
```

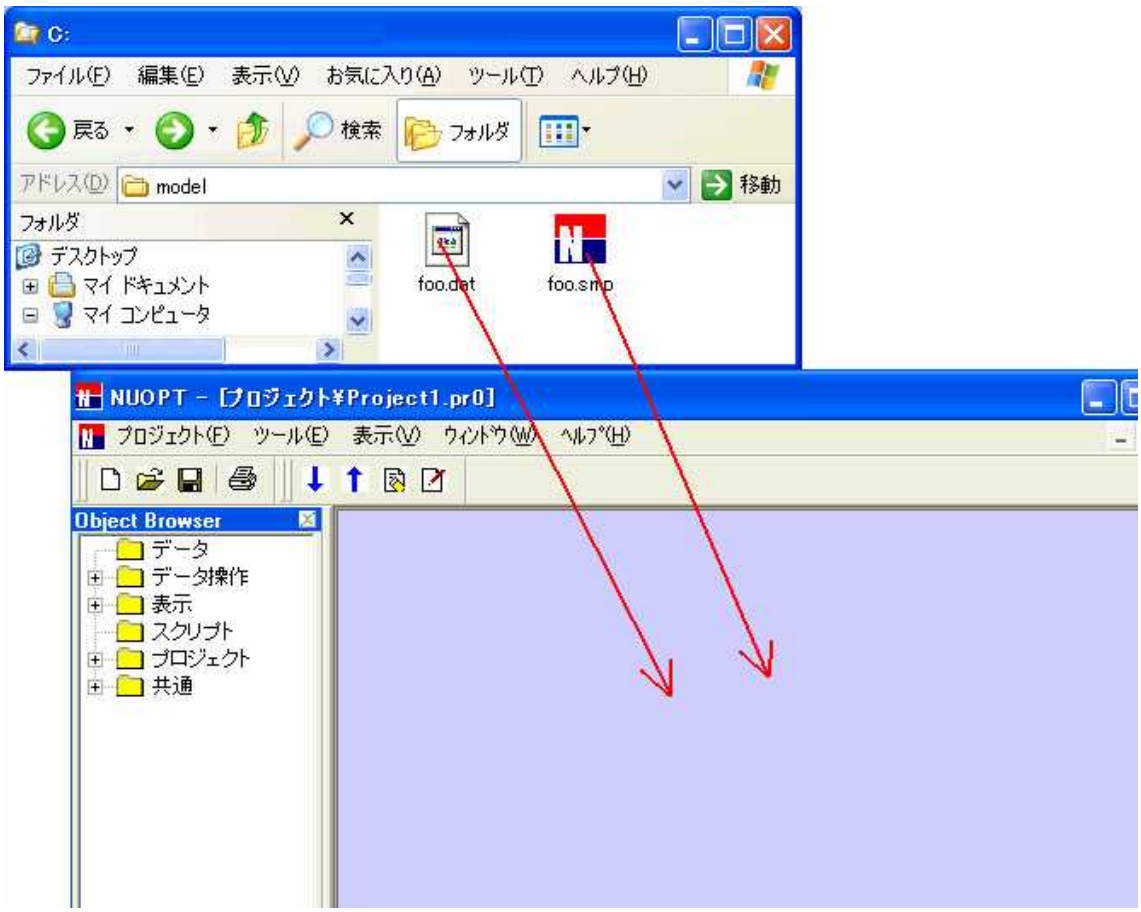
3. 最適化計算を行う

まず，NUOPT GUI を起動します．



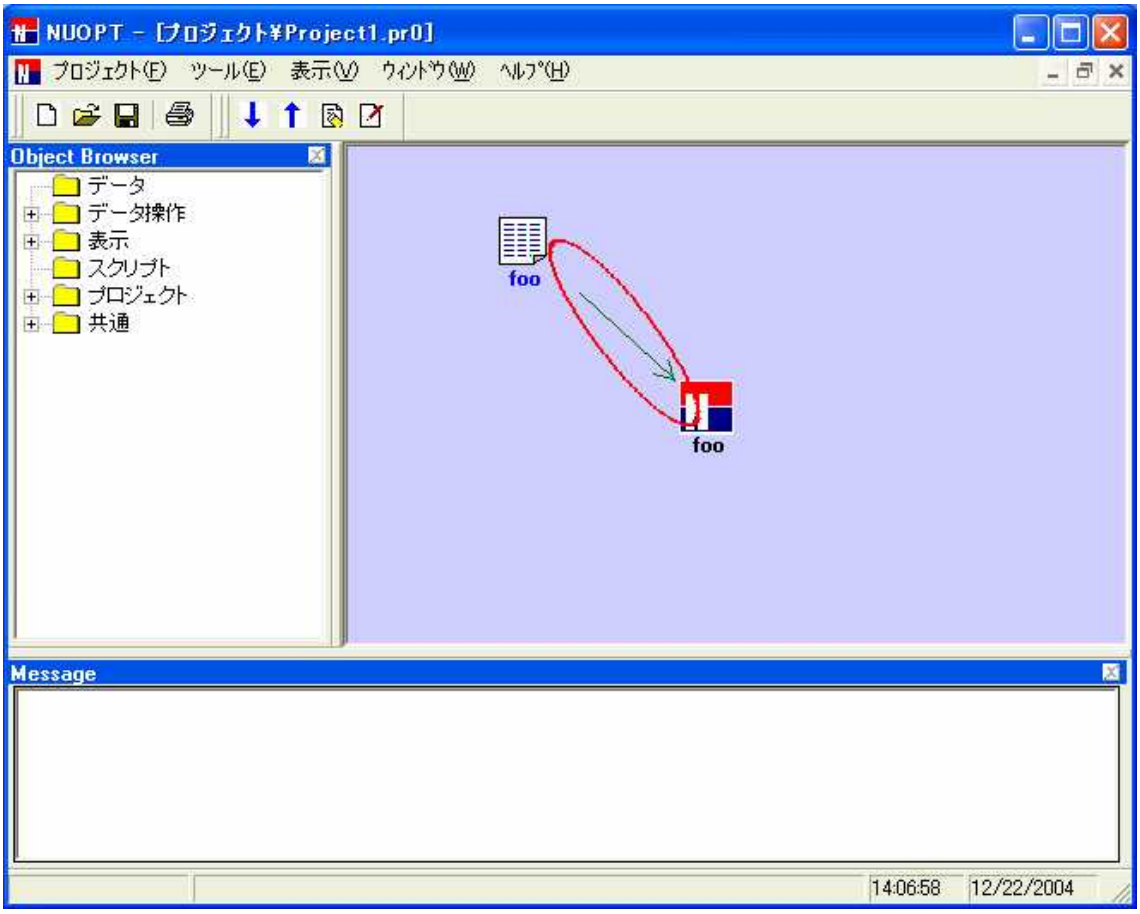
すると，次のような画面が立ち上がります．

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.

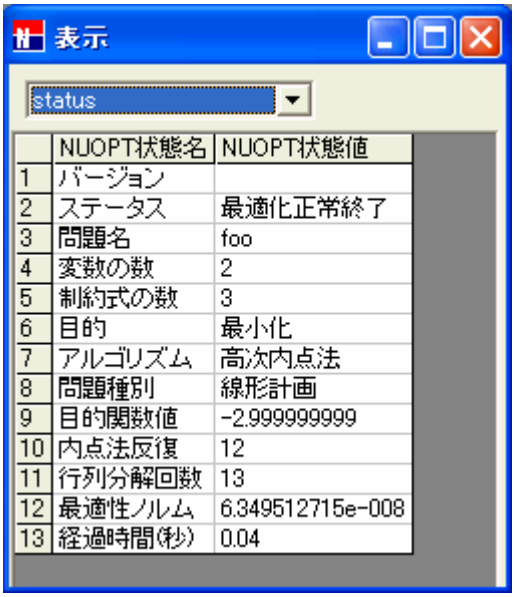


次に, プロジェクトボードに表示されたデータアイコンからモデルアイコンに矢印を繋げます. アイコンを矢印によって連結するには以下の操作を行ってください.

(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
```

3.1.2 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

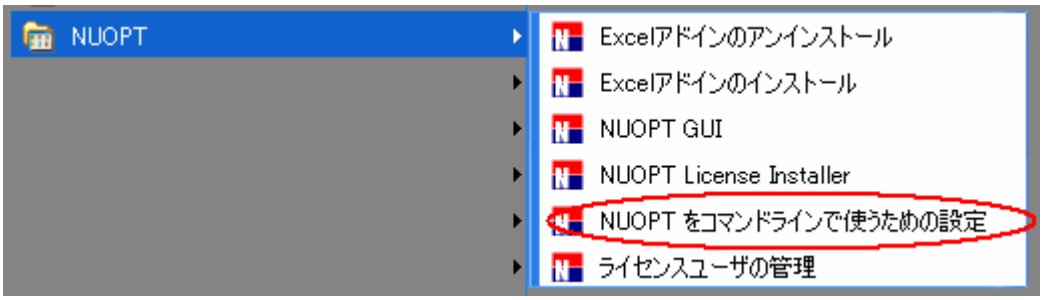
これらの操作については「**3.1.1** GUI を用いる方法」と同様の作業を行ってください.

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.smp
```

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.exe foo.dat
```

そうしますと，「3.1.1 GUI 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．

3.2 UNIX・Linux 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・Linux 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
```

4. 資源制約付きスケジューリング問題を記述する（RCPSP チュートリアル）

本章では，資源制約付きスケジューリング問題の一種である人員スケジューリング問題を，SIMPLE を用いて記述する方法を紹介します．

4.1 資源制約付きスケジューリング問題とは

資源制約付きスケジューリング問題とは

- 幾つかの作業が存在し，一定の資源下でそれら最後の作業の完了時刻を最小化する問題
- 納期のある幾つかの作業が存在し，一定の資源下でそれらの納期遅れを最小化する問題

のことを指します．NUOPT では資源制約付きスケジューリング問題ソルバ rcpsp を用いてこれらの問題を解く事ができます．

4.2 人員スケジューリング問題

次のような人員スケジュール問題を考えます．

（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．

この問題に対する SIMPLE の定式化は以下のようになります．

```
//
//  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);
```

それでは，このモデルに対する定式化の手順を見ていきましょう．

rcpsp を利用する際には，必ず以下の構成要素を定義しなければなりません．

- 作業集合
- モード集合
- 資源集合
- Activity

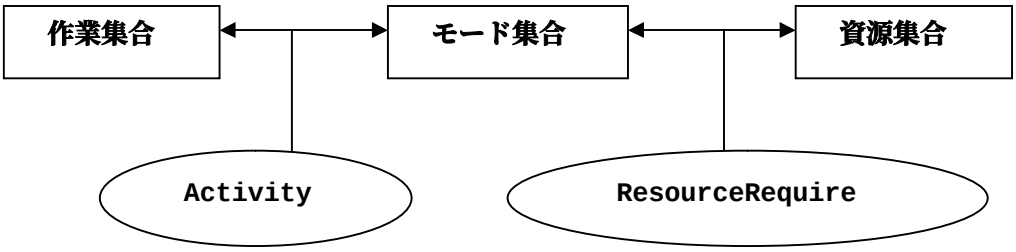
- ResourceRequire
- ResourceCapacity
- 作業時間集合
- 期間集合

作業集合は「必ず実施しなければならない作業」から構成される集合です. 作業集合の要素に, 実施する必要の無い作業が入っていてはいけません. モード集合は「作業に対する対処方法」から構成される集合です.「作業に対する対処方法」の事を, rcpsp ではモードと呼びます. 資源集合は「モードの利用に必要な資源」から構成される集合です. rcpsp を利用する際には, まずこの 3 種類の集合を定義しなければなりません.

次に「どの作業をどのモードで処理するか」に相当する変数 Activity を定義します. rcpsp が決定するのは, この Activity の値です. さらに「各モードはどの資源をどの程度必要とするか」に相当する定数 ResourceRequire を定めます. 最後に「資源はどれだけ利用できるか」に相当する定数 ResourceCapacity を定めます.

作業時間集合は 1 から「モードが資源を利用する最大期間」までの整数からなる集合です. 期間集合は 0 から「スケジューリングする期間」までの整数からなる集合です.

ResourceRequire の値は作業時間集合毎に, ResourceCapacity の値は, 期間集合毎に設定します.



上記の考え方に沿って, この例題の定式化を試みます.

例題において実施しなければならない作業は 6 つの仕事です. そこでこれらを作業集合 J として定義します.

```
Set J="1 .. 6";
```

それぞれの作業には, {A に任せる, B に任せる, C に任せる} の三種類のモード（対処方法）が存在します. そこでこれらをモード集合 M として定義します.

```
Set M="A_does B_does C_does";
```

モードが利用する資源は， A,B,C の 3 人のみですから，これらを資源集合 R として定義します．

```
Set R="A B C";
```

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. 作業

   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

次に，これら 3 つの集合の関連付けを行います．

全ての作業は，モード集合のいずれかのモードで行われる事を示す為 Activity の引数には作業集合 S の添字と，モード集合 M が与えられます．

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

モードに対応する資源を定義する定数 ResourceRequire は次のように定義されます．引数にはモード集合 M と資源集合 R 以外に，作業時間集合 D を新たに定義する必要があります．今回の例では資源を用いる最大期間が 11 日なので， D="1 .. 11" と定めています．

```
Set M="A_does B_does C_does";
Set R="A B C";
Set D="1 .. 11";
Element d(set=D);
ResourceRequire req(mode=M, resource=R, duration=D);
```

モード A_does は資源 A を 6 日間，モード B_does は資源 B を 8 日間，モード C_does は資源 C を 11 日間用いるので， ResourceCapacity それぞれの値は，次のように設定されます．

```
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;
```

次は資源供給量 ResourceCapacity の設定です．各人は同時に 2 つ以上の仕事をすることはできないので，最後の期間まで資源の上限値は全て 1 です．スケジューリングの期間は全体で 40 日なので，期間集合 T は T="0 ..40" で定めます．期間集合は 0 はじまりでなければなりません．これらをまとめると，次のように設定されます．

```
Set R="A B C";
Element r(set=R);
Set T="0 .. 40";
Element t(set=T);
ResourceCapacity cap(resource=R, timeStep=T);
cap[r,t] = 1 // 資源の上限値
```

次に問題の種類を指定します． rcpsp で扱う事の出来る問題は，

- 幾つかの作業が存在し，一定の資源下で最後の作業の完了時刻を最小化する問題
- 納期のある幾つかの作業が存在し，一定の資源下でそれらの納期遅れを最小化する問題

の二種類ですが，今回扱う問題は前者です．これは目的関数で指定します．

```
Objective f(type=minimize);
f = completiontime; // 完了時刻最小化を示す
```

最後に，終了条件を指定します．今回は終了条件として，計算時間 2 秒を設定します．

```
options.maxtim = 2;
```

以上をまとめると，本例題の定式化が完了します．

SIMPLE モデルを実行させると，次のような実行結果が得られます.

```
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.

4.3 ガントチャート出力

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）
```

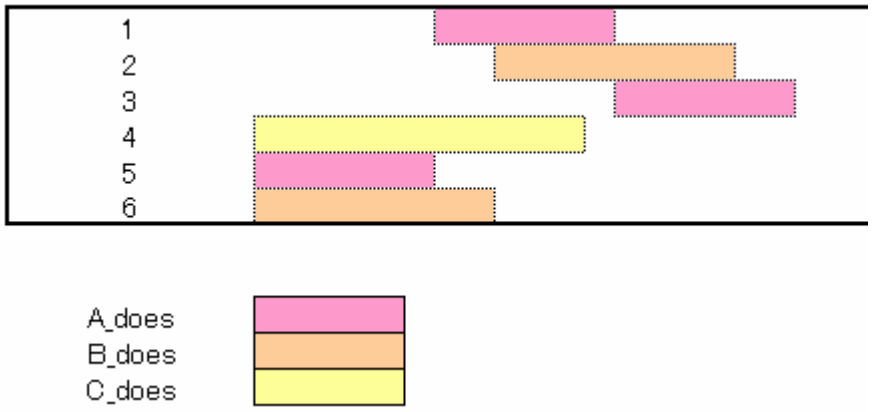
また，アクティビティ 1 .. 3 までをガントチャートに加えたい場合には,

```
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[1])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.

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，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.

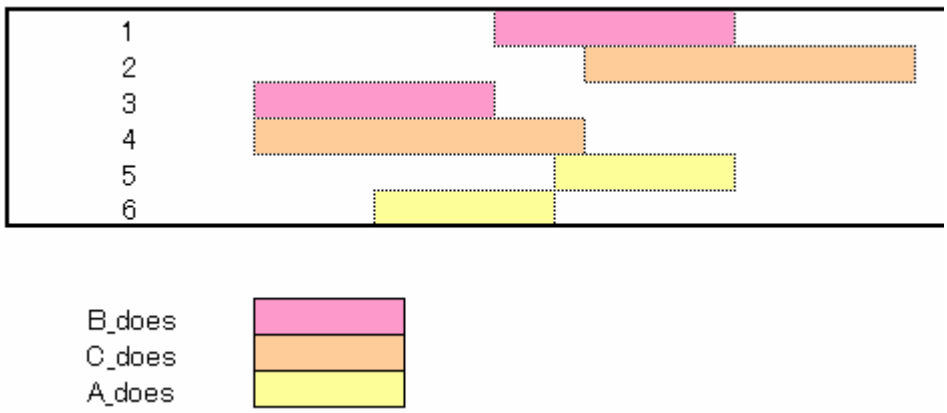
では ResourceCapacity の定義に以下の一行を加えて実行してみましょう.

```
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.

例題１で A が３日目に休む場合のスケジュール

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



4.4 納期遅れ最小化

例題 1 では，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，「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」という問題も多く存在します．

（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]);
```

目的関数には次のようにして納期遅れを設定します.

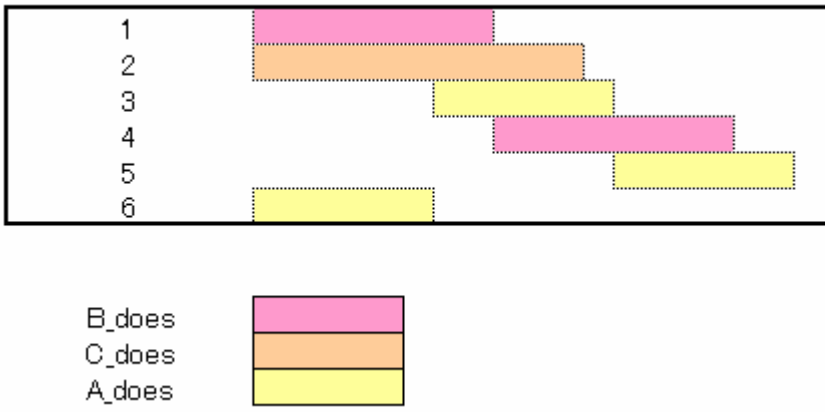
```
// 納期遅れ最小化
Objective f(type=minimize);
f = tardiness;
```

これらを反映した例題 2 の SIMPLE モデルは次のようになります.

```
//
//  例題 2 (納期遅れ最小化問題)
//
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);
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;
Set J = "1 .. 6";
Element j(set=J);
Parameter due(index=j);
due[j] = 10, 1<=j<=3;
due[j] = 17, 4<=j<=5;
due[j] = 6, j==6;
Activity act(name="act",index=j,mode=M,duedate=due[j]);
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 = tardiness; // 納期遅れ最小化
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);
```

この結果のガントチャート出力は，例えば以下のようになります．

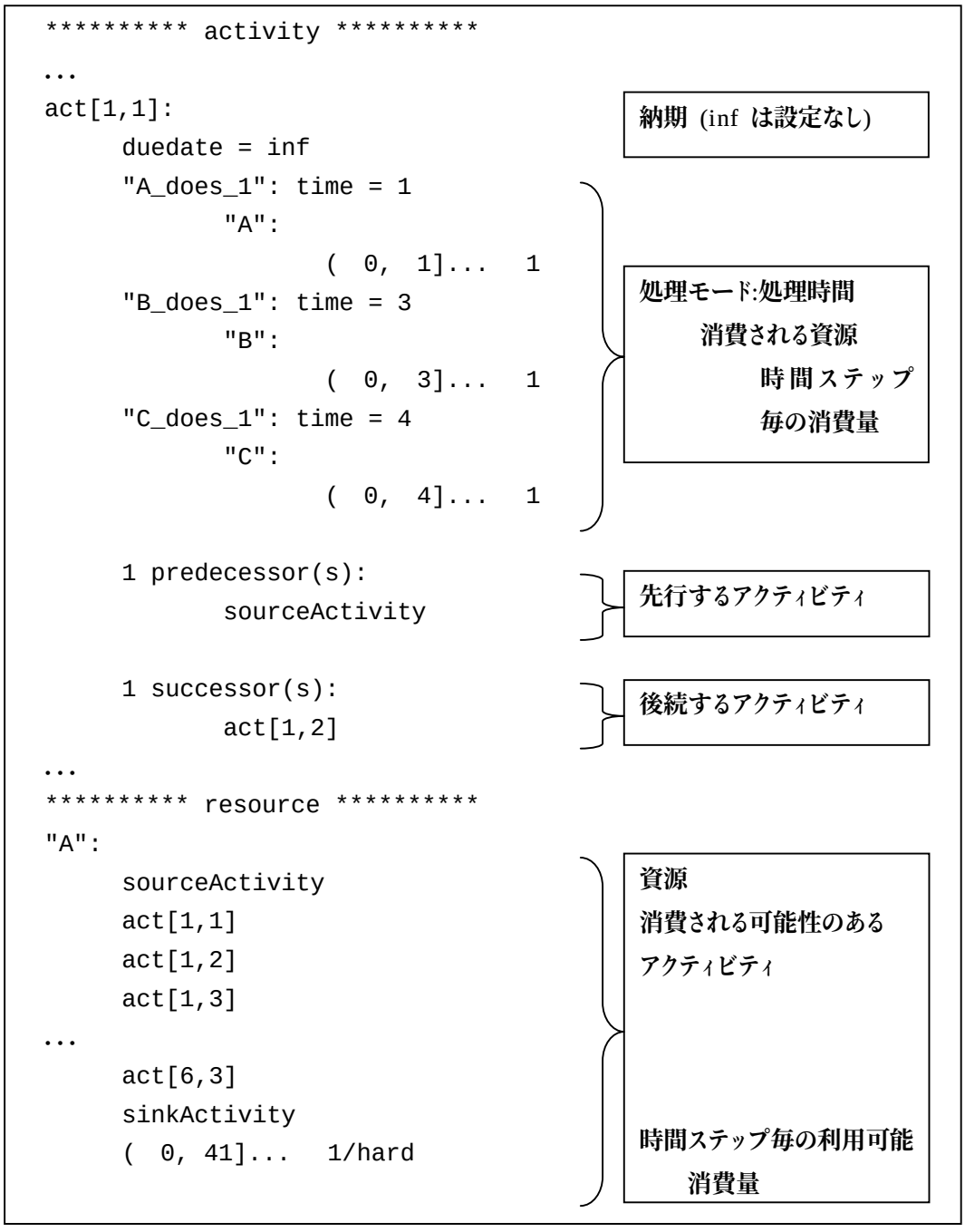
例題２（納期遅れ最小化）の例

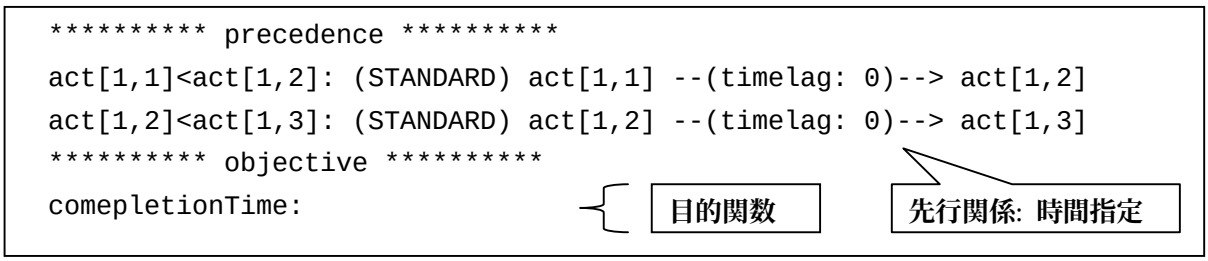


4.5 rcpsp における showSystem 関数の出力

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 関数を用いて，モデルの内容を表示させると便利です． Activity に沿った出力が成されます．

以下は，ある資源制約スケジューリング問題を showSystem 関数で展開した出力の一部です．




********** 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]
********** objective **********
comepletionTime:
目的関数
先行関係: 時間指定

5. 例題集

本章では，様々な数理計画問題を紹介し，それらに対する SIMPLE での定式化の例を与えます．扱う問題の数は合計 14 種類で，その構成は以下の通りです．表における ○ は，それぞれの数理計画問題が，どのようなカテゴリーの問題群に所属するかを表わしています．例えば，ナップサック問題は混合線形整数計画問題です．

LP は線形計画問題， MIP(MILP) は混合線形整数計画問題， QP は凸二次計画問題， WCSP は制約充足問題， RCPSP は資源制約付きスケジューリング問題を意味します．

問題	LP	MIP	QP	WCSP	RCPSP
配合問題	○				
輸送問題	○				
多期間計画問題	○				
ナップサック問題		○			
集合被覆問題		○			
最大流問題		○			
最小費用流問題		○			
多品種流問題		○			
p メディアン問題		○			
p センター問題		○			
割り当て問題		○		○	
最小二乗問題			○		
ポートフォリオ最適化問題			○		
ジョブショップスケジューリング問題					○

5.1 配合問題

配合問題の例として，ここでは特定の組成を持つ合金を生成する問題を扱います．この他にも薬剤の調合や必要な栄養素を含む献立を考えるダイエット問題など配合問題として扱えるものは多岐にわたります．

（例題）
鉛，亜鉛，スズの構成比率が，それぞれ 30%，30%，40%となるような合金を，市販の合金を混ぜ合わせ，できるだけ安いコストで生成することを考えます．現在手に入れることができる市販の合金は９種類で，それらの構成比率と単位量あたりのコストは以下の通りです．

市販の合金	1	2	3	4	5	6	7	8	9
鉛(%)	20	50	30	30	30	60	40	10	10
亜鉛(%)	30	40	20	40	30	30	50	30	10
スズ(%)	50	10	50	30	40	10	10	60	80
コスト($/lb)	7.3	6.9	7.3	7.5	7.6	6.0	5.8	4.3	4.1

所望の組成を持つ合金をコストを一番安く生成するには，市販の合金をどのように混ぜ合わせれば良いでしょうか．

この問題を NUOPT で解くために定式化を行います．本例題は文献[1]からの引用です．

まず，変数として市販の合金 1,2,3,…,9 の混合比率，つまり混ぜ合わせる割合を，それぞれ $x_1, x_2, x_3, \ldots, x_9$ としましょう．

次に，最小化するべき目的関数は，各市販の合金について「単位量当たりのコスト」と「混合比率」の積の総和として表現することができます．

最後に制約条件です．まず，混合比率は負の値をとれませんので，各変数に対して非負制約が必要です．混合比率の総和は 1 ですので，その制約も加えます．また，生成する合金の組成についての制約は，鉛，亜鉛，スズに対して，各市販の合金についての「構成比率」と「混合比率」の積の総和が，それぞれ 30%，30%，40%と等しい，という形になります．

以上のことから，次のように定式化することができます.

