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

導入ガイド

LOG OPT Co., Ltd.

ご注意

- このソフトウエアおよびマニュアルの著作権は LOGOPT 社にあります.
- このソフトウエ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.
- このソフトウエ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 入門 **4**

2.1　仕事の割当 1 4
2.2　仕事の割当 2 7
2.3　仕事の割当 3 8
2.4　SCOP の使用法 10

第 3 章　組合せ最適化問題への適用例 **14**

3.1　多制約ナップサック問題 14
3.2　グラフ分割問題 17
3.3　最大安定集合問題 19
3.4　グラフ彩色問題 21
3.5　2 次割当問題 23
3.6　巡回セールスマン問題 26
3.7　ビンパッキング問題 28
3.8　n クイーン問題 31

第 4 章　応用例 **34**

4.1　時間割作成 34
4.2　スタッフスケジューリング 35
4.3　枝重み容量制約付きグラフ彩色問題 38
4.4　生産ラインへの投入順決定問題 39

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 の特徴は，以下の通り．

- 世界的に有名な最適化理論の研究者である茨木先生（京都大学名誉教授）と野々部先生（法政大学）の開発したメタヒューリスティクスを基礎としているため，世界最速クラスの探索能力をもち，超大規模な問題でも，限られた計算時間内に，極めて効率的に求解することができる．
- スタッフスケジューリングや時間割作成など，数理計画ソルバーでは解きにくいタイプの応用に対しても，自然なモデル化が可能．
- 簡易モデリング言語によるデータ入力と，ライブラリ呼び出しによる利用が可能．ライブラリ呼び出しは，最適化を必要とする他のシステムに組み込んで利用するときに便利である．

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．

以下では，モデルについて例題を交えて丁寧に解説すると同時に，実際問題をモデルに帰着させるための様々なテクニックを紹介する．

第2章　SCOP入門

ここでは，SCOP で対象とする重み付き制約充足問題について解説するとともに，簡単な例題を通して SCOP の使用法を解説する．

まず，基礎となる制約充足問題について述べる．制約充足問題（constraint satisfaction problem）は，以下の 3 つの要素から構成される．

変数（variable）： 分からないもの，最適化によって決めるもの．制約充足問題では，変数は，与えられた集合（以下で述べる「領域」）から 1 つの要素を選択することによって決められる．

領域（domain）： 変数ごとに決められた変数の取り得る値の集合．

制約（constraint）： 幾つかの変数が同時にとることのできる値に制限を付加するための条件．

制約充足問題とは，制約をできるだけ満たすように，変数に領域の中の 1 つの値を割り当てることを目的とした問題である．SCOP では，制約に重みを付加した問題（重み付き制約充足問題：weighted constraint satisfaction problem）を対象とする．すべての変数に領域内の値を割り当てたものを解（solution）とよぶ．SCOP では，単に制約を満たす解を求めるだけでなく，制約からの逸脱量の重み付き和（ペナルティ）を最小にする解を探索する．これによって，通常の制約計画では取り扱いがしにくかった目的関数（objective function）の概念をモデルに組み込むことが可能になる．

以下では，簡単な例を用いて，SCOP の基本的なモデルと使用法を解説する．

2.1　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，変数の定義を行う．SCOP では，変数は予約語 `variable` と `in` を用いて，

```
variable  変数名 in { 領域 }
```

と記述する．ここで領域（domain）とは，変数のとりえる値の集合であり，

```
値 1，値 2，値 3，...
```

とカンマ（,）区切りで記述する．

3 人の作業員は A,B,C，仕事の番号は $0,1,2$ であったので，3 人の作業員に割り当てられる仕事を，変数 A,B,C で表し，各々の変数の領域をすべて $\{0,1,2\}$ とする．これを SCOP で表現するには，以下のように記述すれば良い．

```
variable  A in {0, 1, 2}
variable  B in {0, 1, 2}
variable  C in {0, 1, 2}
```

次に，制約の記述を行う．SCOP では，制約は予約語 weight と type を用いて，

```
制約名:  weight= 制約の重み  type=制約の種類   [制約本体]
```

