

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

scop.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

- このソフトウェアおよびマニュアルの著作権は LOGOPT 社にあります。
- このソフトウェアおよびマニュアルの一部または全部を無断で複製することはできません。
- このソフトウェアおよびマニュアルを運用した結果の影響については、一切責任を負いかねますのでご了承下さい。
- このマニュアルに記載されている事柄は、将来予告なしに変更することがあります。

目次

第 1 章	はじめに	3
第 2 章	scop.py クラスライブラリの使用法	4
2.1	問題クラス Problem	4
2.2	線形制約クラス Linear	5
2.3	2 次制約クラス Quadratic	6
2.4	相異制約クラス Alldiff	7
第 3 章	scop.py の使用例	8
3.1	仕事の割当 1	8
3.2	仕事の割当 2	10
3.3	仕事の割当 3	12
3.4	ビンパッキング問題	13
3.5	スタッフスケジューリング	15
第 4 章	代表的な組合せ最適化問題	18
4.1	グラフ分割問題	18
4.2	最大安定集合問題	20
4.3	グラフ彩色問題	22
4.4	2 次割当問題	25
4.5	巡回セールスマン問題	30
4.6	多制約ナップサック問題	32
付録 A	scop.py	34

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, 他的高级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 (scop.py) の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ある。

第2章 scop.py クラスライブラリの使用法

scop.py は、以下のクラスから構成されている。

- 問題クラス Problem
- 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 問題クラス Problem

Python から SCOP を呼び出して使うときに、最初にすべきことは問題クラスのオブジェクトを生成することである。たとえば、「問題名」と名付けた問題を生成したいときには、以下のように記述する。

```
問題名=Problem()
```

Problem クラスは、以下のメソッド（「問題名.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)

問題に 1 つの変数を追加する。引数の変数名は文字列 (string) で与え、領域はリスト (list) 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。

addVariables(変数のリスト, 領域)

問題に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と、共通の領域を表すリストである。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。

addConstraint(制約オブジェクト)

制約オブジェクトを問題に追加する。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り、以下の項で解説する。

`solve`(制限時間, 乱数の種)

問題の求解を行う。引数の制限時間と乱数の種はオプションであり、省略してもかまわない。制限時間の規定値は 600 (秒)、乱数の種の規定値は 1 である。制限時間を変更したい場合には、第 1 引数に正数値を入れるか、「time=正数値」を引数に入れて関数呼び出しを行う。乱数の種を変更したい場合には、第 2 引数に数値を入れるか、「seed=数値」を引数に入れて関数呼び出しを行う。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。返値は、解 (変数名をキー、値を領域の要素とした辞書) と逸脱した制約 (制約名をキー、逸脱量を値とした辞書) のタプルである。

また、問題クラス `Problem` は、問題の情報を文字列として返すことができる。たとえば、「問題名」と名付けた問題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 問題名
```

で得ることができる。

2.2 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)
```

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。) 重みは、制約の重要性を表す正数もしくは文字列 “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。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。

`Linear` クラスは、以下のメソッド (「問題名.メソッド名」でアクセスするメンバ関数) をもつ。

`addTerm`(係数, 変数名, 値)

線形制約 (の左辺) に 1 つの項を追加する。変数名が値をとるときに 1、それ以外るとき 0 となる変数を x [変数名, 値] とすると、追加される項は、

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

と記述される。引数の係数は整数値、変数名は文字列 (string) 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。

`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。

2.3 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)
```

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。

Quadratic クラスは，以下のメソッド（「問題名.メソッド名」でアクセスするメンバ関数）をもつ。

addTerm(係数,変数名 1,値 1,変数名 2,値 2)

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。変数名 $i(i = 1, 2)$ が値 i をとるときに 1，それ以外のとき 0 となる変数を x [変数名 i , 値 i] とすると，追加される項は，

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

と記述される。引数の係数は整数値，変数名は文字列（string）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。

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。

2.4 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。

```
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、問題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(変数名)

相異制約の変数を 1 つ制約に追加する。

addVariables(変数名のリスト)

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、scop.py の使用法を解説する。

第3章 scop.py の使用例

3.1 仕事の割り当 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。

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

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、scop モジュールを読み込み、問題クラス Problem のオブジェクト p を生成する。

```
import scop
p=scop.Problem()
```

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、この p に対して変数を追加する。問題クラスの 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。

```
p.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=scop.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ため、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=scop.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("<=")

```

最後に、生成した制約 `con1`、`con2` を問題 `p` に `addConstraint` メソッドを用いて追加し (1,2 行目)、`solve` メソッドで解を探索する (3 行目)。ここで、`solve` メソッドで名前付き引数で指定した `time=1` は、制限時間 1 秒で探索することを指定している。

```

1 p.addConstraint(con1)
2 p.addConstraint(con2)
3 sol,violated=p.solve(time=1)

```

`solve` の返値は、解と逸脱した制約の辞書であるので、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]

```

問題を確認するには、問題クラスのオブジェクト `p` を `print` で表示させれば良い。

```
print p
```

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 C B
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