変数

x_1 **市販の合金１の混合比率**
x_2 **市販の合金２の混合比率**
x_3 **市販の合金３の混合比率**
x_4 **市販の合金４の混合比率**
x_5 **市販の合金５の混合比率**
x_6 **市販の合金６の混合比率**
x_7 **市販の合金７の混合比率**
x_8 **市販の合金８の混合比率**
x_9 **市販の合金９の混合比率**

目的関数(最小化) **総コスト**

$$7.3x_1+6.9x_2+7.3x_3+7.5x_4+7.6x_5+6.0x_6+5.8x_7+4.3x_8+4.1x_9$$

非負制約

$x_1 \geq 0$
$x_2 \geq 0$
$x_3 \geq 0$
$x_4 \geq 0$
$x_5 \geq 0$
$x_6 \geq 0$
$x_7 \geq 0$
$x_8 \geq 0$
$x_9 \geq 0$

混合比率の制約

$$x_1+x_2+x_3+x_4+x_5+x_6+x_7+x_8+x_9=1$$
$$0.2x_1+0.5x_2+0.3x_3+0.3x_4+0.3x_5+0.6x_6+0.4x_7+0.1x_8+0.1x_9=0.3$$
$$0.3x_1+0.4x_2+0.2x_3+0.4x_4+0.3x_5+0.3x_6+0.5x_7+0.3x_8+0.1x_9=0.3$$
$$0.5x_1+0.1x_2+0.5x_3+0.3x_4+0.4x_5+0.1x_6+0.1x_7+0.6x_8+0.8x_9=0.4$$

この問題は，目的関数，制約式全て線形なので，線形計画問題となります.

定式化した結果を SIMPLE で記述すると以下のようになります.

```
// 変数
Variable x1(name="市販の合金１の混合比率");
Variable x2(name="市販の合金２の混合比率");
Variable x3(name="市販の合金３の混合比率");
Variable x4(name="市販の合金４の混合比率");
Variable x5(name="市販の合金５の混合比率");
Variable x6(name="市販の合金６の混合比率");
Variable x7(name="市販の合金７の混合比率");
Variable x8(name="市販の合金８の混合比率");
Variable x9(name="市販の合金９の混合比率");
// 目的関数
Objective z(name="総コスト");
z=7.3*x1+6.9*x2+7.3*x3+7.5*x4+7.6*x5+6.0*x6+5.8*x7+4.3*x8+4.1*x9;
// 非負制約
x1 >= 0;
x2 >= 0;
x3 >= 0;
x4 >= 0;
x5 >= 0;
x6 >= 0;
x7 >= 0;
x8 >= 0;
x9 >= 0;
// 混合比率の制約
x1+x2+x3+x4+x5+x6+x7+x8+x9 == 1;
0.2*x1+0.5*x2+0.3*x3+0.3*x4+0.3*x5+0.6*x6+0.4*x7+0.1*x8+0.1*x9==0.3;
0.3*x1+0.4*x2+0.2*x3+0.4*x4+0.3*x5+0.3*x6+0.5*x7+0.3*x8+0.1*x9==0.3;
0.5*x1+0.1*x2+0.5*x3+0.3*x4+0.4*x5+0.1*x6+0.1*x7+0.6*x8+0.8*x9==0.4;
// 求解
solve();
// 出力
z.val.print();
```

より汎用的に問題を定式化すると以下のようになります.

集合	$Alloy = \{1,2,3,4,5,6,7,8,9\}$	**市販の合金**
	$Blend = \{Lead, Zinc, Tin\}$	**構成金属**
定数	$r_{ij}, i \in Alloy,\ j \in Blend$	**市販の合金 i における構成金属 j の比率**
	$c_i, i \in Alloy$	**市販の合金 i の単位量あたりのコスト**
	$b_j,\ j \in Blend$	**構成金属 j の目標比率**
変数	$x_i, i \in Alloy$	**市販の合金 i の混合比率**
目的関数(最小化)	$\sum_{i \in Alloy} c_i x_i$	**総コスト**
制約	$x_i \geq 0, i \in Alloy$	**非負制約**
	$\sum_{i \in Alloy} x_i = 1$	
	$\sum_{i \in Alloy} r_{ij} x_i = b_j,\ j \in Blend$	**混合比の制約**

次に，定数（構成比率，コスト，目標比率）をデータファイルから与える SIMPLE モデルを示します．このようにモデルとデータを分離することにより，市販の合金の数や構成金属の種類の数が変わったとしてもデータファイルを変更するだけで対応できるようになります．

```
// 集合と添字
Set Alloy(name="市販の合金集合");
Element i(set=Alloy);
Set Blend(name="構成金属集合");
Element j(set=Blend);

// パラメータ
Parameter r(name="構成比率", index=(i,j));
Parameter c(name="コスト", index=i);
Parameter b(name="目標比率", index=j);

// 変数
Variable x(name="混合比率", index=i);

// 目的関数
Objective z(name="総コスト");
z = sum(c[i]*x[i],i);

// 非負制約
x[i] >= 0;

// 混合比の制約
sum(x[i],i) == 1;
sum(r[i,j]*x[i],i) == b[j];

// 求解
solve();

// 出力
z.val.print();
x.val.print();
```

データファイル（.dat 形式）は以下のようになります.

```
構成比率 =
[1,Lead] 0.2 [1,Zinc] 0.3 [1, Tin] 0.5
[2,Lead] 0.5 [2,Zinc] 0.4 [2, Tin] 0.1
[3,Lead] 0.3 [3,Zinc] 0.2 [3, Tin] 0.5
[4,Lead] 0.3 [4,Zinc] 0.4 [4, Tin] 0.3
[5,Lead] 0.3 [5,Zinc] 0.3 [5, Tin] 0.4
[6,Lead] 0.6 [6,Zinc] 0.3 [6, Tin] 0.1
[7,Lead] 0.4 [7,Zinc] 0.5 [7, Tin] 0.1
[8,Lead] 0.1 [8,Zinc] 0.3 [8, Tin] 0.6
[9,Lead] 0.1 [9,Zinc] 0.1 [9, Tin] 0.8
;

コスト =
[1] 7.3
[2] 6.9
[3] 7.3
[4] 7.5
[5] 7.6
[6] 6.0
[7] 5.8
[8] 4.3
[9] 4.1
;

目標比率 =
[Lead] 0.3
[Zinc] 0.3
[ Tin] 0.4
;
```

このモデルを実行すると，市販の合金 6 を 40%，市販の合金 8 を 60%混ぜ合わせるのが最適で，そのときの総コストは 4.98 であることがわかります.

5.2 輸送問題

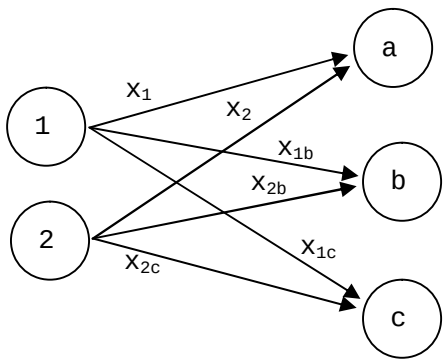
輸送問題は複数の供給地から複数の需要地への物の流れ方を決める問題ということができます．供給地と需要地をノード，物の流れをアークとすれば，輸送問題はネットワークによって表現できる問題の一種と捉えることができます. 輸送問題に対する定式化の方法は他のネットワークによって表現できる問題にも応用できます.

（例題）
ある配送業者は二つの工場 1，2 から三つの店舗 a，b，c への製品の輸送を請け負っているとします．各工場，店舗について，それぞれ供給可能量と需要量が決められており，それらを満たしつつ, 最もコストがかからない製品の運び方を決定することが目的です．各工場の供給可能量，各店舗の需要量，輸送コストは以下の通りです.

工場	供給可能量	店舗	需要量
1	250	a	200
2	450	b	200
		c	200

輸送コスト			
	a	b	c
1	3.4	2.2	2.9
2	3.4	2.4	2.5

この問題を NUOPT で解くために定式化を行います．本例題は文献[1]からの引用です．
まず，変数として各工場から各店舗への輸送量を以下の図のように設定します.



次に，目的関数は，工場から店舗への各経路について「単位量あたりの輸送コスト」と「輸送

量」の積の総和として表現することができます.

最後に制約条件です．輸送量は負にはなり得ないので，各変数に対して非負制約が必要です．さらに，各工場について工場からの輸送量の和が供給可能量以下である，各店舗について店舗への輸送量の和が需要量と等しい，という制約が加わります.

以上のことから，次のように定式化することができます.

変数	x_{1a}	工場１から店舗ａ への輸送量
	x_{1b}	工場１から店舗ｂ への輸送量
	x_{1c}	工場１から店舗ｃ への輸送量
	x_{2a}	工場２から店舗ａ への輸送量
	x_{2b}	工場２から店舗ｂ への輸送量
	x_{2c}	工場２から店舗ｃ への輸送量

目的関数(最小化) 総コスト

$$3.4x_{1a} + 2.2x_{1b} + 2.9x_{1c} + 3.4x_{2a} + 2.4x_{2b} + 2.5x_{2c}$$

非負制約	$x_{1a} \geq 0$	
	$x_{1b} \geq 0$	
	$x_{1c} \geq 0$	
	$x_{2a} \geq 0$	
	$x_{2b} \geq 0$	
	$x_{2c} \geq 0$	
工場の生産量の制約	$x_{1a} + x_{1b} + x_{1c} \leq 250$	工場１について
	$x_{2a} + x_{2b} + x_{2c} \leq 450$	工場２について
店舗の需要量の制約	$x_{1a} + x_{2a} = 200$	店舗ａについて
	$x_{1b} + x_{2b} = 200$	店舗ｂについて
	$x_{1c} + x_{2c} = 200$	店舗ｃについて

ネットワークで表現できる問題の多くは, 上で見てきたように各アークに対して変数を定義し, 各ノードについて制約をたてることによって定式化されます.

定式化した結果を SIMPLE で記述すると以下のようになります.

```
// 変数
Variable x1a(name="工場１から店舗ａへの輸送量");
Variable x1b(name="工場１から店舗ｂへの輸送量");
Variable x1c(name="工場１から店舗ｃへの輸送量");
Variable x2a(name="工場２から店舗ａへの輸送量");
Variable x2b(name="工場２から店舗ｂへの輸送量");
Variable x2c(name="工場２から店舗ｃへの輸送量");

// 非負制約
x1a >= 0;
x1b >= 0;
x1c >= 0;
x2a >= 0;
x2b >= 0;
x2c >= 0;

// 工場の生産量の制約
x1a + x1b + x1c <= 250;
x2a + x2b + x2c <= 450;

// 店舗の需要量の制約
x1a + x2a == 200;
x1b + x2b == 200;
x1c + x2c == 200;

// 目的関数
Objective z(name="総コスト", type=minimize);
z = 3.4*x1a + 2.2*x1b + 2.9*x1c + 3.4*x2a + 2.4*x2b + 2.5*x2c;

// 求解
solve();

// 出力
z.val.print();
```

より汎用的に問題を定式化すると以下のようになります.

集合	$Cannery = \{1, 2\}$	**工場**
	$Warehouse = \{a, b, c\}$	**店舗**
定数	$upper_i, i \in Cannery$	**工場** i **の供給可能量**
	$demand_j, j \in Warehouse$	**店舗** j **の需要量**
	$c_{ij}, i \in Cannery,\ j \in Warehouse$	**工場** i **から店舗** j **への単位量あたりの輸送コスト**
変数	$x_{ij}, i \in Cannery,\ j \in Warehouse$	**工場** i **から店舗** j **への輸送量**
目的関数（最小化）	$\sum_{i \in Cannery} \sum_{j \in Warehouse} c_{ij} x_{ij}$	**総コスト**
制約	$x_{ij} \geq 0$	**非負制約**
	$\sum_{j \in Warehouse} x_{ij} \leq upper_i$	**工場の生産量の制約**
	$\sum_{i \in Cannery} x_{ij} = demand_j$	**店舗の需要量の制約**

次に，入力データとモデルを分離した SIMPLE モデルを示します．

```
// 集合と添字
Set Cannery(name="工場");
Element i(set=Cannery);
Set Warehouse(name="店舗");
Element j(set=Warehouse);

// パラメータ
Parameter upper(name="供給可能量", index=i);
Parameter demand(name="需要量", index=j);
Parameter cost(name="輸送コスト", index=(i,j));

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

// 目的関数
Objective z(name="総コスト", type=minimize);
z = sum(cost[i,j]*x[i,j],(i,j));

// 非負制約
x[i,j] >= 0;

// 工場の生産量の制約
sum(x[i,j],j) <= upper[i];

// 店舗の需要量の制約
sum(x[i,j],i) == demand[j];

// 求解
solve();

// 出力
z.val.print();
x.val.print();
```

データファイル（.dat 形式）は以下のようになります.

```
供給可能量 =
[1] 250
[2] 450
;

需要量 =
[a] 200
[b] 200
[c] 200
;

輸送コスト =
[1,a] 3.4
[1,b] 2.2
[1,c] 2.9
[2,a] 3.4
[2,b] 2.4
[2,c] 2.5
;
```

このモデルを実行すると，以下のような結果が得られます.

```
総コスト=1620
輸送量[1,"a"]=26.4602
輸送量[1,"b"]=200
輸送量[1,"c"]=1.79049e-008
輸送量[2,"a"]=173.54
輸送量[2,"b"]=3.58099e-008
輸送量[2,"c"]=200
```

5.3 多期間計画問題

多期間計画問題とは，多期間にわたり各期間ごとに意思決定をする問題のことをいいます．多期間計画問題を定式化する場合は，期間ごとに変数を定義するのが一般的です．

（例題）

2 種類の原料 A, B を加工して 2 種類の製品 I, II を生産している工場が，向こう 3 ヶ月間の生産計画を立てようとしています．各製品を 1 単位生産するために必要な原料の使用量，各製品の生産／在庫コスト，各製品の月ごとの出荷量，各原料の月ごとの利用可能量は以下のように与えられています．

原料使用量		
	I	II
A	2	7
B	5	3

生産／在庫コスト		
	I	II
生産	75	50
在庫	8	7

製品の出荷量		
	I	II
1	30	20
2	60	50
3	80	90

原料の利用可能量		
	A	B
1	920	790
2	750	600
3	500	480

各月に出荷する製品をその月中に全て生産できるとは限らないので，前の月に生産した製品を在庫として保管して来月に出荷することも考えられます．このような状況の下で，要求された製品の出荷量と与えられた原料の利用可能量の制約を満たしつつ総コストを最小にするには，各月における各製品の生産量と在庫量をどのように決定すればよいでしょうか．

この問題を NUOPT で解くために定式化を行います．本例題は文献[2]からの引用です．

まず，変数として各月における製品 I，II の生産量をそれぞれ $x_{I1}, x_{II1}, x_{I2}, x_{II2}, x_{I3}, x_{II3}$ とし，在庫量をそれぞれ $y_{I1}, y_{II1}, y_{I2}, y_{II2}$ としましょう．

次に，最小化するべき目的関数は，各生産量／在庫量とその単位量あたりのコストとの積の総和として表現することができます．

最後に制約条件です．まず，各変数に対して非負制約が必要です．次に，1 ヶ月に利用できる原料が決められているので，各月ごとに各原料に関する制約が必要です．さらに，各製品に対して在庫量は次の月に持ち越せますので，それを踏まえた出荷量の制約を加えます．

以上のことから，次のように定式化することができます．

変数	$x_{\mathrm{I}1}$	**製品Ｉの１ヶ月目の生産量**
	$x_{\mathrm{II}1}$	**製品IIの１ヶ月目の生産量**
	$x_{\mathrm{I}2}$	**製品Ｉの２ヶ月目の生産量**
	$x_{\mathrm{II}2}$	**製品IIの２ヶ月目の生産量**
	$x_{\mathrm{I}3}$	**製品Ｉの３ヶ月目の生産量**
	$x_{\mathrm{II}3}$	**製品IIの３ヶ月目の生産量**
	$y_{\mathrm{I}1}$	**製品Ｉの１ヶ月目の在庫量**
	$y_{\mathrm{II}1}$	**製品IIの１ヶ月目の在庫量**
	$y_{\mathrm{I}2}$	**製品Ｉの２ヶ月目の在庫量**
	$y_{\mathrm{II}2}$	**製品IIの２ヶ月目の在庫量**

目的関数(最小化) **総コスト**

$$75x_{\mathrm{I}1}+50x_{\mathrm{II}1}+8y_{\mathrm{I}1}+7y_{\mathrm{II}1}+75x_{\mathrm{I}2}+50x_{\mathrm{II}2}+8y_{\mathrm{I}2}+7y_{\mathrm{II}2}+75x_{\mathrm{I}3}+50x_{\mathrm{II}3}$$

非負制約	$x_{\mathrm{I}1}\geq 0$，$x_{\mathrm{II}1}\geq 0$	
	$x_{\mathrm{I}2}\geq 0$，$x_{\mathrm{II}2}\geq 0$	
	$x_{\mathrm{I}3}\geq 0$，$x_{\mathrm{II}3}\geq 0$	
	$y_{\mathrm{I}1}\geq 0$，$y_{\mathrm{II}1}\geq 0$	
	$y_{\mathrm{I}2}\geq 0$，$y_{\mathrm{II}2}\geq 0$	
原料の制約	$2x_{\mathrm{I}1}+7x_{\mathrm{II}1}\leq 920$	**原料Ａについて（１ヶ月目）**
	$5x_{\mathrm{I}1}+3x_{\mathrm{II}1}\leq 790$	**原料Ｂについて（１ヶ月目）**
	$2x_{\mathrm{I}2}+7x_{\mathrm{II}2}\leq 750$	**原料Ａについて（２ヶ月目）**
	$5x_{\mathrm{I}2}+3x_{\mathrm{II}2}\leq 600$	**原料Ｂについて（２ヶ月目）**
	$2x_{\mathrm{I}3}+7x_{\mathrm{II}3}\leq 500$	**原料Ａについて（３ヶ月目）**
	$5x_{\mathrm{I}3}+3x_{\mathrm{II}3}\leq 480$	**原料Ｂについて（３ヶ月目）**
出荷量の制約	$x_{\mathrm{I}1}-y_{\mathrm{I}1}=30$	**製品Ｉについて（１ヶ月目）**
	$x_{\mathrm{II}1}-y_{\mathrm{II}1}=20$	**製品IIについて（１ヶ月目）**
	$x_{\mathrm{I}2}+y_{\mathrm{I}1}-y_{\mathrm{I}2}=60$	**製品Ｉについて（２ヶ月目）**
	$x_{\mathrm{II}2}+y_{\mathrm{II}1}-y_{\mathrm{II}2}=50$	**製品IIについて（２ヶ月目）**
	$x_{\mathrm{I}3}+y_{\mathrm{I}2}=80$	**製品Ｉについて（３ヶ月目）**
	$x_{\mathrm{II}3}+y_{\mathrm{II}2}=90$	**製品IIについて（３ヶ月目）**

問題を SIMPLE で記述すると以下のようになります.

```
// 変数
Variable x11(name="製品Ｉの１ヶ月目の生産量");
Variable x21(name="製品IIの１ヶ月目の生産量");
Variable x12(name="製品Ｉの２ヶ月目の生産量");
Variable x22(name="製品IIの２ヶ月目の生産量");
Variable x13(name="製品Ｉの３ヶ月目の生産量");
Variable x23(name="製品IIの３ヶ月目の生産量");
Variable y11(name="製品Ｉの１ヶ月目の在庫量");
Variable y21(name="製品IIの１ヶ月目の在庫量");
Variable y12(name="製品Ｉの２ヶ月目の在庫量");
Variable y22(name="製品IIの２ヶ月目の在庫量");
// 目的関数
Objective z(name="総コスト", type=minimize);
z=75*x11+50*x21+8*y11+7*y21+75*x12+50*x22+8*y12+7*y22+75*x13+50*x23;
// 非負制約
x11 >= 0; x21 >= 0;
x12 >= 0; x22 >= 0;
x13 >= 0; x23 >= 0;
y11 >= 0; y21 >= 0;
y12 >= 0; y22 >= 0;
// 原料の制約
2*x11 + 7*x21 <= 920;
5*x11 + 3*x21 <= 790;
2*x12 + 7*x22 <= 750;
5*x12 + 3*x22 <= 600;
2*x13 + 7*x23 <= 500;
5*x13 + 3*x23 <= 480;
// 出荷量の制約
x11      - y11 == 30;
x21      - y21 == 20;
x12 + y11 - y12 == 60;
x22 + y21 - y22 == 50;
x13 + y12      == 80;
x23 + y22      == 90;
// 求解
solve();
// 出力
z.val.print();
```

より汎用的に問題を定式化すると以下のようになります.