と記述する．ここで「制約名」とは，ユーザーが制約に適当につけた名前である．制約名の後ろには，必ずコロン（:）をつける必要がある．weight= の後ろに記述した「制約の重み」とは，制約の重要さを表す数値であり，必ず正の整数，もしくは無限大を表す予約語 inf を入力する．inf を入れた場合には，制約は「ハードな制約」もしくは「絶対制約」とよばれ，その逸脱量を最小にするように解の探索が行われる．それ以外の場合には，制約は「ソフトな制約」もしくは「考慮制約」とよばれ，制約を逸脱した量に重みを乗じたものの和を最小にするように探索が行われる．

type= の後ろに記述した「制約の種類」とは，SCOP で扱うことのできる制約のタイプであり，linear，alldiff，quadratic の 3 種類がある．linear は線形の制約，alldiff は変数の値の番号（インデックス）がすべて異なることを表す制約, quadratic は変数の 2 次式の制約である．ここでは，linear，alldiff の 2 種類を用いて記述する．quadratic については，2.3 節で例を用いて説明する（3.2 節，3.5 節も参照されたい）.「制約本体」は，制約の種類によって異なるので，以下で例とともに解説する．

SCOP においては，目的関数も制約の 1 つと考える．ここでは，目的関数は割り当てに伴う費用の和であるので，制約の重みが 1 の線形制約として記述する．

線形制約の本体は,

```
係数 1 ( 変数名 1 , 値 1 ) , 係数 2 ( 変数名 2 , 値 2 ), ....   <= (>=, =) 右辺定数
```

と記述する．これによって，変数名 1 の変数が値 1 をとったときに，係数 1 が制約の左辺に加えられる．変数名 2 以降についても同様である．ここで，係数は線形制約の各項の係数であり，整数値（負でも良い）を入力する．右辺定数も整数値（負でも良い）として入力し，制約は，以下（<=），以上（>=），等しい（=）のいずれかの型をとる．

obj と名付けた目的関数を表す制約は，以下のように書ける．

```
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)
 <=0
```

費用の和を最小化したいので，線形制約は，費用の和が 0 以下であると記述する．すると，SCOP では，この制約の逸脱量に重み 1 を乗じたものをペナルティとして計算し，その和を最小化するように探索を行う．

次に，各作業員は，1 つの仕事に割り当てられ，それらは重複してはならないことを制約として記述する．

これは，変数の値の番号（インデックス）がすべて異なることを表す制約 `alldiff` を用いて表すことができる．

`alldiff` 制約の本体は，

```
変数名１ ，変数名２ ， ... ，最後の変数名;
```

と，変数名をカンマ（,）区切りで並べることによって記述される．ただし，式の最後には，式が終わったことを表すために，必ずセミコロン（;）を入れる．

例題の場合には，変数 A, B, C がすべて異なる必要があるので，以下のように，ハードな制約として記述する．

```
constraint: weight=inf  type=alldiff A B C;
```

以上をまとめると，割当問題の SCOP 入力は，以下のようになる．

```
variable  A in {0, 1, 2}
variable  B in {0, 1, 2}
variable  C in {0, 1, 2}
constraint: weight=inf  type=alldiff A B C;
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)
 <=0
```

このデータを SCOP の入力データ（ファイル名は `ex1-scop.dat`）として入れると，以下のような結果が得られる．

```
scop  < ex1-scop.dat
# reading data ... done: 0.00(s)

penalty = 1/32 (hard/soft), time = 0.00(s), iteration = 0
# improving the initial solution greedily
penalty = 0/47 (hard/soft), time = 0.00(s), iteration = 0
# start tabu search
penalty = 0/37 (hard/soft), time = 0.01(s), iteration = 2
# penalty = 0/37 (hard/soft)
# cpu time = 0.01/0.05(s)
# iteration = 2/100

[best solution]
A: 0
B: 2
C: 1

penalty: 0/37 (hard/soft)

[Violated constraints]
obj: 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．

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

ハードな制約の逸脱量は 0 であるので，各仕事は重複なく 1 人の作業員に割り当てられている．また，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 37，制約 `obj` 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．

2.2　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．

たとえば，仕事 0 に割り当てられた人数の合計が 1 以上であることを表す制約は，以下のように書ける．

