制約計画ソルバー SCOP (Solver for Constraint Programing)

scop2.py モジュール使用法ガイド
(Python 言語からの呼び出し方法)

LOG OPT Co., Ltd.

ご注意

目 次

第1章　はじめに

SCOP（Solver for COnstraint Programing：スコープ）は，大規模な制約計画問題を高速に解くためのソルバーである．ここで，制約計画（constraint programming）とは，従来の数理計画を補完する最適化理論の体系であり，組合せ最適化問題に特化した求解原理を用いるため，従来の数理計画ソルバーで解けない大規模な問題に対しても，効率的に良好な解を探索することができる．

SCOP のトライアル・バージョンは，LOGOPT 社のホームページ

`http://www.logopt.com/scop.htm`

から無料でダウンロードして試用することができる．このトライアル・バージョンは，変数の数 15 までの問題を解くことができる．また，本ドキュメントで使用されたプログラムも，同じ場所からダウンロードできる．

超高級プログラミング言語 Python は，初学者に優しい「お気楽」なプログラミング言語である．Python の予約語（キーワード）は，他の高級プログラミング言語と比べて圧倒的に少ない．また，Python，様々な便利な機能を搭載している．たとえば，変数の宣言が必要なく，メモリ管理も必要なく，多くのプラットフォームで動作し，オブジェクト指向であり，しかもフリーソフトである．

SCOP を Python 言語から呼び出して使用できると便利である．ここでは，制約計画ソルバー SCOP を，Python 言語から直接呼び出して，求解するためのモジュール (scop2.py) の使用法について解説する．このモジュールは，すべて Python で書かれたクラスで構成されており，ソースコードもトライアル・バージョンと同じ場所からダウンロードできる．また，ソース自身も公開されているので，ユーザが書き換え可能でる．

第2章　クラスライブラリの使用法

scop2.py は，以下のクラスから構成されている．

- モデルクラス Model
- 変数クラス Variable
- 制約クラス (Constraint): これは，以下のクラスのスーパークラスである．
  - 線形制約クラス Linear
  - 2 次制約クラス Quadratic
  - 相異制約クラス Alldiff

以下では，各クラスの解説を行う．

2.1　モデルクラス Model

Python から SCOP を呼び出して使うときに，最初にすべきことはモデルクラスのオブジェクトを生成することである．たとえば，m と名付けたモデルオブジェクトを生成したいときには，以下のように記述する．

```
1 from scop2 import *
2 m = Model()
```

1 行目は，scop2 モジュールからすべて（ワイルドカード * で表す）を import によって読み込み，その後で 2 行目では，モデルクラスからモデルオブジェクト m を生成している．

Model クラスは，以下のメソッド（「モデル名. メソッド名」でアクセスするメンバ関数）をもつ．

addVariable(変数名，領域)

モデルに 1 つの変数を追加する．引数の変数名 (引数名は name) は文字列で与え，領域はリスト (引数名は domain) とする．領域を定義するためのリストの要素は，文字列でも数値でもかまわない．

以下に例を示す．

```
1 x = model.addVarriable("var")                          # domain is set to []
2 x = model.addVariable(name="var",domain=[1,2,3])       # arguments by name
3 x = model.addVariable("var",["A","B","C"])             # arguments by position
```

1 行目の例では，変数名を “var”と設定した空の領域をもつ変数を追加している．2 行目の例では，名前付き引数で変数名と領域を設定している．領域は 1, 2, 3 の数値である．3 行目の例では，領域を文字列として変数を追加している．

addVariables(変数のリスト，領域)

モデルに，同一の領域をもつ複数の変数を同時に追加する．引数は変数名を要素としたリスト (引数名は names) と，共通の領域を表すリスト (引数名は domain) である．領域を定義するためのリストの要素は，文字列でも数値でもかまわない．

以下に例を示す．

```
1 varlist=["var1","var2","var3"]
2 x = model.addVariables(varlist)                          # domain  is set to []
3 x = model.addVariables(names=varlist,domain=[1,2,3]      # arguments by name
4 x = model.addVariables(varlist,["A","B","C"]             # arguments by position
```

1 行目では変数名のリストを `varlist` に保管している．その後，2,3,4 行目では，3 通りの方法で，複数の変数を追加している．

addConstriant(制約オブジェクト)

制約オブジェクトをモデルに追加する．制約オブジェクトは，制約クラスを用いて生成されたオブジェクトである．制約オブジェクトの生成法については，以下の項で解説する．

以下の例では，制約オブジェクト `L` をモデルに追加している．

```
model.addConstraint(L)
```

optimize( )

モデルの求解（最適化）を行う．最適化のためのパラメータは，パラメータ属性 Params で設定する．返値は，最適解の情報を保管した辞書と，破った制約の情報を保管した辞書のタプル（組）である．

たとえば，以下のプログラムでは最適解を辞書 `sol` に破った制約を辞書 `violated` に保管する．

```
sol,violated= m.optimize()
```

最適解や破った制約は，変数や制約の名前をキーとし，解の値や制約逸脱量を値とした辞書であるので，たとえば最適解を出力するには，以下のように記述すれば良い．

```
for x in sol:
    print x, sol[x]
```

結果は以下のようになる．

```
A 1
C 1
B 2
E 0
D 2
```

モデルオブジェクトは，求解（最適化）の際に用いるパラメータを表す属性 Params をもつ．

Params 属性で設定可能なパラメータは，以下の通り．

制限時間 (TimeLimit)

制限時間は正数値を設定する必要があり，その規定値は 600（秒）である．

たとえば，モデルオブジェクト m のパラメータである制限時間 (TimeLimit) を 1 秒に変更するには，以下のように記述すれば良い．

```
m.Params.TimeLimit=1
```

乱数の種 (RandomSeed)

SCOP では探索にランダム性を加味しているので，乱数の種を変えると，得られる解が変わる可能性がある．

乱数の種の規定値は 1 である．

出力フラグ (OutputFlag)

最適化の過程を出力する際の詳細さを制御するためのパラメータである．真 (True もしくは正の値) に設定すると詳細な情報を出力し，偽 (False もしくは 0) に設定すると最小限の情報を出力する．規定値は偽 (0) である．

目標値 (Target)

制約の逸脱量が目標値以下になったら自動終了させるためのパラメータである．規定値は 0 である．

また，モデルオブジェクトは，モデルの情報を文字列として返すことができる．たとえば,「モデル名」と名付けたモデルクラスのオブジェクトの情報（変数と制約の数，制約の種類と展開した式）は,

```
print モデル名
```

で得ることができる．

たとえば，後述する例題のモデル model を作成した後で,

```
print model
```

とすると，以下の結果が画面に出力される．

```
number of variables = 3
number of constraints= 2
variable A:[0, 1, 2]
variable B:[0, 1, 2]
variable C:[0, 1, 2]
AD: weight= inf  type=alldiff A C B ;
L: weight= 1 type=linear 15(A,0) 20(A,1) 30(A,2) 7(B,0) 15(B,1) 12(B,2) 25(C,0) 10(C,1) 13(C,2) <=0
```

2.2 変数クラス Variable

すでに述べたように，変数クラス Variable のオブジェクトは，モデルオブジェクトの addVariable もしくは addVariables メソッドを用いて生成される．

変数クラスは，以下の属性をもつ．

name

変数の名称である．

たとえば，変数オブジェクト var の名前を ‘‘NewVarName"に変えるには，以下のように記述する．

```
var.name( "NewVarName" )
```

domain

変数の領域を表すリストである．

たとえば，変数オブジェクト var の領域に “C”を追加するには，リストの append メソッドを用いて，以下のように記述する．

```
var.domain.append( "C" )
```

また，変数オブジェクトは，変数の情報を文字列として返すことができる．たとえば，

```
var  = model.addVariable("x", ["A","B","C"])
print var
```

によって，変数オブジェクト var の情報（変数名と領域）を出力すると，以下の結果が得られる．

```
variable x: ["A","B","C"]
```

2.3 線形制約クラス Linear

線形制約クラス Linear のオブジェクトは，以下のように生成する．

```
オブジェクト=Linear(制約名,重み,右辺定数,制約の向き)
```

引数の意味は以下の通り．

制約名 (name)

制約名は，制約を区別するための名称であり，個有の名前を文字列で入力する必要がある．(名前が重複した場合には，前に定義した制約が無視される．)

重み (weight)

重みは，制約の重要性を表す正数もしくは文字列 “inf” である．ここで “inf” は無限大を表し，絶対制約を定義するときに用いられる．重みは省略することができ，その場合の既定値は 1 である．

右辺定数 (rhs)

右辺定数は，制約の右辺を表す定数（整数値）である．右辺定数は省略することができ，その場合の既定値は 0 である．

制約の向き (direction)

制約の向きは，‘<=”, “>=”, “=” のいずれかの文字列とし，規定値は “<=” である．

Linear クラスは，以下のメソッドをもつ．

addTerms(係数（リスト），変数オブジェクト（リスト），値（リスト）)

線形制約（の左辺）に 1 つもしくは複数の項を追加する．変数が値をとるときに 1，それ以外のとき 0 となる仮想の変数 –これを制約最適化では値（あたい）変数とよぶ–を x[変数名, 値] とすると， 追加される項は，

$$\text{係数} \times x[\text{変数}, \text{値}]$$

を表す．

addTerms メソッドは，1 つの項を追加するか，複数の項を一度に追加する．1 つの項を追加する場合には，引数の係数は整数値，変数は変数オブジェクトで与え，値は変数の領域の要素とする（値は，文字列でも数値でもかまわない）．複数の項を一度に追加する場合には，同じ長さをもつ，係数，変数オブジェクト，値のリストで与える．

たとえば，項をもたない線形制約オブジェクト L に対して，

```
L.addTerms(1, y, "A")
```

と 1 つの項を追加すると，制約の左辺は

```
1 x[y, "A"]
```

となる．

同様に，項をもたない線形制約オブジェクト L に対して，

```
L.addTerms([2, 3, 1], [y, y, z], ["C", "D", "C"])
```

と 3 つの項を同時に追加すると，制約の左辺は以下のようになる．

```
 2 x[y,"C"] +3 x[y,"D"] + 1 x[z,"C"]
```

setRhs(右辺定数)

線形制約の右辺定数を設定する．引数は整数値であり，規定値は 0 である．

setDirection(制約の向き)

制約の向きを設定する．引数は “<=”, “>=”, “=” のいずれかの文字列とし，規定値は “<=” である．

setWeight(重み)

制約の重みを設定する．引数は正数値もしくは文字列 “inf” であり，“inf” は無限大を表し，絶対制約を定義するときに用いられる．

また，線形制約クラス Linear は，制約の情報を文字列として返すことができる．たとえば，L1 と名付けた線形制約クラスのオブジェクトの情報（重みと展開した式）は，

```
print L1
```

で得ることができる．

2.4 2 次制約クラス Quadratic

2 次制約クラス Quadratic のオブジェクトは，以下のように生成する．

```
オブジェクト=Quadratic(制約名,重み,右辺定数,制約の向き)
```

引数は，線形制約クラス Linear と同じ意味をもつ．

Quadratic クラスは，以下のメソッドをもつ．

addTerms(係数（リスト）, 変数オブジェクト 1（リスト）, 値 1（リスト）, 変数オブジェクト 2（リスト）, 値 2（リス

2 次制約（の左辺）に 2 つの変数の積から成る項を追加する．変数オブジェクト $i(i = 1, 2)$ が値 $i(i = 1, 2)$ をとるときに 1，それ以外のとき 0 となる変数（値変数）を x[変数 i, 値 i] とすると，追加される項は，

$$\text{係数} \times x[\text{変数 } 1, \text{値 } 1] \times x[\text{変数 } 2, \text{値 } 2]$$

と記述される．

addTerms メソッドは，1 つの項を追加するか，複数の項を一度に追加する．1 つの項を追加する場合には，引数の係数は整数値，変数は変数オブジェクトで与え，値は変数の領域の要素とする（値は，文字列でも数値でもかまわない）．複数の項を一度に追加する場合には，同じ長さをもつ，係数，変数オブジェクト，値のリストで与える．

たとえば，項をもたない 2 次制約オブジェクト Q に対して，

```
L.addTerms(1, y, "A", z, "B")
```

と 1 つの項を追加すると，制約の左辺は

```
1 x[y,"A"] * x[z,"B"]
```

となる．

同様に，項をもたない 2 次制約オブジェクト Q に対して，

```
L.addTerms([2, 3, 1], [y, y, z], ["C", "D", "C"], [x, x, y], ["A", "B", "C"])
```

と 3 つの項を同時に追加すると，制約の左辺は以下のようになる．

```
2 x[y,"C"] * x[x,"A"] + 3 x[y,"D"] * x[x,"B"] + 1 x[z,"C"] * x[y,"C"]
```

setRhs(右辺定数)

線形制約の右辺定数を設定する．引数は整数値であり，規定値は 0 である．

setDirection(制約の向き)

制約の向きを設定する．引数は “<=”, “>=”, “=” のいずれかの文字列とし，規定値は “<=” である．

setWeight(重み)