集合	$Product = \{\mathrm{I}, \mathrm{II}\}$	**製品**
	$Material = \{A, B\}$	**原料**
	$Period = \{1, 2, 3\}$	**期間**
定数	$use_{ij}, i \in Product,\ j \in Material$	**製品 i を生産するのに必要な原料 j の使用量**
	$out_{it}, i \in Prodcut, t \in Period$	**t ヶ月目の製品 i の出荷量**
	$upper_{jt},\ j \in Material, t \in Period$	**t ヶ月目の原料 j の取扱可能量**
	$costp_i, i \in Product$	**製品 i の生産コスト**
	$costi_i, i \in Product$	**製品 i の在庫コスト**
変数	$x_{it}, i \in Product, t \in Period$	**生産量**
	$y_{it}, i \in Product, t \in Period$	**在庫量**
目的関数 (最小化)	$\sum_{t \in Period} \sum_{i \in Product} costp_{it} x_{it} + costi_{it} y_{it}$	**総コスト**
制約	$x_{it} \geq 0$	**生産量の非負制約**
	$y_{it} \geq 0$	**在庫量の非負制約**
	$\sum_{i \in Product} use_{ij} x_{it} \leq upper_{jt}$	**原料の使用量の制約**
	$x_{i1} - y_{i1} = out_{i1}$	**１ヶ月目の出荷量について**
	$x_{i2} + y_{i1} - y_{i2} = out_{i2}$	**２ヶ月目の出荷量について**
	$x_{i3} + y_{i2} = out_{i3}$	**３ヶ月目の出荷量について**

次に，定数をデータファイルから与える場合のモデルを示します．

```
// 集合と添字
Set Product(name="製品");
Element i(set=Product);
Set Material(name="原料");
Element j(set=Material);
OrderedSet Period(name="期間");
Element t(set=Period);
// パラメータ
Parameter use(name="原料使用量", index=(i,j));
Parameter out(name="出荷量", index=(i,t));
Parameter upper(name="取扱可能量", index=(j,t));
Parameter costp(name="生産コスト", index=i);
Parameter costi(name="在庫コスト", index=i);
// 変数
Variable x(name="生産量", index=(i,t));
Variable y(name="在庫量", index=(i,t));
// 目的関数
Objective z(name="総コスト", type=minimize);
z = sum(sum(costp[i]*x[i,t]+ costi[i]*y[i,t],i),t);
// 非負制約
x[i,t] >= 0;
y[i,t] >= 0;
// 原料の制約
sum(use[i,j]*x[i,t],i) <= upper[j,t];
// 在庫量の制約
x[i,t]          - y[i,t] == out[i,t], t == 1;
x[i,t] + y[i,t-1] - y[i,t] == out[i,t], t == 2;
x[i,t] + y[i,t-1]        == out[i,t], t == 3;
// 求解
solve();
// 出力
z.val.print();
x.val.print();
y.val.print();
```

データファイル（.dat 形式）は以下のようになります.

```
原料使用量 =
["I", "A"] 2
["I", "B"] 5
["II", "A"] 7
["II", "B"] 3
;

出荷量 =
["I", 1] 30
["I", 2] 60
["I", 3] 80
["II", 1] 20
["II", 2] 50
["II", 3] 90
;

取扱可能量 =
["A", 1] 920
["A", 2] 750
["A", 3] 500
["B", 1] 790
["B", 2] 600
["B", 3] 480
;

生産コスト =
["I"] 75
["II"] 50
;

在庫コスト =
["I"] 8
["II"] 7
;
```

このモデルを実行すると，総コスト 21199.2 である最適解を得ます.

5.4 ナップサック問題

ナップサック問題は, ナップサックの中にいくつかの品物を詰め込み入れた品物の総価値を最大にするという問題です．ただし，ナップサックと品物にはそれぞれ容量が与えられていて，入れた品物の総容量がナップサックの容量を超えてはいけないという条件があります. この問題は, 組合せ最適化問題の代表的な例の一つとしてよく知られていて, プロジェクトの選択や物資の購入などの問題に応用されています.

（例題）
容量 65 のナップサックに次の表にある品物を詰め込むことにします．この時，詰め込んだ品物の総価値を最大にするためには何をいくつ詰め込むと良いでしょうか．ただし，同じ品物を何個詰め込んでも良いものとします.

品物	１個あたりの価値	１個あたりの容量
缶コーヒー	１２０	１０
水入りペットボトル	１３０	１２
バナナ	８０	7
りんご	１００	9
おにぎり	２５０	２１
パン	１８５	１６

まず，変数は各品物を詰め込む個数です．よって，この変数は整数値しか取らないということになります. ただし,「-1 個詰め込む」というようなありえない答えを排除する必要があります．このため，各変数は 0 以上の値しか取らないということを制約条件として明示しておく必要があります．なお,「0 個詰め込む」は「その品物を詰め込まない」と解釈します.

次に，最大化することになる目的関数は詰め込んだものの総価値です．これは，各品物について「1 個あたりの価値」と「その品物を詰め込んだ個数」の積を求め，その総和を取ることで表現できます.

制約条件は, 先ほど述べた変数に関するものの他に, 詰め込んだ品物の総容量がナップサックの容量を超えないというものがあります．目的関数の時と同様に考えると，各品物に関する「1 個あたりの容量」と「その品物を詰め込んだ個数」の積の総和をとると詰め込んだ品物の総容量が得られます．よって，この総和がナップサックの容量である 65 を超えないということを式で表せばよいことになります.

以上のことから，この例題は次のように定式化することが出来ました.

整数変数	$coffee$	**缶コーヒーの個数**
	$water$	**水入りペットボトルの個数**
	$banana$	**バナナの個数**
	$apple$	**りんごの個数**
	$rice\_ball$	**おにぎりの個数**
	$bread$	**パンの個数**
目的関数（最大化）	$120coffee+130water+80banana+100apple+250rice\_ball+185bread$	**総価値を最大化する**
制約条件	$10coffee+12water+7banana+9apple+21rice\_ball+16bread \leq 65$	**容量に関する制約**
	$coffee \geq 0$	**缶コーヒーは 0 個以上詰め込む**
	$water \geq 0$	**水入りペットボトルは 0 個以上詰め込む**
	$banana \geq 0$	**バナナは 0 個以上詰め込む**
	$apple \geq 0$	**りんごは 0 個以上詰め込む**
	$rice\_ball \geq 0$	**おにぎりは 0 個以上詰め込む**
	$bread \geq 0$	**パンは 0 個以上詰め込む**

定式化した結果を SIMPLE で記述すると次のようになります.

```
//整数変数を宣言する
IntegerVariable coffee(name= "缶コーヒーの個数");
IntegerVariable water(name="水入りペットボトルの個数");
IntegerVariable banana(name="バナナの個数");
IntegerVariable apple(name="りんごの個数");
IntegerVariable rice_ball(name="おにぎりの個数");
IntegerVariable bread(name="パンの個数");

//総価値を最大化する
Objective total_value(name="総価値",type=maximize);
total_value=120*coffee+130*water+80*banana+100*apple+250*rice_ball+185*bread;

//容量に関する制約
10*coffee+12*water+7*banana+9*apple+21*rice_ball+16*bread<=65;

//各品物は 0 個以上詰め込む
coffee>=0;
water>=0;
banana>=0;
apple>=0;
rice_ball>=0;
bread>=0;

//求解し結果を出力する
solve();
coffee.val.print();
water.val.print();
banana.val.print();
apple.val.print();
rice_ball.val.print();
bread.val.print();
total_value.val.print();
```

このモデルを NUOPT で実行すると，最後に

```
缶コーヒーの個数=3
水入りペットボトルの個数=0
バナナの個数=2
りんごの個数=0
おにぎりの個数=1
パンの個数=0
総価値=770
```

という表示がされます．そして，この表示から「缶コーヒーを 3 個，バナナを 2 個，そしておにぎりを 1 個詰め込むと良い」というこの例題の答えを確認できます．

ところで，このモデルについて品物の種類などを変更したい場合 SIMPLE での記述を修正する箇所が多く大変な手間がかかってしまいます．この対策として，ナップサックの容量，品物の価値および品物の容量を別に用意した dat ファイルから与えることにします．このようにすることで，SIMPLE での記述が汎用的なものになり，品物の種類が変わったとしても dat ファイルの変更のみで対応できるようになります．そのために，ここでは「品物の集合」という概念を導入します．すると定式化は次のように書き直すことができます．

集合	$Object = \{$缶コーヒー,水入りペットボトル,バナナ,りんご,おにぎり,パン$\}$	品物の集合
整数変数	$count_i \quad , i \in Object$	品物 i を詰め込む個数
定数	$capacity$	ナップサックの容量
	$value_i \quad , i \in Object$	品物 i の 1 個あたりの価値
	$weight_i \quad , i \in Object$	品物 i の 1 個あたりの容量
目的関数（最大化）	$\sum_{i \in Object} value_i \cdot count_i$	総価値を最大化する
制約条件	$\sum_{i \in Object} weight_i \cdot count_i \le capacity$	容量に関する制約
	$count \ge 0, i \in Object$	各品物は 0 個以上詰め込む

この定式化を SIMPLE で記述すると，以下のような簡潔なものになります．なお，品物の集合の具体的な要素については NUOPT では dat ファイルから自動的に認識します．

```
//品物の集合を宣言する
Set Object;
Element object(set=Object);

//データファイルから与えるパラメータを宣言する
Parameter capacity(name="ナップサックの容量");
Parameter value(name="品物の価値",index=object);
Parameter weight(name="品物の容量",index=object);

//整数変数を宣言する
IntegerVariable count(name="詰め込む個数",index=object);

//総価値の最大化
Objective total_value(name="総価値",type=maximize);
total_value=sum(value[object]*count[object],object);

//容量に関する制約
sum(weight[object]*count[object],object)<=capacity;

//各品物は 0 個以上詰め込む
count[object]>=0;

//求解し結果を出力する
solve();
count.val.print();
total_value.val.print();
```

なお，今回の例題についての dat ファイルは次のようになります．

```
ナップサックの容量=65;
品物の価値=
[缶コーヒー]120
[水入りペットボトル]130
[バナナ]80
[りんご]100
[おにぎり]250
[パン]185;
品物の容量=
[缶コーヒー]10
[水入りペットボトル]12
[バナナ]7
[りんご]9
[おにぎり]21
[パン]16;
```

最後に，この例題では「缶コーヒーが 3 個」というように容量が許す限り同じ品物を何個でも詰め込むことができました．それでは，各品物についてナップサックに詰め込むことができるのは 1 個だけということにするとどのようになるでしょうか．なお，この制限を加えると 0-1 ナップサック問題という典型的な 0-1 整数計画問題になります．SIMPLE では，0-1 変数であるという宣言を簡単に行うことができます．具体的には，先ほど汎用化させた SIMPLE モデル中で変数を宣言している部分を

```
IntegerVariable count(name="詰め込む個数",index=object,type=binary);
```

とするだけです．さらに，この宣言から各変数がとりうる値は 0 か 1 しかないということが明らかなため，各品物は 0 個以上詰め込むという制約「count[object]>=0;」を記述する必要がなくなります．以上の点についてモデルファイルを書き換えた上で，NUOPT で実行させると，

```
詰め込む個数["おにぎり"]=1
詰め込む個数["りんご"]=1
詰め込む個数["バナナ"]=1
詰め込む個数["パン"]=1
詰め込む個数["缶コーヒー"]=0
詰め込む個数["水入りペットボトル"]=1
総価値=745
```

という結果が得られ，缶コーヒー以外の品物を詰め込むと良いという事がわかります．

5.5 集合被覆問題

集合 U とその部分集合の族および各部分集合に対応するコストが与えられているものとします．この時，全ての U の要素をカバーするように部分集合の族から部分集合を選び，その際にかかるコストを最小にするという問題が集合被覆問題です. 応用例には乗務員スケジューリング問題などがあります．なお，集合被覆問題の場合 U の各要素について複数個の部分集合でカバーすることを許しています．このことに関連して，複数個の部分集合でカバーすることを許さない場合，集合分割問題と呼ばれ，選挙区の設定問題などに応用されています．

（例題）
ある企業は A，B，C，D，E，F，G の 7 つのエリアがある都市で宅配便の配達事業を始めるため，配達員を採用することになりました．この都市には配達員の候補は 10 人いて，各人が配達できるエリアおよび配達を依頼した際にかかるコストは次の表のようになっています．

候補者	配達可能エリア	コスト
佐藤	A，B，C	200
鈴木	A，D，F	280
高橋	B，E	175
田中	C，D，E，F，G	560
渡辺	A，F	205
伊藤	B，D，F	245
山本	D	80
中村	C，G	195
小林	C，F，G	265
斉藤	B，E，G	190

この時，最も少ないコストで 7 つのエリアすべてに配達するためには誰を採用すると良いでしょうか．ただし，配達可能エリアの一部のみを依頼する（例：伊藤に B と D のみ依頼する）ことはできないものとします．

この例題の変数は，各候補者について「採用する」もしくは「採用しない」という状態を表現できるものとなります．よって，ここでは各候補者に対し採用する場合は 1，採用しない場合は 0 を取るような変数 $x_{(候補者)}$ を導入します．

この時，目的関数はどのように表すことができるでしょうか．まず，佐藤を例に実際にかかる

コストを表現します．佐藤を採用した場合（$x_{佐藤}=1$の場合）には200のコストがかかります．

一方，採用しなかった場合（$x_{佐藤}=0$の場合）は佐藤については何もコストがかかりません．言い換えると，かかったコストが0であるということになります．よって，先ほど導入した変数を用い2つの場合をまとめると，佐藤については実際には$200x_{佐藤}$のコストがかかったと表現できます．他の候補者に対しても同様に実際のコストを表し，その総和を取ると実際にかかった総コストとなります．この総コストが，最小化することになる目的関数です．

制約条件については，「各エリアに１人以上配置されている」という条件を式で表現することになります．例えば，エリアAの場合佐藤・鈴木・渡辺の中から１人以上採用することで条件を満たすことができます．このことを式で表すと$x_{佐藤}+x_{鈴木}+x_{渡辺}\geq 1$となります．他のエリアについても同様に考えることで制約条件を表現できます．

以上のことから，この例題は0-1整数計画問題として次のように定式化できます．

変数	$x_{佐藤}$，$x_{鈴木}$，$x_{高橋}$，$x_{田中}$，$x_{渡辺}$，$x_{伊藤}$，$x_{山本}$，$x_{中村}$，$x_{小林}$，$x_{斉藤}$	各候補者について採用する時1，採用しない時0を取る変数
目的関数（最小化）	$200x_{佐藤}+280x_{鈴木}+175x_{高橋}+560x_{田中}+205x_{渡辺}+245x_{伊藤}+80x_{山本}+195x_{中村}+265x_{小林}+190x_{斉藤}$	総コストの最小化
制約	$x_{佐藤}+x_{鈴木}+x_{渡辺}\geq 1$	エリアAで配達可能にする
	$x_{佐藤}+x_{高橋}+x_{伊藤}+x_{斉藤}\geq 1$	エリアBで配達可能にする
	$x_{佐藤}+x_{田中}+x_{中村}+x_{小林}\geq 1$	エリアCで配達可能にする
	$x_{鈴木}+x_{田中}+x_{伊藤}+x_{山本}\geq 1$	エリアDで配達可能にする
	$x_{高橋}+x_{田中}+x_{斉藤}\geq 1$	エリアEで配達可能にする
	$x_{鈴木}+x_{田中}+x_{渡辺}+x_{伊藤}+x_{小林}\geq 1$	エリアFで配達可能にする
	$x_{田中}+x_{中村}+x_{小林}+x_{斉藤}\geq 1$	エリアGで配達可能にする

この定式化した結果についてそのまま SIMPLE で記述すると次のようになります.

```
//変数の宣言
IntegerVariable x_sato(name="佐藤",type=binary);
IntegerVariable x_suzuki(name="鈴木",type=binary);
IntegerVariable x_takahashi(name="高橋",type=binary);
IntegerVariable x_tanaka(name="田中",type=binary);
IntegerVariable x_watanabe(name="渡辺",type=binary);
IntegerVariable x_ito(name="伊藤",type=binary);
IntegerVariable x_yamamoto(name="山本",type=binary);
IntegerVariable x_nakamura(name="中村",type=binary);
IntegerVariable x_kobayashi(name="小林",type=binary);
IntegerVariable x_saito(name="斉藤",type=binary);
//目的関数の設定
Objective total_cost(type=minimize);
total_cost=200*x_sato+280*x_suzuki+175*x_takahashi+560*x_tanaka+
205*x_watanabe+245*x_ito+80*x_yamamoto+195*x_nakamura+
265*x_kobayashi+190*x_saito;
//各エリアに１人以上配置する
x_sato+x_suzuki+x_watanabe>=1;
x_sato+x_takahashi+x_ito+x_saito>=1;
x_sato+x_tanaka+x_nakamura+x_kobayashi>=1;
x_suzuki+x_tanaka+x_ito+x_yamamoto>=1;
x_takahashi+x_tanaka+x_saito>=1;
x_suzuki+x_tanaka+x_watanabe+x_ito+x_kobayashi>=1;
x_tanaka+x_nakamura+x_kobayashi+x_saito>=1;
//求解し解を出力する
solve();
x_sato.val.print();
x_suzuki.val.print();
x_takahashi.val.print();
x_tanaka.val.print();
x_watanabe.val.print();
x_ito.val.print();
x_yamamoto.val.print();
x_nakamura.val.print();
x_kobayashi.val.print();
x_saito.val.print();
total_cost.val.print();
```

この記述を見て分かるように，このままですと修正が大変ですし汎用的でもありません．このため，汎用的になるように定式化した結果を見直すことにします．

前節のナップサック問題の時と同様に「候補者の集合」および「エリアの集合」という概念を導入します．また，各候補者のコストは定数として外部から与えることにします．

さらに，制約条件についても見直します．例えば，エリア A に 1 人以上配置されているという制約条件 $x_{佐藤} + x_{鈴木} + x_{渡辺} \geq 1$ については，ほかの候補者も考慮すると

$$1x_{佐藤} + 1x_{鈴木} + 0x_{高橋} + 0x_{田中} + 1x_{渡辺} + 0x_{伊藤} + 0x_{山本} + 0x_{中村} + 0x_{小林} + 0x_{斉藤} \geq 1$$

と記述できます．ここで，この式の各変数についている係数 0 または 1 を外部から定数として与えることにします．また，他の地域についても同様に定数を導入します．すると，各エリアに対する制約条件は全て同じ枠組みで定式化できます．その結果，SIMPLE では一般的な形での記述で対応でき，よりわかりやすいものになります．

以上のことを反映し，汎用化させた結果は以下のようになります．

集合	$Man = \{佐藤, 鈴木, 高橋, 田中, 渡辺, 伊藤, 山本, 中村, 小林, 斉藤\}$	候補者集合
	$Area = \{A, B, C, D, E, F, G\}$	エリア集合
変数	$x_i \quad , i \in Man$	候補者 i を採用する時 1, 採用しない時 0 を取る変数
定数	$deliver_{ij} \ , i \in Man, j \in Area$	候補者 i がエリア j に配達可能な時 1, 不可能な時 0 を取る
	$cost_i \quad , i \in Man$	候補者 i を採用した際のコスト
目的関数 （最小化）	$\sum_{i \in Man} cost_i x_i$	総コストの最小化
制約	$\sum_{i \in Man} deliver_{ij} x_i \geq 1, \ j \in Area$	すべてのエリアで配達可能になる ようにする

これを SIMPLE で記述すると，次のようになります．なお，制約条件についてはデータファイルの内容をもとに各エリアに対して自動的に生成されます．

```
//候補者集合およびエリア集合の宣言
Set Man;
Element i(set=Man);
Set Area;
Element j(set=Area);
//パラメータおよび変数の宣言
Parameter deliver(name="配達可能エリア",index=(i,j));
Parameter cost(name="コスト",index=i);
IntegerVariable x(name="採用",index=i,type=binary);
//総コストを最小化する
Objective total_cost(name="総コスト",type=minimize);
total_cost=sum(cost[i]*x[i],i);
//各エリアに配置する
sum(deliver[i,j]*x[i],i)>=1;
//求解し結果を表示する
solve();
x.val.print();
total_cost.val.print();
```

実行させる際には，次の配達可能エリアを表す csv ファイル（左）とコストを表す dat ファイル（右）を与えます．

```
配達可能エリア,A,B,C,D,E,F,G
佐藤,1,1,1,0,0,0,0
鈴木,1,0,0,1,0,1,0
高橋,0,1,0,0,1,0,0
田中,0,0,1,1,1,1,1
渡辺,1,0,0,0,0,1,0
伊藤,0,1,0,1,0,1,0
山本,0,0,0,1,0,0,0
中村,0,0,1,0,0,0,1
小林,0,0,1,0,0,1,1
斉藤,0,1,0,0,1,0,1
```

```
コスト=
[佐藤]200
[鈴木]280
[高橋]175
[田中]560
[渡辺]205
[伊藤]245
[山本]80
[中村]195
[小林]265
[斉藤]190;
```

このモデルファイルを実行させることにより，佐藤・伊藤・斉藤の 3 人を採用すると良く，この場合の総コストは 635 であるということが分かります．

ここで，NUOPT の解法について少し述べておきます．今回の例題では厳密解法を用い求解をしました．一方で，0-1 整数計画問題の場合には近似解法である wcsp を用い近似解を求めることもできます． wcsp を用いた際の近似解を求めたい場合には，SIMPLE モデル中の solve(); より前に

```
options.method="wcsp";
```

と記述します．なお，wcsp を用いる場合には終了条件に関する定数をいくつか設定することができます．例えば，wcsp を最大で 1 秒間実行し終了したい場合には，モデルファイルに

```
options.maxtim=1;
```

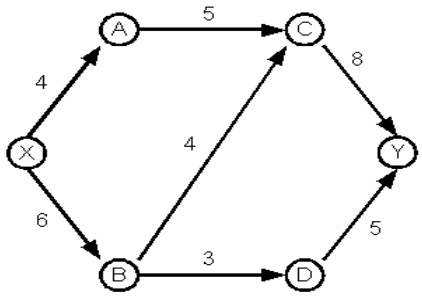
と記述します．詳しいことにつきましては NUOPT/SIMPLE マニュアルを参考にしてください．

最後に，この 3 人を採用した場合，エリアBは 3 人とも担当可能になっています．このようなダブりを許さないように，例題に「1 つのエリアを 2 人以上で担当することを許さない」という制約を加えてみるとどのようになるでしょうか．これは先ほどの SIMPLE モデル中の制約条件「sum(deliver[i,j]*x[i],i)>=1;」を「sum(deliver[i,j]*x[i],i)==1;」とすることで表現できます．実行すると，鈴木・高橋・中村を採用すると良く，この場合の総コストは 650 であるという結果が得られます．

5.6 最大流問題

この節では, 有向グラフをもとにしたデータ構造であるネットワークに関する基本的な問題の一つである最大流問題を取り上げます．最大流問題とは，ネットワーク上で始点（Source）から終点（Sink）まで流すことができる量の最大値を求める問題です．ただし，各辺には流すことができる量の上限が与えられています．さらに，始点と終点以外の点については,「流入してくる総量と流出していく総量は一致する」という関係（「保存則」と言います）が成り立つ必要があります．最大流問題は，配送や通信の分野などで応用が可能です．

（例題）
ある企業は，X 町にある工場で製造したテレビを Y 市のデパートまで納品しています．この際，下の図のように，工場からデパートまで輸送する間にいくつかの配送センターを経由しています．また，2 つの地点の間で 1 日に運べるテレビの台数の上限が図の数字の通りに決まっています．この時，工場からデパートへ 1 日に納品できるテレビは最大で何台でしょうか．ただし，各配送センターは他の地点から輸送されてきたテレビを全て他の地点に輸送するものとします．



まず変数として，ネットワーク上で始点（X 町の工場）から終点（Y 市のデパート）へ 1 日に輸送するテレビの台数（今後は「総輸送量」と呼びます）と各辺に対し輸送するテレビの台数を用意します．こうすると，目的関数は変数「総輸送量」自身となります．また，この例題に出てくる変数は全てテレビの台数を表しますので，整数変数ということになります．

次に，制約条件について考えます．まず，2 点間の輸送量については，負の値は許さず，対応する辺に与えられている上限を超えてはならないという制約条件が必要です．このことは，上下限制約として表現できます．さらに，各地点に対しては保存則が成立している必要があります．ただし，工場については，各地点への輸送量の和が総輸送量と一致するということになります．また，デパートについては，各地点からの輸送量の和が総輸送量と一致しなければなりません．

以上のことから，この例題は次のように定式化できます．

整数変数	$total$	総輸送量
	f_{XA}	工場から A 地点への輸送量
	f_{XB}	工場から B 地点への輸送量
	f_{AC}	A 地点から C 地点への輸送量
	f_{BC}	B 地点から C 地点への輸送量
	f_{BD}	B 地点から D 地点への輸送量
	f_{CY}	C 地点からデパートへの輸送量
	f_{DY}	D 地点からデパートへの輸送量
目的関数（最大化）	$total$	総輸送量
制約条件	$total = f_{XA} + f_{XB}$	工場での輸送量
	$f_{XA} = f_{AC}$	A 地点での輸送量
	$f_{XB} = f_{BC} + f_{BD}$	B 地点での輸送量
	$f_{AC} + f_{BC} = f_{CY}$	C 地点での輸送量
	$f_{BD} = f_{DY}$	D 地点での輸送量
	$f_{CY} + f_{DY} = total$	デパートでの輸送量
	$0 \le f_{XA} \le 4$	工場から A 地点への輸送量の上下限
	$0 \le f_{XB} \le 6$	工場から B 地点への輸送量の上下限
	$0 \le f_{AC} \le 5$	A 地点から C 地点への輸送量の上下限
	$0 \le f_{BC} \le 4$	B 地点から C 地点への輸送量の上下限
	$0 \le f_{BD} \le 3$	B 地点から D 地点への輸送量の上下限
	$0 \le f_{CY} \le 8$	C 地点からデパートへの輸送量の上下限
	$0 \le f_{DY} \le 5$	D 地点からデパートへの輸送量の上下限