```

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。

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

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。

3.2 仕事の割り当て

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。

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

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 SCOP を用いて解くための Python プログラムは、以下のよう書ける。

```
1 import scop
2 p=scop.Problem()
3 workers=["A","B","C","D","E"]
4 Cost=[[15, 20, 30],[7, 15, 12],[25,10,13],[15,18,3],[5,12,17]]
5 LB=[1,2,2]
6 p.addVariables(workers,range(3))
7 LB_Constraint={} #dictionary for keeping lower bound constraints
8 for j in range(3):
9     LB_Constraint[j]=scop.Linear("LB_constraint"+str(j),"inf")
10    for i in range(len(workers)):
11        LB_Constraint[j].addTerm(1,workers[i],j)
```

```

12     LB_Constraint[j].setRhs(LB[j])
13     LB_Constraint[j].setDirection(">=")
14     p.addConstraint(LB_Constraint[j])
15     con1=scop.Linear("linear_constraint")
16     for i in range(len(workers)):
17         for j in range(3):
18             con1.addTerm(Cost[i][j],workers[i],j)
19     p.addConstraint(con1)
20     sol,violated=p.solve(time=1)
21
22     print p
23
24     print "solution"
25     for x in sol:
26         print x,sol[x]
27     print "violated constraint(s)"
28     for v in violated:
29         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。

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．

	0	1	2
A	15	20	30
B	7	15	12
C	25	10	13
D	15	18	3
E	5	12	17

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．

3.4 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 3.1）．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のように書ける．

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 のビンが無数準備されている．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，必要なビンの数を最小にするような詰めかたを求めよ．

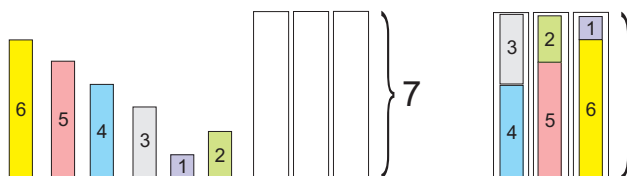


図 3.1: 箱詰め担当部長のビンパッキング問題の問題例と SCOP による解．

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、以下ようになる。

```
1  bpp=Problem()
2
3  Items=[6,5,4,3,1,2]
4  B=7
5  num_bins=3
6  n=len(Items)
7
8  for i in range(n):
9      bpp.addVariable("Item"+str(i),range(num_bins))
10
11  Bin={}
12  for j in range(num_bins):
13      Bin[j]=Linear("Bin"+str(j),weight="inf")
14      Bin[j].setWeight(1)
15      Bin[j].setDirection("<=")
16      Bin[j].setRhs(B)
17      for i in range(n):
18          Bin[j].addTerm(Items[i],"Item"+str(i),j)
19      bpp.addConstraint(Bin[j])
20
21  sol,violated=bpp.solve(time=1,seed=3)
22
23  print bpp
24
25  print "solution="
26  for i in sol:
27      print i,sol[i]
28
29  print "violated constraints=",violated
```

実行結果は、以下ようになる。

```
number of variables = 6
number of constraints= 3
variable Item0:[0, 1, 2]
variable Item1:[0, 1, 2]
variable Item2:[0, 1, 2]
variable Item3:[0, 1, 2]
variable Item4:[0, 1, 2]
variable Item5:[0, 1, 2]
Linear: Bin0: weight=1
6*x[Item0:0] 5*x[Item1:0] 4*x[Item2:0] 3*x[Item3:0] 1*x[Item4:0] 2*x[Item5:0] <=7
Linear: Bin1: weight=1
6*x[Item0:1] 5*x[Item1:1] 4*x[Item2:1] 3*x[Item3:1] 1*x[Item4:1] 2*x[Item5:1] <=7
Linear: Bin2: weight=1
6*x[Item0:2] 5*x[Item1:2] 4*x[Item2:2] 3*x[Item3:2] 1*x[Item4:2] 2*x[Item5:2] <=7

solution=
Item2 0
Item3 0
Item0 2
Item1 1
Item4 2
Item5 1
violated constraints= {}
```

3.5 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 $0, 1, \dots, 6$ (`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、以下ようになる。

```

1 import scop
2 p=scop.Problem()
3 periods=range(7)
4 shifts=range(4) #three shifts named 0 (off), 1, 2, and 3
5 staffs=["A","B","C","D"]
6 variables=[] #list of variables
7 for i in staffs:
8     for t in periods:
9         variables.append(i+str(t))
10
11 p.addVariables(variables, shifts)
12
13 LB={} #dictionary for holding lower bound constraints
14 for i in staffs:
15     LB[i]=scop.Linear("LB"+i) #weight is set to default (1)
16     for t in periods:
17         for s in shifts[1:]:
18             LB[i].addTerm(1, i+str(t), s)
19     LB[i].setRhs(5)
20     LB[i].setDirection(">=")
21     p.addConstraint(LB[i])
22
23 UB={} #dictionary for holding upper bound constraints
24 for t in periods:
25     for s in shifts[1:]:
26         UB[(t, s)]=scop.Linear("UB"+str(t)+str(s)) #weight is set to default (1)
27         for i in staffs:
28             UB[(t, s)].addTerm(1, i+str(t), s)
29         UB[(t, s)].setRhs(1)
30         UB[(t, s)].setDirection("<=")
31         p.addConstraint(UB[(t, s)])
32
33 #forbid two different shifts on two consecutive days
34 Forbid={}
35 for i in staffs:
36     for t in periods:
37         for s in shifts[1:]:
38             Forbid[(i, t, s)]=scop.Linear("Forbid"+i+str(t)+str(s))
39             Forbid[(i, t, s)].addTerm(1, i+str(t), s)
40             for k in shifts[1:]:
41                 if k!=s:
42                     if t==periods[-1]:
43                         Forbid[(i, t, s)].addTerm(1, i+str(0), k)
44                     else:
45                         Forbid[(i, t, s)].addTerm(1, i+str(t+1), k)
46             Forbid[(i, t, s)].setRhs(1)
47             Forbid[(i, t, s)].setDirection("<=")
48             p.addConstraint(Forbid[(i, t, s)])
49
50 #shifts 2 and 3 must do at least two consecutive days
51 Cons={}
52 for i in staffs:
53     for t in periods:
54         for s in shifts[2:]:
55             Cons[(i, t)]=scop.Linear("Cons"+i+str(t))
56             Cons[(i, t)].addTerm(-1, i+str(t), s)
57             if t==0:
58                 Cons[(i, t)].addTerm(1, i+str(periods[-1]), s)
59             else:
60                 Cons[(i, t)].addTerm(1, i+str(t-1), s)
61             if t==periods[-1]:
62                 Cons[(i, t)].addTerm(1, i+str(0), s)

```