制約の重みを設定する．引数は正数値もしくは文字列 “inf” であり，“inf” は無限大を表し，絶対制約を定義するときに用いられる．

また，2 次制約クラス Quadratic は，制約の情報を文字列として返すことができる．たとえば，Q1 と名付けた線形制約クラスのオブジェクトの情報（重みと展開した式）は，

```
print Q1
```

で得ることができる．

2.5 相異制約クラス Alldiff

相異制約クラス Alldiff のオブジェクトは，以下のように生成する．

```
オブジェクト = Alldiff(制約名, 変数名のリスト, 重み)
```

変数名のリストは，すべての値の番号（インデックス）が異なることを要求される変数のリストであり，省略も可能である．その場合の既定値は，空のリストとなる．ここで追加する変数は，モデルクラスに追加された変数である必要がある．

制約名と重みについては，線形制約クラス Linear と同じように設定する．

addVariable(変数オブジェクト)

相異制約の変数を 1 つ制約に追加する．

addVariables(変数オブジェクトのリスト)

相異制約の変数を複数同時に（リストとして）追加する．

setWeight(重み)

制約の重みを設定する．引数は正数値もしくは文字列 “inf ” であり，“inf ” は無限大を表し，絶対制約を定義するときに用いられる．

また，相異制約クラス Alldiff は，制約の情報を文字列として返すことができる．たとえば，A1 と名付けた相異制約クラスのオブジェクトの情報（重みと式に含まれる変数）は，

```
print A1
```

で得ることができる．

次章では，幾つかの例を用いて，scop2.py の使用法を解説する．

第3章　簡単な例題

本章では，簡単な例題を通して scop2.py の使用法を学ぶ．

3.1　仕事の割当 1

最初の例題は，3 人の作業員 A,B,C を 3 つの仕事 $0,1,2$ に割り当てる問題である．すべての仕事には，1 人の作業員を割り当てる必要があるが，作業員と仕事には相性があり，割り当てにかかる費用（単位は万円）は，以下のようになっている．

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{pmatrix} 15 & 20 & 30 \\ 7 & 15 & 12 \\ 25 & 10 & 13 \end{pmatrix} \end{array}$$

総費用を最小にするような作業員の割り当てを決めることが問題の目的である．

まず，scop2 モジュールを読み込み，モデルクラス Model のオブジェクト m を生成する．

```
from scop2 import *
m = Model()
```

作業員はリスト workers で，割当費用はリストのリスト Cost に保管しておく．

```
workers=["A","B","C"]
Cost=[ [15, 20, 30],
       [7, 15, 12],
       [25,10,13] ]
```

次に，モデルオブジェクト m に対して変数を追加する．モデルクラスの addVariables メソッドを用いて，すべての作業員に同じ領域（値の集合）$0,1,2$ を設定する．これには，Python の range() 関数を用いれば良い．

```
m.addVariables(workers,range(3))
```

続いて制約クラスのオブジェクトを生成する．この問題は，1 人の作業員に 1 つの仕事を割り当てることを表す相異制約と，目的関数を表す線形制約で表現できる．まず，all_diff_constraint と名付けた相異制約 con1 を作っておく．

```
1 con1=Alldiff("all_diff_constraint",workers)
2 con1.setWeight("inf")
```

1 行目では，制約の重みを省略しているので，重みは既定値の 1 となっている．2 行目で setWeight メソッドを用いて重みを無限大（inf）に変更したので，この制約は絶対制約となる．もちろんこれは，以下のように 1 行で書いても良い．

```
con1 = scop.Alldiff("all_diff_constraint",workers,"inf")
```

次に linear_constraint と名付けた線形制約 con2 を生成する．この制約の重みは，既定値の 1 とするので，引数の weight は省略している．その後の 2 行目から 4 行目では，addTerm メソッドを用いて，左辺に項を追加している．この項は，i 番目の作業員が仕事 j に割り当てられたときに費用 Cost[i][j] がかかることを表す．5 行目は右辺定数が 0，6 行目は制約が以下（<=）を表すことを指定している．実は線形制約クラスの既定値は <= 0 であるので，最後の 2 行は省略しても良い．

```
1 con2 = Linear("linear_constraint")
2 for i in range(len(workers)):
3     for j in range(3):
4         con2.addTerm(Cost[i][j],workers[i],j)
5 con2.setRhs(0)
6 con2.setDirection("<=")
```

また，上のプログラムの 1 行目で，引数に重み，右辺定数と制約の向きを指定して制約を生成しても同じである．

```
con2=Linear("L",weight=1,rhs=0,direction="<=")
```

最後に，生成した制約 con1，con2 を問題 p に addConstrant メソッドを用いて追加し（1,2 行目）し，パラメータ TimeLimit を 1 に設定（3 行目：制限時間 1 秒で探索することを指定）し，optimize メソッドで解を探索する（4 行目）．

```
1 m.addConstraint(con1)
2 m.addConstraint(con2)
3 m.Params.TimeLimit=1
4 sol,violated=m.optimize()
```

optimize メソッドの返値は，解と逸脱した制約の辞書であり，それぞれ sol，violated に保持している．これを表示するには，以下のように辞書のキーと値を標準出力に Python の print コマンドで出力すれば良い（ここで Python のバージョンは 2.x を仮定している．3.x の場合には，print() と括弧付きの関数で呼び出す必要がある）．

```
print "solution"
for x in sol:
    print x,sol[x]

print "violated constraint(s)"
for v in violated:
    print v,violated[v]
```

上の Python プログラムを実行すると，結果は以下のように出力される．

```
solution
A 0
C 1
B 2
violated constraint(s)
linear_constraint 37
```

結果から，作業員 A には仕事 0 を，作業員 B には仕事 2 を，作業員 C には仕事 1 を割り当てるのが最良であることが分かる．割り当てられた作業員と仕事の対に対応する費用を丸で囲んで表すと，以下のようになる．

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{pmatrix} \textcircled{15} & 20 & 30 \\ 7 & 15 & \textcircled{12} \\ 25 & \textcircled{10} & 13 \end{pmatrix} \end{array}$$

相異制約 all_diff_constraint の逸脱量は 0 であるので，上では表示されていない．逸脱があるのは，線形制約 linear_constraint であり，逸脱量は 37，制約の重みは 1 であったので，費用は $37(=15+12+10)$ 万円になることが分かる．

なお，作成したモデルを確認するには，モデルオブジェクト m を print で表示させれば良い．

```
print m
```

これによって，以下の出力が得られる．

```
number of variables = 3
number of constraints= 2
variable A:[0, 1, 2]
variable B:[0, 1, 2]
variable C:[0, 1, 2]
Alldiff: all_diff_constraint: weight=inf
A B C
Linear: linear_constraint: weight=1
15*x[A:0] 20*x[A:1] 30*x[A:2] 7*x[B:0] 15*x[B:1] 12*x[B:2] 25*x[C:0] 10*x[C:1] 13*x[C:2] <=0
```

この操作は Python の shell で対話的にできるので，デバッグの際には便利である．もちろん，変数や制約も個別に print で出力することができる．

3.2 仕事の割当 2

次に，5 人の作業員 A,B,C,D,E を 3 つの仕事 $0,1,2$ に割り当てる問題を考える．ここでは，各仕事にかかる作業員の最低人数が与えられており，それぞれ $1,2,2$ 人必要であり，割り当ての際の費用（単位は万円）は，以下のようになっているものとする．

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{pmatrix} 15 & 20 & 30 \\ 7 & 15 & 12 \\ 25 & 10 & 13 \\ 15 & 18 & 3 \\ 5 & 12 & 17 \end{pmatrix} \end{array}$$

作業員の最低人数は，線形制約として表現できる．これらの制約はハードな制約とするため，制約の重み（weight）は無限大（inf）と設定する．

この問題を SCOP2 を用いて解くための Pythn プログラムは，以下のように書ける．

```
from scop2 import *

m=Model()
workers=["A","B","C","D","E"]
Cost=[[15, 20, 30],[7, 15, 12],[25,10,13],[15,18,3],[5,12,17]]
LB=[1,2,2]
varlist=m.addVariables(workers,range(3))
LBC={} #dictionary for keeping lower bound constraints
for j in range(len(LB)):
    LBC[j]=Linear("LB%s"%j,"inf",LB[j],">=")
    coeffs=[1 for i in range(5)]
    values=[j for i in range(5)]
    LBC[j].addTerms(coeffs,varlist,values)
    m.addConstraint(LBC[j])
con1=Linear("L")
for i in range(len(workers)):
    for j in range(3):
        con1.addTerms(Cost[i][j],varlist[i],j)
m.addConstraint(con1)
m.Params.TimeLimit=1
sol,violated=m.optimize()
print m
print "solution"
for x in sol:
    print x,sol[x]
print "violated constraint(s)"
for v in violated:    print v,violated[v]
```

上のプログラムを実行すると，以下の結果が得られる．

```
number of variables = 5
number of constraints= 4
variable A:[0, 1, 2]
variable B:[0, 1, 2]
variable C:[0, 1, 2]
variable D:[0, 1, 2]
variable E:[0, 1, 2]
Linear: LB_constraint0: weight=inf
1*x[A:0] 1*x[B:0] 1*x[C:0] 1*x[D:0] 1*x[E:0] >=1
Linear: LB_constraint1: weight=inf
1*x[A:1] 1*x[B:1] 1*x[C:1] 1*x[D:1] 1*x[E:1] >=2
Linear: LB_constraint2: weight=inf
1*x[A:2] 1*x[B:2] 1*x[C:2] 1*x[D:2] 1*x[E:2] >=2
Linear: linear_constraint: weight=1
15*x[A:0] 20*x[A:1] 30*x[A:2] 7*x[B:0] 15*x[B:1] 12*x[B:2] 25*x[C:0]
10*x[C:1] 13*x[C:2] 15*x[D:0] 18*x[D:1] 3*x[D:2] 5*x[E:0] 12*x[E:1] 17*x[E:2] <=0

solution
A 1
C 1
B 2
E 0
D 2
violated constraint(s)
linear_constraint 50
```

結果から分かるように，作業員 A には仕事 1 を，作業員 B には仕事 2 を，作業員 C には仕事 1 を，作業員 D には仕事 2 を，作業員 E には仕事 0 を割り当てるのが最良であることが分かる．割り当てに対応する費用を丸で囲んで表すと，以下のようになる．

$$
\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}
\begin{array}{c}
\begin{array}{ccc} 0 & 1 & 2 \end{array} \\
\begin{pmatrix}
15 & \textcircled{20} & 30 \\
7 & 15 & \textcircled{12} \\
25 & \textcircled{10} & 13 \\
15 & 18 & \textcircled{3} \\
\textcircled{5} & 12 & 17
\end{pmatrix}
\end{array}
$$

ハードな制約の逸脱量は 0 であるので，すべての仕事の必要人数は確保され，割当費用の合計が 0 以下であると定義したソフトな制約の逸脱量は 50 であるので，費用は $50(=20+12+10+3+5)$ 万円になることが分かる．

3.3 仕事の割当 3

上の例題と同じ状況で，仕事を割り振ろうとしたところ，作業員 A と C は仲が悪く，一緒に仕事をさせると喧嘩を始めることが判明した．作業員 A と C を同じ仕事に割り振らないようにするには，どうしたら良いかを考えてみる．

上の例題ではデータを保管するのにリストを用いたが，ここでは辞書を用いて実装しよう．実際問題を解く際には，辞書の方が融通が利いて実装しやすい．

まず，作業員と仕事はリストに保管した後で，割当費用と仕事に割り当てる作業員数の下限は辞書として保管する．

```
from scop2 import *
m=Model()
workers=["A","B","C","D","E"]
Jobs    =[0,1,2]
Cost={ ("A",0):15, ("A",1):20, ("A",2):30,
       ("B",0): 7, ("B",1):15, ("B",2):12,
       ("C",0):25, ("C",1):10, ("C",2):13,
       ("D",0):15, ("D",1):18, ("D",2): 3,
       ("E",0): 5, ("E",1):12, ("E",2):17
       }
LB={0: 1,
    1: 2,
    2: 2
    }
```

ここで Cost は，作業員と仕事のタプルをキーとし，費用を値とした辞書であり，LB は仕事をキーとし，下限を値とした辞書である．

変数 x も辞書として以下のように定義しておく．

```
x={}
for i in workers:
    x[i]= m.addVariable(i,Jobs)
```

変数 x はキーを作業員，値を変数オブジェクト（領域は仕事のリスト）とした辞書である．

作業員数の下限制約を以下のように定義し，制約も辞書

t LBC に保管しておく．

```
LBC={}
for j in Jobs:
    LBC[j]=Linear("LB%s"%j,"inf",LB[j],">=")
```

```
4       for i in workers:
5           LBC[j].addTerms(1,x[i],j)
6       m.addConstraint(LBC[j])
```

目的関数（割当費用の合計）を表す制約を以下のように記述する．

```
1 obj=Linear("obj",1,0,"<=")
2 for i in workers:
3     for j in Jobs:
4         obj.addTerms(Cost[i,j],x[i],j)
```

最後に，追加された作業員 A と C を同じ仕事に割り当てることを禁止する制約は，2 次制約（重みは 100）として以下のように記述する．

```
1 conf=Quadratic("conflict",100,0,"=")
2 for j in Jobs:
3     conf.addTerms(1,x["A"],j,x["C"],j)
4 m.addConstraint(conf)
```