これを SIMPLE で記述すると以下のようになります.

```
//変数の宣言
IntegerVariable total(name="工場からデパートへの総輸送量");
IntegerVariable f_XA(name="工場から地点Aへの輸送量");
IntegerVariable f_XB(name="工場から地点Bへの輸送量");
IntegerVariable f_AC(name="地点Aから地点Cへの輸送量");
IntegerVariable f_BC(name="地点Bから地点Cへの輸送量");
IntegerVariable f_BD(name="地点Bから地点Dへの輸送量");
IntegerVariable f_CY(name="地点Cからデパートへの輸送量");
IntegerVariable f_DY(name="地点Dからデパートへの輸送量");
//総輸送量の最大化
Objective obj(name="工場からデパートへの総輸送量",type=maximize);
obj=total;
//各地点での輸送量
total==f_XA+f_XB;
f_XA==f_AC;
f_XB==f_BC+f_BD;
f_AC+f_BC==f_CY;
f_BD==f_DY;
f_CY+f_DY==total;
//輸送量に関する上下限制約
0<=f_XA<=4;
0<=f_XB<=6;
0<=f_AC<=5;
0<=f_BC<=4;
0<=f_BD<=3;
0<=f_CY<=8;
0<=f_DY<=5;
//求解して値を出力する
solve();
f_XA.val.print();
f_XB.val.print();
f_AC.val.print();
f_BC.val.print();
f_BD.val.print();
f_CY.val.print();
f_DY.val.print();
total.val.print();
```

このモデルファイルを実行させると，

```
工場から地点 A への輸送量=4
工場から地点 B への輸送量=6
地点 A から地点 C への輸送量=4
地点 B から地点 C への輸送量=4
地点 B から地点 D への輸送量=2
地点 C からデパートへの輸送量=8
地点 D からデパートへの輸送量=2
工場からデパートへの総輸送量=10
```

という表示がされ，1 日に最大 10 台納品できるという結果を確認できます．

ここで，SIMPLE での記述について，より簡潔な形に書き直していくことにします．そのために，都市の集合という概念を導入します．また，2 点間の輸送量の上限値を定数として外部から与えることにします．すると，定式化については次のように書き直すことができます．

集合	$City = \{A, B, C, D, X, Y\}$	都市の集合
整数変数	$total$	総輸送量
	$f_{ij}\ , i \in City, j \in City$	i から j への輸送量
目的関数（最大化）	$total$	総輸送量
定数	$upper_{ij}\ , i \in City, j \in City$	i から j への輸送量の上限
制約条件	$total = \sum_{j \in City} f_{Xj}$	工場での輸送量
	$\sum_{i} f_{ik} = \sum_{j} f_{kj}\ , k \in City, k \neq X, k \neq Y$	配送センターでの輸送量
	$\sum_{i \in City} f_{iY} = total$	デパートでの輸送量
	$0 \leq f_{ij} \leq upper_{ij}\ , i \in City, j \in City$	i から j への輸送量の上下限

これより，SIMPLE での記述は次のようなものになります．

```
//都市の集合の宣言
Set City;
Element i(set=City),j(set=City);
//変数の宣言
IntegerVariable total(name="工場からデパートへの総輸送量");
IntegerVariable f(index=(City,City));
//2 点間の輸送量の上限をパラメータとして宣言
Parameter upper(name="輸送量の上限",index=(City,City));
//総輸送量の最大化
Objective obj(name="工場からデパートへの総輸送量",type=maximize);
obj=total;
//各地点での輸送量
total==sum(f["X",j],j);
sum(f[i,j],j)==sum(f[j,i],j),i!="X",i!="Y";
sum(f[i,"Y"],i)==total;
//輸送量に関する上下限制約
0<=f[i,j]<=upper[i,j];
//求解して解を出力する
solve();
total.val.print();
```

なお，実行時には次の csv ファイルを与えます．

```
輸送量の上限,A,B,C,D,X,Y
A,0,0,5,0,0,0
B,0,0,4,3,0,0
C,0,0,0,0,0,8
D,0,0,0,0,0,5
X,4,6,0,0,0,0
Y,0,0,0,0,0,0
```

ところで，SIMPLE では Graph というグラフ構造に対応するデータ構造が用意されていて，ネットワーク問題を記述する際に利用できます．そこで，この例題を Graph を用いて記述してみます．

すると，SIMPLE モデルは次のようになります．

```
//ネットワークを Graph を用い宣言する
Graph g(name="network");
Element i(set=g.nodes);
Element e(set=g.arcs);
Element eout(set=out(g,i));
Element ein(set=in(g,i));
//変数の宣言
IntegerVariable f(name="2 点間の輸送量",index=g.arcs);
IntegerVariable total(name="工場からデパートへの総輸送量");
//2 点間の輸送量の上限をパラメータとして宣言
Parameter upper_flow(index=g.arcs);
//総輸送量の最大化
Objective obj(type=maximize);
obj=total;
//輸送量に関する制約条件
sum(f[eout],eout)==total,i=="X";
sum(f[ein],ein)==sum(f[eout],eout),i!="X",i!="Y";
sum(f[ein],ein)==total,i=="Y";
0<=f[e]<=upper_flow[e];
//求解して解を出力する
solve();
f.val.print();
total.val.print();
```

ここで，実行時には次のような dat ファイルを与えます．

```
network.nodes=X A B C D Y;
network.arcs=X,A X,B A,C B,C B,D C,Y D,Y;
upper_flow=[X,A]4 [X,B]6 [A,C]5 [B,C]4 [B,D]3 [C,Y]8 [D,Y]5;
```

このように記述しておくと，dat ファイルの修正のみで新たな配送センターができたなどの状況の変化に柔軟に対応でき大変便利です．ただし，工場が X でデパートが Y であるということを変更したい場合はモデルファイルも修正する必要があります．

5.7 最小費用流問題

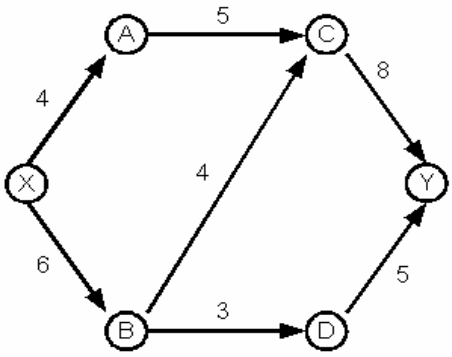
前節の最大流問題では, テレビの輸送を例にネットワークを流れる量をできるだけ多くしようという方針で最適化を行いました. このため, テレビを輸送する際にかかる燃料費などの様々なコストを考慮していませんでした. また, 工場から 1 日に出荷するテレビの量が決まっているような場合には別の方針で最適化を行う必要があります.

決まった量をネットワーク上で始点から終点まで流す際にかかる費用（コスト）を最小にしようという問題のことを最小費用流問題といいます. なお, 最大流問題のときと同様に, 保存則が成立する必要があります. また,「決まった量」が 1 である場合には最小路問題と呼ばれ, カーナビゲーションシステムでのルート探索や鉄道の経路案内などに応用されています.

（例題）

ある企業は, X 町の工場で製造したテレビを Y 市のデパートまで 1 日に 8 台納品しています. この際, 右下の図のように, 工場からデパートまで輸送する間にいくつかの配送センターを経由しています. また, 2 つの地点の間で 1 日に運べるテレビの台数の上限が図の数字のように決まっています. さらに, 2 つの地点の間でテレビを 1 台輸送する毎に左下の表に書かれている費用がかかります. このとき, テレビを 8 台とも工場からデパートへ輸送する際にかかる総費用は最低いくらでしょうか. ただし, 各配送センターは他の地点から輸送されてきたテレビを全て他の地点に輸送するものとします.

輸送元	輸送先	1 台あたりの費用
工場 X	A	250
工場 X	B	200
A	C	270
B	C	300
B	D	220
C	デパート Y	190
D	デパート Y	170



定式化について, 前節の最大流問題の時との違いは

・総輸送量（前節の例題での total という変数）が 8 に決まっている.

・目的関数が総費用の最小化に変わっている.

の 2 点です. この 2 点以外については, 前節の例題と同じ定式化となりますので必要に応じて前節を参考にしてください. なお, 総費用は各輸送ルートについて「1 台あたりの費用」と「輸

送した台数」の積を求め総和をとることで求められます.

以上のことから，この例題は次のように定式化できます.

整数変数	f_{XA}	工場から A 地点への輸送量
	f_{XB}	工場から B 地点への輸送量
	f_{AC}	A 地点から C 地点への輸送量
	f_{BC}	B 地点から C 地点への輸送量
	f_{BD}	B 地点から D 地点への輸送量
	f_{CY}	C 地点からデパートへの輸送量
	f_{DY}	D 地点からデパートへの輸送量
目的関数（最小化）	$250f_{XA}+200f_{XB}+270f_{AC}+300f_{BC}+220f_{BD}+190f_{CY}+170f_{DY}$	総費用
制約条件	$8=f_{XA}+f_{XB}$	工場での輸送量
	$f_{XA}=f_{AC}$	A 地点での輸送量
	$f_{XB}=f_{BC}+f_{BD}$	B 地点での輸送量
	$f_{AC}+f_{BC}=f_{CY}$	C 地点での輸送量
	$f_{BD}=f_{DY}$	D 地点での輸送量
	$f_{CY}+f_{DY}=8$	デパートでの輸送量
	$0\le f_{XA}\le 4$	工場から A 地点への輸送量の上下限
	$0\le f_{XB}\le 6$	工場から B 地点への輸送量の上下限
	$0\le f_{AC}\le 5$	A 地点から C 地点への輸送量の上下限
	$0\le f_{BC}\le 4$	B 地点から C 地点への輸送量の上下限
	$0\le f_{BD}\le 3$	B 地点から D 地点への輸送量の上下限
	$0\le f_{CY}\le 8$	C 地点からデパートへの輸送量の上下限
	$0\le f_{DY}\le 5$	D 地点からデパートへの輸送量の上下限

この結果を SIMPLE で記述すると以下のようになります.

```
//変数の宣言
IntegerVariable f_XA(name="工場から地点 A への輸送量");
IntegerVariable f_XB(name="工場から地点 B への輸送量");
IntegerVariable f_AC(name="地点 A から地点 C への輸送量");
IntegerVariable f_BC(name="地点 B から地点 C への輸送量");
IntegerVariable f_BD(name="地点 B から地点 D への輸送量");
IntegerVariable f_CY(name="地点 C からデパートへの輸送量");
IntegerVariable f_DY(name="地点 D からデパートへの輸送量");
//総費用の最小化
Objective totalcost(name="総費用",type=minimize);
totalcost=250*f_XA+200*f_XB+270*f_AC+300*f_BC+220*f_BD+190*f_CY+
170*f_DY;
//各地点での輸送量
8==f_XA+f_XB;
f_XA==f_AC;
f_XB==f_BC+f_BD;
f_AC+f_BC==f_CY;
f_BD==f_DY;
f_CY+f_DY==8;
//輸送量に関する上下限制約
0<=f_XA<=4;
0<=f_XB<=6;
0<=f_AC<=5;
0<=f_BC<=4;
0<=f_BD<=3;
0<=f_CY<=8;
0<=f_DY<=5;
//求解して値を出力する
solve();
f_XA.val.print();
f_XB.val.print();
f_AC.val.print();
f_BC.val.print();
f_BD.val.print();
f_CY.val.print();
f_DY.val.print();
totalcost.val.print();
```

このモデルを実行させると，

```
工場から地点 A への輸送量=2
工場から地点 B への輸送量=6
地点 A から地点 C への輸送量=2
地点 B から地点 C への輸送量=3
地点 B から地点 D への輸送量=3
地点 C からデパートへの輸送量=5
地点 D からデパートへの輸送量=3
総費用=5260
```

という表示がされ，結果を確認できます．

ところで，このままの記述では都市の数が多い問題に応用することが困難です．そこで，都市の集合という概念を導入し，必要なデータを定数として外部から与えるようにすることで大規模な問題に対しても応用しやすいようにします．まず，この方針で定式化をしなおすと次のようになります．なお，保存則については「他の地点からの輸送量の和」と「他の地点への輸送量の和」の差のデータを外部から与える形式を取りました．ちなみに，この値が負の地点が工場，正の地点がデパート，そして 0 の地点が配送センターということになります．

集合	$City = \{A, B, C, D, X, Y\}$	都市の集合
整数変数	$f_{ij}\ , i \in City, j \in City$	i から j への輸送量
定数	$upper_{ij}\ , i \in City, j \in City$	i から j への輸送量の上限
	$cost_{ij}\ , i \in City, j \in City$	i から j への輸送の際にかかる 1 台あたりの費用
	$supply_i\ , i \in City$	i での流入量と流出量の差
目的関数（最小化）	$\sum_{j \in City} \sum_{i \in City} cost_{ij} \cdot f_{ij}$	総費用
制約条件	$\sum_{i \in City} f_{ik} - \sum_{j \in City} f_{kj} = supply_k\ , k \in City$	各地点での輸送量
	$0 \le f_{ij} \le upper_{ij}\ , i \in City, j \in City$	i から j への輸送量の上下限

この定式化をしなおした結果を SIMPLE で記述すると次のようになります.

```
//都市の集合の宣言
Set City;
Element i(set=City),j(set=City),k(set=City);

//変数の宣言
IntegerVariable f(name="輸送量",index=(City,City));

//パラメータの宣言
Parameter upper(name="輸送量の上限",index=(City,City));
Parameter cost(name="輸送費用",index=(City,City));
Parameter supply(name="supply",index=City);

//総費用の最小化
Objective total_cost(name="総費用",type=minimize);
total_cost=sum(sum(cost[i,j]*f[i,j],i),j);

//各地点での輸送量
sum(f[i,k],i)-sum(f[k,j],j)==supply[k];

//輸送量に関する上下限制約
0<=f[i,j]<=upper[i,j];

//求解して解を出力する
solve();
total_cost.val.print();
```

なお, 実行時には次の 2 つの csv ファイル (左・中) と 1 つの dat ファイル (右) を与えます.

```
輸送量の上限,A,B,C,D,X,Y
A,0,0,5,0,0,0
B,0,0,4,3,0,0
C,0,0,0,0,0,8
D,0,0,0,0,0,5
X,4,6,0,0,0,0
Y,0,0,0,0,0,0
```

```
輸送費用,A,B,C,D,X,Y
A,0,0,270,0,0,0
B,0,0,300,220,0,0
C,0,0,0,0,0,190
D,0,0,0,0,0,170
X,250,200,0,0,0,0
Y,0,0,0,0,0, 0
```

```
supply=
[A]0
[B]0
[C]0
[D]0
[X]-8
[Y]8;
```

最後に，この例題についても Graph を用いて書き換えてみます．すると，SIMPLE での記述は以下のようになります．

```
//Graph に関する宣言
Graph g(name="network");
Element i(set=g.nodes);
Element e(set=g.arcs);
Element eout(set=out(g,i));
Element ein(set=in(g,i));
//変数の宣言
IntegerVariable f(name="２点間の輸送量",index=g.arcs);
//パラメータの宣言
Parameter supply(name="supply",index=g.nodes);
Parameter upper_flow(name="輸送量の上限",index=g.arcs);
Parameter cost(name="輸送費用",index=g.arcs);
//総費用の最小化
Objective total_cost(name="総費用",type=minimize);
total_cost=sum(cost[e]*f[e],e);
//輸送量に関する制約条件
sum(f[ein],ein)-sum(f[eout],eout)==supply[i];
0<=f[e]<=upper_flow[e];
//求解し解を出力する
solve();
f.val.print();
total_cost.val.print();
```

ここで，実行時には次のような dat ファイルを与えます．

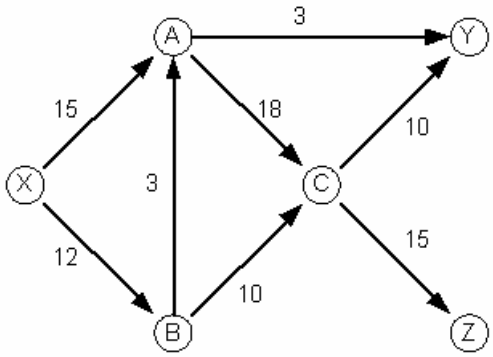
```
network.nodes=X A B C D Y;
network.arcs=X,A X,B A,C B,C B,D C,Y D,Y;
supply=[X]-8 [A]0 [B]0 [C]0 [D]0 [Y]8;
輸送量の上限=[X,A]4 [X,B]6 [A,C]5 [B,C]4 [B,D]3 [C,Y]8 [D,Y]5;
輸送費用=[X,A]250 [X,B]200 [A,C]270 [B,C]300 [B,D]220 [C,Y]190
[D,Y]170;
```

5.8 多品種流問題

今まで述べてきた最大流問題や最小費用流問題では，ネットワーク上を１種類のものしか流れていませんでした．しかし，現実世界ではネットワーク上を複数のものが流れるというケースがよくあります．多品種流問題は，複数の品物をそれぞれ始点から終点まで輸送するという場合に利用される問題で，通信など多くの分野で応用されています．

（例題）

ある企業は，X 町の工場で製造したテレビを Y 市のデパートまで１日に 8 台納品しています．さらに，同じ工場で製造したパソコンを Z 市の専門店まで１日に 13 台納品しています．この際，下の図のように，工場からデパートまで輸送する間にいくつかの配送センターを経由しています．また，2 つの地点の間で１日に運べるテレビとパソコンの合計台数の上限が図の数字のように決まっています．さらに，2 つの地点の間でテレビやパソコンを１台輸送する毎に下の表に書かれているだけの費用がかかります．このとき，テレビとパソコンを決められた台数だけ納品する際にかかる総費用は１日につき最低いくらでしょうか．



輸送元	輸送先	テレビの輸送費	パソコンの輸送費
工場 X	A	12	11
工場 X	B	10	12
A	C	4	5
A	デパート Y	11	-
B	A	6	7
B	C	11	9
C	デパート Y	9	-
C	専門店 Z	-	5

まず，各輸送ルートについてテレビの輸送台数とパソコンの輸送台数が変数として必要です．ただし，デパートにパソコンを輸送したり専門店にテレビを輸送したりすることはありませんからこれらに対応する変数はここでは宣言しないことにします．

次に，目的関数については最小費用流問題の時と同様に考えることになります．ただし，テレビとパソコンで輸送コストが異なる点には注意が必要です．

最後に，制約条件を考えます．各輸送ルートについては問題文にあるようにテレビの輸送量とパソコンの輸送量の和が上限を超えないという制約条件が必要です．また，保存則については地点と商品に関し場合分けをする必要があります．さらに，「-1 個輸送する」ということはありえませんので変数について非負制約が必要です．

以上のことから，次のように定式化できます．

整数変数	TV_{XA}, PC_{XA}	工場から A 地点への輸送量
	TV_{XB}, PC_{XB}	工場から B 地点への輸送量
	TV_{AC}, PC_{AC}	A 地点から C 地点への輸送量
	TV_{AY}	A 地点からデパートへの輸送量
	TV_{BA}, PC_{BA}	B 地点から A 地点への輸送量
	TV_{BC}, PC_{BC}	B 地点から C 地点への輸送量
	TV_{CY}	C 地点からデパートへの輸送量
	PC_{CZ}	C 地点から専門店への輸送量

目的関数（最小化）

$$12TV_{XA}+10TV_{XB}+4TV_{AC}+11TV_{AY}+6TV_{BA}+11TV_{BC}+9TV_{CY}$$
$$+11PC_{XA}+12PC_{XB}+5PC_{AC}+7PC_{BA}+9PC_{BC}+5PC_{CZ}$$

総輸送費

制約条件

$$TV_{XA}+PC_{XA}\le 15, TV_{XB}+PC_{XB}\le 12, TV_{AC}+PC_{AC}\le 18,$$
$$TV_{AY}\le 3, TV_{BA}+PC_{BA}\le 3, TV_{BC}+PC_{BC}\le 10, TV_{CY}\le 10, PC_{CZ}\le 15$$

各輸送ルートの輸送量の上限を超えない

$$TV_{XA}+TV_{XB}=8,\ TV_{AC}+TV_{AY}=TV_{XA}+TV_{BA},\ TV_{BA}+TV_{BC}=TV_{XB},$$
$$TV_{CY}=TV_{AC}+TV_{BC},\ 8=TV_{AY}+TV_{CY}$$

テレビに関する保存則

$$PC_{XA}+PC_{XB}=13,\ PC_{AC}+PC_{AY}=PC_{XA}+PC_{BA},\ PC_{BA}+PC_{BC}=PC_{XB},$$
$$PC_{CZ}=PC_{AC}+PC_{BC},\ 13=PC_{CZ}$$

パソコンに関する保存則

$$TV_{XA}\ge 0, TV_{XB}\ge 0, TV_{AC}\ge 0, TV_{AY}\ge 0, TV_{BA}\ge 0, TV_{BC}\ge 0, TV_{CY}\ge 0,$$
$$PC_{XA}\ge 0, PC_{XB}\ge 0, PC_{AC}\ge 0, PC_{BA}\ge 0, PC_{BC}\ge 0, PC_{CZ}\ge 0$$

輸送量に関する非負制約

これを SIMPLE で記述すると次のようになります.

```
//変数の宣言
IntegerVariable
      TV_XA,TV_XB,TV_AC,TV_AY,TV_BA,TV_BC,TV_CY,
      PC_XA,PC_XB,PC_AC,PC_BA,PC_BC,PC_CZ;
//目的関数の宣言
Objective total_cost(name="総輸送費",type=minimize);
total_cost=12*TV_XA+10*TV_XB+4*TV_AC+11*TV_AY+6*TV_BA+11*TV_BC
  +9*TV_CY+11*PC_XA+12*PC_XB+5*PC_AC+7*PC_BA+9*PC_BC+5*PC_CZ;
//輸送ルートごとの輸送量の上限
TV_XA+PC_XA<=15;
TV_XB+PC_XB<=12;
TV_AC+PC_AC<=18;
TV_AY<=3;
TV_BA+PC_BA<=3;
TV_BC+PC_BC<=10;
TV_CY<=10;
PC_CZ<=15;
//テレビに関する保存則
TV_XA+TV_XB==8;
TV_AC+TV_AY==TV_XA+TV_BA;
TV_BA+TV_BC==TV_XB;
TV_CY==TV_AC+TV_BC;
8==TV_AY+TV_CY;
//パソコンに関する保存則
PC_XA+PC_XB==13;
PC_AC==PC_XA+PC_BA;
PC_BA+PC_BC==PC_XB;
PC_CZ==PC_AC+PC_BC;
13==PC_CZ;
//非負制約
TV_XA>=0;TV_XB>=0;TV_AC>=0;TV_AY>=0;TV_BA>=0;TV_BC>=0;TV_CY>=0;
PC_XA>=0;PC_XB>=0;PC_AC>=0;PC_BA>=0;PC_BC>=0;PC_CZ>=0;
```

ここで，SIMPLE での記述をより簡潔なものにしていくことにします．そのためには，都市の集合の概念を導入し，さまざまなデータに対応できるようにすることが有効です．この際，輸送量の上限値などの具体的なデータを出来るだけ外部から与えるようにしておきます．

すると，定式化については次のように表現できます.