```

63         else:
64             Cons[(i, t)].addTerm(1, i+str(t+1), s)
65             Cons[(i, t)].setRhs(0)
66             Cons[(i, t)].setDirection(">=")
67             p.addConstraint(Cons[(i, t)])
68
69 sol, violated=p.solve(time=1)
70 print "solution"
71 for x in sol:
72     print x, sol[x]
73 print "violated constraint(s)"
74 for v in violated:
75     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)

```

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 モジュールの使用法を解説する。また、モデル化の比較のため、数理計画ソルバー Gurobi¹を Python から呼び出して求解するためのプログラムも示す。

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？

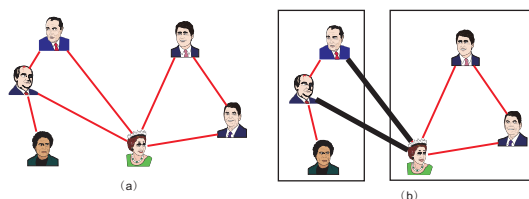


図 4.1: グラフ分割問題の例。(a) 線の引いてある人同士は仲良しであることを表すグラフ。(b) 仲良しが別のチームになることを最小にする等分割。違うチームに所属する仲良しのペアは太線で表されており、2 本である。よって、この等分割の目的関数値は 2 となる。

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、次のように書ける。

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

¹Gurobi Optimizer は、<http://www.gurobi.com/>からダウンロードできる。本マニュアルは、Gurobi Optimization Inc. からの正式な承認を得て紹介している。

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。

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

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。

```
import scop
p=scop.Problem()
nodes=["n"+str(i) for i in range(6)]
adj=[[1,4],[0,2,4],[1],[4,5],[0,1,3,5],[3,4]]
n=len(nodes)
p.addVariables(nodes,[0,1])
con1=scop.Linear("constraint","inf")
for i in nodes:
    con1.addTerm(1,i,1)
con1.setRhs(n/2)
con1.setDirection("=")
p.addConstraint(con1)
con2=scop.Quadratic("obj")
for i in range(n):
    for j in adj[i]:
        con2.addTerm(1,nodes[i],1,nodes[j],0)
        con2.addTerm(1,nodes[i],0,nodes[j],1)
con2.setRhs(0)
con2.setDirection("<=")
p.addConstraint(con2)
p.solve(1)
```

市販の数理計画ソルバーは、通常、上のように（下に凸でない）2 次の項を目的関数に含んだ最小化問題には対応していない。したがって、一般的な（混合）整数計画ソルバーで求解するためには、2 次の項を線形関数に変形してから解く必要がある。

枝 (i, j) が L と R の間にあるとき、言い換えれば $i \in L, j \in R$ 、もしくは $i \in R, j \in L$ が成立するとき 1、それ以外のとき 0 になる 0-1 変数 y_{ij} を導入する。すると、上の 2 次整数計画問題は、以下の線形整数計画に帰着される。

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

数理計画ソルバー Gurobi を Python から呼び出して最適化するプログラムを以下に示す。

```

from gurobipy import *
n=6
nodes=range(n)
adj=[[1,4],[0,2,4],[1],[4,5],[0,1,3,5],[3,4]]
model=Model("gpp")
x={}
y={}
for i in range(n):
    x[i]=model.addVar(0,1,0,GRB.BINARY,"x"+str(i))
    for j in adj[i]:
        y[i,j]=model.addVar(0,1,1,GRB.BINARY,"y"+str(i)+str(j))
model.update()
con1=LinExpr()
for i in range(n):
    con1.addTerms(1,x[i])
model.addConstr(lhs=con1,sense=GRB.EQUAL,rhs=n/2,name="constraint1")
for i in range(n):
    for j in adj[i]:
        con2=LinExpr()
        con2.addTerms(1,x[i])
        con2.addTerms(-1,x[j])
        con2.addTerms(-1,y[i,j])
        model.addConstr(lhs=con2,sense=GRB.LESS_EQUAL,rhs=0,name="con2"+str(i)+str(j))
        con3=LinExpr()
        con3.addTerms(1,x[j])
        con3.addTerms(-1,x[i])
        con3.addTerms(-1,y[i,j])
        model.addConstr(lhs=con3,sense=GRB.LESS_EQUAL,rhs=0,name="con3"+str(i)+str(j))
model.update()
model.write("gpp.lp")
model.optimize()
print "Opt. value=",model.ObjVal
for v in model.getVars():
    if v.X>0.001:
        print v.VarName,v.X

```

4.2 最大安定集合問題

あなたは 6 人のお友達から何人が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だろうか？

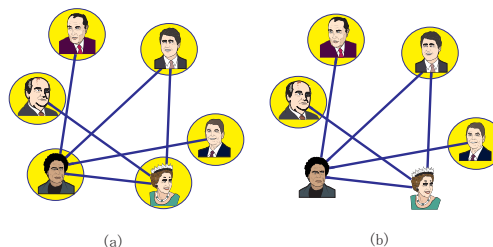


図 4.2: 最大安定集合問題の例。(a) 線の引いてある人同士は仲が悪いことを表すグラフ。(b) 仲が悪い同士を連れて行かないでピクニックに行くときの最大人数。丸で囲んだ人を連れて行くと目的関数値は 4 となり、これが最適解になる。

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。

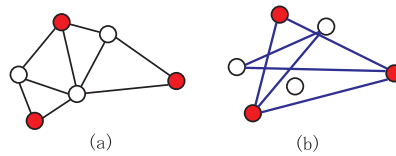


図 4.3: (a) 最大安定集合 . (b) 補グラフ上の最大クリーク .

最大安定集合問題は、次のように定義される問題である .

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、それ以外るとき 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 .

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

上の定式化を SCOP を用いて求解するには、以下のようによれば良い .