これは変数 xx["A"] と x["C"] が同じ値 j を持ったときに左辺に 1 を加える制約であり，右辺が 0 であるので，なるべく作業員 A と C を同じ仕事に割り当てないことになる．

この制約を追加したプログラムを実行すると，以下の結果が得られる．

```
number of variables = 5
number of constraints= 5
variable A:[0, 1, 2]
variable B:[0, 1, 2]
variable C:[0, 1, 2]
variable D:[0, 1, 2]
variable E:[0, 1, 2]
LB0: weight= inf type=linear 1(A,0) 1(B,0) 1(C,0) 1(D,0) 1(E,0) >=1
LB1: weight= inf type=linear 1(A,1) 1(B,1) 1(C,1) 1(D,1) 1(E,1) >=2
LB2: weight= inf type=linear 1(A,2) 1(B,2) 1(C,2) 1(D,2) 1(E,2) >=2
obj: weight= 1 type=linear 15(A,0) 20(A,1) 30(A,2) 7(B,0) 15(B,1) 12(B,2) 25(C,0)
 10(C,1) 13(C,2) 15(D,0) 18(D,1) 3(D,2) 5(E,0) 12(E,1) 17(E,2) <=0
conflict: weight= 100 type=quadratic 1(A,0) (C,0) 1(A,1) (C,1) 1(A,2) (C,2) =0

solution
A 0
C 1
B 2
E 1
D 2
obj  52
```

結果から分かるように，作業員 A には仕事 0 を，作業員 B には仕事 2 を，作業員 C には仕事 1 を，作業員 D には仕事 2 を，作業員 E には仕事 1 を割り当てるのが最良であることが分かる．割り当てに対応する費用を丸で囲んで表すと，以下のようになる．

$$
\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}
\begin{array}{c}
\begin{array}{ccc} 0 & 1 & 2 \end{array} \\
\left(\begin{array}{ccc}
\textcircled{15} & 20 & 30 \\
7 & 15 & \textcircled{12} \\
25 & \textcircled{10} & 13 \\
15 & 18 & \textcircled{3} \\
5 & \textcircled{12} & 17
\end{array}\right)
\end{array}
$$

確かに，作業員 A と C は，異なる仕事に割り振られており，ハードな制約の逸脱量は 0 であるので，すべての仕事の必要人数は確保され，割当費用の合計が 0 以下であると定義したソフトな制約の逸脱量は 52 であるので，費用は $52(= 15 + 12 + 10 + 3 + 12)$ 万円になることが分かる．

第4章　代表的な組合せ最適化問題

ここでは，代表的な組合せ最適化問題を例として，SCOP Python モジュール scop2.py の使用法を解説する．

4.1　グラフ分割問題

いま，6 人のお友達を 2 つのチームに分けてミニサッカーをしようとしている（図 4.1）．もちろん，公平を期すために，同じ人数になるように 3 人ずつに分ける．ただし，お友達同士には仲良しがいて，仲良しが別のチームになることは極力避けたいと考えている．さて，どのようにチーム分けをしたら良いだろうか？

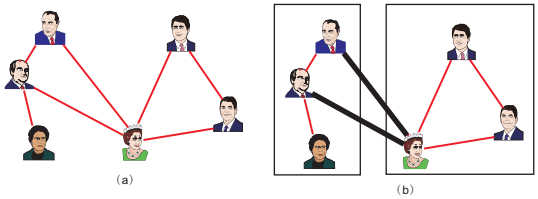

図 4.1: グラフ分割問題の例．(a) 線の引いてある人同士は仲良しであることを表すグラフ．(b) 仲良しが別のチームになることを最小にする等分割．違うチームに所属する仲良しのペアは太線で表されており，2 本である．よって，この等分割の目的関数値は 2 となる．

上の問題は，**グラフ分割問題**（graph partitioning problem）とよばれる組合せ最適化問題の例である．実際には，サッカーのチーム分けではなく，VLSI 設計をはじめとする多くの真面目な応用をもつ $\mathcal{NP}$-困難問題である．

グラフ分割問題をきちんと定義すると，次にように書ける．

グラフ分割問題
点数 $n = |V|$ が偶数である無向グラフ $G = (V, E)$ が与えられたとき，点集合 V の**等分割**（uniform partition, eqipartition）(L, R) とは，$L \cap R = \emptyset$, $L \cup R = V$, $|L| = |R| = n/2$ を満たす点の部分集合の対である．グラフ分割問題とは，L と R の間にある枝の本数を最小にする等分割 (L, R) を求める問題.

問題を明確化するために，グラフ分割問題を整数計画問題として定式化しておく．無向グラフ $G = (V, E)$ に対して，$L \cap R = \emptyset$（共通部分がない），$L \cup R = V$（合わせると点集合全体になる）を満たす非順序対 (L, R) を**分割**（partition）もしくは **2 分割**（bipartition）とよぶ．分割 (L, R) において，L は左側，R は右側を表すが，これらは逆にしても同じ分割であるので，非順序対とよばれる．点 i が，分割 (L, R) の L 側に含まれているとき 1，それ以外の（R 側に含まれている）とき 0 の 0-1 変数 x_i を導入する．このとき，等分割であるためには，x_i の合計が $n/2$

である必要がある．枝 (i,j) が L と R をまたいでいるときには，$x_i(1-x_j)$ もしくは $(1-x_i)x_j$ が 1 になることから，以下の定式化を得る．

$$\begin{aligned} \text{minimize} \quad & \sum_{(i,j)\in E} x_i(1-x_j)+(1-x_i)x_j \\ \text{subject to} \quad & \sum_{i\in V} x_i = n/2 \\ & x_i \in \{0,1\} \qquad\qquad \forall i \in V \end{aligned}$$

この定式化は，目的関数を 2 次の考慮制約として，制約式を線形の絶対制約として入力することによって，SCOP で容易に求解できる．

目的関数は 1 本の 2 次制約として，以下のように記述できる．

```
con2=Quadratic( "obj")
for i in range(n):
    for j in adj[i]:
        con2.addTerms(1,varlist[i],1,varlist[j],0)
        con2.addTerms(1,varlist[i],0,varlist[j],1)
m.addConstraint(con2)
```

しかし，大規模問題例においては，以下のプログラムに示すように，目的関数の項ごとに 2 次の制約を付加した方が，効率良く探索できる．これを理解するには，SCOP に搭載されている近傍探索法（正確に言うと禁断探索法）のメカニズムを知る必要がある．近傍とは，1 つの変数に割り当てられた値を，領域内の他の値に変更することである．1 本の 2 次関数として目的関数を表した場合には，変数が他の値に変更される度に，長い 2 次式の和を計算して，目的関数値の変化量を評価する必要がある．一方，個別の項を制約として表した場合には，変数が他の値に変化したときの目的関数値の変化量の計算が短時間でできる．したがって，後者（以下のプログラム）の方が，同じ計算時間ならより良い解を算出することが期待できる．

```
from scop2 import *

m=Model()

nodes=["n%i"%i for i in range(6)]
adj=[[1,4],[0,2,4],[1],[4,5],[0,1,3,5],[3,4]]
n=len(nodes)

varlist=m.addVariables(nodes,[0,1])

con1=Linear("constraint","inf",n/2,"=")
for i in range(len(nodes)):
    con1.addTerms(1,varlist[i],1)
m.addConstraint(con1)

con2={}
for i in range(n):
    for j in adj[i]:
        con2[i,j]= Quadratic( "obj_%s_%s"%(i,j) )
        con2[i,j].addTerms(1,varlist[i],1,varlist[j],0)
        con2[i,j].addTerms(1,varlist[i],0,varlist[j],1)
        m.addConstraint(con2[i,j])

m.Params.TimeLimit=1
sol,violated=m.optimize()

print m

print "solution"
```

```
for x in sol:
    print x,sol[x]
print "violated constraint(s)"
for v in violated:
    print v,violated[v]
    from scop2 import *
```

目的関数を 1 本の 2 次制約とした場合の結果は，以下のようになる．

```
number of variables = 6
number of constraints= 2
variable n0:[0, 1]
variable n1:[0, 1]
variable n2:[0, 1]
variable n3:[0, 1]
variable n4:[0, 1]
variable n5:[0, 1]
constraint: weight= inf type=linear 1(n0,1) 1(n1,1) 1(n2,1) 1(n3,1) 1(n4,1) 1(n5,1) =3
obj: weight= 1 type=quadratic 1(n0,1) (n1,0) 1(n0,0) (n1,1) 1(n0,1) (n4,0) 1(n0,0) (n4,1) 1(n1,1) (n0,0) 1(n1,0

solution
n0 0
n1 0
n2 0
n3 1
n4 1
n5 1
violated constraint(s)
obj 4
```

4.2 最大安定集合問題

あなたは 6 人のお友達から何人か選んで一緒にピクニックに行こうと思っている．しかし，図 4.2 で線で結んである人同士はとても仲が悪く，彼らが一緒にピクニックに行くとせっかくの楽しいピクニックが台無しになってしまう．なるべくたくさんの仲間でピクニックに行くには誰を誘えばいいんだろう？

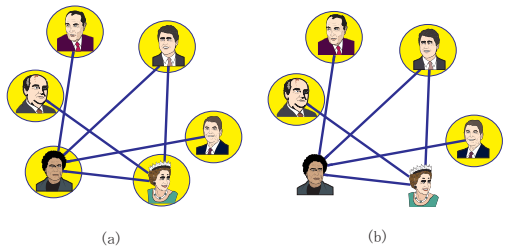

図 4.2: 最大安定集合問題の例．(a) 線の引いてある人同士は仲が悪いことを表すグラフ．(b) 仲が悪い同士を連れて行かないでピクニックに行くときの最大人数．丸で囲んだ人を連れて行くと目的関数値は 4 となり，これが最適解になる．

これは，**最大安定集合問題**（maximum stable set problem）とよばれるグラフ理論の基礎的な問題の一例である．

最大安定集合問題は，次のように定義される問題である．

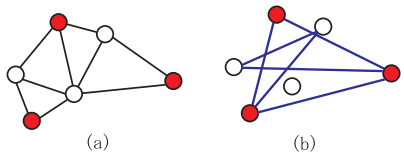

図 4.3: (a) 最大安定集合．(b) 補グラフ上の最大クリーク．

最大安定集合問題
点数 n の無向グラフ $G = (V, E)$ が与えられたとき，点の部分集合 S $(\subseteq V)$ は，すべての S 内の点の間に枝がないとき**安定集合**（stable set）とよばれる．最大安定集合問題とは，集合に含まれる要素数（位数）$|S|$ が最大になる安定集合 S を求める問題.

この問題のグラフの補グラフ（枝の有無を反転させたグラフ）を考えると，以下に定義される**最大クリーク問題**（maximum clique problem）になる．これらの 2 つの問題は（お互いに簡単な変換によって帰着されるという意味で）同値である（図 4.3）.

最大クリーク問題
無向グラフ $G = (V, E)$ が与えられたとき，点の部分集合 C $(\subseteq V)$ は，C によって導かれた誘導部分グラフが**完全グラフ**（complete graph）になるとき**クリーク**（clique）とよばれる（完全グラフとは，すべての点の間に枝があるグラフである）．最大クリーク問題とは，位数 $|C|$ が最大になるクリーク C を求める問題.

これらの問題は，符号理論，信頼性，遺伝学，考古学，VLSI 設計など広い応用をもつ.

点 i が安定集合 S に含まれるとき 1，それ以外のとき 0 の 0-1 変数を用いると，最大安定集合問題は，以下のように定式化できる.