集合	$City = \{A, B, C, X, Y, Z\}$	**都市の集合**
整数変数	$TV_{ij}\ , i \in City, j \in City$	**i から j へのテレビの輸送量**
	$PC_{ij}\ , i \in City, j \in City$	**i から j へのパソコンの輸送量**
定数	$upper_{ij}\ , i \in City, j \in City$	**i から j への輸送量の上限**
	$TV\_cost_{ij}\ , i \in City, j \in City$	**i から j への輸送の際にかかるテレビ 1 台あたりの費用**
	$PC\_cost_{ij}\ , i \in City, j \in City$	**i から j への輸送の際にかかるパソコン 1 台あたりの費用**
	$TV\_supply_{i}\ , i \in City$	**i でのテレビの流入量と流出量の差**
	$PC\_supply_{i}\ , i \in City$	**i でのパソコンの流入量と流出量の差**
目的関数（最小化）	$\sum_{j \in City} \sum_{i \in City} (TV\_cost_{ij} \cdot TV_{ij} + PC\_cost_{ij} \cdot PC_{ij})$	**総輸送費**
制約条件	$TV_{ij} + PC_{ij} \le upper_{ij}\ , i \in City, j \in City$	**i から j への輸送量の上限**
	$\sum_{i \in City} TV_{ik} - \sum_{j \in City} TV_{kj} = TV\_supply_{k}\ , k \in City$	**各地点でのテレビの輸送量**
	$\sum_{i \in City} PC_{ik} - \sum_{j \in City} PC_{kj} = PC\_supply_{k}\ , k \in City$	**各地点でのパソコンの輸送量**
	$0 \le TV_{ij}\ , i \in City, j \in City$	**テレビの輸送量の非負制約**
	$0 \le PC_{ij}\ , i \in City, j \in City$	**パソコンの輸送量の非負制約**

SIMPLE で記述すると次のようになり, 先ほどのものより簡潔になっていることが分かります.

```
//都市の集合の宣言
Set City;
Element i(set=City),j(set=City),k(set=City);
//変数の宣言
IntegerVariable TV(name="テレビの輸送台数",index=(City,City));
IntegerVariable PC(name="パソコンの輸送台数",index=(City,City));
//パラメータの宣言
Parameter upper(name="輸送量の上限",index=(City,City));
Parameter TV_cost(name="テレビの輸送費用",index=(City,City));
Parameter PC_cost(name="パソコンの輸送費用",index=(City,City));
Parameter TV_supply(name="TV_supply",index=City);
Parameter PC_supply(name="PC_supply",index=City);
//目的関数の宣言
Objective total_cost(name="総輸送費",type=minimize);
total_cost=sum(TV_cost[i,j]*TV[i,j]+PC_cost[i,j]*PC[i,j],(i,j));
//輸送ルートごとの輸送量の上限
TV[i,j]+PC[i,j]<=upper[i,j];
//テレビに関する保存則
sum(TV[i,k],i)-sum(TV[k,j],j)==TV_supply[k];
//パソコンに関する保存則
sum(PC[i,k],i)-sum(PC[k,j],j)==PC_supply[k];
//非負制約
TV[i,j]>=0;
PC[i,j]>=0;
```

なお, 実行時には次の 3 つの csv ファイル（輸送量の上限・テレビの輸送費用・パソコンの輸送費用）と 1 つの dat ファイル（TV_supply および PC_supply）を与えます.

```
輸送量の上限,A,B,C,X,Y,Z
A,0,0,18,0,3,0
B,3,0,10,0,0,0
C,0,0,0,0,10,15
X,15,12,0,0,0,0
Y,0,0,0,0,0,0
Z,0,0,0,0,0,0
```

```
テレビの輸送費用,A,B,C,X,Y,Z
A,0,0,4,0,11,0
B,6,0,11,0,0,0
C,0,0,0,0,9,0
X,12,10,0,0,0,0
Y,0,0,0,0,0,0
Z,0,0,0,0,0,0
```

```
パソコンの輸送費用,A,B,C,X,Y,Z
A,0,0,5,0,0,0
B,7,0,9,0,0,0
C,0,0,0,0,0,5
X,11,12,0,0,0,0
Y,0,0,0,0,0,0
Z,0,0,0,0,0,0
```

```
TV_supply=[A]0 [B]0 [C]0
          [X]-8 [Y]8 [Z]0;
PC_supply=[A]0 [B]0 [C]0
          [X]-13 [Y]0 [Z]13;
```

最後に，この例題についても，Graph を用いた SIMPLE での記述を以下に示しておきます．

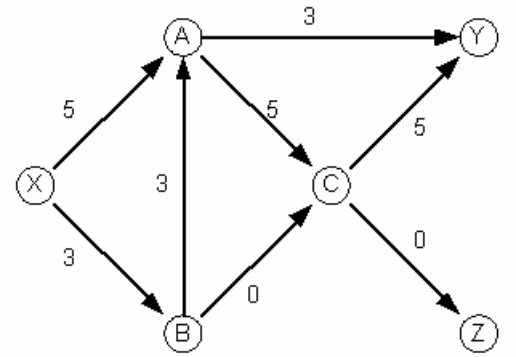
```
//Graph に関する宣言
Graph g(name="network");
Element i(set=g.nodes);
Element e(set=g.arcs);
Element eout(set=out(g,i));
Element ein(set=in(g,i));
//変数の宣言
IntegerVariable TV(name="２点間のテレビの輸送量",index=g.arcs);
IntegerVariable PC(name="２点間のパソコンの輸送量",index=g.arcs);
//パラメータの宣言
Parameter supply_TV(name="supply_TV",index=g.nodes);
Parameter supply_PC(name="supply_PC",index=g.nodes);
Parameter upper_flow(name="輸送量の上限",index=g.arcs);
Parameter cost_TV(name="テレビの輸送費用",index=g.arcs);
Parameter cost_PC(name="パソコンの輸送費用",index=g.arcs);
//総費用の最小化
Objective total_cost(name="総輸送費",type=minimize);
total_cost=sum(cost_TV[e]*TV[e]+cost_PC[e]*PC[e],e);
//輸送量に関する制約条件
TV[e]+PC[e]<=upper_flow[e];
sum(TV[ein],ein)-sum(TV[eout],eout)==supply_TV[i];
sum(PC[ein],ein)-sum(PC[eout],eout)==supply_PC[i];
0<=TV[e];
0<=PC[e];
//求解し解を出力する
solve();
simple_printf("%s から %s への輸送量：テレビ %d 台 パソコン %d 台
¥n",e,TV[e],PC[e]);
total_cost.val.print();
```

実行時には次のデータファイルを与えます.

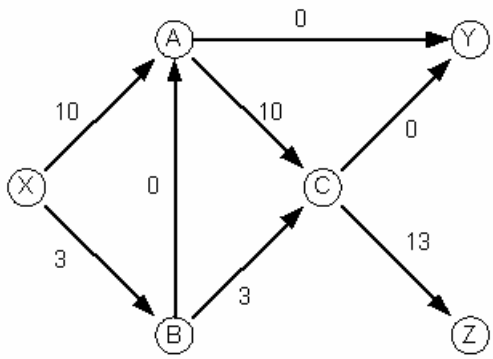
```
network.nodes=X A B C Y Z;
network.arcs=X,A X,B A,C A,Y B,A B,C C,Y C,Z;
supply_TV=[X]-8 [A]0 [B]0 [C]0 [Y]8 [Z]0;
supply_PC=[X]-13 [A]0 [B]0 [C]0 [Y]0 [Z]13;
輸送量の上限=[X,A]15 [X,B]12 [A,C]18 [A,Y]3 [B,A]3 [B,C]10 [C,Y]10
[C,Z]15;
テレビの輸送費用=[X,A]12 [X,B]10 [A,C]4 [A,Y]11 [B,A]6 [B,C]11 [C,Y]9
[C,Z]0;
パソコンの輸送費用=[X,A]11 [X,B]12 [A,C]5 [A,Y]0 [B,A]7 [B,C]9 [C,Y]0
[C,Z]5;
```

このモデルを実行した結果から, 次の図のようにテレビとパソコンを輸送すると最適でありこのときにかかる総費用は 494 であることが分かります.

テレビの輸送台数



パソコンの輸送台数



5.9 p メディアン問題

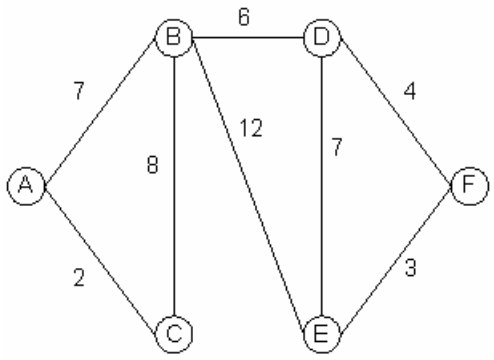
施設をどの地点に配置すると最適なのかを求める施設配置問題は, 都市計画などに応用されています．施設配置問題の中からこの節で p メディアン問題を，次の節で p センター問題を取り上げます．なお，本チュートリアルでは各地点は需要を有する地点（今後は「需要点」と呼びます）でありさらに施設を配置することが可能な地点でもあると仮定します．

まず，メディアンについて説明します．メディアンとは各需要点から施設を配置した地点（今後は「施設点」と呼びます）への移動距離の総和を最小にする点のことです．そして，このメディアンを求める問題のことをメディアン問題と呼びます．なお，一般にはどの程度の需要があるかは需要点ごとに異なりますので重み付きの総移動距離を最小にすることになります．また，施設を１箇所ではなく p 箇所に配置するような場合に p メディアン問題と呼びます．

（例題）

ある企業は，A 市から F 市までの 6 市の中から 2 つの市に新たにデパートを出店しようと考えています．この時，出店地までの総移動距離を最小にするためにはどの市に出店すると良いでしょうか．なお，各市の人口は左下の表のようになっています．また，2 市間の距離は右下の図のようになっています．ただし，直接結ばれていない 2 都市間の距離は他の都市を経由したときの最短距離を採用します．

都市	人口
A 市	800
B 市	550
C 市	780
D 市	600
E 市	1020
F 市	360



まず, この問題では各都市に対しデパートを出店するのかどうかを表す変数が必要であることがわかります. しかし, これだけでは各都市からどちらの出店地が近いのかを求める必要があり, 複雑な式になってしまいます．そこで，都市 i について都市 j に配分する場合 1，そうでない場合は 0 を取るような 0-1 変数 x_{ij} を導入します．なお，$x_{ii}=1$の時には都市 i にデパートを出店し，$x_{ii}=0$ の時には出店しないという解釈をします．

次に，制約条件について考えていきます．先ほど述べたように，x_{ii} は都市 i に出店するかど

うかを表しています．この例題では 2 都市に出店しますので，$\sum_i x_{ii}$ は 2 である必要があります．また，各都市について近い方のデパートに配分しなければなりません（同距離の場合には任意にどちらかを選択することになります）．x_{ij} を用いて表現すると，各 i について $\sum_j x_{ij}$ が 1 であるということになります．最後に，今のままですとデパートよりも近くにある出店していない都市に配分されてしまう可能性があります．これを防ぐためには，$x_{jj}=0$ の場合に $x_{ij}=1$ となってはいけないという制約条件が必要です．言い換えると，x_{ij} は x_{jj} より大きい値をとらないということになります．よって，式で表すと $x_{ij} \le x_{jj}$ となります．ただし，この制約条件は $i=j$ の場合には無意味ですから $i \neq j$ の場合のみで十分であることに注意してください．

目的関数は，重み付きの総移動距離ということになります．まず，都市 i から都市 j への移動距離は $x_{ij}=1$ の場合にのみ考慮する必要があります．よって，まず 2 市 i,j 間の距離と x_{ij} の積を求めます．そして，j について和をとることで都市 i からの重みなしの移動距離が得られます．そして，この移動距離に重みである人口を掛けあわせることで都市 i からの重みをつけた移動距離を得ます．重みをつけた移動距離を全ての都市について求め，和をとることで目的関数である重み付きの総移動距離となります．なお，図より 2 都市間の距離は次の表の通りになることが分かります．

	A市	B市	C市	D市	E市	F市
A市	0	7	2	13	19	17
B市	7	0	8	6	12	10
C市	2	8	0	14	20	18
D市	13	6	14	0	7	4
E市	19	12	20	7	0	3
F市	17	10	18	4	3	0

以上のことをまとめると，次のように定式化できます.

変数

$$x_{AA}, x_{AB}, x_{AC}, x_{AD}, x_{AE}, x_{AF}, x_{BA}, x_{BB}, x_{BC}, x_{BD}, x_{BE}, x_{BF}, x_{CA}, x_{CB}, x_{CC}, x_{CD}, x_{CE}, x_{CF},$$
$$x_{DA}, x_{DB}, x_{DC}, x_{DD}, x_{DE}, x_{DF}, x_{EA}, x_{EB}, x_{EC}, x_{ED}, x_{EE}, x_{EF}, x_{FA}, x_{FB}, x_{FC}, x_{FD}, x_{FE}, x_{FF}$$

0 または 1 をとる出店するのかと最も近い出店地はどこなのかを表す変数

目的関数（最小化）

$$\begin{aligned} & 800(7x_{AB} + 2x_{AC} + 13x_{AD} + 19x_{AE} + 17x_{AF}) \\ & + 550(7x_{BA} + 8x_{BC} + 6x_{BD} + 12x_{BE} + 10x_{BF}) \\ & + 780(2x_{CA} + 8x_{CB} + 14x_{CD} + 20x_{CE} + 18x_{CF}) \\ & + 600(13x_{DA} + 6x_{DB} + 14x_{DC} + 7x_{DE} + 4x_{DF}) \\ & + 1020(19x_{EA} + 12x_{EB} + 20x_{EC} + 7x_{ED} + 3x_{EF}) \\ & + 360(17x_{FA} + 10x_{FB} + 18x_{FC} + 4x_{FD} + 3x_{FE}) \end{aligned}$$

重み付き総移動距離

制約条件

$$x_{AA} + x_{BB} + x_{CC} + x_{DD} + x_{EE} + x_{FF} = 2$$ 2 都市に出店

$$x_{AA} + x_{AB} + x_{AC} + x_{AD} + x_{AE} + x_{AF} = 1$$ A 市は 1 箇所に配分

$$x_{BA} + x_{BB} + x_{BC} + x_{BD} + x_{BE} + x_{BF} = 1$$ B 市は 1 箇所に配分

$$x_{CA} + x_{CB} + x_{CC} + x_{CD} + x_{CE} + x_{CF} = 1$$ C 市は 1 箇所に配分

$$x_{DA} + x_{DB} + x_{DC} + x_{DD} + x_{DE} + x_{DF} = 1$$ D 市は 1 箇所に配分

$$x_{EA} + x_{EB} + x_{EC} + x_{ED} + x_{EE} + x_{EF} = 1$$ E 市は 1 箇所に配分

$$x_{FA} + x_{FB} + x_{FC} + x_{FD} + x_{FE} + x_{FF} = 1$$ F 市は 1 箇所に配分

$$x_{AB} \le x_{BB}, x_{AC} \le x_{CC}, x_{AD} \le x_{DD}, x_{AE} \le x_{EE}, x_{AF} \le x_{FF},$$
$$x_{BA} \le x_{AA}, x_{BC} \le x_{CC}, x_{BD} \le x_{DD}, x_{BE} \le x_{EE}, x_{BF} \le x_{FF},$$
$$x_{CA} \le x_{AA}, x_{CB} \le x_{BB}, x_{CD} \le x_{DD}, x_{CE} \le x_{EE}, x_{CF} \le x_{FF},$$
$$x_{DA} \le x_{AA}, x_{DB} \le x_{BB}, x_{DC} \le x_{CC}, x_{DE} \le x_{EE}, x_{DF} \le x_{FF},$$
$$x_{EA} \le x_{AA}, x_{EB} \le x_{BB}, x_{EC} \le x_{CC}, x_{ED} \le x_{DD}, x_{EF} \le x_{FF},$$
$$x_{FA} \le x_{AA}, x_{FB} \le x_{BB}, x_{FC} \le x_{CC}, x_{FD} \le x_{DD}, x_{FE} \le x_{EE}$$

出店している都市に配分

この結果をそのまま SIMPLE で記述すると次のようになります.

```
//変数の宣言
IntegerVariable
     x_AA(type=binary),x_AB(type=binary),x_AC(type=binary),
     x_AD(type=binary),x_AE(type=binary),x_AF(type=binary),
     x_BA(type=binary),x_BB(type=binary),x_BC(type=binary),
     x_BD(type=binary),x_BE(type=binary),x_BF(type=binary),
     x_CA(type=binary),x_CB(type=binary),x_CC(type=binary),
     x_CD(type=binary),x_CE(type=binary),x_CF(type=binary),
     x_DA(type=binary),x_DB(type=binary),x_DC(type=binary),
     x_DD(type=binary),x_DE(type=binary),x_DF(type=binary),
     x_EA(type=binary),x_EB(type=binary),x_EC(type=binary),
     x_ED(type=binary),x_EE(type=binary),x_EF(type=binary),
     x_FA(type=binary),x_FB(type=binary),x_FC(type=binary),
     x_FD(type=binary),x_FE(type=binary),x_FF(type=binary);
//重み付き総移動距離の最小化
Objective total_distance(type=minimize);
total_distance=800*(7*x_AB+2*x_AC+13*x_AD+19*x_AE+17*x_AF)
     +550*(7*x_BA+8*x_BC+6*x_BD+12*x_BE+10*x_BF)
     +780*(2*x_CA+8*x_CB+14*x_CD+20*x_CE+18*x_CF)
     +600*(13*x_DA+6*x_DB+14*x_DC+7*x_DE+4*x_DF)
     +1020*(19*x_EA+12*x_EB+20*x_EC+7*x_ED+3*x_EF)
     +360*(17*x_FA+10*x_FB+18*x_FC+4*x_FD+3*x_FE);
//２都市に出店する
x_AA+x_BB+x_CC+x_DD+x_EE+x_FF==2;
//各都市を配分する
x_AA+x_AB+x_AC+x_AD+x_AE+x_AF==1;
x_BA+x_BB+x_BC+x_BD+x_BE+x_BF==1;
x_CA+x_CB+x_CC+x_CD+x_CE+x_CF==1;
x_DA+x_DB+x_DC+x_DD+x_DE+x_DF==1;
x_EA+x_EB+x_EC+x_ED+x_EE+x_EF==1;
x_FA+x_FB+x_FC+x_FD+x_FE+x_FF==1;
//出店しない都市には配分しない
x_AB<=x_BB; x_AC<=x_CC; x_AD<=x_DD; x_AE<=x_EE; x_AF<=x_FF;
x_BA<=x_AA; x_BC<=x_CC; x_BD<=x_DD; x_BE<=x_EE; x_BF<=x_FF;
x_CA<=x_AA; x_CB<=x_BB; x_CD<=x_DD; x_CE<=x_EE; x_CF<=x_FF;
x_DA<=x_AA; x_DB<=x_BB; x_DC<=x_CC; x_DE<=x_EE; x_DF<=x_FF;
x_EA<=x_AA; x_EB<=x_BB; x_EC<=x_CC; x_ED<=x_DD; x_EF<=x_FF;
x_FA<=x_AA; x_FB<=x_BB; x_FC<=x_CC; x_FD<=x_DD; x_FE<=x_EE;
```

ところで, この記述ですと都市の数が増えた場合に記述が大変であるなどの理由で汎用的ではありません．そこで，モデルファイルを汎用的なものにしていくことにします．そのためには, 生データをできるだけ外部から与える必要があります．ここでは, 人口と距離に関するデータをデータファイルから与えることにします．この時, 様々なデータに対応できるように都市の集合という概念を導入しておきます．すると，定式化について次のように書き直せます．

集合	$City = \{A, B, C, D, E, F\}$	都市の集合
変数	$x_{ij}, i \in City, j \in City$	0 または 1 をとる出店するのかとどこに配分されるのかを表す変数
定数	$distance_{ij}, i \in City, j \in City$	都市 i から都市 j までの距離
	$population_i, i \in City$	都市 i の人口
目的関数（最小化）	$\sum_{i \in City} \left(population_i \cdot \sum_{j \in City, i \neq j} (distance_{ij} \cdot x_{ij}) \right)$	重み付き総移動距離
制約条件	$\sum_{i \in City} x_{ii} = 2$	2 都市に出店
	$\sum_{j \in City} x_{ij} = 1$	各都市は 1 箇所に配分
	$x_{ij} \leq x_{jj}, i \in City, j \in City, i \neq j$	出店している都市に配分

上記の式を SIMPLE で記述すると，次のようになります．なお，NUOPT での実行時には都市の集合の具体的な要素はデータファイルから自動的に認識します．また，simple_printf() を用い出店する都市のみを表示するように工夫しました．

```
//都市の集合の宣言
Set City;
Element i(set=City);
Element j(set=City);
//パラメータの宣言
Parameter distance(name="2 都市間の距離",index=(City,City));
Parameter population(name="都市の人口",index=City);
//変数の宣言
IntegerVariable x(index=(City,City),type=binary);
//目的関数の宣言
Objective total_distance(name="総移動距離",type=minimize);
total_distance=sum(population[i]*sum(distance[i,j]*x[i,j],(j,i!=j
)),i);
//制約条件
sum(x[i,i],i)==2;
sum(x[i,j],j)==1;
x[i,j]<=x[j,j],i!=j;
//求解し出店する都市を出力する
solve();
simple_printf("%s 市に出店する．¥n",i,x[i,i].val==1);
```

なお，実行時に次の 2 都市間の距離を表す csv ファイル（左）と都市の人口を表す dat ファイル（右）を与えます．

```
2 都市間の距離,A,B,C,D,E,F
A,0,7,2,13,19,17
B,7,0,8,6,12,10
C,2,8,0,14,20,18
D,13,6,14,0,7,4
E,19,12,20,7,0,3
F,17,10,18,4,3,0
```

```
都市の人口=
["A"]800
["B"]550
["C"]780
["D"]600
["E"]1020
["F"]360;
```

このモデルを実行すると次の表示がされ，A 市と E 市に出店すると良いことがわかります．

```
"A" 市に出店する．
"E" 市に出店する．
```

5.10 p センター問題

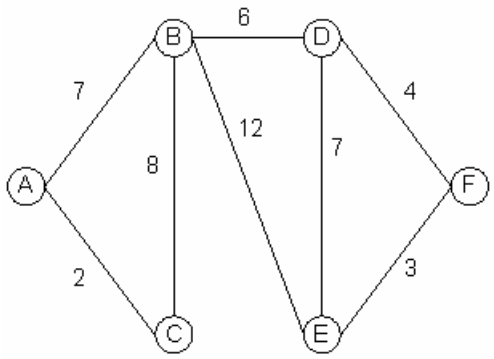
前節の p メディアン問題は，需要点と施設点の間の（重み付き）総移動距離を最小にするという目的で作られた問題でした．このため，需要点から施設点までの平均的な移動距離を最小にするような解を期待できます．しかし，解の中に需要点と施設点の間が極端に離れている組み合わせが含まれている可能性があります．この点，この節で扱う p センター問題は，（重み付き）移動距離が極端に大きい組み合わせを作らないように配置をする問題と見ることができます．

ここで，センターとは最も遠い需要点からの（重み付き）距離を最小にする施設点のことで，この点を求める問題のことをセンター問題といいます．また，p 箇所に施設を配置するような場合に p センター問題と呼び，消防署のような施設を配置するような場合に用いられています．

（例題）

ある企業は，A 市から F 市までの 6 市の中から 2 つの市に新たにデパートを出店しようと考えています．この時，都市から出店地までの重み付き移動距離の最大値を最小にするためにはどの市に出店すると良いでしょうか．なお，各市の人口は左下の表のようになっています．また，2 市間の距離は右下の図のようになっています．なお，直接結ばれていない 2 都市間の距離は他の都市を経由したときの最短距離を採用します．

都市	人口
A 市	800
B 市	550
C 市	780
D 市	600
E 市	1020
F 市	360



前節の例題との違いは目的関数の部分だけです．このため，前節の例題での変数と制約条件についてはこの例題でもそのまま使用しますので詳細は前節を参照してください．

目的関数については，都市から出店地までの重み付き移動距離の最大値ということになります．まず，都市 i から都市 j までの重み付き移動距離を求めます．これは $x_{ij}=1$ の時に意味を持つものですから i-j 間の距離，i の人口そして x_{ij} の 3 つを掛け合わせたものとなります．なお，$i=j$ の時にはこの積は常に 0 になりますので計算には含めないことにします．次に，今回のようなミニマックス問題の定式化での一つのテクニックとして変数 v を新たに導入します．この時，重み付き移動距離がそれぞれ v 以下であるという制約条件を加えることで v に重み付き移

動距離の最大値という意味を持たせることができます．よって，変数 v 自身が目的関数となります．

以上のことから，次のように定式化できます．

変数

$x_{AA}, x_{AB}, x_{AC}, x_{AD}, x_{AE}, x_{AF}, x_{BA}, x_{BB}, x_{BC}, x_{BD}, x_{BE}, x_{BF}, x_{CA}, x_{CB}, x_{CC}, x_{CD}, x_{CE}, x_{CF},$
$x_{DA}, x_{DB}, x_{DC}, x_{DD}, x_{DE}, x_{DF}, x_{EA}, x_{EB}, x_{EC}, x_{ED}, x_{EE}, x_{EF}, x_{FA}, x_{FB}, x_{FC}, x_{FD}, x_{FE}, x_{FF}$