```
import scop
p=scop.Problem()
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)
p.addVariables(nodes,[0,1])
for i in range(n):
    for j in adj[i]:
        if i<j:
            con1=scop.Linear("constraint")
            con1.addTerm(1,nodes[i],1)
            con1.addTerm(1,nodes[j],1)
            con1.setRhs(1)
            con1.setDirection("<=")
            p.addConstraint(con1)
obj=scop.Linear("obj")
for i in range(n):
```

```

obj.addTerm(1,nodes[i],1)
obj.setRhs(n)
obj.setDirection(">=")
p.addConstraint(obj)
p.solve(1)

```

ちなみに数理計画ソルバー Gurobi によるプログラムは、以下ようになる。

```

from gurobipy import *
n=6
nodes=range(n)
adj=[[2],[3],[0,3,4,5],[1,2,5],[2],[2,3]]
model=Model("ssp")
x={}
for i in range(n):
    x[i]=model.addVar(0,1,1,GRB.BINARY,"x"+str(i))
model.update()
for i in range(n):
    for j in adj[i]:
        if i<j:
            con1=LinExpr()
            con1.addTerms(1,x[i])
            con1.addTerms(1,x[j])
            model.addConstr(lhs=con1,sense=GRB.LESS_EQUAL,rhs=1,name="edge"+str(i)+str(j))

model.ModelSense=-1
model.update()
model.write("ssp.lp")
model.optimize()
print "Opt.value=",model.ObjVal
for v in model.getVars():
    if v.X>0.001:
        print v.VarName,v.X

```

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？

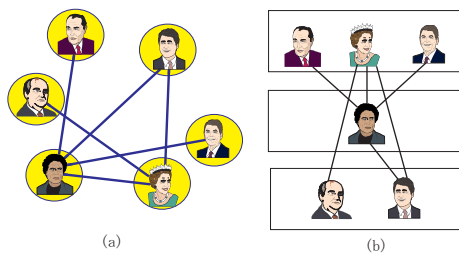


図 4.4: グラフ彩色問題の例。(a) 線の引いてある人同士は仲が悪いことを表すグラフ。(b) 3 つのクラスに分けると仲の悪い友達と同じクラスに入らない。目的関数値(クラス数)は 3 となり、これが最適解になる。

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 $\Upsilon = \{V_1, \dots, V_K\}$ 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 ($i = 1, \dots, K$) 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 $\Upsilon = \{V_1, \dots, V_K\}$ 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 $\{1, 2, \dots, K\}$ 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.

```
import scop
p=scop.Problem()
K=3
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)
p.addVariables(nodes, range(K))
for i in range(n):
    for j in adj[i]:
        if i<j:
            con1=scop.Alldiff("alldiff"+str(i)+str(j), [nodes[i], nodes[j]], "inf")
            p.addConstraint(con1)
p.solve()
```

グラフ彩色問題の整数計画問題として定式化を行うために, 彩色可能な色数の上界 K_{\max} が分かっているものと仮定する. すなわち, 最適な K 彩色は, $1 \leq K \leq K_{\max}$ の整数から選択される.

点 i に塗られた色が k のとき 1, それ以外のとき 0 の 0-1 変数 x_{ik} と, 色クラス V_k に含まれる点が 1 つでもあるときに 1, それ以外の (色クラスが空の) とき 0 の 0-1 変数 y_k を用いると, グラフ彩色問題は, 以下のように定式化できる.

$$\begin{aligned} & \text{minimize} && \sum_{k=1}^{K_{\max}} y_k \\ & \text{subject to} && \sum_{k=1}^{K_{\max}} x_{ik} = 1 \quad \forall i \in V \\ & && x_{ik} + x_{jk} \leq y_k \quad \forall (i, j) \in E, k = 1, \dots, K_{\max} \\ & && x_{ik} \in \{0, 1\} \quad \forall i \in V, k = 1, \dots, K_{\max} \\ & && y_k \in \{0, 1\} \quad \forall k = 1, \dots, K_{\max} \end{aligned}$$

上の定式化における最初の制約は, 各点 i に必ず 1 つの色が塗られることを表す. 2 番目の制約は, 枝 (i, j) の両端点の点 i と点 j が, 同じ色クラスに割り当てられることを禁止する制約

$$x_{ik} + x_{jk} \leq 1 \quad \forall (i, j) \in E, k = 1, \dots, K_{\max}$$

と変数 x と y の繋ぎ条件 ($y_k = 1$ のときのみ色 k で彩色可能)

$$\sum_{i \in V} x_{ik} \leq ny_k \quad \forall k = 1, \dots, K_{\max}$$

を同時に表したものである .

市販の数理計画ソルバーの多くは、分枝限定法とよばれる解法を利用して求解している . 上のグラフ彩色問題の定式化においては、色クラスはすべて無記名で扱われ、解に対称性がある . たとえば、解 $V_1 = \{1, 2, 3\}, V_2 = \{4, 5\}$ と解 $V_1 = \{4, 5\}, V_2 = \{1, 2, 3\}$ はまったく同じものであるが、上の定式化では異なるベクトル x, y で表される . この場合、変数 x, y をもとに分枝しても下界が改良されない現象が発生する . グラフ彩色問題を、市販の数理計画ソルバーで求解する際には、解の対称性を避けるために、以下の制約を付加することが推奨される .

$$y_k \geq y_{k+1} \quad \forall k = 1, \dots, K_{\max} - 1$$

上の制約は、添え字の小さい色クラスを優先して用いることを規定し、この制約を加えるだけで求解時間が劇的に改善する場合もある .

数理計画ソルバー Gurobi によるプログラムの例を以下に示す .