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in V} x_i \\ \text{subject to} \quad & x_i + x_j \leq 1 \quad \forall (i,j) \in E \\ & x_i \in \{0,1\} \quad \forall i \in V \end{aligned}$$

上の定式化を SCOP を用いて求解するには，以下のようにすれば良い.

```
m=Model()
nodes=["n"+str(i) for i in range(6)]
adj=[[2],[3],[0,3,4,5],[1,2,5],[2],[2,3]]
n=len(nodes)
varlist=m.addVariables(nodes,[0,1])
for i in range(n):
    for j in adj[i]:
        if i<j:
            con1=Linear("constraint"+str(i)+str(j),"inf",1)
            con1.addTerms(1,varlist[i],1)
            con1.addTerms(1,varlist[j],1)
            m.addConstraint(con1)
obj=Linear("obj",1,n,">=")
for i in range(n):
    obj.addTerms(1,varlist[i],1)
m.addConstraint(obj)
m.Params.TimeLimit=1
sol,violated=m.optimize()
print m
```

```
print "solution"
for x in sol:
    print x,sol[x]
print "violated constraint(s)"
for v in violated:
    print v,violated[v]
```

上のプログラムの実行結果は，以下のようになる．

```
number of variables = 6
number of constraints= 7
variable n0:[0, 1]
variable n1:[0, 1]
variable n2:[0, 1]
variable n3:[0, 1]
variable n4:[0, 1]
variable n5:[0, 1]
constraint02: weight= inf type=linear 1(n0,1) 1(n2,1) <=1
constraint13: weight= inf type=linear 1(n1,1) 1(n3,1) <=1
constraint23: weight= inf type=linear 1(n2,1) 1(n3,1) <=1
constraint24: weight= inf type=linear 1(n2,1) 1(n4,1) <=1
constraint25: weight= inf type=linear 1(n2,1) 1(n5,1) <=1
constraint35: weight= inf type=linear 1(n3,1) 1(n5,1) <=1
obj: weight= 1 type=linear 1(n0,1) 1(n1,1) 1(n2,1) 1(n3,1) 1(n4,1) 1(n5,1) >=6

solution
n0 1
n1 1
n2 0
n3 0
n4 1
n5 1
violated constraint(s)
obj 2
```

4.3 グラフ彩色問題

あなたは，お友達のクラス分けで悩んでいる．お友達同士で仲が悪い組は，図 4.4 で線で結んである．仲が悪いお友達を同じクラスに入れると喧嘩を始めてしまう．なるべく少ないクラスに分けるには，どのようにすればいいんだろう？

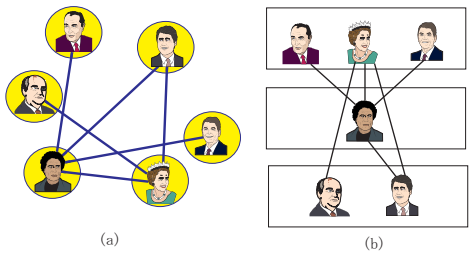

図 4.4: グラフ彩色問題の例．(a) 線の引いてある人同士は仲が悪いことを表すグラフ．(b) 3 つのクラスに分けると仲の悪い友達は同じクラスに入らない．目的関数値（クラス数）は 3 となり，これが最適解になる．

これは**グラフ彩色問題**（graph coloring problem）とよばれる古典的な最適化問題の例である．

グラフ彩色問題は，以下のように定義される問題である．

グラフ彩色問題
点数 n の無向グラフ $G=(V,E)$ の K **分割**（K partition）とは，点集合 V の K 個の部分集合への分割 $\Upsilon=\{V_1,\cdots,V_K\}$ で，$V_i\cap V_j=\emptyset,\forall i\neq j$（共通部分がない），$\bigcup_{j=1}^{K}V_j=V$（合わせると点集合全体になる）を満たすものを指す．各 V_i $(i=1,\cdots,K)$ を**色クラス**（color class）とよぶ． K 分割は，すべての色クラス V_i が安定集合（点の間に枝がない）のとき K **彩色**（K coloring）とよばれる．グラフ彩色問題とは，与えられた無向グラフ $G=(V,E)$ に対して，最小の K（これを彩色数とよぶ) を導く K 彩色 $\Upsilon=\{V_1,\cdots,V_K\}$ を求める問題.

グラフ彩色問題は，時間割作成，周波数割当など様々な応用をもつ．

グラフが K 色で彩色可能か否かを判定する問題は，SCOP を用いると簡単に解くことができる.（最小の彩色数を求めるためには，K をパラメータとして色々変えながら SCOP を用いて求解する必要がある.）

まず，点ごとに領域を $\{1,2,\cdots,K\}$ をもつ変数を定義する．次に，枝 $(i,j)\in E$ の両端点 i,j が異なる色に彩色されるように相異制約を付加する．

例題を SCOP で求解するには，以下のようにすれば良い．

```
from scop2 import *
m=Model()
K=3
nodes=["n%s"%i for i in range(6)]
adj=[[2],[3],[0,3,4,5],[1,2,5],[2],[2,3]]
n=len(nodes)

varlist=m.addVariables(nodes,range(K))

for i in range(n):
    for j in adj[i]:
        if i<j:
            con1=Alldiff("alldiff_%s_%s"%(i,j),[varlist[i],varlist[j]],"inf")
            m.addConstraint(con1)

m.Params.TimeLimit=1
sol,violated=m.optimize()
print m
print "solution"
for x in sol:
    print x,sol[x]
print "violated constraint(s)"
for v in violated:
    print v,violated[v]
```

実行結果は，以下のようになる．

```
number of variables = 6
number of constraints= 6
variable n0:[0, 1, 2]
variable n1:[0, 1, 2]
variable n2:[0, 1, 2]
variable n3:[0, 1, 2]
variable n4:[0, 1, 2]
variable n5:[0, 1, 2]
alldiff_0_2: weight= inf  type=alldiff n0 n2 ;
alldiff_1_3: weight= inf  type=alldiff n3 n1 ;
alldiff_2_3: weight= inf  type=alldiff n2 n3 ;
alldiff_2_4: weight= inf  type=alldiff n2 n4 ;
alldiff_2_5: weight= inf  type=alldiff n2 n5 ;
alldiff_3_5: weight= inf  type=alldiff n3 n5 ;
```

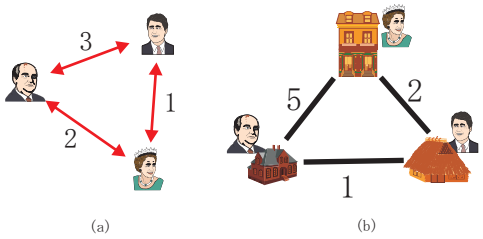

図 4.5: 2 次割当問題の例．(a) 3 人のお友達が打ち合わせをする頻度．枝の上の数字は週に何回打ち合わせをするかを表す．(b) 3 箇所の家とお友達の割り当ての例．枝の上の数字は地点間の距離を表す．お友達同士の行き来する頻度と割り当てられた家の間の距離の和は，$2 \times 5 + 3 \times 1 + 1 \times 2 = 15$ となり，この割り当ての目的関数値は，その 2 倍（お友達同士は往復するから）で $2 \times 15 = 30$ となる．

```
solution
n0 0
n1 0
n2 1
n3 2
n4 2
n5 0
violated constraint(s)
```

4.4　2 次割当問題

いま，3 人のお友達が 3 箇所の家に住もうとしている．3 人は毎週何回か重要な打ち合わせをする必要があり，打ち合わせの頻度は，図 4.5 (a) のようになっている．家の間の移動距離は，図 4.5 (b) のようになっており，3 人は打ち合わせのときに移動する距離を最小にするような場所に住むことを希望している．さて，誰をどの家に割り当てたらよいのだろうか？

この問題は，**2 次割当問題**（quadratic assignment problem）とよばれ，$\mathcal{NP}$-困難な問題の中でも特に悪名高い問題の例である．

2 次割当問題をきちんと定義すると，以下のようになる．

2 次割当問題
2 次割当問題とは，集合 $V = \{1, \cdots, n\}$ および 2 つの $n \times n$ 行列 $F = [f_{ij}]$，$D = [d_{k\ell}]$ が与えられたとき，

$$\sum_{i \in V} \sum_{j \in V} f_{ij} d_{\pi(i)\pi(j)}$$

を最小にする順列 $\pi : V \to \{1 \cdots, n\}$ を求める問題．

この問題は，Koopmans–Beckmann によって導入された問題であり，施設の配置を決定する応用から生まれた．n 個の施設があり，それを n 箇所の地点に配置することを考える．施設 i, j 間には物の移動量 f_{ij} があり，地点 k, ℓ 間を移動するには距離 $d_{k\ell}$ がかかるものとする．問題の目的は，物の総移動距離を最小にするように，各地点に 1 つ

ずつ施設を配置することである．順列 π は施設 i を地点 $\pi(j)$ に配置することを表す．

図 4.5 の例題では，行列 $F=[f_{ij}]$，$D=[d_{k\ell}]$ は，

$$F=\begin{pmatrix}0&2&3\\2&0&1\\3&1&0\end{pmatrix}\quad D=\begin{pmatrix}0&5&1\\5&0&2\\1&2&0\end{pmatrix}$$

となる．順列 π は，距離行列 D の行と列の交換を表す．行列 D の行 i を $\pi(i)$ 行と交換し，同時に列 j を $\pi(j)$ 行と交換した行列を D' とする．この行列の i,j 成分と行列 F の i,j 成分の積を，すべての i,j に対して加えたものが，2 次割当問題の目的関数となる．たとえば，$\pi=(2,1,3)$，写像で書くと $\pi(1)=2,\pi(2)=1,\pi(3)=3$ の順列に対して，D' は

$$D'=\begin{pmatrix}0&5&2\\5&0&1\\2&1&0\end{pmatrix}$$

となり，F との成分ごとの積和は，$2\times5+3\times2+1\times1=17$ の 2 倍で 34 となる．

2 次割当問題は $\mathcal{NP}$-困難な問題の中でも極めて解きにくい問題の 1 つであり，巡回セールスマン問題（4.6 節）を特別な場合として含んでいる．帰着は容易であり，

$$D=\begin{array}{c}\\1\\2\\\vdots\\n-1\\n\end{array}\begin{array}{c}\begin{array}{cccccc}1&2&3&\cdots&n-1&n\end{array}\\\begin{pmatrix}0&1&0&\cdots&0&0\\0&0&1&&0&0\\\vdots&\vdots&&\ddots&&\vdots\\0&0&0&\cdots&1&0\\1&0&0&\cdots&0&0\end{pmatrix}\end{array}$$

と設定すれば良い．

施設 i が地点 k に配置されるとき 1，それ以外のとき 0 となる 0-1 変数 x_{ik} を用いると，2 次割当問題は以下のように定式化できる．

$$\begin{aligned}\text{minimize}\quad&\sum_{i,j\in V,i\neq j}\sum_{k,\ell\in V,k\neq\ell}f_{ij}d_{k\ell}x_{ik}x_{j\ell}\\\text{subject to}\quad&\sum_{i\in V}x_{ik}=1&&\forall k\in V\\&\sum_{k\in V}x_{ik}=1&&\forall i\in V\\&x_{ik}\in\{0,1\}&&\forall i,k\in V\end{aligned}$$

SCOP では，2 次の制約（目的関数）をそのまま記述できるので，プログラムは容易である．

```
from scop2 import *
m=Model()

n=3
d=[[0,2,3],[2,0,1],[3,1,0]]
f=[[0,5,1],[5,0,2],[1,2,0]]

nodes=["n%s"%i for i in range(n)]
```

```
varlist=m.addVariables(nodes,range(n))

con1=Alldiff("AD",varlist,"inf")
m.addConstraint(con1)

obj=Quadratic("obj")
for i in range(n-1):
    for j in range(i+1,n):
        for k in range(n):
            for ell in range(n):
                if k !=ell:
                    obj.addTerms(f[i][j]*d[k][ell],varlist[i],k,varlist[j],ell)
m.addConstraint(obj)

m.Params.TimeLimit=1
sol,violated=m.optimize()

print m

print "solution"
for x in sol:
    print x,sol[x]
print "violated constraint(s)"
for v in violated:
    print v,violated[v]
```

上のプログラムの実行結果は，以下のようになる．

```
number of variables = 3
number of constraints= 2
variable n0:[0, 1, 2]
variable n1:[0, 1, 2]
variable n2:[0, 1, 2]
AD: weight= inf  type=alldiff n0 n2 n1 ;
obj: weight= 1 type=quadratic 10(n0,0) (n1,1) 15(n0,0) (n1,2) 10(n0,1) (n1,0) 5(n0,1) (n1,2) 15(n0,2)
 (n1,0) 5(n0,2) (n1,1) 2(n0,0) (n2,1) 3(n0,0) (n2,2) 2(n0,1) (n2,0) 1(n0,1) (n2,2) 3(n0,2) (n2,0)
 1(n0,2) (n2,1) 4(n1,0) (n2,1) 6(n1,0) (n2,2) 4(n1,1) (n2,0) 2(n1,1) (n2,2) 6(n1,2) (n2,0) 2(n1,2) (n2,1) <=0

solution
n0 2
n1 1
n2 0
violated constraint(s)
obj 12
```

4.5 ビンパッキング問題

あなたは，大企業の箱詰め担当部長だ．あなたの仕事は，色々な大きさのものを，決められた大きさの箱に「上手に」詰めることである．この際，使う箱の数をなるべく少なくすることが，あなたの目標だ．(なぜって，あなたの会社が利用している宅配業者では，運賃は箱の数に比例して決められるから．) 1 つの箱に詰められる荷物の上限は 7 kg と決まっており，荷物の重さは分かっている．詰め込む荷物の重量リストを，到着順に $(6, 5, 4, 3, 1, 2)$ とする（図 4.6）．しかも，あなたの会社で扱っている荷物は，どれも重たいものばかりなので，容積は気にする必要はない（すなわち箱の容量は十分と仮定する）．さて，どのように詰めて運んだら良いだろうか？

この実際問題は，**箱詰め問題**もしくは**ビンパッキング問題**（bin packing problem）とよばれる問題の一例である．ビンパッキング問題を数学的に記述すると次にように書ける．

ビンパッキング問題

n 個のアイテムから成る有限集合 N とサイズ B のビンが無限個準備されている．個々のアイテム $i \in N$ のサイズ $0 \leq w_i \leq B$ は分かっているものとする．これら n 個のアイテムを，サイズ B のビンに詰めることを考えるとき，必要なビンの数を最小にするような詰めかたを求めよ．

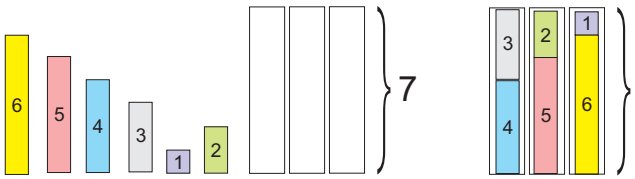

図 4.6: 箱詰め担当部長のビンパッキング問題の問題例と SCOP による解.