0 または 1 をとる出店するのかと最も近い出店地はどこなのかを表す変数

v 重み付き移動距離の最大値

目的関数（最小化）

v 重み付き移動距離の最大値

制約条件

$v \geq 800 \times 7x_{AB}, v \geq 800 \times 2x_{AC}, v \geq 800 \times 13x_{AD}, v \geq 800 \times 19x_{AE}, v \geq 800 \times 17x_{AF},$
$v \geq 550 \times 7x_{BA}, v \geq 550 \times 8x_{BC}, v \geq 550 \times 6x_{BD}, v \geq 550 \times 12x_{BE}, v \geq 550 \times 10x_{BF},$
$v \geq 780 \times 2x_{CA}, v \geq 780 \times 8x_{CB}, v \geq 780 \times 14x_{CD}, v \geq 780 \times 20x_{CE}, v \geq 780 \times 18x_{CF},$
$v \geq 600 \times 13x_{DA}, v \geq 600 \times 6x_{DB}, v \geq 600 \times 14x_{DC}, v \geq 600 \times 7x_{DE}, v \geq 600 \times 4x_{DF},$
$v \geq 1020 \times 19x_{EA}, v \geq 1020 \times 12x_{EB}, v \geq 1020 \times 20x_{EC}, v \geq 1020 \times 7x_{ED}, v \geq 1020 \times 3x_{EF},$
$v \geq 360 \times 17x_{FA}, v \geq 360 \times 10x_{FB}, v \geq 360 \times 18x_{FC}, v \geq 360 \times 4x_{FD}, v \geq 360 \times 3x_{FE}$

各重み付き移動距離は最大値を越えない

$x_{AA} + x_{BB} + x_{CC} + x_{DD} + x_{EE} + x_{FF} = 2$ 2 都市に出店

$x_{AA} + x_{AB} + x_{AC} + x_{AD} + x_{AE} + x_{AF} = 1$ A 市は 1 箇所に配分

$x_{BA} + x_{BB} + x_{BC} + x_{BD} + x_{BE} + x_{BF} = 1$ B 市は 1 箇所に配分

$x_{CA} + x_{CB} + x_{CC} + x_{CD} + x_{CE} + x_{CF} = 1$ C 市は 1 箇所に配分

$x_{DA} + x_{DB} + x_{DC} + x_{DD} + x_{DE} + x_{DF} = 1$ D 市は 1 箇所に配分

$x_{EA} + x_{EB} + x_{EC} + x_{ED} + x_{EE} + x_{EF} = 1$ E 市は 1 箇所に配分

$x_{FA} + x_{FB} + x_{FC} + x_{FD} + x_{FE} + x_{FF} = 1$ F 市は 1 箇所に配分

$x_{AB} \leq x_{BB}, x_{AC} \leq x_{CC}, x_{AD} \leq x_{DD}, x_{AE} \leq x_{EE}, x_{AF} \leq x_{FF},$
$x_{BA} \leq x_{AA}, x_{BC} \leq x_{CC}, x_{BD} \leq x_{DD}, x_{BE} \leq x_{EE}, x_{BF} \leq x_{FF},$
$x_{CA} \leq x_{AA}, x_{CB} \leq x_{BB}, x_{CD} \leq x_{DD}, x_{CE} \leq x_{EE}, x_{CF} \leq x_{FF},$
$x_{DA} \leq x_{AA}, x_{DB} \leq x_{BB}, x_{DC} \leq x_{CC}, x_{DE} \leq x_{EE}, x_{DF} \leq x_{FF},$
$x_{EA} \leq x_{AA}, x_{EB} \leq x_{BB}, x_{EC} \leq x_{CC}, x_{ED} \leq x_{DD}, x_{EF} \leq x_{FF},$
$x_{FA} \leq x_{AA}, x_{FB} \leq x_{BB}, x_{FC} \leq x_{CC}, x_{FD} \leq x_{DD}, x_{FE} \leq x_{EE}$

出店している都市に配分

この結果を SIMPLE で記述すると次のようになります．

```
//変数の宣言
IntegerVariable
     x_AA(type=binary),x_AB(type=binary),x_AC(type=binary),
     x_AD(type=binary),x_AE(type=binary),x_AF(type=binary),
     x_BA(type=binary),x_BB(type=binary),x_BC(type=binary),
     x_BD(type=binary),x_BE(type=binary),x_BF(type=binary),
     x_CA(type=binary),x_CB(type=binary),x_CC(type=binary),
     x_CD(type=binary),x_CE(type=binary),x_CF(type=binary),
     x_DA(type=binary),x_DB(type=binary),x_DC(type=binary),
     x_DD(type=binary),x_DE(type=binary),x_DF(type=binary),
     x_EA(type=binary),x_EB(type=binary),x_EC(type=binary),
     x_ED(type=binary),x_EE(type=binary),x_EF(type=binary),
     x_FA(type=binary),x_FB(type=binary),x_FC(type=binary),
     x_FD(type=binary),x_FE(type=binary),x_FF(type=binary);
Variable v;
//重み付き移動距離の最大値の最小化
Objective max_distance(type=minimize);
max_distance=v;
v>=800*7*x_AB;  v>=800*2*x_AC;  v>=800*13*x_AD;  v>=800*19*x_AE;
v>=800*17*x_AF; v>=550*7*x_BA;  v>=550*8*x_BC;   v>=550*6*x_BD;
v>=550*12*x_BE; v>=550*10*x_BF; v>=780*2*x_CA;   v>=780*8*x_CB;
v>=780*14*x_CD; v>=780*20*x_CE; v>=780*18*x_CF; v>=600*13*x_DA;
v>=600*6*x_DB;  v>=600*14*x_DC; v>=600*7*x_DE;   v>=600*4*x_DF;
v>=1020*19*x_EA;v>=1020*12*x_EB;v>=1020*20*x_EC;v>=1020*7*x_ED;
v>=1020*3*x_EF; v>=360*17*x_FA; v>=360*10*x_FB; v>=360*18*x_FC;
v>=360*4*x_FD;  v>=360*3*x_FE;
//出店と配分に関する制約条件
x_AA+x_BB+x_CC+x_DD+x_EE+x_FF==2;
x_AA+x_AB+x_AC+x_AD+x_AE+x_AF==1;
x_BA+x_BB+x_BC+x_BD+x_BE+x_BF==1;
x_CA+x_CB+x_CC+x_CD+x_CE+x_CF==1;
x_DA+x_DB+x_DC+x_DD+x_DE+x_DF==1;
x_EA+x_EB+x_EC+x_ED+x_EE+x_EF==1;
x_FA+x_FB+x_FC+x_FD+x_FE+x_FF==1;
x_AB<=x_BB; x_AC<=x_CC; x_AD<=x_DD; x_AE<=x_EE; x_AF<=x_FF;
x_BA<=x_AA; x_BC<=x_CC; x_BD<=x_DD; x_BE<=x_EE; x_BF<=x_FF;
x_CA<=x_AA; x_CB<=x_BB; x_CD<=x_DD; x_CE<=x_EE; x_CF<=x_FF;
x_DA<=x_AA; x_DB<=x_BB; x_DC<=x_CC; x_DE<=x_EE; x_DF<=x_FF;
x_EA<=x_AA; x_EB<=x_BB; x_EC<=x_CC; x_ED<=x_DD; x_EF<=x_FF;
x_FA<=x_AA; x_FB<=x_BB; x_FC<=x_CC; x_FD<=x_DD; x_FE<=x_EE;
```

このままでは大変分かりにくいですので，前節と同様に SIMPLE での記述を簡潔なものにしていくことにします．この際，定式化については次のようなものにします．

集合	$City = \{A, B, C, D, E, F\}$	都市の集合
変数	$x_{ij}, i \in City, j \in City$	0または1をとる出店するのかと最も近い出店地はどこなのかを表す変数
	v	重み付き移動距離の最大値
定数	$distance_{ij}, i \in City, j \in City$	都市 i から都市 j までの距離
	$population_i, i \in City$	都市 i の人口
目的関数（最小化）	v	重み付き移動距離の最大値
制約条件	$v \geq population_i \cdot distance_{ij} \cdot x_{ij}, i \in City, j \in City, i \neq j$	各重み付き移動距離は最大値を越えない
	$\sum_{i \in City} x_{ii} = 2$	2 都市に出店
	$\sum_{j \in City} x_{ij} = 1$	各都市は 1 箇所に配分
	$x_{ij} \leq x_{jj}, i \in City, j \in City, i \neq j$	出店している都市に配分

すると，SIMPLE では次のような簡潔な記述になります．

```
//都市の集合の宣言
Set City;
Element i(set=City);
Element j(set=City);
//パラメータの宣言
Parameter distance(name="２都市間の距離",index=(City,City));
Parameter population(name="都市の人口",index=City);
//変数の宣言
IntegerVariable x(index=(City,City),type=binary);
Variable v;
//目的関数の宣言
Objective max_distance(type=minimize);
max_distance=v;
//制約条件
v>=population[i]*distance[i,j]*x[i,j];
sum(x[i,i],i)==2;
sum(x[i,j],j)==1;
x[i,j]<=x[j,j],i!=j;
//求解し出店する都市を出力する
solve();
simple_printf("%s 市に出店する．¥n",i,x[i,i].val==1);
```

実行する際には，前節の例題と同様に次の 2 都市間の距離を表す csv ファイル（左）と各都市の人口を表す dat ファイル（右）を与えます．

```
２都市間の距離,A,B,C,D,E,F
A,0,7,2,13,19,17
B,7,0,8,6,12,10
C,2,8,0,14,20,18
D,13,6,14,0,7,4
E,19,12,20,7,0,3
F,17,10,18,4,3,0
```

```
都市の人口=
["A"]800
["B"]550
["C"]780
["D"]600
["E"]1020
["F"]360;
```

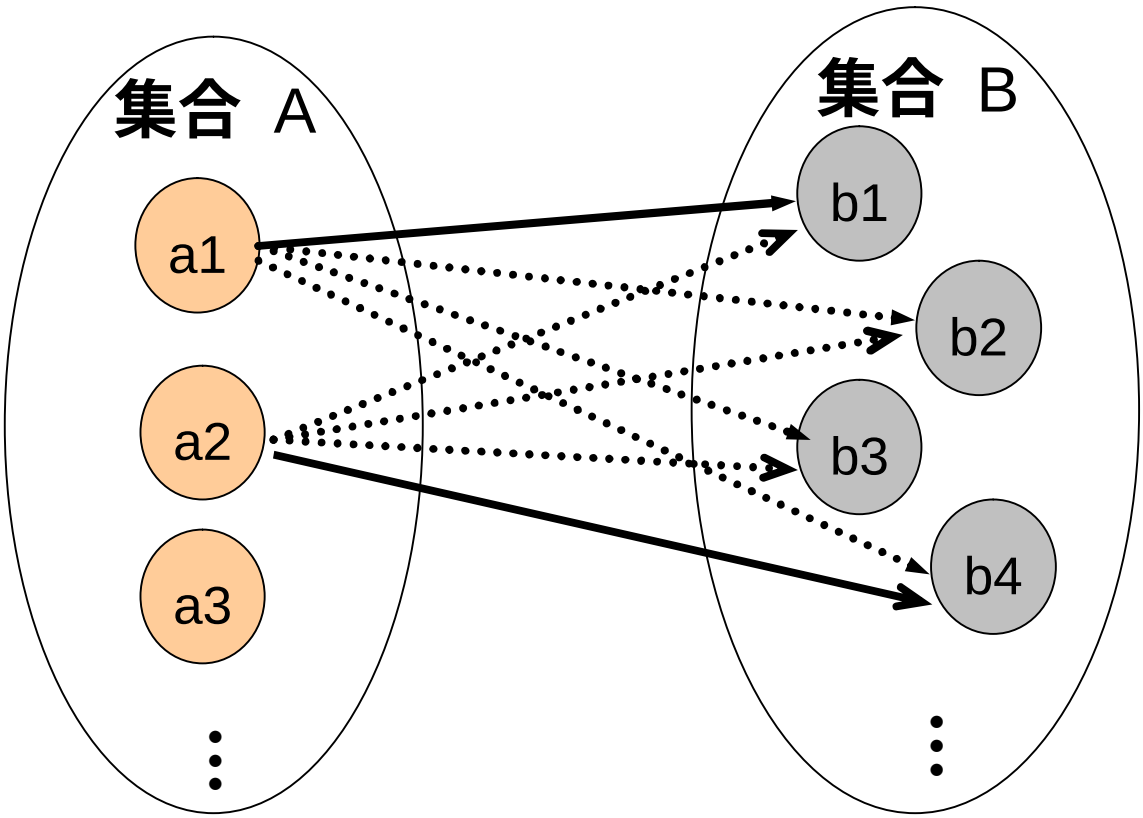
このモデルを実行すると，最後に次の表示がされ，前節の p メディアン問題の時の結果とは異なり，A 市と F 市に出店すると良いことがわかります．

5.11 割り当て問題

割り当て問題は現在多くの業務に利用されています. ここでは基本的な割り当て問題の例題を取り上げ, 割り当て問題の定式化のテクニック, モデル化の例, 大規模な問題に対するアプローチ法を紹介します.

5.11.1 割り当て問題とは

割り当て問題とは, 集合 A の要素を集合 B の要素のどれに割り当てるかを決定する問題です.



また, 割り当て問題はマス目を埋める問題に置き換えることができます. 上記の図を, マス目を埋めるイメージで表すと以下の図のようになります.

	a1	a2	a3		
b1	○				
b2			○		
b3					
b4		○			

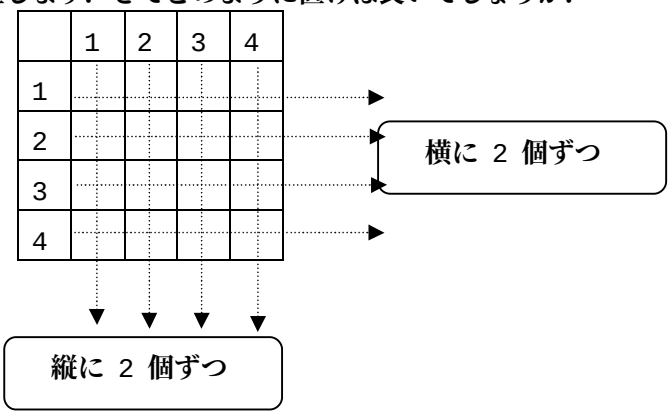
割り当て問題はマスの埋め方のルールを制約条件として与えることによって, 個々の問題に対応する定式化を行うことができます. 割り当て問題の応用例として, 例えば以下のものがあります.

- 人員配置問題
- シフトスケジューリング問題
- 配車計画問題
- 訪問スケジューリング決定問題
- マーケットテリトリー決定問題

5.11.2 基礎的なマス埋め割り当て問題

ここではまず, マス目を埋めるだけの問題を考えていきます.

（例題）
４×４のマス目があります. このマスに石を置きたいのですが, 全ての縦横に関して石の数が２つになるように配置します. さてどのように置けば良いでしょうか.



この問題を定式化するためには, 0-1 変数を用いる必要があります. この問題の場合以下の

図のように，各マスに対して 0-1 変数を対応させます.

	1	2	3	4
1	x11	x12	x13	X14
2	x21	x22	x23	X24
3	x31	x32	x33	X34
4	x41	x42	x43	X44

定式化をすると以下のようになります.

0-1 変数	$x_{11}, x_{12}, x_{13}, x_{14}$ $x_{21}, x_{22}, x_{23}, x_{24}$ $x_{31}, x_{32}, x_{33}, x_{34}$ $x_{41}, x_{42}, x_{43}, x_{44}$	それぞれのマスに石を置くならば 1 置かないならば 0 .
目的関数		この問題には目的関数はない
制約条件	$x_{11} + x_{12} + x_{13} + x_{14} = 2$	1行目を横に見たときに石を2つ置く
	$x_{21} + x_{22} + x_{23} + x_{24} = 2$	2行目を横に見たときに石を2つ置く
	$x_{31} + x_{32} + x_{33} + x_{34} = 2$	3行目を横に見たときに石を2つ置く
	$x_{41} + x_{42} + x_{43} + x_{44} = 2$	4行目を横に見たときに石を2つ置く
	$x_{11} + x_{21} + x_{31} + x_{41} = 2$	1列目を縦に見たときに石を2つ置く
	$x_{12} + x_{22} + x_{32} + x_{42} = 2$	2列目を縦に見たときに石を2つ置く
	$x_{13} + x_{23} + x_{33} + x_{43} = 2$	3列目を縦に見たときに石を2つ置く
	$x_{14} + x_{24} + x_{34} + x_{44} = 2$	4列目を縦に見たときに石を2つ置く

定式化した結果を SIMPLE で記述すると以下のようになります.

```
IntegerVariable x_11(type = binary);
IntegerVariable x_21(type = binary);
IntegerVariable x_31(type = binary);
IntegerVariable x_41(type = binary);

IntegerVariable x_12(type = binary);
IntegerVariable x_22(type = binary);
IntegerVariable x_32(type = binary);
IntegerVariable x_42(type = binary);

IntegerVariable x_13(type = binary);
IntegerVariable x_23(type = binary);
IntegerVariable x_33(type = binary);
IntegerVariable x_43(type = binary);

IntegerVariable x_14(type = binary);
IntegerVariable x_24(type = binary);
IntegerVariable x_34(type = binary);
IntegerVariable x_44(type = binary);

x_11 + x_12 + x_13 + x_14 == 2;
x_21 + x_22 + x_23 + x_24 == 2;
x_31 + x_32 + x_33 + x_34 == 2;
x_41 + x_42 + x_43 + x_44 == 2;

x_11 + x_21 + x_31 + x_41 == 2;
x_12 + x_22 + x_32 + x_42 == 2;
x_13 + x_23 + x_33 + x_43 == 2;
x_14 + x_24 + x_34 + x_44 == 2;

// 以下は出力用のプログラム
solve();
simple_printf("%d %d %d %d¥n",x_11,x_12,x_13,x_14);
simple_printf("%d %d %d %d¥n",x_21,x_22,x_23,x_24);
simple_printf("%d %d %d %d¥n",x_31,x_32,x_33,x_34);
simple_printf("%d %d %d %d¥n",x_41,x_42,x_43,x_44);
```

このモデルを NUOPT で実行すると，最後に

```
 1 0 1 0
 1 1 0 0
 0 1 0 1
 0 0 1 1
```

という表示がされ，この例題の答えを確認できます．どの縦横にも二箇所ずつ石が置いてある（1 と表示されている）のが分かります．

さて，次にこの問題を SIMPLE の添え字機能を用いてモデル化してみます．定式化は以下のように変更します．

集合	$I=\{1,2,3,4\}$	行
	$J=\{1,2,3,4\}$	列
0-1 変数	$x_{ij}, i\in I, j\in J$	マス(i,j)に石を置くならば$x_{ij}=1$，置かないならば$x_{ij}=0$とする．
目的関数		この問題には目的関数はない
制約条件	$\sum_i x_{ij}=2$	各列 j は横に見たときに石を２つ置く
	$\sum_j x_{ij}=2$	各行 i は縦に見たときに石を２つ置く

添え字を用いることにより，以下のように簡単にモデル記述することができます．

```
Set I = "1 2 3 4";
Set J = "1 2 3 4";
Element i(set = I);
Element j(set = J);
IntegerVariable x(type = binary,index = (i,j));
sum(x[i,j],i) == 2;
sum(x[i,j],j) == 2;

// 以下は出力用のプログラム
solve();
simple_printf("%d %d %d %d¥n",
x[i,j],x[i,j+1],x[i,j+2],x[i,j+3],j == 1);
```

5.11.3 仕事割り当て問題

ここでは，仕事を人に効率よく割り当てる問題を取り上げます．次の例題を考えます．

（例題）「安藤」「佐藤」「鈴木」「山本」「渡辺」 の 5 人に仕事を割り当てます．仕事は 「接客」「厨房」「レジ打ち」「発注」「ごみ捨て」「買出し」「掃除」「仕込み」の６つです．各人を仕事に割り当てるにはコストがかかり，それは個人・仕事によってきまります．また，各人はそれぞれの仕事に対して熟練度があり，熟練度が高いほどコストがかかる傾向があります．以下は熟練度とコストをまとめたものです．

熟練度	安藤	佐藤	鈴木	山本	渡辺
接客	-1	3	-2	3	-4
厨房	5	-2	3	-4	5
レジ打ち	0	3	-2	3	-1
発注	-3	-1	1	1	2
ごみ捨て	1	-1	3	1	-1
買出し	-1	-1	1	0	2
掃除	2	-2	2	-3	4
仕込み	5	-2	0	1	5

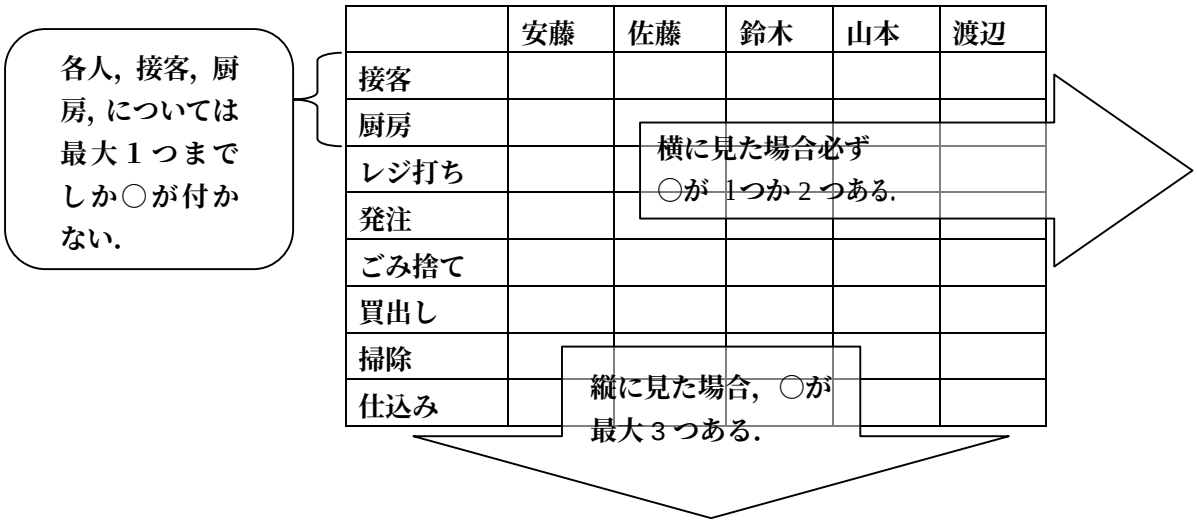
コスト	安藤	佐藤	鈴木	山本	渡辺
接客	570	1400	520	1410	450
厨房	1800	1000	1700	1050	2300
レジ打ち	800	1500	500	1500	600
発注	500	600	1000	1000	1200
ごみ捨て	800	600	1200	800	600
買出し	600	600	800	700	1300
掃除	1200	500	1200	500	1300
仕込み	1500	1000	1200	1200	1500

また，以下の点を守らなくてはいけません．

- 各人に割り振る仕事は，最大で３つまでとする．
- 「接客」「厨房」「レジ打ち」「掃除」「仕込み」は２人を割り当てる．
- 「発注」「ごみ捨て」「買出し」は１人を割り当てる
- 「接客」「厨房」は別の人が担当する．
- 仕事を担当する人の熟練度の和を仕事のクオリティーとする．
- クオリティーは負になってはいけない．

このとき，コストの合計を最小にするような割り当て方を求めてください．

この問題も前節の問題と同様に以下のようなマス目に○をつける問題として考えることができます.