```

from gurobipy import *
model=Model("gcp")
K=3
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)
x={}
y={}
for i in range(n):
    for k in range(K):
        x[i,k]=model.addVar(0,1,0,GRB.BINARY,"x"+str(i)+str(k))
for k in range(K):
    y[k]=model.addVar(0,1,1,GRB.BINARY,"y"+str(k))

model.update()

for i in range(n):
    lin=LinExpr()
    for k in range(K):
        lin.addTerms(1,x[i,k])
    model.addConstr(lhs=lin,sense=GRB.EQUAL,rhs=1,name="constraint1_"+str(i))

for i in range(n):
    for j in adj[i]:
        if i<j:
            for k in range(K):
                lin=LinExpr()
                lin.addTerms(1,x[i,k])
                lin.addTerms(1,x[j,k])
                lin.addTerms(-1,y[k])
                model.addConstr(lhs=lin,sense=GRB.LESS_EQUAL,rhs=0,
                    name="constraint2_"+str(i)+str(j)+str(k))

for k in range(K-1):
    lin=LinExpr()
    lin.addTerms(1,y[k])
    lin.addTerms(-1,y[k+1])
    model.addConstr(lhs=lin,sense=GRB.LESS_EQUAL,rhs=0,name="constraint3_"+str(k))

model.update()
model.write("gcp.lp")

```

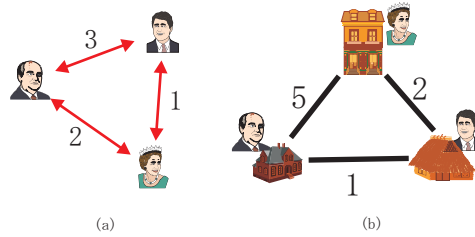


図 4.5: 2 次割当問題の例 . (a) 3 人のお友達が打ち合わせをする頻度 . 枝の上の数字は週に何回打ち合わせをするかを表す . (b) 3 箇所の家とお友達の割り当ての例 . 枝の上の数字は地点間の距離を表す . お友達同士の行き来する頻度と割り当てられた家との距離の和は, $2 \times 5 + 3 \times 1 + 1 \times 2 = 15$ となり, この割り当ての目的関数値は, その 2 倍 (お友達同士は往復するから) で $2 \times 15 = 30$ となる .

```

model.optimize ()
print "Opt. value=" , model.ObjVal
for v in model.getVars ():
    if v.X>0.001:
        print v.VarName, v.X

```

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 . 家との移動距離は, 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, 以下ようになる .

2 次割当問題

2 次割当問題とは, 集合 $V = \{1, \dots, n\}$ 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 $\pi : V \rightarrow \{1, \dots, n\}$ 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 箇所の施設があり, 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 $F = [f_{ij}]$ ， $D = [d_{kl}]$ 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， $2 \times 5 + 3 \times 2 + 1 \times 1 = 17$ 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 (4.5 節) を特別な場合として含んでいる．帰着は容易であり，

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

と設定すれば良い．

施設 i が地点 k に配置されるとき 1，それ以外るとき 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．

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

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．

```
import scop
n=3
d=[[0,2,3],[2,0,1],[3,1,0]]
f=[[0,5,1],[5,0,2],[1,2,0]]
p=scop.Problem()
nodes=["n"+str(i) for i in range(n)]
p.addVariables(nodes, range(n))
con1=scop.Alldiff("AD", nodes, "inf")
p.addConstraint(con1)
obj=scop.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.addTerm(f[i][j]*d[k][ell], nodes[i], k, nodes[j], ell)
p.addConstraint(obj)
p.solve(1)

```

市販の数値計画ソルバーは、通常、上のような（下に凸でない）2 次関数を含んだ最小化問題には対応していない。整数線形計画に帰着させるためには、様々な方法が提案されているが、ここでは 2 つの定式化を紹介しよう。

はじめの定式化は、施設 i を地点 k に配置したときの、施設 i と他の施設の間の費用の合計を表す実数変数 w_{ik} を追加したものであり、 $O(n^2)$ 個の変数を用いたコンパクトなものである。

施設 i を地点 k に配置したときの費用の上界となるパラメータ M_{ik} を導入する。

$$M_{ik} = \sum_{j, \ell \in V} f_{ij} d_{k\ell}$$

線形整数計画による定式化は以下のように書ける。

$$\begin{aligned}
 & \text{minimize} && \sum_{i, k \in V} w_{ik} \\
 & \text{subject to} && \sum_{i \in V} x_{ik} = 1 && \forall k \in V \\
 & && \sum_{k \in V} x_{ik} = 1 && \forall i \in V \\
 & && M_{ik}(x_{ik} - 1) + \sum_{j, \ell \in V} f_{ij} d_{k\ell} x_{j\ell} \leq w_{ik} && \forall i, k \in V \\
 & && x_{ik} \in \{0, 1\} && \forall i, k \in V \\
 & && w_{ik} \geq 0 && \forall i, k \in V
 \end{aligned}$$

この定式化は変数の数は少ないが、得られる下界が弱い。この定式化を数値計画ソルバー Gurobi で記述すると、以下ようになる。

```

from gurobipy import *
n=3
d=[[0,2,3],[2,0,1],[3,1,0]]
f=[[0,5,1],[5,0,2],[1,2,0]]
model=Model("qap")
w={}
x={}
for i in range(n):
    for j in range(n):
        w[i,j]=model.addVar(0,100000,1,GRB.CONTINUOUS,"w"+str(i)+str(j))
for i in range(n):
    for j in range(n):
        x[i,j]=model.addVar(0,1,0,GRB.BINARY,"x"+str(i)+str(j))

model.update()
M={}
for i in range(n):
    for k in range(n):
        M[i,k]=0.0
        for j in range(n):
            for ell in range(n):
                M[i,k]+=f[i][j]*d[k][ell]

for i in range(n):
    lin=LinExpr()

```

```

for j in range(n):
    lin.addTerms(1,x[i,j])
model.addConstr(lhs=lin,sense=GRB.EQUAL,rhs=1,name="assign1_"+str(i))

for j in range(n):
    lin=LinExpr()
    for i in range(n):
        lin.addTerms(1,x[i,j])
    model.addConstr(lhs=lin,sense=GRB.EQUAL,rhs=1,name="assign2_"+str(j))