この問題は，通常の数理計画ソルバーが苦手とするタイプの問題であり，現実的には，大きい順に詰めるなどのヒューリスティクスが使われることが多い．ここでは，SCOP を用いて解く方法について考える．SCOP を用いることによって，現実問題で頻繁にあらわれる付加条件付きのビンパッキング問題に対しても簡単に対応ができる．

この問題を解くための Python のプログラムは，以下のようになる．

```
1  from scop2 import *
2  bpp=Model()
3
4  Items=[6,5,4,3,1,2]
5  B=7
6  num_bins=3
7  n=len(Items)
8
9  x={}
10 for i in range(n):
11       x[i] = bpp.addVariable("x_%s"%i,range(num_bins))
12 Bin={}
13 for j in range(num_bins):
14       Bin[j]=Linear("Bin_%s"%j,weight=1,rhs=B,direction="<=")
15       for i in range(n):
16            Bin[j].addTerms(Items[i],x[i],j)
17       bpp.addConstraint(Bin[j])
18
19 sol,violated=bpp.optimize()
20
21 print bpp
22
23 print "solution="
24 for i in sol:
25     print i,sol[i]
26
27 print "violated constraints=",violated
```

実行結果は，以下のようになる．

```
number of variables = 6
number of constraints= 3
variable x_0:[0, 1, 2]
variable x_1:[0, 1, 2]
variable x_2:[0, 1, 2]
variable x_3:[0, 1, 2]
variable x_4:[0, 1, 2]
variable x_5:[0, 1, 2]
```

```
Bin_0: weight= 1 type=linear 6(x_0,0) 5(x_1,0) 4(x_2,0) 3(x_3,0) 1(x_4,0) 2(x_5,0) <=7
Bin_1: weight= 1 type=linear 6(x_0,1) 5(x_1,1) 4(x_2,1) 3(x_3,1) 1(x_4,1) 2(x_5,1) <=7
Bin_2: weight= 1 type=linear 6(x_0,2) 5(x_1,2) 4(x_2,2) 3(x_3,2) 1(x_4,2) 2(x_5,2) <=7

solution=
x_4 2
x_5 0
x_2 1
x_3 1
x_0 2
x_1 0
violated constraints= {}
```

4.6 巡回セールスマン問題

あなたは休暇を利用してヨーロッパめぐりをしようと考えている．現在スイスのチューリッヒに宿を構えているあなたの目的は，スペインのマドリッドで闘牛を見ること，イギリスのロンドンでビックベンを見物すること，イタリアのローマでコロシアムを見ること，ドイツのベルリンで本場のビールを飲むことである．
あなたはレンタルヘリコプターを借りてまわることにしたが，移動距離に比例した高額なレンタル料を支払わなければならない．したがって，あなたはチューリッヒを出発した後，なるべく短い距離で他の 4 つの都市（マドリッド，ロンドン，ローマ，ベルリン）を経由し，再びチューリッヒに帰って来ようと考えた．都市の間の移動距離を測ってみたところ図 4.7 のようになっていることがわかった．さて，どのような順序で旅行すれば，移動距離が最小になるだろうか?

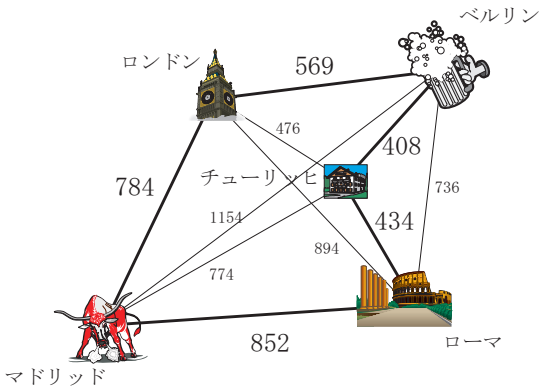

図 4.7: ヨーロッパ旅行のグラフ表現（枝上の数値は距離で単位はマイル）と最適巡回路（太線）．

この問題は，一般に**巡回セールスマン問題**（traveling salesman problem）とよばれる古典的な組合せ最適化問題である．

巡回セールスマン問題は，以下のように定義される問題である．

巡回セールスマン問題
n 個の点から成るグラフ $G = (V, E)$, 枝上の距離（重み, 費用）関数 $D : E \to \mathrm{R}$ が与えられたとき，すべての点集合 V をちょうど 1 回ずつ経由する巡回路で，枝上の距離の合計（これを巡回路の長さとよぶ）を最小にするものを求める問題.

上の例題を SCOP を用いて解くには，2 次割当問題の特殊形と考えて解けば良い．

```
from scop2 import *
m=Model()
cities=["T","L","M","R","B"]
d=[[0,476,774,434,408],
   [476,0,784,894,569],
   [774,784,0,852,1154],
   [434,894,852,0,569],
   [408,569,1154,569,0]]
n=len(cities)
varlist=m.addVariables(cities,range(n))
con1=Alldiff("AD",varlist,"inf")
m.addConstraint(con1)
obj=Quadratic("obj")
for i in range(n):
    for j in range(n):
        if i!=j:
            for k in range(n):
                if k ==n-1:
                    ell=0
                else:
                    ell=k+1
                obj.addTerms(d[i][j],varlist[i],k,varlist[j],ell)
m.addConstraint(obj)
m.Params.TimeLimit=1
sol,violated=m.optimize()
print m
print "solution"
for x in sol:
    print x,sol[x]
print "violated constraint(s)"
for v in violated:
    print v,violated[v]
```

グラフ分割問題のところでも述べたように，目的関数の 2 次関数は，項別に宣言した方が良い．そのためには，目的関数の宣言部を以下のように書き直す．

```
obj={}
for i in range(n):
    for j in range(n):
        if i!=j:
            for k in range(n):
                if k ==n-1:
                    ell=0
                else:
                    ell=k+1
                obj[i,j,k,ell]=Quadratic("obj_%s_%s_%s_%s"%(i,j,k,ell))
                obj[i,j,k,ell].addTerms(d[i][j],varlist[i],k,varlist[j],ell)
                m.addConstraint(obj[i,j,k,ell])
```

目的関数を 1 つの 2 次関数として書いたときの結果は，以下のようになる．

```
number of variables = 5
number of constraints= 2
variable T:[0, 1, 2, 3, 4]
variable L:[0, 1, 2, 3, 4]
variable M:[0, 1, 2, 3, 4]
variable R:[0, 1, 2, 3, 4]
variable B:[0, 1, 2, 3, 4]
AD: weight= inf  type=alldiff T M L B R ;
obj: weight= 1 type=quadratic 476(T,0) (L,1) 476(T,1) (L,2) 476(T,2) (L,3) 476(T,3) (L,4) 476(T,4) (L,0)
 774(T,0) (M,1) 774(T,1) (M,2) 774(T,2) (M,3) 774(T,3) (M,4) 774(T,4) (M,0) 434(T,0) (R,1) 434(T,1) (R,2)
  434(T,2) (R,3) 434(T,3) (R,4) 434(T,4) (R,0) 408(T,0) (B,1) 408(T,1) (B,2) 408(T,2) (B,3) 408(T,3) (B,4)
  408(T,4) (B,0) 476(L,0) (T,1) 476(L,1) (T,2) 476(L,2) (T,3) 476(L,3) (T,4) 476(L,4) (T,0) 784(L,0) (M,1)
```

```
  784(L,1) (M,2) 784(L,2) (M,3) 784(L,3) (M,4) 784(L,4) (M,0) 894(L,0) (R,1) 894(L,1) (R,2) 894(L,2) (R,3)
  894(L,3) (R,4) 894(L,4) (R,0) 569(L,0) (B,1) 569(L,1) (B,2) 569(L,2) (B,3) 569(L,3) (B,4) 569(L,4) (B,0)
  774(M,0) (T,1) 774(M,1) (T,2) 774(M,2) (T,3) 774(M,3) (T,4) 774(M,4) (T,0) 784(M,0) (L,1) 784(M,1) (L,2)
  784(M,2) (L,3) 784(M,3) (L,4) 784(M,4) (L,0) 852(M,0) (R,1) 852(M,1) (R,2) 852(M,2) (R,3) 852(M,3) (R,4)
  852(M,4) (R,0) 1154(M,0) (B,1) 1154(M,1) (B,2) 1154(M,2) (B,3) 1154(M,3) (B,4) 1154(M,4) (B,0) 434(R,0) (T,1)
  434(R,1) (T,2) 434(R,2) (T,3) 434(R,3) (T,4) 434(R,4) (T,0) 894(R,0) (L,1) 894(R,1) (L,2) 894(R,2) (L,3)
  894(R,3) (L,4) 894(R,4) (L,0) 852(R,0) (M,1) 852(R,1) (M,2) 852(R,2) (M,3) 852(R,3) (M,4) 852(R,4) (M,0)
  569(R,0) (B,1) 569(R,1) (B,2) 569(R,2) (B,3) 569(R,3) (B,4) 569(R,4) (B,0) 408(B,0) (T,1) 408(B,1) (T,2)
  408(B,2) (T,3) 408(B,3) (T,4) 408(B,4) (T,0) 569(B,0) (L,1) 569(B,1) (L,2) 569(B,2) (L,3) 569(B,3) (L,4)
  569(B,4) (L,0) 1154(B,0) (M,1) 1154(B,1) (M,2) 1154(B,2) (M,3) 1154(B,3) (M,4) 1154(B,4) (M,0) 569(B,0)(R,1)
  569(B,1) (R,2) 569(B,2) (R,3) 569(B,3) (R,4) 569(B,4) (R,0) <=0

solution
B 2
R 0
M 4
T 1
L 3
violated constraint(s)
obj 3047
```

4.7　多制約ナップサック問題

あなたは，ぬいぐるみ専門の泥棒だ．ある晩，あなたは高級ぬいぐるみ店にこっそり忍び込んで，盗む物を選んでいる．狙いはもちろん，マニアの間で高額で取り引きされているクマさん人形だ．クマさん人形は，現在 4 体販売されていて，それらの値段と重さと容積は，図 4.8 のようになっている．あなたは，転売価格の合計が最大になるようにクマさん人形を選んで逃げようと思っているが，あなたが逃走用に愛用しているナップサックはとても古く，7 kg より重い荷物を入れると，底がぬけてしまうし，10000cm^3（10 ℓ）を超えた荷物を入れると破けてしまう．さて，どのクマさん人形をもって逃げれば良いだろうか？

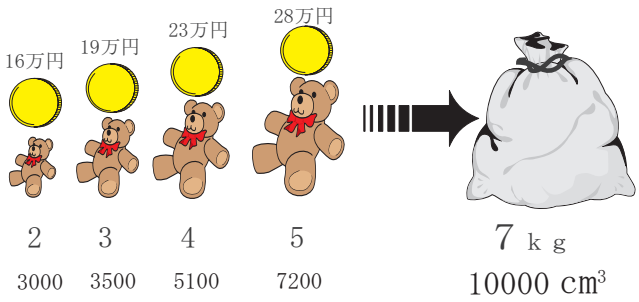

図 4.8: クマさん人形のラインアップと愛用のナップサック．

この問題は，**多制約ナップサック問題**（multi-constrained knapsack problem）とよばれる組合せ最適化問題であり，制約が 1 本の問題（ナップサック問題）でも $\mathcal{NP}$-困難である．ナップサック問題は，**分枝限定法**（branch and bound method）や**動的計画**（dynamic programming）で容易に解くことができるが，制約の数が増えた場合には解くことが困難になる．

多制約（0-1）ナップサック問題は，以下のように定義される．

多制約ナップサック問題

n 個のアイテムからなる有限集合 N，m 本の制約の添え字集合 M，各々のアイテム $j \in N$ の価値 v_j (≥ 0)，アイテム $j \in N$ の制約 $i \in M$ に対する重み a_{ij} (≥ 0)，および制約 $i \in M$ に対する制約の上限値 b_i (≥ 0) が与えられたとき，選択したアイテムの重みの合計が各制約 $i \in M$ の上限値 b_i を超えないという条件の下で，価値の合計を最大にするように N からアイテムを選択する問題.

ナップサック問題は，アイテム j $(\in N)$ をナップサックに詰めるとき 1，それ以外のとき 0 になる 0-1 変数 x_j を使うと，以下のように整数計画問題として定式化できる.

$$\begin{aligned} \text{maximize} \quad & \sum_{j \in N} v_j x_j \\ \text{subject to} \quad & \sum_{j \in N} a_{ij} x_j \leq b_i \quad \forall i \in M \\ & x_j \in \{0,1\} \qquad \forall j \in N \end{aligned}$$

問題例によっては，市販の数理計画ソルバーに上の定式化をそのまま入れただけでは，最適解を得ることが難しい場合がある．試しに，区間 $(0,1]$ の一様乱数 $U(0,1]$ を用いて以下のような（比較的難しいと言われている）問題例を作成してみた．制約式の係数 a_{ij} を $1 - 1000 \log_2 U(0,1]$ とし，右辺定数 b_i を $0.25 \sum_j a_{ij}$，目的関数の係数 v_j を $10 \sum_i a_{ij}/m + 10U(0,1]$ とする．変数の数 $n = 100$，制約の数 $m = 5$ の問題例を，市販の数理計画ソルバー Xpress-MP で求解したところ，2 時間かけても最適解を得ることができず，メモリ上限を超過してしまった.