以上を踏まえて，以下のように定式化することができます．

集合	JOB＝｛接客，厨房，レジ打ち，発注，ごみ捨て，買出し，掃除，仕込み｝ 仕事の集合	
	$PEOPLE$＝｛安藤，佐藤，鈴木，山本，渡辺｝ 人の集合	
0-1 変数	$x_{jp}, j \in JOB, p \in PEOPLE$	仕事 j を人 p に割り当てるならば $x_{jp}=1$， そうでないならば $x_{jp}=0$ とする
定数	$cost_{jp}$	仕事 j を人 p に割り当てる際のコスト
	$jyukuren_{jp}$	仕事 j を人 p が行う際の熟練度
目的関数（最小化）	$\sum_{j,p} cost_{jp} \times x_{jp}$	コストの総和
制約条件	$\sum_{p} x_{jp} = 2, j \in$ ｛接客，厨房，レジ打ち，掃除，仕込み｝	
	「接客」「厨房」「レジ打ち」「掃除」「仕込み」は２人を割り当てる．	
	$\sum_{p} x_{jp} = 1, j \in$ ｛発注，ごみ捨て，買出し｝	
	「発注」「ごみ捨て」「買出し」は１人を割り当てる	
	$\sum_{j} x_{jp} \le 3$	各人には，最大３つまでの仕事を割り当てることができる
	$\sum_{p} jyukuren_{jp} \times x_{jp} \ge 0$	各仕事のクオリティーが負になってはいけない
	$\sum_{j, j \in MAINJOB} x_{jp} = 1, MAINJOB =$ ｛接客，厨房｝	
	$\sum_{j, j \in MAINJOB, (j,p) \in JPPari} x_{jp} = 1, MAINJOB =$ ｛接客，厨房｝	
	接客，厨房は違う人が担当する（同じ人が接客と厨房を兼ねない）．	

今回は定数に関してはファイルからデータを与えてみます. モデル部分は以下のようになります.

```
Set Job;
Set People;
Element j(set = Job);
Element p(set = People);

IntegerVariable x(type = binary,index = (j,p));
Parameter cost(index = (j,p),name = "コスト");
Parameter jyukuren(index = (j,p),name = "熟練度");

sum(x[j,p],p) == 2,
  j == "接客" || j == "厨房" || j == "レジ打ち" ||
  j == "掃除" || j == "仕込み";
sum(x[j,p],p) == 1,
  j == "発注" || j == "ごみ捨て" || j == "買出し" ;
sum(x[j,p],j) <= 3;
sum(x[j,p]*jyukuren[j,p],p) >= 0;
sum(x[j,p],(j, j == "接客" || j == "厨房")) <= 1;

Objective total_cost(type = minimize,name = "総コスト");
total_cost = sum(cost[j,p]*x[j,p],(j,p));

// 以下出力用プログラム
solve();
total_cost.val.print();
simple_printf("%s,%s¥n",j,p,x[j,p].val == 1);
simple_printf("¥n");
simple_printf("%s のクオリティーは  %d¥n",
j,sum(jyukuren[j,p]*x[j,p].val,p));
```

以下が csv 形式の入力ファイルです. コストに関してと熟練度に関しての二種類のファイルを用意する必要があります

```
熟練度,安藤,佐藤,鈴木,山本,渡辺
接客,-1,3,-2,3,-4
厨房,5,-2,3,-4,5
レジ打ち,0,3,-2,3,-1
発注,-3,-1,1,1,2
ごみ捨て,1,-1,3,1,-1
買出し,-1,-1,1,0,2
掃除,2,-2,2,-3,4
仕込み,5,-2,0,1,5
```

```
コスト,安藤,佐藤,鈴木,山本,渡辺
接客,570,1400,520,1410,450
厨房,1800,1000,1700,1050,2300
レジ打ち,800,1500,500,1500,600
発注,500,600,1000,1000,1200
ごみ捨て,800,600,1200,800,600
買出し,600,600,800,700,1300
掃除,1200,500,1200,500,1300
仕込み,1500,1000,1200,1200,1500
```

このモデルを NUOPT で実行すると，最後に

```
総コスト=13380
"ごみ捨て","安藤"
"レジ打ち","佐藤"
"レジ打ち","渡辺"
"仕込み","山本"
"仕込み","鈴木"
"厨房","佐藤"
"厨房","鈴木"
"接客","安藤"
"接客","山本"
"掃除","安藤"
"掃除","佐藤"
"買出し","山本"
"発注","鈴木"

"ごみ捨て"のクオリティーは  1
"レジ打ち"のクオリティーは  2
"仕込み"のクオリティーは  1
"厨房"のクオリティーは  1
"接客"のクオリティーは  2
"掃除"のクオリティーは  0
"買出し"のクオリティーは  0
"発注"のクオリティーは  1
```

という表示がされ，この例題の答えを確認できます.

（例題）先ほどの問題に以下の条件を付け加えて下さい.

安藤に「接客」をさせることはできない
佐藤に「厨房」をさせることはできない
鈴木に「レジ打ち」をさせることはできない
山本に「発注」をさせることはできない
渡辺に「ごみ捨て」をさせることはできない

さて, この問題ですが通常に考えると先ほどの問題に以下の制約を付け加えることで解くことができます．また記述する位置は x の宣言以降，solve() の手前であればどこでもかまいません.

```
x["接客,安藤"] == 0;
x["厨房,佐藤"] == 0;
x["レジ打ち,鈴木"] == 0;
x["発注,山本"] == 0;
x["ごみ捨て,渡辺"] == 0;
```

上記記述を追加し，モデルを実行すると結果は以下のように変わります.

```
総コスト=13470
"ごみ捨て","安藤"
"レジ打ち","佐藤"
"レジ打ち","渡辺"
"仕込み","山本"
"仕込み","鈴木"
"厨房","安藤"
"厨房","山本"
"接客","佐藤"
"接客","鈴木"
"掃除","安藤"
"掃除","佐藤"
"買出し","山本"
"発注","鈴木"

"ごみ捨て"のクオリティーは  1
"レジ打ち"のクオリティーは  2
"仕込み"のクオリティーは  1
"厨房"のクオリティーは  1
"接客"のクオリティーは  1
"掃除"のクオリティーは  0
"買出し"のクオリティーは  0
"発注"のクオリティーは  1
```

結果を見ると，総コストが多くなっていることが分かります.

以上の考え方は以下の図のように変数を準備し，色が付いているところが 0 であるという制約を設けたと考えることができます．（なお図中のレジはレジ打ち，ごみはごみ捨て，買出は買出し，仕込は仕込みです）

	安藤	佐藤	鈴木	山本	渡辺
接客	x[接客，安藤]	x[接客，佐藤]	x[接客，鈴木]	x[接客，山本]	x[接客，渡辺]
厨房	x[厨房，安藤]	x[厨房，佐藤]	x[厨房，鈴木]	x[厨房，山本]	x[厨房，渡辺]
レジ打ち	x[レジ，安藤]	x[レジ，佐藤]	x[レジ，鈴木]	x[レジ，山本]	x[レジ，渡辺]
発注	x[発注，安藤]	x[発注，佐藤]	x[発注，鈴木]	x[発注，山本]	x[発注，渡辺]
ごみ捨て	x[ごみ，安藤]	x[ごみ，佐藤]	x[ごみ，鈴木]	x[ごみ，山本]	x[ごみ，渡辺]
買出し	x[買出，安藤]	x[買出，佐藤]	x[買出，鈴木]	x[買出，山本]	x[買出，渡辺]
掃除	x[掃除，安藤]	x[掃除，佐藤]	x[掃除，鈴木]	x[掃除，山本]	x[掃除，渡辺]
仕込み	x[仕込，安藤]	x[仕込，佐藤]	x[仕込，鈴木]	x[仕込，山本]	x[仕込，渡辺]

ここで，上の図の色のついている部分は 1 になることがないので，はじめから変数を準備する必要がないと考えることが出来ます．特に大規模の割り当て問題においては，無駄な変数を準備することにより不用意に問題の規模を大きくしてしまうと，計算のパフォーマンスを著しく低下させてしまう要因になります．ここでは明らかな制約に関しては「制約を付け加えることなく，そのような選択肢を元々準備しない」という方法を紹介します．以下の図のようなイメージです．

	安藤	佐藤	鈴木	山本	渡辺
接客		x[接客，佐藤]	x[接客，鈴木]	x[接客，山本]	x[接客，渡辺]
厨房	x[厨房，安藤]		x[厨房，鈴木]	x[厨房，山本]	x[厨房，渡辺]
レジ打ち	x[レジ，安藤]	x[レジ，佐藤]		x[レジ，山本]	x[レジ，渡辺]
発注	x[発注，安藤]	x[発注，佐藤]	x[発注，鈴木]		x[発注，渡辺]
ごみ捨て	x[ごみ，安藤]	x[ごみ，佐藤]	x[ごみ，鈴木]	x[ごみ，山本]	
買出し	x[買出，安藤]	x[買出，佐藤]	x[買出，鈴木]	x[買出，山本]	x[買出，渡辺]
掃除	x[掃除，安藤]	x[掃除，佐藤]	x[掃除，鈴木]	x[掃除，山本]	x[掃除，渡辺]
仕込み	x[仕込，安藤]	x[仕込，佐藤]	x[仕込，鈴木]	x[仕込，山本]	x[仕込，渡辺]

定式化する際に，上記を表す集合 *JPPair* を準備する必要があります．

- $(j, p) \in JPPair \Leftrightarrow$ j を pに割り当てることができる

以下が *JPPair* を用いた定式化です.

集合	*JOB*＝｛接客，厨房，レジ打ち，発注，ごみ捨て，買出し，掃除，仕込み｝ 仕事の集合， $j \in JOB$ *PEOPLE*＝｛安藤，佐藤，鈴木，山本，渡辺｝ 人の集合， $p \in PEOPLE$ $(j, p) \in JPPair \Leftrightarrow$ j を pに割り当てることができる	
0-1 変数	$x_{jp}, (j, p) \in JPPair$	仕事 j を人 p に割り当てるならば $x_{jp} = 1$，そうでないならば $x_{jp} = 0$ とする
定数	$cost_{jp}, (j, p) \in JPPair$	仕事 j を人 p に割り当てる際のコスト
	$jyukuren_{jp}, (j, p) \in JPPair$	仕事 j を人 p が行う際の熟練度
目的関数 （最小化）	$\sum_{j,p,(j,p)\in JPPair} cost_{jp} \times x_{jp}$ コストの総和	
制約条件	$\sum_{p,(j,p)\in JPPair} x_{jp} = 2, j \in$ ｛接客，厨房，レジ打ち，掃除，仕込み｝ 「接客」「厨房」「レジ打ち」「掃除」「仕込み」は２人を割り当てる.	
	$\sum_{p,(j,p)\in JPPair} x_{jp} = 1, j \in$ ｛発注，ごみ捨て，買出し｝ 「発注」「ごみ捨て」「買出し」は１人を割り当てる	
	$\sum_{j,(j,p)\in JPPair} x_{jp} \leq 3$ 各人には，最大３つまで仕事を割り当てることができる	
	$\sum_{p,(j,p)\in JPPair} jyukuren_{jp} \times x_{jp} \geq 0$ 各仕事のクオリティーが負になってはいけない	
	$\sum_{(j,p), j\in MAINJOB, (j,p)\in JPPair} x_{jp} = 1, MAINJOB =$ ｛接客，厨房｝ 接客，厨房は違う人が担当する.	

モデルは以下のようになります.

```
Set Job;
Set People;
Element j(set = Job);
Element p(set = People);

Set JPPair(dim = 2,superSet = (Job,People));
IntegerVariable x(type = binary,index = JPPair);
Parameter cost(index = JPPair,name = "コスト");
Parameter jyukuren(index = JPPair,name = "熟練度");

sum(x[j,p],(p,(j,p)<JPPair)) == 2,
  j == "接客" || j == "厨房" || j == "レジ打ち" ||
  j == "掃除" || j == "仕込み";
sum(x[j,p],(p,(j,p)<JPPair)) == 1,
  j == "発注" || j == "ごみ捨て" || j == "買出し" ;
sum(x[j,p],(j,(j,p)<JPPair)) <= 3;
sum(x[j,p]*jyukuren[j,p],(p,(j,p)<JPPair)) >= 0;
sum(x[j,p],(j,(j,p)<JPPair,j == "接客" || j == "厨房")) <= 1;

Objective total_cost(type = minimize,name = "総コスト");
total_cost = sum(cost[j,p]*x[j,p],(j,p,(j,p)<JPPair));

// 以下出力用プログラム
solve();
total_cost.val.print();
x.val.print();
```

上記記述内の JPPair には無駄な [j,p] のペアを入れてはいけないので，入力ファイルもそれに伴い以下のように変わります．

```
j,p,熟練度,コスト
接客,佐藤,3,1400
接客,鈴木,-2,520
接客,山本,3,1410
接客,渡辺,-4,450
厨房,安藤,5,1800
厨房,鈴木,3,1700
厨房,山本,-4,1050
厨房,渡辺,5,2300
レジ打ち,安藤,0,800
レジ打ち,佐藤,3,1500
レジ打ち,山本,3,1500
レジ打ち,渡辺,-1,600
発注,安藤,-3,500
発注,佐藤,-1,600
発注,鈴木,1,1000
発注,渡辺,2,1200
ごみ捨て,安藤,1,800
ごみ捨て,佐藤,-1,600
ごみ捨て,鈴木,3,1200
ごみ捨て,山本,1,800
買出し,安藤,-1,600
買出し,佐藤,-1,600
買出し,鈴木,1,800
買出し,山本,0,700
買出し,渡辺,2,1300
掃除,安藤,2,1200
掃除,佐藤,-2,500
掃除,鈴木,2,1200
掃除,山本,-3,500
掃除,渡辺,4,1300
仕込み,安藤,5,1500
仕込み,佐藤,-2,1000
仕込み,鈴木,0,1200
仕込み,山本,1,1200
仕込み,渡辺,5,1500
```

またこのモデルを制約充足問題ソルバ wcsp を用いる場合にはモデルを以下のように変更する必要があります.

```
options.method = "wcsp";// 解法に wcsp を用いる指定
options.maxtim = 10;// 計算時間の設定

Set Job;
Set People;
Element j(set = Job);
Element p(set = People);

Set JPPair(dim = 2,superSet = (Job,People));
IntegerVariable x(type = binary,index = JPPair);
Parameter cost(index = JPPair,name = "コスト");
Parameter jyukuren(index = JPPair,name = "熟練度");

sum(x[j,p],(p,(j,p)<JPPair)) == 2,
  j == "接客" || j == "厨房" ||
  j == "レジ打ち" || j == "掃除" || j == "仕込み";

// wcsp 特有の関数 selection
selection(x[j,p],(p,(j,p)<JPPair)),
  j == "発注" || j == "ごみ捨て" || j == "買出し";

sum(x[j,p],(j,(j,p)<JPPair)) <= 3;
sum(x[j,p]*jyukuren[j,p],(p,(j,p)<JPPair)) >= 0;
sum(x[j,p],(j,(j,p)<JPPair,j == "接客" || j == "厨房")) <= 1;

Objective total_cost(type = minimize,name = "総コスト");
total_cost = sum(cost[j,p]*x[j,p],(j,p,(j,p)<JPPair));

// 以下出力用プログラム
solve();
total_cost.val.print();
x.val.print();
```

大規模な割り当て問題を解く際の問題規模に対する対応として上記で紹介した手法を試してみてください.

5.12 最小二乗問題

最小二乗問題は，科学や工学の分野でよく利用される基本的な問題です．この問題の典型的な例としては，曲線の当てはめがあります．実験により得られたデータをモデル曲線に当てはめ定数を推定していくことを考えます．すると，一般に各観測点においてデータとモデルの間には誤差が発生してしまいます．この時，「各観測点における誤差の二乗和を最小にする」という基準を採用しなるべく良い曲線に当てはめようとすると最小二乗問題となります．

（例題）

A さんは，実験開始から t 秒後の y という値を計測するという実験を行いました．その結果をまとめると，下の表のようになりました．A さんはこの結果をプロットしたところ，$f(t)=at^2+bt+c$ という二次関数モデルに当てはめられることに気付きました．そこで，各計測時刻でのデータ値とモデルから得られる値との誤差の二乗和が最小になるように a,b,c を推定してください．

t	y	t	y
0.5	1.22470	3.0	3.22933
1.0	0.87334	3.5	4.44744
1.5	0.99577	4.0	6.13509
2.0	1.34215	4.5	8.14697
2.5	2.12172	5.0	10.50759

この例題において，変数となるのは推定する a,b,c の 3 つです．また，ある計測時刻 t でのデータ値 $y(t)$ とモデルから得られる値 $f(t)$ との誤差は $at^2+bt+c-y(t)$ となります．よって，各計測時刻について誤差の二乗を求め，その総和をとることで最小化する目的関数が得られます．

以上のことから，この例題は次のように定式化できます.

変数	a	**2 次の項の係数**
	b	**1 次の項の係数**
	c	**定数項**
目的関数（最小化）	$(0.5^2a+0.5b+c-1.22470)^2+(1.0^2a+1.0b+c-0.87334)^2$ $+(1.5^2a+1.5b+c-0.99577)^2+(2.0^2a+2.0b+c-1.34215)^2$ $+(2.5^2a+2.5b+c-2.12172)^2+(3.0^2a+3.0b+c-3.22933)^2$ $+(3.5^2a+3.5b+c-4.44744)^2+(4.0^2a+4.0b+c-6.13509)^2$ $+(4.5^2a+4.5b+c-8.14697)^2+(5.0^2a+5.0b+c-10.50759)^2$	
		各計測時刻における誤差の二乗和

ここで，この結果を SIMPLE で何も工夫せずに記述すると次のようになります.

```
//変数の宣言
Variable a(name="２次の項の係数");
Variable b(name="１次の項の係数");
Variable c(name="定数項");
//誤差の二乗和の最小化
Objective err(type=minimize);
err=(0.5*0.5*a+0.5*b+c-1.2247)*(0.5*0.5*a+0.5*b+c-1.2247)
+(1.0*1.0*a+1.0*b+c-0.87334)*(1.0*1.0*a+1.0*b+c-0.87334)
+(1.5*1.5*a+1.5*b+c-0.99577)*(1.5*1.5*a+1.5*b+c-0.99577)
+(2.0*2.0*a+2.0*b+c-1.34215)*(2.0*2.0*a+2.0*b+c-1.34215)
+(2.5*2.5*a+2.5*b+c-2.12172)*(2.5*2.5*a+2.5*b+c-2.12172)
+(3.0*3.0*a+3.0*b+c-3.22933)*(3.0*3.0*a+3.0*b+c-3.22933)
+(3.5*3.5*a+3.5*b+c-4.44744)*(3.5*3.5*a+3.5*b+c-4.44744)
+(4.0*4.0*a+4.0*b+c-6.13509)*(4.0*4.0*a+4.0*b+c-6.13509)
+(4.5*4.5*a+4.5*b+c-8.14697)*(4.5*4.5*a+4.5*b+c-8.14697)
+(5.0*5.0*a+5.0*b+c-10.50759)*(5.0*5.0*a+5.0*b+c-10.50759);
//求解し結果を表示する
solve();
a.val.print();
b.val.print();
c.val.print();
```

これでは, 何回も実験を行うような場合モデルファイルの修正に大変な手間がかかってしまい効率的ではありません. そこで, SIMPLE の特長を生かし様々な測定結果に対応できるように定式化から見直していきます.

まず, ある計測時刻 t でのデータ値 $y(t)$ とモデルから得られる値 $f(t)$ との誤差を $e(t)$ と表すことにします. すると, 先ほどの議論から $e(t) = at^2 + bt + c - y(t)$ となります. これより, 目的関数は $e(t)^2$ を t について和をとったものと言えます. SIMPLE においては, Expression を用いることで数式を宣言できます. よって, $e(t)$ を Expression で記述することでモデルファイルをより簡潔なものにできます. また, 二乗和を求める部分についても観測時間の集合を導入することで sum()を用い簡単に記述できます. 最後に, 各計測時刻 t における $y(t)$ の値は直接モデルファイルに記述するのではなく csv ファイルから与えることにします.

以上のことから, 次のように定式化しなおすことができました.

集合	$ObserveTime = \{0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0\}$	観測時間の集合
変数	a	2 次の項の係数
	b	1 次の項の係数
	c	定数項
定数	$y(t), t \in ObserveTime$	時刻 t における観測値
目的関数（最小化）	$\sum_{t \in ObserveTime} e(t)^2 = \sum_{t \in ObserveTime} (at^2 + bt + c - y(t))^2$	各観測時刻における誤差の二乗和

この結果を SIMPLE で記述すると次のようになり，先ほどのものに比べ汎用性が高くさらに見やすいものになっています.

```
//観測時間の集合の宣言
Set ObserveTime;
Element t(set=ObserveTime);
//観測値をパラメータとして宣言
Parameter y(index=t);
//変数の宣言
Variable a(name="２次の項の係数");
Variable b(name="１次の項の係数");
Variable c(name="定数項");
//誤差を Expression を用いて表現する
Expression e(index=t);
e[t]=a*t*t+b*t+c-y[t];
//誤差の二乗和の最小化
Objective err(type=minimize);
err=sum(e[t]*e[t],t);
//求解して結果を出力する
solve();
a.val.print();
b.val.print();
c.val.print();
```

なお，実行させる際には次のような csv ファイルを与えます.

```
t,y
0.5,1.22470
1.0,0.87334
1.5,0.99577
2.0,1.34215
2.5,2.12172
3.0,3.22933
3.5,4.44744
4.0,6.13509
4.5,8.14697
5.0,10.50759
```

このモデルを実行すると，最後に

```
２次の項の係数=0.644402
１次の項の係数=-1.47656
定数項=1.76057
```

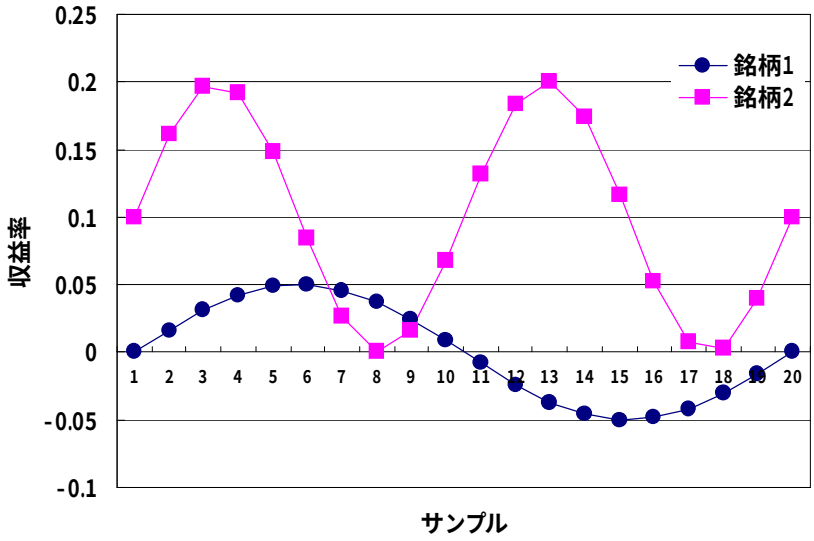
という表示がされます．このことから， $f(t) = 0.644402t^2 - 1.47656t + 1.76057$ **という二次関数モデルが推定できたことになります．**

5.13 ポートフォリオ最適化問題

ここでは, ポートフォリオが与える収益率の分布を評価する統計量として平均と分散を用いるマルコビッツモデルの例を示します. 具体的にはポートフォリオのもたらす収益率の変動の大きさ（リスク）を分散で計測することにし，それを最小化するような資産配分を求めます.

（例題）

銘柄1, 2に対する収益率のサンプル20期分が以下の図のように得られているとします.



この収益率のサンプルを用いてポートフォリオの収益率の分散が最小となる投資配分を求めます．ただし，空売りはできないものとし，またポートフォリオの収益率の期待値は0以上とします.

この問題を定式化すると以下のようになります.

集合	$Asset = \{1, 2\}$	**銘柄の集合**
	$Sample = \{1, 2, 3 \ldots 20\}$	**サンプルの集合**
定数	$r_{tj}, t \in Sample, j \in Asset$	**サンプル t における銘柄 j の期待収益率**
	$\overline{r}_j = \sum_{t \in Sample} r_{tj} / \lvert Sample \rvert$	**平均期待収益率**
変数	$x_j, j \in Asset$	**銘柄 j の組入れ比率**
目的関数(最小化)	$\sum_{t \in Sample} dev_t^{\,2} / \lvert Sample \rvert$	**ポートフォリオが与える期待収益率の分散**
	$dev_t = \sum_{j \in Asset} r_{tj} x_j - \overline{r}_p$	
	$\overline{r}_p = \sum_{j \in Asset} \overline{r}_j x_j$	
制約条件	$x_j \geq 0, \forall j \in Asset$	**非負制約（空売り禁止）**
	$\sum_{j \in Asset} x_j = 1$	**組入れ比率の総和は `1`**
	$\overline{r}_p \geq 0$	**ポートフォリオが与える平均期待収益率は `0` 以上**

この問題は目的関数が二次であり制約式は全て線形ですので二次計画問題になります.