for i in range(n):
    for k in range(n):
        lin=LinExpr()
        lin.addTerms(M[i,k],x[i,k])
        lin.addTerms(-1,w[i,k])
        for j in range(n):
            for ell in range(n):
                if i==j and k==ell:
                    pass
                else:
                    lin.addTerms(f[i][j]*d[k][ell],x[j,ell])
        model.addConstr(lhs=lin,sense=GRB.LESS_EQUAL,rhs=M[i,k],name="const"+str(i)+str(k))

model.update()
model.write("qap.lp")
model.Params.MIPfocus=3
model.optimize()
print "Opt. value=",model.ObjVal
for v in model.getVars():
    if v.X>0.001:
        print v.VarName,v.X

```

次の定式化は、施設 i が地点 k に配置され、かつ施設 j が地点 ℓ に配置されるとき 1、それ以外るとき 0 となる 4 添え字の 0-1 変数 $y_{ikj\ell}$ を用いたものである。変数の数は $O(n^4)$ 個と多いが、得られる下界は強くなり、 $n = 10$ 程度の問題まで市販の数値計画ソルバーで解くことができる（ただし、 $n = 10$ 程度の問題ならすべての順列を列挙するいわゆる全列挙法でも解くことができる。）線形整数計画による定式化は以下のように書ける。

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

この定式化を数値計画ソルバー Gurobi で記述すると、以下のようになる。

```

from gurobipy import *
n=3
d=[[0,2,3],[2,0,1],[3,1,0]]
f=[[0,5,1],[5,0,2],[1,2,0]]
model=Model("qap2")
x={}
y={}

```

```

for i in range(n):
    for j in range(n):
        x[i,j]=model.addVar(0,1,0,GRB.BINARY,"x"+str(i)+str(j))

for i in range(n):
    for j in range(n):
        if i!=j:
            for k in range(n):
                for ell in range(n):
                    if k!=ell:
                        y[i,k,j,ell]=model.addVar(0,1,f[i][j]*d[k][ell],
                                                    GRB.BINARY,"y"+str(i)+str(k)+str(j)+str(ell))
model.update()

for i in range(n):
    lin=LinExpr()
    for j in range(n):
        lin.addTerms(1,x[i,j])
    model.addConstr(lhs=lin,sense=GRB.EQUAL,rhs=1,name="assign1_"+str(i))

for j in range(n):
    lin=LinExpr()
    for i in range(n):
        lin.addTerms(1,x[i,j])
    model.addConstr(lhs=lin,sense=GRB.EQUAL,rhs=1,name="assign2_"+str(j))

for k in range(n):
    for j in range(n):
        for ell in range(n):
            if k!=ell:
                lin=LinExpr()
                lin.addTerms(-1,x[j,ell])
                for i in range(n):
                    if i!=j:
                        lin.addTerms(1,y[i,k,j,ell])
                model.addConstr(lhs=lin,sense=GRB.LESS_EQUAL,
                                rhs=0,name="const1"+str(k)+str(j)+str(ell))

for i in range(n):
    for j in range(n):
        for ell in range(n):
            if i!=j:
                lin=LinExpr()
                lin.addTerms(-1,x[j,ell])
                for k in range(n):
                    if k!=ell:
                        lin.addTerms(1,y[i,k,j,ell])
                model.addConstr(lhs=lin,sense=GRB.EQUAL,rhs=0,
                                name="const2"+str(i)+str(j)+str(ell))

for i in range(n):
    for j in range(n):
        if i!=j:
            for k in range(n):
                for ell in range(n):
                    if k!=ell:
                        lin=LinExpr()
                        lin.addTerms(1,y[i,k,j,ell])
                        lin.addTerms(-1,y[j,ell,i,k])
                        model.addConstr(lhs=lin,sense=GRB.EQUAL,rhs=0,
                                        name="const3"+str(i)+str(j)+str(k)+str(ell))

model.update()
model.write("qap2.lp")
model.optimize()
print "Opt. value=",model.ObjVal
for v in model.getVars():
    if v.X>0.001:
        print v.VarName,v.X

```

4.5 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ウムを見ること、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。都市間の移動距離を測ってみたところ図 4.6 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？

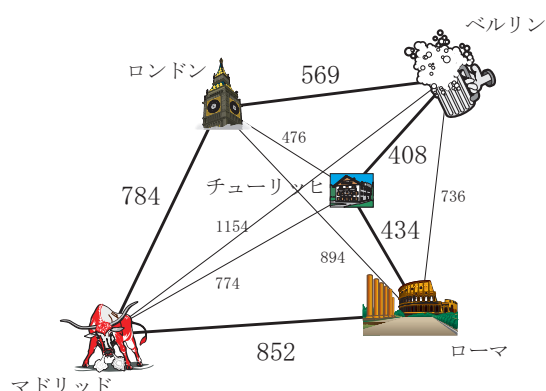


図 4.6: ヨーロッパ旅行のグラフ表現（枝上の数値は距離で単位はマイル）と最適巡回路（太線）。

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 $D : E \rightarrow \mathbb{R}$ 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呼ぶ）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。

```
import scop
n=5
p=scop.Problem()
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)
p.addVariables(cities,range(n))
con1=scop.Alldiff("AD",cities,"inf")
p.addConstraint(con1)
obj=scop.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.addTerm(d[i][j], cities[i], k, cities[j], ell)
p.addConstraint(obj)
p.solve(1)

```

また、数理計画ソルバー Gurobi で解くためには、Miller-Tucker-Zemlin 制約（たとえば久保幹雄著「サプライ・チェーン最適化ハンドブック」（朝倉書店）参照）を用いると容易である。