SCOP を用いれば，極めて短時間に良好な近似解を得ることができる.

例題を SCOP で求解するプログラムは，以下のようになる.

```
rom scop2 import *

model=Model()

v=[16,19,23,28]
a=[[2,3,4,5],[3000,3500,5100,7200]]
b=[7,10000]
n=len(v)
m=len(b)
items=["item%s"%j for j in range(n)]
varlist=model.addVariables(items,[0,1])
for i in range(m):
    con1=Linear("mkp_%s"%i,"inf",b[i])
    for j in range(n):
        con1.addTerms(a[i][j],varlist[j],1)
    model.addConstraint(con1)

con2=Linear("obj",1,sum(v),">=")
for j in range(n):
    con2.addTerms(v[j],varlist[j],1)
model.addConstraint(con2)

model.Params.TimeLimit=1
sol,violated=model.optimize()

print model

print "solution"
for x in sol:
    print x,sol[x]
print "violated constraint(s)"
for v in violated:
    print v,violated[v]
```

```
number of variables = 4
number of constraints= 3
variable item0:[0, 1]
variable item1:[0, 1]
variable item2:[0, 1]
variable item3:[0, 1]
mkp_0: weight= inf type=linear 2(item0,1) 3(item1,1) 4(item2,1) 5(item3,1) <=7
mkp_1: weight= inf type=linear 3000(item0,1) 3500(item1,1) 5100(item2,1) 7200(item3,1) <=10000
obj: weight= 1 type=linear 16(item0,1) 19(item1,1) 23(item2,1) 28(item3,1) >=86

solution
item2 1
item3 0
item0 0
item1 1
violated constraint(s)
obj 44
```

第5章　応用例

本章では，SCOP を様々な現実問題に応用した例を示す．

5.1　スタッフスケジューリング

多くの職場では，スタッフスケジューリングは重要な意思決定問題の 1 つである．この問題は，社員，アルバイト，パートなどから，必要なスタッフをどのように確保し，かつ費用を最小化することを目的とした組合せ最適化問題であるが,「人」がからむ複雑な制約のため，しばしばモデル化が困難になる．ここでは，SCOP を用いたモデル化を解説する．

スタッフスケジューリング問題の基本モデルは，以下のデータを必要とする．

スタッフの集合： 職場で働くことができる人員の候補集合であり，社員，アルバイト，パートなどのグループに分けて管理される．通常は，期・シフトごとに，グループ別の必要最低人数が与えられている．また，スタッフごとに時給や希望シフトなどの情報が与えられているものとする．

期の集合： スケジューリングを組む対象となる期間の集合．通常は，日単位で管理を行い，30 日程度が対象期間となる．

シフトの集合: 期ごとに決められる仕事の種類の集合．たとえば，朝，昼，夜の 3 シフトから構成される職場の場合には，シフトの集合は，{ 朝，昼，夜，休 } と定義される．

この問題は，SCOP を用いると，以下のように自然にモデル化できる．

スタッフ・期ごとに，シフトの集合を領域とした変数 X を定義する．各期・シフトに対して，グループごとに必要なスタッフの数の上下限を線形制約として表現する．スタッフの希望シフトや，禁止されている連続シフトなどの制約は，線形制約もしくは 2 次制約として表現する．その他の，実際問題ごとに必要な付加条件も，SCOP を用いて表現できる．実際に，SCOP を用いたソルバーは，看護婦スケジューリングなどの複雑な実際問題を解決することに成功している．

以下に，簡単なスタッフスケジューリング問題の例を示す．この例題は，カーネギーメロン大の John Hooker の 2009 年の講演の例を改訂したものである．

問題の仮定は以下の通り．

- 1 シフトは 8 時間で，3 シフトの交代制とする．

- 4 人のスタッフは，1 日の高々1 つのシフトしか行うことができない．
- 繰り返し行われる 1 週間のスケジュールの中で，スタッフは最低 5 日間は勤務しなければならない．
- 各シフトに割り当てられるスタッフの数は，ちょうど 1 人でなければならない．
- 異なるシフトを翌日に行ってはいけない．(異なるシフトに移るときには，必ず休日を入れる．)
- シフト 2, 3 は，少なくとも 2 日間は連続で行わなければならない．

これを SCOP で解くために，休日を表すシフト 0 を導入する．スタッフは A, B, C, D を格納した文字列のリストで表し，期は $0, 1, ..., 6$ （`range(7)`）で表す．変数は，スタッフの番号 i と期の番号 t を添え字とし，その領域をシフトに休日を加えた集合 $\{0, 1, 2, 3\}$ （`range(4)`）とする．

この問題を解くための Python のプログラムは，以下のようになる．

```
from scop2 import *

m=Model()

periods=range(7)
shifts=range(4) #three shifts named 0 (off), 1, 2, and 3
staffs=["A","B","C","D"]

var={} #list of variables
for i in range(len(staffs)):
    for t in periods:
        var[i,t]=m.addVariable(name=staffs[i]+str(t),domain=shifts)

LB={} #dictionary for holding lower bound constraints
for i in range(len(staffs)):
    LB[i]=Linear("LB_%s"%i,rhs=5,direction=">=") #weight is set to default (1)
    for t in periods:
        for s in shifts[1:]:
            LB[i].addTerms(1,var[i,t],s)
    m.addConstraint(LB[i])


UB={} #dictionary for holding upper bound constraints
for t in periods:
    for s in shifts[1:]:
        UB[t,s]=Linear("UB_%s_%s"%(t,s),rhs=1) #weight is set to default (1)
        for i in range(len(staffs)):
            UB[t,s].addTerms(1,var[i,t],s)
        m.addConstraint(UB[t,s])

#forbid two different shifts on two consecutive days
Forbid={}
for i in range(len(staffs)):
    for t in periods:
        for s in shifts[1:]:
            Forbid[(i,t,s)]=Linear("Forbid_%s_%s_%s"%(i,t,s),1,1)
            Forbid[(i,t,s)].addTerms(1,var[i,t],s)
            for k in shifts[1:]:
                if k!=s:
                    if t==periods[-1]:
                        Forbid[(i,t,s)].addTerms(1,var[i,0],k)
                    else:
                        Forbid[(i,t,s)].addTerms(1,var[i,t+1],k)
            m.addConstraint(Forbid[(i,t,s)])


#shifts 2 and 3 must do at least two consecutive days
```

```
Cons={}
for i in range(len(staffs)):
     for t in periods:
        for s in shifts[2:]:
             Cons[(i,t)]=Linear("Cons_%s_%s"%(i,t),direction=">=")
             Cons[(i,t)].addTerms(-1,var[i,t],s)
             if t==0:
                  Cons[(i,t)].addTerms(1,var[i,periods[-1]],s)
             else:
                  Cons[(i,t)].addTerms(1,var[i,t-1],s)
             if t==periods[-1]:
                  Cons[(i,t)].addTerms(1,var[i,0],s)
             else:
                  Cons[(i,t)].addTerms(1,var[i,t+1],s)
             m.addConstraint(Cons[(i,t)])

m.Params.TimeLimit=1
sol,violated=m.optimize()

#print m

print "solution"
for x in sol:
    print x,sol[x]
print "violated constraint(s)"
for v in violated:
    print v,violated[v]
```

実行結果は，以下のようになり，容易にすべての制約を満たしたシフトを得ることができる．

```
solution
A1 3
A0 0
A3 3
A2 3
A5 2
A4 0
A6 2
C3 0
C2 2
C1 2
C0 2
C6 0
C5 3
C4 3
B4 1
B5 0
B6 3
B0 3
B1 0
B2 1
B3 1
D6 1
D4 2
D5 0
D2 0
D3 2
D0 1
D1 1
violated constraint(s)
```

5.2 n クイーン問題

制約計画の例題で頻繁に用いられるパズルとして n クイーン問題がある．この問題は，以下のように定義される．

n クイーン問題
$n \times n$ のチェス盤の上に，将棋の飛車と角行の動きを同時にできる駒（クイーン）をお互いに動きを妨げないように n 個置け．

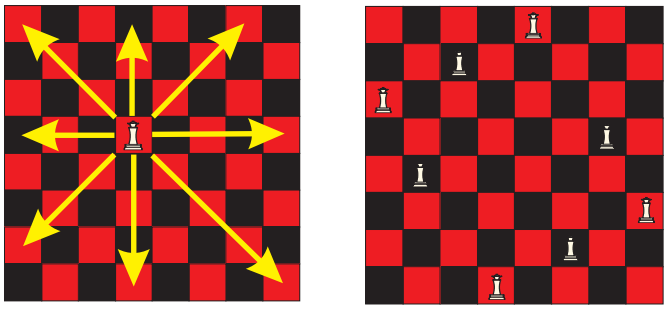

図 5.1: クイーンの動きと 8 クイーン問題の解の一例．

まず，各行に置くクイーンの位置（列番号）を変数とする．8 個のクイーンを配置する場合には，これらの変数の領域は $\{0, 1, ..., 7\}$ となる．列番号は互いに異なる必要があるので，これを `alldiff` 条件で記述する．また，クイーンが斜めの動きでお互いに取り合わないという条件は，線形制約とすることによってモデル化できる．

実は，この問題は最適化問題としてみると非常に簡単な問題である．問題を面白いものとするために，クイーンを i 行 j 列に配置したときに $i \times j$ の費用がかかるという目的関数を追加してみる．SCOP では，目的関数を表す制約 `obj` を，重み 1 の線形制約として追加するだけで良い．これは，以下の Python プログラムで記述できる．

```
from scop2 import *
m=Model()
n=8
varlist=[]
for i in range(n):
    varlist.append("x"+str(i))

var=m.addVariables(varlist,range(n))
con1=Alldiff("AD",var,"inf")
m.addConstraint(con1)

for k in range(2,2*n-1):
    con2=Linear("rightdown_%s"%k,"inf",1,"<=")
    for i in range(n):
        j=k-n+i
        if j>=0 and j<=n-1:
            con2.addTerms(1,var[i],j)
    m.addConstraint(con2)
for k in range(2,2*n-1):
    con3=Linear("leftdown_%s"%k,"inf",1,"<=")
    for i in range(n):
        j=k-i-1
        if j>=0 and j<=n-1:
            con3.addTerms(1,var[i],j)
    m.addConstraint(con3)
obj=Linear("obj",1,0,"<=")
for i in range(n):
```

```
    for j in range(n):
        obj.addTerms((i+1)*(j+1),var[i],j)
m.addConstraint(obj)
m.Params.TimeLimit=1
sol,violated=m.optimize()
print m
print "solution"
for x in sol:
    print x,sol[x]
print "violated constraint(s)"
for v in violated:
    print v,violated[v]
```