定式化した結果を SIMPLE で記述すると以下のようになります.

```
// 集合と添字
Set Asset(name="アセット");
Element j(set=Asset);
Set Sample(name="サンプル");
Element t(set=Sample);

// パラメータ
Parameter r(name="期待収益率", index=(t,j));
Parameter rb(name="平均期待収益率", index=j);
rb[j] = sum(r[t,j],t)/Sample.card();

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

// 式
Expression rpb(name="期待収益");
rpb = sum(rb[j]*x[j],j);
Expression dev(name="偏差", index=t);
dev[t] = sum(r[t,j]*x[j],j)-rpb;

// 目的関数
Objective V(name="リスク");
V = sum(dev[t]*dev[t],t)/Sample.card();

// 制約
x[j] >= 0.0;
sum(x[j],j) == 1.0;
rpb >= 0.0;

// 求解
solve();

// 出力
V.val.print();
rpb.val.print();
x.val.print();
```

データファイル（.dat 形式）は以下のようになります.

```
期待収益率 =
[ 1,1]  0.0000 [ 1,2] 0.100
[ 2,1]  0.0162 [ 2,2] 0.161
[ 3,1]  0.0307 [ 3,2] 0.197
[ 4,1]  0.0419 [ 4,2] 0.192
[ 5,1]  0.0485 [ 5,2] 0.148
[ 6,1]  0.0498 [ 6,2] 0.084
[ 7,1]  0.0458 [ 7,2] 0.026
[ 8,1]  0.0368 [ 8,2] 0.000
[ 9,1]  0.0238 [ 9,2] 0.016
[10,1]  0.0082 [10,2] 0.068
[11,1] -0.0082 [11,2] 0.132
[12,1] -0.0238 [12,2] 0.184
[13,1] -0.0368 [13,2] 0.200
[14,1] -0.0458 [14,2] 0.174
[15,1] -0.0498 [15,2] 0.116
[16,1] -0.0485 [16,2] 0.052
[17,1] -0.0419 [17,2] 0.008
[18,1] -0.0307 [18,2] 0.003
[19,1] -0.0162 [19,2] 0.039
[20,1]  0.0000 [20,2] 0.100
;
```

このモデルを実行すると以下のような解が得られます.

```
リスク=0.000951734
期待収益=0.0199126
組入比率[1]=0.800874
組入比率[2]=0.199126
```

5.14 ジョブショップスケジューリング問題

ジョブショップスケジューリング問題とは，生産計画等の現場で現れ，仕事(ジョブ)を機械に効率的に割り振る事で，完了時刻，納期遅れ等の最小化を目的としたスケジューリング問題です．ここでは，特に機械の数が複数で，仕事が複数の作業(オペレーション)から構成され，各仕事の全ての作業が処理されると，仕事の処理が完了する直列機械(serial machines)について言及します．本問題は，作業が処理される順序によって，以下の 3 つに大別されます．

(1) オープンショップ問題
各作業が処理される機械の順序は決まっていない．

(2) フローショップ問題
各作業が処理される機械の順序が同じ．

(3) ジョブショップ問題
各作業が処理される機械の順序が決められているが，これらの順序は仕事ごとに異なっていてもよい．フローショップ問題の一般形

5.14.1 オープンショップ問題

（例題）オープンショップ問題
３つの作業から構成される，3 つの仕事があるとします．各作業を処理する機械と処理時間は以下のように与えられています．

	作業 1(機械 1)	作業 2(機械 2)	作業 3(機械 3)
仕事 a	5 時間	8 時間	4 時間
仕事 b	5 時間	4 時間	6 時間
仕事 c	6 時間	5 時間	5 時間

各機械が作業を処理している間は，他の作業を処理する事は出来ません．また，各作業を処理する順番に制限はありません．この場合の最後の作業の完了時刻が最小となるようにするには，各仕事をどのように機械に割り振ればいいでしょうか．

この問題を資源制約付きスケジューリング問題で定式化すると，例えば以下の様になります．各仕事の各作業は同時には 2 つ以上処理出来ない事を，ダミー資源を用いて表現している事に注意して下さい．

資源	machine1	機械 1，供給量:常に 1
	machine2	機械 2，供給量:常に 1
	machine3	機械 3，供給量:常に 1
	d1	ダミー資源 1，供給量:常に 1
	d2	ダミー資源 2，供給量:常に 1
	d3	ダミー資源 3，供給量:常に 1
モード	mode_a_1	処理時間:5
		必要量：machine1,d1 を処理時間中常に 1
	mode_a_2	処理時間:8
		必要量：machine2,d1 を処理時間中常に 1
	mode_a_3	処理時間:4
		必要量：machine3,d1 を処理時間中常に 1
	mode_b_1	処理時間:5
		必要量：machine1,d2 を処理時間中常に 1
	mode_b_2	処理時間:4
		必要量：machine2,d2 を処理時間中常に 1
	mode_b_3	処理時間:6
		必要量：machine3,d2 を処理時間中常に 1
	mode_c_1	処理時間:6
		必要量：machine1,d3 を処理時間中常に 1
	mode_c_2	処理時間:5
		必要量：machine2,d3 を処理時間中常に 1
	mode_c_3	処理時間:5
		必要量：machine3,d3 を処理時間中常に 1
アクティビティ	$act_{a,1}$	仕事 a の作業 1
		処理モード：mode_a_1
	$act_{a,2}$	仕事 a の作業 2
		処理モード：mode_a_2
	$act_{a,3}$	仕事 a の作業 3
		処理モード：mode_a_3
	$act_{b,1}$	仕事 b の作業 1
		処理モード：mode_b_1
	$act_{b,2}$	仕事 b の作業 2
		処理モード：mode_b_2
	$act_{b,3}$	仕事 b の作業 3
		処理モード：mode_b_3

	$act_{c,1}$	仕事 c の作業 1 処理モード：mode_c_1
	$act_{c,2}$	仕事 c の作業 2 処理モード：mode_c_2
	$act_{c,3}$	仕事 c の作業 3 処理モード：mode_c_3
目的関数(最小化)	completionTime	最後の作業の完了時刻
スケジュール期間	T	30

これを SIMPLE で記述すると次のようになります.

```
Set M; // モード
Element m(set=M);
Set R; // 資源
Element r(set=R);
Set D; // 各モードの作業時間の最大
Element d(set=D);
ResourceRequire  req(name="req", mode=M, resource=R, duration=D);
// 必要資源量
Set T(name=”T”); // スケジュール期間
Element t(set=T);
ResourceCapacity cap(name="cap",resource=R,timeStep=T); // 資源供給
量
cap[r,t] = 1;
Set J; // 仕事
Set S; // 作業
Element j(set=J);
Element s(set=S);
Set AvailMode(name="AvailMode", index=(j,s)); // 各仕事のオペレーショ
ンにおいて処理されるモード
Activity act(name="act", index=(j,s),mode= AvailMode[j,s]);
Objective f(type=minimize); // 最後の作業の完了時刻の最小化
f = completionTime;
options.maxtim = 15;
showSystem();
solve();
simple_printf("act[%s,%d] = %d¥n", j, s,  act[j,s].startTime);
```

データファイルは次のようになります.

```
T = 0 .. 30; // スケジューリング全体期間
AvailMode =
[a,1] mode_a_1
[a,2] mode_a_2
[a,3] mode_a_3
[b,1] mode_b_1
[b,2] mode_b_2
[b,3] mode_b_3
[c,1] mode_c_1
[c,2] mode_c_2
[c,3] mode_c_3
;
```

```
req =
[mode_a_1,machine1,1] 1 [mode_a_2,machine2,1] 1  [mode_a_3,machine3,1] 1
[mode_a_1,machine1,2] 1 [mode_a_2,machine2,2] 1  [mode_a_3,machine3,2] 1
[mode_a_1,machine1,3] 1 [mode_a_2,machine2,3] 1  [mode_a_3,machine3,3] 1
[mode_a_1,machine1,4] 1 [mode_a_2,machine2,4] 1  [mode_a_3,machine3,4] 1
[mode_a_1,machine1,5] 1 [mode_a_2,machine2,5] 1
                        [mode_a_2,machine2,6] 1
                        [mode_a_2,machine2,7] 1
                        [mode_a_2,machine2,8] 1

[mode_b_1,machine1,1] 1 [mode_b_2,machine2,1] 1  [mode_b_3,machine3,1] 1
[mode_b_1,machine1,2] 1 [mode_b_2,machine2,2] 1  [mode_b_3,machine3,2] 1
[mode_b_1,machine1,3] 1 [mode_b_2,machine2,3] 1  [mode_b_3,machine3,3] 1
[mode_b_1,machine1,4] 1 [mode_b_2,machine2,4] 1  [mode_b_3,machine3,4] 1
[mode_b_1,machine1,5] 1                         [mode_b_3,machine3,5] 1
                                                 [mode_b_3,machine3,6] 1

[mode_c_1,machine1,1] 1 [mode_c_2,machine2,1] 1 [mode_c_3,machine3,1] 1
[mode_c_1,machine1,2] 1 [mode_c_2,machine2,2] 1 [mode_c_3,machine3,2] 1
[mode_c_1,machine1,3] 1 [mode_c_2,machine2,3] 1 [mode_c_3,machine3,3] 1
[mode_c_1,machine1,4] 1 [mode_c_2,machine2,4] 1 [mode_c_3,machine3,4] 1
[mode_c_1,machine1,5] 1 [mode_c_2,machine2,5] 1 [mode_c_3,machine3,5] 1
[mode_c_1,machine1,6] 1

[mode_a_1,d1,1] 1 [mode_a_2,d1,1] 1  [mode_a_3,d1,1] 1
[mode_a_1,d1,2] 1 [mode_a_2,d1,2] 1  [mode_a_3,d1,2] 1
[mode_a_1,d1,3] 1 [mode_a_2,d1,3] 1  [mode_a_3,d1,3] 1
[mode_a_1,d1,4] 1 [mode_a_2,d1,4] 1  [mode_a_3,d1,4] 1
[mode_a_1,d1,5] 1 [mode_a_2,d1,5] 1
                  [mode_a_2,d1,6] 1
                  [mode_a_2,d1,7] 1
                  [mode_a_2,d1,8] 1
```

```
[mode_b_1,d2,1] 1 [mode_b_2,d2,1] 1 [mode_b_3,d2,1] 1
[mode_b_1,d2,2] 1 [mode_b_2,d2,2] 1 [mode_b_3,d2,2] 1
[mode_b_1,d2,3] 1 [mode_b_2,d2,3] 1 [mode_b_3,d2,3] 1
[mode_b_1,d2,4] 1 [mode_b_2,d2,4] 1 [mode_b_3,d2,4] 1
[mode_b_1,d2,5] 1                   [mode_b_3,d2,5] 1
                                     [mode_b_3,d2,6] 1

[mode_c_1,d3,1] 1 [mode_c_2,d3,1] 1 [mode_c_3,d3,1] 1
[mode_c_1,d3,2] 1 [mode_c_2,d3,2] 1 [mode_c_3,d3,2] 1
[mode_c_1,d3,3] 1 [mode_c_2,d3,3] 1 [mode_c_3,d3,3] 1
[mode_c_1,d3,4] 1 [mode_c_2,d3,4] 1 [mode_c_3,d3,4] 1
[mode_c_1,d3,5] 1 [mode_c_2,d3,5] 1 [mode_c_3,d3,5] 1
[mode_c_1,d3,6] 1
;
```

実行すると，各作業の開始時刻

```
act["a",1] = 0
act["a",2] = 5
act["a",3] = 13
act["b",1] = 6
act["b",2] = 13
act["b",3] = 0
act["c",1] = 11
act["c",2] = 0
act["c",3] = 6
```

が出力されます.

5.14.2 フローショップ問題

> （例題）フローショップ問題
> 上述のオープンショップ問題の条件の下，各仕事は，作業 1, 作業 2, 作業 3 の順で処理されなければならないとします．この場合の最後の作業の完了時刻が最小となるようにするには，どのように機械を割り振ればよいでしょうか.

この問題を記述するには, 各作業の処理順序を制約する以下の先行制約を加える必要があります.

先行制約	$act_{a,1} \prec act_{a,2}$	仕事 a の作業 1 は，仕事 a の作業 2 に先行する
	$act_{a,2} \prec act_{a,3}$	仕事 a の作業 2 は，仕事 a の作業 3 に先行する
	$act_{b,1} \prec act_{b,2}$	仕事 b の作業 1 は，仕事 b の作業 2 に先行する
	$act_{b,2} \prec act_{b,3}$	仕事 b の作業 2 は，仕事 b の作業 3 に先行する
	$act_{c,1} \prec act_{c,2}$	仕事 c の作業 1 は，仕事 c の作業 2 に先行する
	$act_{c,2} \prec act_{c,3}$	仕事 c の作業 2 は，仕事 c の作業 3 に先行する

上記の先行制約を加えた SIMPLE のモデルは以下の様になります．尚，各仕事の各作業は同時には 2 つ以上処理出来ない事は，先行制約で表現されている事に注意してください．

```
Set M; // モード
Element m(set=M);
Set R; // 資源
Element r(set=R);
Set D; // 各モードの作業時間の最大
Element d(set=D);
ResourceRequire  req(name="req", mode=M, resource=R, duration=D);
// 必要資源量
Set T(name=”T”); // スケジュール期間
Element t(set=T);
ResourceCapacity cap(name="cap",resource=R,timeStep=T); // 資源供給
量
cap[r,t] = 1;
Set J; // 仕事
Set S; // 作業
Element j(set=J);
Element s(set=S);
Set AvailMode(name="AvailMode", index=(j,s)); // 各仕事のオペレーショ
ンにおいて処理されるモード
Activity act(name="act", index=(j,s),mode= AvailMode[j,s]);
act[j,s-1] < act[j,s], s > 1 ; // 先行制約
Objective f(type=minimize); // 最後の作業の完了時刻の最小化
f = completionTime;
options.maxtim = 15;
showSystem();
solve();
simple_printf("act[%s,%d] = %d¥n", j, s,  act[j,s].startTime);
```

先行制約により，ダミー資源を用いる必要が無くなった為，データファイルの１つが以下の様に修正されます．

```
req =
[mode_a_1,machine1,1] 1 [mode_a_2,machine2,1] 1  [mode_a_3,machine3,1] 1
[mode_a_1,machine1,2] 1 [mode_a_2,machine2,2] 1  [mode_a_3,machine3,2] 1
[mode_a_1,machine1,3] 1 [mode_a_2,machine2,3] 1  [mode_a_3,machine3,3] 1
[mode_a_1,machine1,4] 1 [mode_a_2,machine2,4] 1  [mode_a_3,machine3,4] 1
[mode_a_1,machine1,5] 1 [mode_a_2,machine2,5] 1
                        [mode_a_2,machine2,6] 1
                        [mode_a_2,machine2,7] 1
                        [mode_a_2,machine2,8] 1

[mode_b_1,machine1,1] 1 [mode_b_2,machine2,1] 1  [mode_b_3,machine3,1] 1
[mode_b_1,machine1,2] 1 [mode_b_2,machine2,2] 1  [mode_b_3,machine3,2] 1
[mode_b_1,machine1,3] 1 [mode_b_2,machine2,3] 1  [mode_b_3,machine3,3] 1
[mode_b_1,machine1,4] 1 [mode_b_2,machine2,4] 1  [mode_b_3,machine3,4] 1
[mode_b_1,machine1,5] 1                         [mode_b_3,machine3,5] 1
                                                 [mode_b_3,machine3,6] 1

[mode_c_1,machine1,1] 1 [mode_c_2,machine2,1] 1 [mode_c_3,machine3,1] 1
[mode_c_1,machine1,2] 1 [mode_c_2,machine2,2] 1 [mode_c_3,machine3,2] 1
[mode_c_1,machine1,3] 1 [mode_c_2,machine2,3] 1 [mode_c_3,machine3,3] 1
[mode_c_1,machine1,4] 1 [mode_c_2,machine2,4] 1 [mode_c_3,machine3,4] 1
[mode_c_1,machine1,5] 1 [mode_c_2,machine2,5] 1 [mode_c_3,machine3,5] 1
[mode_c_1,machine1,6] 1
;
```

5.14.3 ジョブショップ問題

（例題）ジョブショップ問題
上述のオープンショップ問題の条件の下，各仕事は以下の順に処理されなければならないとします.

	作業 1(機械 1)	作業 2(機械 2)	作業 3(機械 3)
仕事 a	1	3	2
仕事 b	1	2	3
仕事 c	2	1	3

この場合の最後の作業の完了時刻が最小となるようにするには，どのように機械を割り振ればよいでしょうか.

この問題は，先行制約が以下の様に変更されます.

先行制約	$act_{a,1} \prec act_{a,3}$	**仕事 a の作業 1 は，仕事 a の作業 3 に先行する**
	$act_{a,3} \prec act_{a,2}$	**仕事 a の作業 3 は，仕事 a の作業 2 に先行する**
	$act_{b,1} \prec act_{b,2}$	**仕事 b の作業 1 は，仕事 b の作業 2 に先行する**
	$act_{b,2} \prec act_{b,3}$	**仕事 b の作業 2 は，仕事 b の作業 3 に先行する**
	$act_{c,2} \prec act_{c,1}$	**仕事 c の作業 2 は，仕事 c の作業 1 に先行する**
	$act_{c,1} \prec act_{c,3}$	**仕事 c の作業 1 は，仕事 c の作業 3 に先行する**

上記の先行制約をデータから与えられるように SIMPLE のモデルとデータを修正します.

```
Set M; // モード
Element m(set=M);
Set R; // 資源
Element r(set=R);
Set D; // 各モードの作業時間の最大
Element d(set=D);
ResourceRequire  req(name="req", mode=M, resource=R, duration=D);
// 必要資源量
Set T(name=”T”); // スケジュール期間
Element t(set=T);
ResourceCapacity cap(name="cap",resource=R,timeStep=T); // 資源供給
量
cap[r,t] = 1;
Set J; // 仕事
Set S; // 作業
Element j(set=J);
Element s(set=S);
Set AvailMode(name="AvailMode", index=(j,s)); // 各仕事のオペレーショ
ンにおいて処理されるモード
Activity act(name="act", index=(j,s),mode= AvailMode[j,s]);
Set Prec(name="Prec", dim=3);
Set U;
Set V;
Element u(set=S);
Element v(set=S);
act[j,u] < act[j,v], (j,u,v) < Prec ; // 先行制約
Objective f(type=minimize); // 最後の作業の完了時刻の最小化
f = completionTime;
options.maxtim = 15;
showSystem();
solve();
simple_printf("act[%s,%d] = %d¥n", j, s,  act[j,s].startTime);
```

データファイルの１つが以下の様に修正されます.

```
T = 0 .. 30; // スケジューリング全体期間
AvailMode =
[a,1] mode_a_1
[a,2] mode_a_2
[a,3] mode_a_3
[b,1] mode_b_1
[b,2] mode_b_2
[b,3] mode_b_3
[c,1] mode_c_1
[c,2] mode_c_2
[c,3] mode_c_3
;

Prec =
a 1 3
a 3 2
b 1 2
b 2 3
c 2 1
c 1 3
;
```

5.14.4 リスケジューリング問題

実際の現場では，ある時，機械が故障した等の突発事故が起こる事がしばしばあります．その場合，過去のスケジュールを固定し，条件を変更した後，再度スケジュールを組みなおします．このような問題をリスケジューリング問題と呼ぶ事にします．

（例題）リスケジューリング問題
上述のジョブショップ問題を解いた結果，各作業の開始時刻は，

	作業 1(機械 1)	作業 2(機械 2)	作業 3(機械 3)
仕事 a	0	14	5
仕事 b	5	10	14
仕事 c	10	0	20

となりました．このスケジュールに基づいて機械を運転していた所，10 時間が過ぎた所で，
機械 2 が故障し，5 時間停止する事になりました．この場合の最後の作業の完了時刻が最小となるようにはどのようにスケジュールを組み直せばよいでしょうか．

この問題を記述するには，過去のスケジュールを固定する必要があります．また，未来のスケジュールが過去に来ないようにするように全ての作業に先行する sourceActivity(自動で定義されている)との先行制約を記述する必要があります．その為，以下の制約が加わります．

アクティビティの開始時刻固定	$fixActivity(act_{a,1}.startTime)$	仕事 a の作業 1 の開始時刻を 0 に固定する
	$fixActivity(act_{a,3}.startTime)$	仕事 a の作業 3 の開始時刻を 5 に固定する
	$fixActivity(act_{b,1}.startTime)$	仕事 b の作業 1 の開始時刻を 5 に固定する
	$fixActivity(act_{c,2}.startTime)$	仕事 c の作業 2 の開始時刻を 0 に固定する
アクティビティのモード固定	$fixActivity(act_{a,1})$	仕事 a の作業 1 のモードを mode_a_1 に固定する
	$fixActivity(act_{a,3})$	仕事 a の作業 3 のモードを mode_a_3 に固定する
	$fixActivity(act_{b,1})$	仕事 b の作業 1 のモードを mode_b_1 に固定する
	$fixActivity(act_{c,2})$	仕事 c の作業 2 のモードを mode_c_2 に固定する
先行制約	$sourceActivity \prec act_{a,2},10$	仕事 a の作業 2 は，時刻 10 以前には処理されない
	$sourceActivity \prec act_{b,2},10$	仕事 b の作業 2 は，時刻 10 以前には処理されない
	$sourceActivity \prec act_{b,3},10$	仕事 b の作業 3 は，時刻 10 以前には処理されない
	$sourceActivity \prec act_{c,1},10$	仕事 c の作業 1 は，時刻 10 以前には処理されない
	$sourceActivity \prec act_{c,3},10$	仕事 c の作業 3 は，時刻 10 以前には処理されない

また，機械 2 は，時刻 10 から 5 時間停止するので，資源が以下の様に修正されます.

資源	machine1	機械 1，供給量:常に 1
	machine2	機械 2，供給量:時刻 10 以上 15 未満 0，その他常に 1
	machine3	機械 3，供給量:常に 1

これを SIMPLE で記述すると以下の様になります.

```
Set M; // モード
Element m(set=M);
Set R; // 資源
Element r(set=R);
Set D; // 各モードの作業時間の最大
Element d(set=D);
ResourceRequire  req(name="req", mode=M, resource=R, duration=D);
// 必要資源量
Set T(name=”T”); // スケジュール期間
Element t(set=T);
ResourceCapacity cap(name="cap",resource=R,timeStep=T); // 資源供給
量
cap[r,t] = 1;
cap["machine2", t] = 0, 10 <= t <= 14; // 故障
Set J; // 仕事
Set S; // 作業
Element j(set=J);
Element s(set=S);
Set AvailMode(name="AvailMode", index=(j,s)); // 各仕事のオペレーショ
ンにおいて処理されるモード
Activity act(name="act", index=(j,s),mode= AvailMode[j,s]);
Set Prec(name="Prec", dim=3);
Set U;
Set V;
Element u(set=S);
Element v(set=S);
act[j,u] < act[j,v], (j,u,v) < Prec ; // 先行制約
Objective f(type=minimize); // 最後の作業の完了時刻の最小化
f = completionTime;
/* --- リスケジューリングの為の制約 --- */
// 過去(0 .. 10) のスケジュールを固定
Set FixAct(name="FixAct", dim=2);
Parameter fixTime(name="fixTime", index=(j,s));
act[j,s].startTime = fixTime[j,s], (j,s) < FixAct;
Parameter fixMode(name="fixMode", index=(j,s));
act[j,s] = fixMode[j,s], (j,s) < FixAct;
FixAct.val.print();
fixActivity(act[j,s].startTime, (j,s) < FixAct);
fixActivity(act[j,s], (j,s) < FixAct);
```

```
// 過去のジョブ以外はステップ 10 以前に来てはならない
Set NotFixAct(name="NotFixAct", dim=2);
NotFixAct = setOf((j,s), (j < J, s < S)) - FixAct;
sourceActivity < act[j,s], (j,s) < NotFixAct, 10;
/* ---------------------------------- */
options.maxtim = 15;
showSystem();
solve();

simple_printf("act[%s,%d] = %d¥n", j, s,  act[j,s].startTime);
```

データファイルの１つが以下の様に修正されます.

```
T = 0 .. 30; // スケジューリング全体期間
AvailMode =
[a,1] mode_a_1
[a,2] mode_a_2
[a,3] mode_a_3
[b,1] mode_b_1
[b,2] mode_b_2
[b,3] mode_b_3
[c,1] mode_c_1
[c,2] mode_c_2
[c,3] mode_c_3
;

Prec =
a 1 3
a 3 2
b 1 2
b 2 3
c 2 1
c 1 3
;

FixAct =
a 1
a 3
b 1
c 2
;

fixTime = [a,1] 0 [a,3] 5 [b,1] 5 [c,2] 0;
fixMode = [a,1] mode_a_1 [a,3] mode_a_3 [b,1] mode_b_1 [c,2] mode_c_2;
```

参考文献

[1] George B. Dantzig, Mukund N.Thapa, Linear Programming 1:Introduction, Springer, 1997

[2] 福島雅夫, 数理計画入門, 朝倉書店, 1996

Q

R

S

T

U

W

あ

え

お

か

き

け

こ

さ

し

す

せ

そ

た

て

な

の

は

ふ

へ

ほ

ま

み

も

ゆ

り

わ