```
constraint0: weight=inf  type=linear 1(A,0) 1(B,0) 1(C,0) 1(D,0) 1(E,0) >=1
```

SCOP のモデル全体は，以下のようになる．

```
variable  A in {0, 1, 2}
variable  B in {0, 1, 2}
variable  C in {0, 1, 2}
variable  D in {0, 1, 2}
variable  E in {0, 1, 2}

constraint0: weight=inf  type=linear 1(A,0) 1(B,0) 1(C,0) 1(D,0) 1(E,0) >=1
constraint1: weight=inf  type=linear 1(A,1) 1(B,1) 1(C,1) 1(D,1) 1(E,1) >=2
constraint2: 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
```

このデータを SCOP の入力データ（ファイル名は ex2-scop.dat）として入れると，以下のような結果が得られる．

```
scop  <ex2-scop.dat
# reading data ... done: 0.00(s)

penalty = 1/52 (hard/soft), time = 0.01(s), iteration = 0
# improving the initial solution greedily
penalty = 0/57 (hard/soft), time = 0.01(s), iteration = 0
# start tabu search
penalty = 0/50 (hard/soft), time = 0.03(s), iteration = 2
# penalty = 0/50 (hard/soft)
# cpu time = 0.03/0.05(s)
# iteration = 2/100

[best solution]
A: 1
B: 2
C: 1
D: 2
E: 0

penalty: 0/50 (hard/soft)

[Violated constraints]
obj: 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．

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

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．

2.3 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．

この制約を記述するためには，変数の 2 次式の制約（制約の種類が qudratic）を用いると便利である．

2 次の制約の本体は，

係数１（ 変数名１ ，値１ ）（ 変数名２ ，値２ ）

係数２（ 変数名３ ，値３ ）（ 変数名４ ，値４ ）

...

<= (>=, =) 右辺定数

と記述する．線形制約の場合と異なり，変数名と値の対を 2 つ続けて入力する．これによって，変数名 1 の変数が値 1 をとり，かつ変数名 2 の変数が値 2 をとったときに，係数 1 が制約の左辺に加えられることが定義される．線形制約の場合と同様に，係数は制約の各項の係数であり，整数値（負でも良い）を入力する．右辺定数も整数値（負でも良い）として入力し，制約は，以下（<=），以上（>=），等しい（=）のいずれかの型をとる．

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（重みは 100）は，以下のように記述される．

```
quad: weight=100 type=quadratic
  1 (A,0) (C,0)   1 (A,1) (C,1)  1 (A,2) (C,2)  <=0
```

このデータを前節のデータに追加したもの（ファイル名は ex3-scop.dat）を SCOP に入れると，以下のような結果が得られる．

```
scop  <ex3-scop.dat
# reading data ... done: 0.00(s)

penalty = 1/52 (hard/soft), time = 0.00(s), iteration = 0
# improving the initial solution greedily
penalty = 0/157 (hard/soft), time = 0.00(s), iteration = 0
# start tabu search
penalty = 0/60 (hard/soft), time = 0.01(s), iteration = 2
penalty = 0/52 (hard/soft), time = 0.01(s), iteration = 8
# penalty = 0/52 (hard/soft)
# cpu time = 0.01/0.03(s)
# iteration = 8/100

[best solution]
A: 0
B: 2
C: 1
D: 2
E: 1

penalty: 0/52 (hard/soft)

[Violated constraints]
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．

2.4 SCOP の使用法

SCOP は，問題例のデータを標準入力から読み込むことによって実行される．データを記述した入力ファイルを *inputfile* として，