SCOP を用いて求解してみると，以下のようになる．

```
number of variables = 8
number of constraints= 28
variable x0:[0, 1, 2, 3, 4, 5, 6, 7]
variable x1:[0, 1, 2, 3, 4, 5, 6, 7]
variable x2:[0, 1, 2, 3, 4, 5, 6, 7]
variable x3:[0, 1, 2, 3, 4, 5, 6, 7]
variable x4:[0, 1, 2, 3, 4, 5, 6, 7]
variable x5:[0, 1, 2, 3, 4, 5, 6, 7]
variable x6:[0, 1, 2, 3, 4, 5, 6, 7]
variable x7:[0, 1, 2, 3, 4, 5, 6, 7]
AD: weight= inf  type=alldiff x0 x3 x2 x5 x1 x7 x6 x4 ;
rightdown_2: weight= inf type=linear 1(x6,0) 1(x7,1) <=1
rightdown_3: weight= inf type=linear 1(x5,0) 1(x6,1) 1(x7,2) <=1
rightdown_4: weight= inf type=linear 1(x4,0) 1(x5,1) 1(x6,2) 1(x7,3) <=1
rightdown_5: weight= inf type=linear 1(x3,0) 1(x4,1) 1(x5,2) 1(x6,3) 1(x7,4) <=1
rightdown_6: weight= inf type=linear 1(x2,0) 1(x3,1) 1(x4,2) 1(x5,3) 1(x6,4) 1(x7,5) <=1
rightdown_7: weight= inf type=linear 1(x1,0) 1(x2,1) 1(x3,2) 1(x4,3) 1(x5,4) 1(x6,5) 1(x7,6) <=1
rightdown_8: weight= inf type=linear 1(x0,0) 1(x1,1) 1(x2,2) 1(x3,3) 1(x4,4) 1(x5,5) 1(x6,6) 1(x7,7) <=1
rightdown_9: weight= inf type=linear 1(x0,1) 1(x1,2) 1(x2,3) 1(x3,4) 1(x4,5) 1(x5,6) 1(x6,7) <=1
rightdown_10: weight= inf type=linear 1(x0,2) 1(x1,3) 1(x2,4) 1(x3,5) 1(x4,6) 1(x5,7) <=1
rightdown_11: weight= inf type=linear 1(x0,3) 1(x1,4) 1(x2,5) 1(x3,6) 1(x4,7) <=1
rightdown_12: weight= inf type=linear 1(x0,4) 1(x1,5) 1(x2,6) 1(x3,7) <=1
rightdown_13: weight= inf type=linear 1(x0,5) 1(x1,6) 1(x2,7) <=1
rightdown_14: weight= inf type=linear 1(x0,6) 1(x1,7) <=1
leftdown_2: weight= inf type=linear 1(x0,1) 1(x1,0) <=1
leftdown_3: weight= inf type=linear 1(x0,2) 1(x1,1) 1(x2,0) <=1
leftdown_4: weight= inf type=linear 1(x0,3) 1(x1,2) 1(x2,1) 1(x3,0) <=1
leftdown_5: weight= inf type=linear 1(x0,4) 1(x1,3) 1(x2,2) 1(x3,1) 1(x4,0) <=1
leftdown_6: weight= inf type=linear 1(x0,5) 1(x1,4) 1(x2,3) 1(x3,2) 1(x4,1) 1(x5,0) <=1
leftdown_7: weight= inf type=linear 1(x0,6) 1(x1,5) 1(x2,4) 1(x3,3) 1(x4,2) 1(x5,1) 1(x6,0) <=1
leftdown_8: weight= inf type=linear 1(x0,7) 1(x1,6) 1(x2,5) 1(x3,4) 1(x4,3) 1(x5,2) 1(x6,1) 1(x7,0) <=1
leftdown_9: weight= inf type=linear 1(x1,7) 1(x2,6) 1(x3,5) 1(x4,4) 1(x5,3) 1(x6,2) 1(x7,1) <=1
leftdown_10: weight= inf type=linear 1(x2,7) 1(x3,6) 1(x4,5) 1(x5,4) 1(x6,3) 1(x7,2) <=1
leftdown_11: weight= inf type=linear 1(x3,7) 1(x4,6) 1(x5,5) 1(x6,4) 1(x7,3) <=1
leftdown_12: weight= inf type=linear 1(x4,7) 1(x5,6) 1(x6,5) 1(x7,4) <=1
leftdown_13: weight= inf type=linear 1(x5,7) 1(x6,6) 1(x7,5) <=1
leftdown_14: weight= inf type=linear 1(x6,7) 1(x7,6) <=1
obj: weight= 1 type=linear 1(x0,0) 2(x0,1) 3(x0,2) 4(x0,3) 5(x0,4) 6(x0,5) 7(x0,6) 8(x0,7) 2(x1,0)
4(x1,1) 6(x1,2) 8(x1,3) 10(x1,4) 12(x1,5) 14(x1,6) 16(x1,7) 3(x2,0) 6(x2,1) 9(x2,2) 12(x2,3) 15(x2,4)
 18(x2,5) 21(x2,6) 24(x2,7) 4(x3,0) 8(x3,1) 12(x3,2) 16(x3,3) 20(x3,4) 24(x3,5) 28(x3,6) 32(x3,7) 5(x4,0)
  10(x4,1) 15(x4,2) 20(x4,3) 25(x4,4) 30(x4,5) 35(x4,6) 40(x4,7) 6(x5,0) 12(x5,1) 18(x5,2) 24(x5,3) 30(x5,4)
   36(x5,5) 42(x5,6) 48(x5,7) 7(x6,0) 14(x6,1) 21(x6,2) 28(x6,3) 35(x6,4) 42(x6,5) 49(x6,6) 56(x6,7) 8(x7,0)
   16(x7,1) 24(x7,2) 32(x7,3) 40(x7,4) 48(x7,5) 56(x7,6) 64(x7,7) <=0

solution
x2 7
x3 3
```

```
x0 4
x1 2
x6 5
x7 1
x4 6
x5 0
violated constraint(s)
obj 150
```

目的関数を表す制約の逸脱量は 150 であり，後ろの（行番号が大きい）クイーンほど前に（列番号の小さい場所に）配置する解になっていることが分かる．

5.3 生産ラインへの投入順決定問題

ここでは，生産ラインへの車の投入順決定の例を示す．これは，Edward Tsang 著 “Foundations of constraint satisfaction” の例を改訂したものである．

コンベアー上に一直線に並んだ車の生産ラインを考える．このラインは，幾つかの作業場から構成され，それぞれの作業場では異なる作業が行われる．いま，4 種類の車を同じ生産ラインで製造しており，それぞれをモデル A, B, C, D とする．本日の製造目標は，それぞれ $30, 30, 20, 40$ 台である．

最初の作業場では，サンルーフの取り付けを行っており，これはモデル B, C だけに必要な作業である．次の作業場では，カーナビの取り付けが行われており，これはモデル A, C だけに必要な作業である．それぞれの作業場は長さをもち，サンルーフ取り付けは車 5 台分，カーナビ取り付けは車 3 台分の長さをもつ．また，作業場には作業員が割り当てられており，サンルーフ取り付けは 3 人，カーナビ取り付けは 2 人の作業員が配置されており，作業場の長さを超えない範囲で別々に作業を行う．

生産ラインへの車の投入順序をうまく決めないと，作業場の範囲内で作業を完了することができない．たとえば，C, A, A, B, C の順で投入すると，サンルーフ取り付けでは，3 人の作業員がそれぞれモデル C, B, C に対する作業を行うので間に合うが，カーナビ取り付けでは， 2 人の作業員では C, A, A の 3 台の車の作業を終えることができない．

これは，作業場の容量制約とよばれ，サンルーフ取り付けの作業場では，すべての連続する 5 台の車の中に，モデル B, C が高々 3 つ入っていること（カーナビ取り付けの作業場では，すべての連続する 3 台の車の中に，モデル A, C が高々 2 つ入っていること）が制約条件になる．

この問題は，$120 (= 30 + 30 + 20 + 40)$ 個の領域 $\{A, B, C, D\}$ をもつ変数を準備し，A, B, C, D の個数の合計がそれぞれ $30, 30, 20, 40$ という制約と，上記の容量制約を作業場ごとに定義すれば，SCOP で容易に解くことができる．

例題を解くためのプログラムは，以下のようになる．

```
1  from scop2 import *
2  m=Model()
3  Type=["A","B","C","D"] #car types
4  Number=[30,30,20,40]   #number of cars needed
5  n=sum(Number) #planning horizon
6  #1st line produces car type B and C that has a workplace with length 5 and 3 workers
7  #2nd line produces car type A anc C that has a workplace with length 3 and 2 workers
8  Need=[["B","C"],["A","C"]]
9  Length=[5,3]
10 Capacity=[3,2]
```

```
11
12 x={}
13 for i in range(n):
14     x[i]=m.addVariable("seq[%s]"%i,Type)
15
16 #production volume constraints
17 for i in range(len(Type)):
18     L1=Linear("req[%s]"%i,direction="=",rhs=Number[i])
19     for j in range(n):
20         L1.addTerms(1,x[j],Type[i])
21     m.addConstraint(L1)
22
23 for i in range(len(Length)):
24     for k in range(n-Length[i]+1):
25         L2=Linear("ub[%s_%s]"%(i,k),direction="<=",rhs=Capacity[i])
26         for t in range(k,k+Length[i]):
27             for j in range(len(Need[i])):
28                 L2.addTerms(1,x[t],Need[i][j])
29         m.addConstraint(L2)
30
31 m.Params.TimeLimit=1
32 m.Params.OutputFlag=False
33 sol,violated=m.optimize()
34
35 print "solution"
36 for x in sol:
37     print x,sol[x]
38 print "violated constraint(s)"
39 for v in violated:
40     print v,violated[v]
```

実行結果は，以下のようになり，容易にすべての制約を満たした投入順を得ることができる．

```
solution
seq[18]  C

seq[101]  A

seq[26]  D

中略

seq[94]  A

seq[28]  D

seq[98]  A

violated constraint(s)
```

5.4 時間割作成

最初の実際例は，時間割の作成である．時間割は，多くの教育機関で悩んでいる問題の 1 つであり，そのため時間割作成に関することだけを取り扱う国際会議まであるほどである．時間割作成問題は，複雑な制約が付いた困難な組合せ最適化問題である．整数計画としての定式化も可能であるが，その求解は極めて困難になる．ここでは，簡単な時間割作成のモデル化を説明し，SCOP によるモデリングの有用性を示す．

時間割作成の基本モデルは，以下の入力データを必要とする．

授業の集合： 数学，英語，社会などの授業があらかじめ与えられているものとする．授業には，それを教える教師と受ける学生も決められている．そのため，同じ時限に授業が行われると,（教師も学生も身体は 1 つであるので）困ることになる．したがって，同じ時限に行うことができない授業についての情報も与えられているものとする．

時限の集合： 週休 2 日で，1 限から 6 限までの授業が可能な場合には，月曜の 1 限から，金曜の 6 限までの連続する期が，時限の集合となる．この時限に授業を割り当てるのが，時間割作成の最初の目標である．

教室の集合: 授業を行うことができる教室の集合が与えられているものとする．ただし，授業と教室の相性もあるので，授業ごとに，その授業を行うことが可能な教室の集合が与えられているものとする．この割り当てを決めることが，時間割作成の第 2 の目標となる．

この問題は，SCOP を用いると，以下のように自然にモデル化できる．

授業に時限を割り当てることを，変数 X として表現する．また，授業に教室を割り当てることを別の変数 Y として表現する．同じ時限に授業ができないことを表現するためには，`alldiff` 型の制約を用いる．また，各時限，教室に割り当て可能な授業の数は高々1 つであるので，これは 2 次の制約 `quadratic` で表現する．

他の実際問題で必要な付加制約も，SCOP を用いて表現することが可能である．実際に，SCOP を用いたソルバーは，時間割作成の国際コンペティション（ITC-2007: International Timetabling Competition）において，優秀な成績を残している．