```

from gurobi import *
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)
model=Model("tsp")
x={}
y={}
for i in range(n):
    for j in range(n):
        if i!=j:
            x[i,j]=model.addVar(0,1,d[i][j],GRB.BINARY,"x"+str(i)+str(j))
for i in range(n):
    y[i]=model.addVar(0,n-1,0,GRB.CONTINUOUS,"y"+str(i))
model.update()
for i in range(n):
    con1=LinExpr()
    for j in range(n):
        if i!=j:
            con1.addTerms(1,x[i,j])
    model.addConstr(lhs=con1,sense=GRB.EQUAL,rhs=1,name="degree1_"+str(i))
for j in range(n):
    con2=LinExpr()
    for i in range(n):
        if i!=j:
            con2.addTerms(1,x[i,j])
    model.addConstr(lhs=con2,sense=GRB.EQUAL,rhs=1,name="degree2_"+str(j))

for i in range(n):
    for j in range(n):
        if i!=j and j!=0:
            con3=LinExpr()
            con3.addTerms(1,y[i])
            con3.addTerms(-1,y[j])
            con3.addTerms(n-1,x[i,j])
            model.addConstr(lhs=con3,sense=GRB.LESS_EQUAL,rhs=n-2,name="edge"+str(i)+str(j))

model.update()
model.write("tsp.lp")
model.optimize()
print "Opt. value=",model.ObjVal
for v in model.getVars():
    if v.X>0.001:
        print v.VarName,v.X

```

4.6 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 4.7 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，10000cm³ (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？

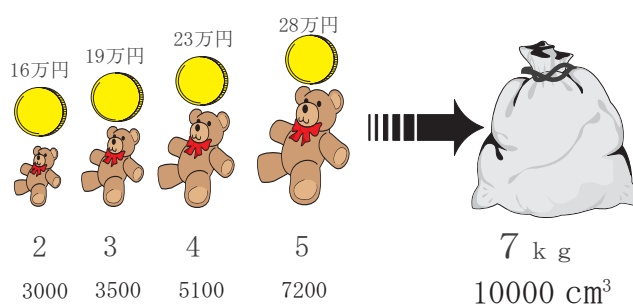


図 4.7: クマさん人形のラインアップと愛用のナップサック．

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 $v_j (\geq 0)$ ，アイテム $j \in N$ の制約 $i \in M$ に対する重み $a_{ij} (\geq 0)$ ，および制約 $i \in M$ に対する制約の上限値 $b_i (\geq 0)$ 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，それ以外るとき 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．

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

問題例によっては，市販の数値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，以下ようになる。

```
import scop
p=scop.Problem()
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"+str(j) for j in range(n)]
p.addVariables(items,[0,1])
for i in range(m):
    con1=scop.Linear("mkp"+str(i),"inf")
    for j in range(n):
        con1.addTerm(a[i][j],items[j],1)
    con1.setRhs(b[i])
    con1.setDirection("<=")
    p.addConstraint(con1)

con2=scop.Linear("obj",1)
for j in range(n):
    con2.addTerm(v[j],items[j],1)
con2.setRhs(sum(v))
con2.setDirection(">=")
p.addConstraint(con2)
p.solve(1)
```

数理計画ソルバー Gurobi によるプログラムを以下に示す。

```
from gurobipy import *
v=[16,19,23,28]
a=[[2,3,4,5],[3000,3500,5100,7200]]
b=[7,10000]
n=len(v)
m=len(b)
model=Model("kp")
var={}
for j in range(n):
    var[j]=model.addVar(0,1,v[j],GRB.BINARY,"x"+str(j))

model.update()

for i in range(m):
    lin=LinExpr()
    for j in range(n):
        lin.addTerms(a[i][j],var[j])
    model.addConstr(lhs=lin,sense=GRB.LESS_EQUAL,rhs=b[i],name="constr"+str(i))

model.ModelSense=-1
model.optimize()
print "Opt.value=",model.ObjVal
for v in model.getVars():
    if v.X>0.001:
        print v.VarName,v.X
```

付録A scop.py

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

```
# file name scop.py
# ver. 1.0 Copyright Log Opt Co., Ltd.
# place scop.exe in the same folder

class Problem(object):
    def __init__(self):
        self.constraints=[] #set of constraints is maintained by a list
        self.variables={} #set of variables is maintained by a dictionary
        self.target=0 #target value of the total penalty

    def __str__(self):
        ret="number of variables = "+str(len(self.variables))+" \n"
        ret+="number of constraints= "+str(len(self.constraints))+"\n"
        varlist=[(i,self.variables[i]) for i in self.variables]
        varlist.sort()
        for (i,j) in varlist:
            ret+="variable "+str(i)+": "+str(j)+"\n"

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

    def setTarget(self, target=0):
        """ set a target value of the solver
        target is a non-negative integer value
        """
        self.target=target

    def addVariable(self, var, dom):
        """ add a variable (var) and its domain (dom)
        var is any hashable object and dom is a list
        """
        if var in self.variables:
            print ("duplicate variable name error of adding a new variable "+ str(var))
            return False
        self.variables[var]=dom

    def addVariables(self, varlist, dom):
        """ add variables (varlist) and their (same) domain (dom)
        varlist is a list of any hashable objects and dom is a list
        """
        for var in varlist:
            self.addVariable(var,dom)

    def addConstraint(self, con):
        """ add a constraint (con)
        con is an object of a constraint class
        """
        if not isinstance(con, Constraint):
            print ("class error of adding a new constraint "+ str(con))
            return False

        #check the feasibility of the constraint added in the class con
        if con.feasible(self.variables):
            self.constraints.append(con)
        else:
```