```
scop < inputfile
```

で実行できる．

SCOP の実行時には，様々なオプションを指定することができる．コンソール上で

```
scop -help
```

とすれば

```
Usage: scop [-options...]
  -noadjust     deactivate weight adjustment mechanism
  -display #    set log display level
  -iteration #  set iteration limit
  -interval #   set log display interval
  -seed #       set random seed
  -target #     set target
  -time #.#     set cpu time limit in second
```

と出力される．まず, これらのオプションについて説明する．

`-noadjust`

タブー探索中のペナルティ重み自動調整機能を無効にする (これにより, 探索中ペナルティ重みは固定される).

通常指定する必要はないが, 制約間のペナルティ重み差が小さい場合, しばしば有効である．

-display

ログの出力レベルを設定する. 規定値（デフォルト値）は 0 である.

0: 最良値更新 + 解移動

1: 最良値更新 + 解移動 (詳細)

2: 最良値更新 + 解移動 + ペナルティ重み調整

3: 最良値更新 + 解移動 (詳細) + ペナルティ重み調整

なお, 解移動情報は -interval オプションで設定された反復ごとにのみ出力される.

-interval

解移動情報を何反復ごとに出力するか設定する．規定値（デフォルト値）は 0 であり，これは解移動情報を表示させないことを意味する．

-iteration

最大反復回数を設定する．規定値（デフォルト値）は無限大であり，実質的な最大反復回数を設定しないことを意味する．

-seed

乱数系列の種を設定する．規定値（デフォルト値）は 1 である．

-target

目標とするペナルティ値を設定する．設定されたペナルティ以下の解が求まった時点でプログラムは終了する．規定値（デフォルト値）は 0 である．

-time

最大計算時間 (秒) を設定する．規定値（デフォルト値）は 600 である．

以下, 入力フォーマットについて述べる．入力フォーマットは,

1. 変数, 値の定義
2. 目標値の設定
3. 制約の記述

の 3 部で構成される．1 と 2 の記述順序は任意であるが, 3 は 1, 2 の後に記述されなくてはならない．なお, # 以降その行末まではコメントとして無視される．また, 改行はコメントの終了を表す以外に意味を持たない.

1. 変数・値の定義

```
variable 変数名 in { 値, 値, ... }
```

変数名および値としては, 英文字 (a–z, A–Z), 数字 (0–9), 角括弧 ([ ]), アンダーバー (_), および @ からなる文字列が使用できる.

2. 目標値の設定

```
target = 目標値
```

ペナルティが目標値以下になった時点でプログラムは終了する. 目標値の設定は省略可能であり, その場合 `target = 0` と見なされる. なお, プログラム実行時に `-target` オプションを使用すると, その値が優先される.

3. 制約の記述

```
制約名: weight = ペナルティ重み type = 制約タイプ　制約記述
```

ペナルティ重みは非負整数もしくは `inf` でなくてはならない (後者の場合, ハードな制約と見なされる). 制約記述は各制約タイプによって異なり, 現在のバージョンでは `linear`, `quadratic`, `alldiff` の 3 種類の制約タイプが使用可能である.

- `linear`

  値変数 x_{ij} に関する線形等式・不等式制約. 制約記述部分は

  ```
  係数（変数名, 値）係数（変数名, 値） ... <= 右辺値
  ```

  で与えられる (係数, 右辺値は整数). `<=` の代わりに `>=` あるいは `=` を使用してもよい.

- `quadratic`

  値変数 x_{ij} に関する 2 次等式・不等式制約. 制約記述部分は

  ```
  係数（変数名, 値）係数（変数名 1, 値 1）（変数名 2, 値 2）
  係数（変数名, 値）係数（変数名 1, 値 1）（変数名 2, 値 2） ... <= 右辺値
  ```

  で与えられる (係数, 右辺値は整数). `<=` の代わりに `>=` あるいは `=` を使用してもよい.

- `alldiff`

  指定された変数集合 V に対し, V に含まれる変数すべてが異なる値をとらなくてはならない. 値が同一であるかどうかは, 値名ではなくインデックス番号により決定される. インデックス番号とは, 「1. 変数・値の定義」における値の記述順序によって定まる非負整数で, 例えば, 変数 `var1` および `var2` がそれぞれ