付 録A　scop.py

以下に SCOP の Python モジュール（scop2.py）のソースコードを示す.

```
# scop interface class
# file name scop2.py
# ver. 2.1 Copyright Log Opt Co., Ltd.
# by M. Kubo 2013

# define the path of the scp solver here:
SCOP = './scop.exe'
#SCOP = './bin/scop'

class Variable():
    """
    SCOP variable class. Variables are associated with a particular model.
    You can create a variable object by adding a variable to a model (using Model.addVariable or
        Model.addVariables)
    instead of by using a Variable constructor.
   """
    def __init__(self,name="",domain=[]):
        self.name=str(name)
        self.domain=list(domain)

    def __str__(self):
        return "variable "+str(self.name)+":"+str(self.domain)

class Parameters():
    """
    SCOP parameter class to control the operation of SCOP.

    TimeLimit:  Limits the total time expended (in seconds). Positive integer. Default=600.
    OutputFlag: Controls the output log. Boolean. Default=False (0).
    RandomSeed: Sets the random seed number. Integer. Default=1.
    Target: Sets the target penalty value;
            optimization will terminate if the solver determines that the optimum penalty value
            for the model is worse than the specified "Target." Non-negative integer. Default=0.
    """
    def __init__(self):
        self.TimeLimit=600
        self.OutputFlag=0
        self.RandomSeed=1
        self.Target =0

class Model(object):
    """
    SCOP model class.

    Attbibutes:
    constraints: Set of constraint objects in the model.
    variables: Set of variable objects in the model.
    Params:  Object including all the parameters of the model.
    varDict: Dictionary that maps variable names to their domains.

    """
    def __init__(self):
        self.constraints = [] # set of constraints is maintained by a list
        self.variables = []   # set of variables is maintained by a list
        self.Params=Parameters()
        self.varDict={}       # dictionary that maps variable names to their domains
```

```
    def __str__(self):
        """ return the information of the problem
            constraints are expanded and are shown in a readable format
        """
        ret="number of variables = "+str(len(self.variables))+" \n"
        ret+="number of constraints= "+str(len(self.constraints))+"\n"
        for v in self.variables:
            ret+=str(v)+"\n"

        for c in self.constraints:
            ret+=str(c)
        return ret

    def update(self):
        """
        prepare a string representing the current model in the scop input format
        """
        f = ""
        #variable declarations
        for var in self.variables:
            domainList = ",".join([str(i) for i in var.domain])
            f += "variable %s in { %s } \n" % (var.name, domainList)
        #target value declaration
        f += "target = %s \n" % str(self.target)
        #constraint declarations
        for con in self.constraints:
            f += str(con)
        return f

    def addVariable(self, name="", domain=[]):
        """
        addVariable ( name="", domain=[] )
        Add a variable to the model.

        Arguments:
        name: Name for new variable. A string object.
        domain: Domain (list of values) of new variable. Each value must be a string or numeric
            object.

        Return value:
        New variable object.

        Example usage:
        x = model.addVarriable("var")                       # domain  is set to []
        x = model.addVariable(name="var",domain=[1,2,3])    # arguments by name
        x = model.addVariable("var",["A","B","C"])          # arguments by position

        """
        var =Variable(name,domain)
        self.variables.append(var)
        # keep variable names using the dictionary varDict to check the validity of constraints
        self.varDict[var.name]=var.domain
        return var

    def addVariables(self, names=[], domain=[]):
        """
        addVariables(names=[], domain=[])
        Add variables and their (identical) domain.

        Arguments:
        names: list of new variables. A list of string objects.
        domain: Domain (list of values) of new variables. Each value must be a string or numeric
            object.

        Return value:
        List of new variable objects.

        Example usage:
        varlist=["var1","var2","var3"]
```

```
        x = model.addVariables(varlist)                          # domain is set to []
        x = model.addVariables(names=varlist,domain=[1,2,3]  # arguments by name
        x = model.addVariables(varlist,["A","B","C"]             # arguments by position

        """
        if type(names)!=type([]):
            print "The first argument (names) must be a list."
            raise NameError
        varlist=[]
        for var in names:
            varlist.append(self.addVariable(var,domain))
        return varlist

    def addConstraint(self, con):
        """
        addConstraint ( con )
        Add a constraint to the model.

        Argument:
        con: A constraint object (Linear, Quadratic or AllDiff).

        Example usage:
        model.addConstraint(L)

        """
        if not isinstance(con,Constraint):
            print "error: %r should be a subclass of Constraint" % con
            raise NameError

        #check the feasibility of the constraint added in the class con
        try:
            if con.feasible(self.varDict):
                self.constraints.append(con)
        except NameError:
            print "Consrtaint %r has an error " % con
            raise  NameError

    def optimize(self):
        """
        optimize ()
        Optimize the model using scop.exe in the same directory.

        Example usage:
        model.optimize()
        """
        time=self.Params.TimeLimit
        seed=self.Params.RandomSeed
        LOG=self.Params.OutputFlag
        self.target=self.Params.Target

        f = self.update()

        f3 = open("scop_input.txt","w")
        f3.write(f)
        f3.close()

        if LOG:
            print "scop input: \n"+f + "\n"
            print "solving using parameters: \n "
            print "  TimeLimit =%s second"%time
            print "  RandomSeed= %s"%seed
            print "  OutputFlag= %s"%LOG +"\n"
        import subprocess
        cmd = SCOP + " -time "+str(time)+" -seed "+str(seed) #solver call
        try:
            pipe = subprocess.Popen(cmd.split(), stdout=subprocess.PIPE, stdin=subprocess.PIPE)
        except OSError:
            print "error: could not execute command '%s'" % cmd
            print "please check that the solver is in the path"
            exit(0)
```

```
        out, err = pipe.communicate(f)                  #get the result
        if LOG:
            print "scop output: \n"+f
            print out, '\n'

        f = open("scop_out.txt","w")
        f.write(out)
        f.close()

        if err!=None:
            f2 = open("scop_error.txt","w")
            f2.write(err)
            f2.close()

        #extract the solution and the violated constraints
        s0 = "[best solution]"
        s1 = "penalty"
        s2 = "[Violated constraints]"
        i0 = out.find(s0) + len(s0)
        i1 = out.find(s1, i0)
        i2 = out.find(s2, i1) + len(s2)

        data = out[i0:i1].strip()
        sol = {}

        if data != "":
            for s in data.split("\n"):
                name, value = s.split(":")
                try:
                    temp=int(value)
                except:
                    sol[name] = value
                else:
                    sol[name]=int(value)

        data = out[i2:].strip()
        violated = {}
        if data != "":
            for s in data.split("\n"):
                try:
                    name, value = s.split(":")
                except:
                    print "Error String=",s

                try:
                    temp=int(value)
                except:
                    violated[name] = value
                else:
                    violated[name] = int(value)

        #return dictionaries containing the solution and the violated constraints
        return sol,violated


class Constraint(object):
    """
     Constraint base class
    """
    def setWeight(self,weight):
        self.weight = str(weight)


class Alldiff(Constraint):
    """
    Alldiff ( name=None, varlist=None, weight=1 )
    Alldiff type constraint constructor.

    Arguments:
    name: Name of all-different type constraint.
```

```
    varlist (optional): List of variables that must have differennt value indices.
    weight (optional): Positive integer representing importance of constraint.

    Attributes:
    name: Name of all-different type  constraint.
    varlist (optional): List of variables that must have differennt value indices.

    weight (optional): Positive integer representing importance of constraint.
    """

    def __init__(self,name=None,varlist=None,weight=1):
        if name==None or name=="":
            print "error: please specify the name when creating a constraint"
            raise NameError
        self.name = name
        self.weight = str(weight)

        if varlist==None:
            self.variables = set([])
        else:
            for var in varlist:
                if not isinstance(var,Variable):
                    print "error: %r should be a subclass of Variable" % var
                    raise NameError
            self.variables = set(varlist)

    def __str__(self):
        """ return the information of the alldiff constraint
        """
        f = self.name+": weight= "+self.weight+"  type=alldiff "
        for var in self.variables:
            f += var.name+" "
        f += "; \n"
        return f

    def addVariable(self,var):
        """
        addVariable ( var )
        Add new variable into all-different type constraint.

        Arguments:
        var: Variable object added to all-different type constraint.

        Example usage:

        AD.addVaeiable( x )

        """
        if not isinstance(var,Variable):
            print "error: %r should be a subclass of Variable" % var
            raise NameError

        if var in self.variables:
            print "duplicate variable name error when adding variable %r" % var
            return False
        self.variables.add(var)

    def addVariables(self, varlist):
        """
        addVariables ( varlist )
        Add variables into all-different type constraint.

        Arguments:
        varlist: List or tuple of variable objects added to all-different type constraint.

        Example usage:

        AD.addVariables( x, y, z )
        AD.addVariables( [x1,x2,x2] )
```

```
        """
        for var in varlist:
            self.addvar(var)

    def feasible(self,allvars):
        """
           return True if the constraint is defined correctly
        """
        for var in self.variables:
            if var.name not in allvars:
                print "no variable in the problem instance named %r" % var.name
                raise NameError
        return True


class Linear(Constraint):
    """
    Linear ( name, weight=1, rhs=0, direction="<=" )
    Linear constraint constructor.

    Arguments:
    name: Name of linear constraint.
    weight (optiona): Positive integer representing importance of constraint.
    rhs: Right-hand-side constant of linear constraint.
    direction: Rirection (or sense) of linear constraint; "<=" (default) or ">=" or "=".

    Attributes:
    name: Name of linear constraint.
    weight (optional): Positive integer representing importance of constraint.
    rhs: Right-hand-side constant of linear constraint.
    direction: Direction (or sense) of linear constraint; "<=" (default) or ">=" or "=".
    terms: List of terms in left-hand-side of constraint.
           Each term is a tuple of coeffcient,variable and its value.
    """
    def __init__(self,name,weight=1,rhs=0,direction="<="):
        """
        Constructor of linear constraint class:
        artuments are:

        """
        if name==None or name=="":
            print "error: please specify the name when creating a constraint"
            raise NameError

        self.name = name
        self.weight = str(weight)
        self.rhs = rhs
        self.direction = direction
        self.terms = []

    def __str__(self):
        """ return the information of the linear constraint
            the constraint is expanded and is shown in a readable format
        """
        f = self.name+": weight= "+self.weight+" type=linear "
        for (coeff,var,value) in self.terms:
            f += str(coeff)+"("+var.name +","+ str(value)+") "
        f += self.direction+str(self.rhs) +"\n"
        return f

    def addTerms(self,coeffs=[],vars=[],values=[]):
        """
        addTerms ( coeffs=[],vars=[],values=[] )
        Add new terms into left-hand-side of linear constraint.

        Arguments:
        coeffs: Coefficients for new terms; either a list of coefficients or a single
            coefficient.
                The three arguments must have the same size.
        vars: Variables for new terms; either a list of variables or a single variable.
```

```
                The three arguments must have the same size.
        values: Values for new terms; either a list of values or a single value.
                The three arguments must have the same size.

        Example usage:

        L.addTerms(1.0, y, "A")
        L.addTerms([2, 3, 1], [y, y, z], ["C", "D", "C"]) #2 X[y,"C"]+3 X[y,"D"]+1 X[z,"C"]

        """
        if type(coeffs) !=type([]): #need a check whether coeffs is numeric ...
            #arguments are nor list; add a term
            if type(coeffs)==type(1):
                self.terms.append( (coeffs,vars,values))
        elif type(coeffs)!=type([]) or type(vars)!=type([]) or type(values)!=type([]):
            print "coeffs, vars, values must be lists"
            raise TypeError
        elif len(coeffs)!=len(vars) or len(coeffs)!=len(values):
            print "length of coeffs, vars, values must be identical"
            raise TypeError
        elif len(coeffs) !=len(vars) or len(coeffs) !=len(values):
            print "error: length of coeffs, vars, and values must be identical"
            raise TypeError
        else:
            for i in range(len(coeffs)):
                self.terms.append( (coeffs[i],vars[i],values[i]))

    def setRhs(self,rhs=0):
        self.rhs = rhs

    def setDirection(self,direction="<="):
        if direction in ["<=",">=","="]:
            self.direction = direction
        else:
            print "direction setting error; direction should be one of '<=', '>=', or '='"
            raise NameError

    def feasible(self,allvars):
        """ return True if the constraint is defined correctly
        """
        for (coeff,var,value) in self.terms:
            if var.name not in allvars:
                print "no variable in the problem instance named %r" % var.name
                raise NameError
            if value not in allvars[var.name]:
                print "no value %r for the variable named %r" % (value, var.name)
                raise NameError
        return True


class Quadratic(Constraint):
    """
    Quadratic ( name, weight=1, rhs=0, direction="<=" )
    Quadratic constraint constructor.

    Arguments:
    name: Name of quadratic constraint.
    weight (optional): Positive integer representing importance of constraint.
    rhs: Right-hand-side constant of linear constraint.
    direction: Direction (or sense) of linear constraint; "<=" (default) or ">=" or "=".

    Attributes:
    name: Name of quadratic constraint.
    weight (optiona): Positive integer representing importance of constraint.
    rhs: Right-hand-side constant of linear constraint.
    direction: Direction (or sense) of linear constraint; "<=" (default) or ">=" or "=".
    terms: List of terms in left-hand-side of constraint.
           each term is a tuple of coeffcient, variable1, value1, variable2 and value2.
    """
```

```
    def __init__(self,name=None,weight=1,rhs=0,direction="<="):
        if name==None or name=="":
            print "error: please specify the name when creating a constraint"
            raise NameError
        self.name = name
        self.weight = str(weight)
        self.rhs = rhs
        self.direction = direction
        self.terms = []

    def __str__(self):
        """ return the information of the quadratic constraint
            the constraint is expanded and is shown in a readable format
        """
        f =  self.name+": weight= "+self.weight+" type=quadratic "
        for (coeff,var1,value1,var2,value2) in self.terms:
            f += str(coeff)+"("+var1.name +","+ str(value1)+") ("+var2.name +","+ str(value2)+")
                "
        f += self.direction+str(self.rhs) +"\n"
        return f

    def addTerms(self,coeffs=[],vars=[],values=[],vars2=[],values2=[]):
        """
        addTerms ( coeffs=[],vars=[],values=[],vars2=[],values2=[])
        Add new terms into left-hand-side of quadratic constraint.

        Arguments:
        coeffs: Coefficients for new terms; either a list of coefficients or a single
            coefficient.
                The five arguments must have the same size.
        vars: Variables for new terms; either a list of variables or a single variable.
                The five arguments must have the same size.
        values: Values for new terms; either a list of values or a single value.
                The five arguments must have the same size.
        vars2: Variables for new terms; either a list of variables or a single variable.
                The five arguments must have the same size.
        values2: Values for new terms; either a list of values or a single value.
                The five arguments must have the same size.
        Example usage:

        L.addTerms(1.0, y, "A", z, "B")
        L.addTerms([2, 3, 1], [y, y, z], ["C", "D", "C"], [x, x, y], ["A", "B", "C"])
                  #2 X[y,"C"] X[x,"A"]+3 X[y,"D"] X[x,"B"]+1 X[z,"C"] X[y,"C"]

        """
        if type(coeffs) !=type([]): #need a check whether coeffs is numeric ...
            self.terms.append( (coeffs,vars,values,vars2,values2))
        elif type(coeffs)!=type([]) or type(vars)!=type([]) or type(values)!=type([]) \
             or type(vars2)!=type([]) or type(values2)!=type([]):
            print "coeffs, vars, values must be lists"
            raise TypeError
        elif len(coeffs)!=len(vars) or len(coeffs)!=len(values) or len(values)!=len(vars) \
             or len(coeffs)!=len(vars2) or len(coeffs)!=len(values2):
            print "length of coeffs, vars, values must be identical"
            raise TypeError
        else:
            for i in range(len(coeffs)):
                self.terms.append( (coeffs[i],vars[i],values[i],vars2[i],values2[i]))

    def setRhs(self,rhs=0):
        self.rhs = rhs

    def setDirection(self,direction="<="):
        if direction in ["<=", ">=", "="]:
            self.direction = direction
        else:
            print "direction setting error"

    def feasible(self,allvars):
        """ return True if the constraint is defined correctly
```

```
        """
        for (coeff,var1,value1,var2,value2) in self.terms:
            if var1.name not in allvars:
                print "no variable in the problem instance named %r" % var1.name
                raise NameError
            if var2.name not in allvars:
                print "no variable in the problem instance named %r" % var2.name
                raise NameError
            if value1 not in allvars[var1.name]:
                print "no value %r for the variable named %r" % (value1, var1.name)
                raise NameError
            if value2 not in allvars[var2.name]:
                print "no value %r for the variable named %r" % (value2, var2.name)
                raise NameError
        return True
```