```

        print "constraint addition error"
        return False

def solve(self , time=600, seed=1):
    """
        solve the instance using scop.exe in the same folder
        input: time= time limit (default 600 seconds)
               seed= random number seed (default 1)
    """
    #prepare the string in a scop input format
    f = ""
    #variable declaration
    for var in self.variables:
        vallist = ""
        for value in self.variables[var]:
            vallist += str(value) + " "
        f += "variable " + var + " in {" + vallist[:-1] + "} \n"
    f += "target=" + str(self.target) + " \n" #target value declaration
    #constraint declaration
    for con in self.constraints:
        if isinstance(con, Alldiff):
            f += con.name + ": weight= " + con.weight + " type=alldiff "
            for var in con.variables:
                f += var + " "
            f += "; \n"
        elif isinstance(con, Linear):
            f += con.name + ": weight= " + con.weight + " type=linear "
            for (coeff, var, value) in con.terms:
                f += str(coeff) + " (" + var + " , " + str(value) + " ) "
            f += con.direction + str(con.rhs) + " \n"
        elif isinstance(con, Quadratic):
            f += con.name + ": weight= " + con.weight + " type=quadratic "
            for (coeff, var1, value1, var2, value2) in con.terms:
                f += str(coeff) + " (" + var1 + " , " + str(value1) + " ) (" + var2 + " , " + str(value2) + " ) "
            f += con.direction + str(con.rhs) + " \n"

    import subprocess
    print "input stream \n" + f
    cmd = "scop -time " + str(time) + " -seed " + str(seed) #solver call
    pipe = subprocess.Popen(cmd, stdout=subprocess.PIPE, stdin=subprocess.PIPE)
    out, err = pipe.communicate(f) #get the result
    print out, '\n'
    #extract the solution and the violated constraints
    s0 = "[best solution]"
    s1 = "penalty"
    s2 = "[Violated constraints]"
    i0 = out.find(s0)
    i1 = out.find(s1, i0)
    i2 = out.find(s2, i1)
    p = i0 + len(s0)
    sollist = [j for j in out[p:i1].split()]
    sol = {}
    for i in range(0, len(sollist), 2):
        varname = sollist[i] + ":" + sollist[i+1]
        value = sollist[i+1]
        sol[varname] = value
    p = i2 + len(s2)
    violist = [j for j in out[p:].split()]
    violated = {}
    for i in range(0, len(violist), 2):
        conname = violist[i] + ":" + violist[i+1]
        value = violist[i+1]
        violated[conname] = value
    #return dictionaries containing the solution and the violated constraints
    return sol, violated

class Constraint(object):
    """
        Constraint base class
    """

```

```

pass

class Alldiff(Constraint):
    """
    Alldiff constraint class
    """

    def __init__(self, name=None, varlist=None, weight=1):
        if name==None or name=="":
            print "please specify the name of the constraint"
            return False
        else:
            self.name=name
            self.weight=str(weight)

        if varlist==None:
            self.variables=set([])
        else:
            self.variables=set(varlist)

    def __str__(self):
        ret="Alldiff: "+self.name+": weight="+str(self.weight)+"\n"
        for i in self.variables:
            ret+=str(i)+" "
        ret+="\n"
        return ret

    def setWeight(self, weight):
        self.weight=str(weight)

    def addVariable(self, var):
        """ add a variable (var)
           var is variable
        """
        if var in self.variables:
            print ("duplicate variable name error when adding a variable "+str(var))
            return False
        self.variables.add(var)

    def addVariables(self, varlist):
        """ add variables (varlist)
           varlist is a list of variables
        """
        for var in varlist:
            self.addvar(var)

    def feasible(self, allvars):
        for var in self.variables:
            if var not in allvars:
                print("no variable in the problem instance named "+str(var))
                return False
        return True

class Linear(Constraint):
    """
    Linear constraint class
    """

    def __init__(self, name=None, weight=1):
        if name==None or name=="":
            print "please specify the name of the constraint"
            return False
        self.name=name
        self.weight=str(weight)
        self.terms=[]
        self.rhs=0
        self.direction="<="

```

```

def __str__(self):
    ret="Linear: "+self.name+": weight="+str(self.weight)+"\n"
    for (i,j,k) in self.terms:
        ret+=str(i)+"*"+str(j)+"x["+str(k)+"] "+str(k)+" ] "
    ret+=self.direction+str(self.rhs)+"\n"
    return ret

def setWeight(self, weight):
    self.weight=str(weight)

def addTerm(self, coeff, var, value):
    """add a term to the left-hand-side of the constraint
       coeff (var, value)
    """
    self.terms.append( (coeff, var, value))

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):
    for (coeff, var, value) in self.terms:
        if var not in allvars:
            print("no variable in the problem instance named "+str(var))
            return False
        if value not in allvars[var]:
            print("no value "+str(value)+" in the variable named "+str(var))
            return False
    return True

class Quadratic(Constraint):
    """
    Quadratic constraint class
    """

    def __init__(self, name=None, weight=1):
        if name==None or name=="":
            print "please specify the name of the constraint"
            return False
        self.name=name
        self.weight=str(weight)
        self.terms=[]
        self.rhs=0
        self.direction="<="

    def __str__(self):
        ret="Quadratic: "+self.name+": weight="+str(self.weight)+"\n"
        for (i,j,k,l,m) in self.terms:
            ret+=str(i)+"*"+str(j)+"x["+str(k)+"]*x["+str(l)+"]"+str(m)+" ] "
        ret+=self.direction+str(self.rhs)+"\n"
        return ret

    def setWeight(self, weight):
        self.weight=str(weight)

    def addTerm(self, coeff, var1, value1, var2, value2):
        """add a term to the left-hand-side of the constraint
           coeff (var, value)
        """
        self.terms.append( (coeff, var1, value1, var2, value2))

    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):
    for (coeff, var1, value1, var2, value2) in self.terms:
        if var1 not in allvars:
            print("no variable in the problem instance named "+str(var1))
            return False
        if var2 not in allvars:
            print("no variable in the problem instance named "+str(var2))
            return False
        if value1 not in allvars[var1]:
            print("no value "+str(value1)+" in the variable named "+str(var1))
            return False
        if value2 not in allvars[var2]:
            print("no value "+str(value2)+" in the variable named "+str(var2))
            return False
    return True
```