  ```
  variable var1 in { val1, val2 }
  variable var2 in { val2, val1 }
  ```

  で定義されたとき, 変数 `var1` の値 `val1`, `val2` のインデックス番号はそれぞれ 0 と 1, 変数 `var2` の値 `val1`, `val2` のインデックス番号は 1, 0 となる. 制約記述部分は

変数名 変数名 ... 変数名 ;

で与えられる. 最後はセミコロン ; で終了しなくてはならない.

第3章　組合せ最適化問題への適用例

本章では，SCOP を様々な組合せ最適化問題に適用する．ここで用いる例は，久保幹雄，ジョア・ペドロ・ペドロソ著「メタヒューリスティクスの数理」（共立出版），久保幹雄著「ロジスティクスの数理」（共立出版），久保幹雄，松井知己著「組合せ最適化（短編集）」（朝倉書店）からとったものである．問題の詳細については，原著を参照されたい．

3.1　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 3.1 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？

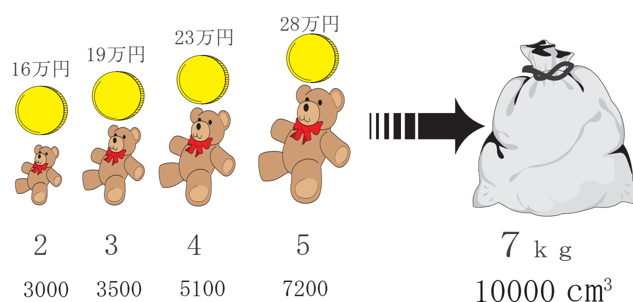

図 3.1: クマさん人形のラインアップと愛用のナップサック．

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.

多制約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.

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

多くの市販の汎用の（混合）整数計画ソルバーは，線形計画問題に対する単体法や内点法と，最も標準的に用いられる厳密解法である分枝限定法から構成されているが，中規模の問題例でも求解が困難になることがある．動的計画は，与えられた数値データが小さな整数の場合には高速であるが，制約の数が増えたり，数値データがある程度大きな整数や実数であったりすると，求解が困難になる.

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，このような小規模問題例は，極めて短時間に良好な近似解を得ることができる.

上で示した例題を，SCOP の入力データ形式で記述すると，以下のようになる.

```
variable  x[0] in {0,1}
variable  x[1] in {0,1}
variable  x[2] in {0,1}
variable  x[3] in {0,1}
constraint0 : weight=inf  type=linear 2(x[0],1) 3(x[1],1) 4(x[2],1) 5(x[3],1) <=7
constraint1 : weight=inf  type=linear 3000(x[0],1) 3500(x[1],1) 5100(x[2],1) 7200(x[3],1) <=10000
object: weight=1  type=linear 16(x[0],1)19(x[1],1)23(x[2],1)28(x[3],1)>=86
```

最後の制約は，目的関数値の最大化を表す．ここでは，最大値は分かっていないので，価値の合計以上を表す制約を設け，その逸脱量（ナップサックに入れられないアイテムの価値の合計）を最小化する.

与えられた多制約ナップサック問題の問題例を，SCOP の入力データに変換する Python プログラムは，以下のようになる．入力データは，制約行列を表すリストのリスト `a`，アイテムの価値を表すリスト `v`，制約の上限値を表すリスト `b` で与える.

```
1 v=[16,19,23,28]
2 a=[[2,3,4,5],[3000,3500,5100,7200]]
3 b=[7,10000]
4 n=len(v)
5 m=len(b)
6 f = open("mkp-scop.dat", 'w')
7 for i in range(n):
```

```
8        f.write("variable   x["+str(i)+"]  in  {0,1}"+"\n")
9
10   for i in range(m):
11       f.write( "constraint"+str(i)+" : weight=inf  type=linear ")
12       for j in range(n):
13           f.write( str(a[i][j])+"(x["+str(j)+"],1) " )
14       f.write( "<="+str(b[i])+"\n")
15
16   f.write( "object: weight=1  type=linear ")
17   for i in range(n):
18       f.write( str(v[i])+"(x["+str(i)+"],1)")
19
20   f.write( ">="+str(sum(v)) +"\n")
21   f.close()
```

上で作成したデータをSCOPの入力として実行した結果は，以下のようになる．解は，アイテム $1,2$ をナップサックに入れるものであり，ソフトな制約のペナルティは 44 となる．これは，ナップサックに入れていないアイテム $0,4$ の価値の合計を表す.

```
scop <mkp-scop.dat
# reading data ... done: 0.00(s)

penalty = 2302/35 (hard/soft), time = 0.00(s), iteration = 0
# improving the initial solution greedily
penalty = 0/47 (hard/soft), time = 0.00(s), iteration = 0
# start tabu search
penalty = 0/44 (hard/soft), time = 0.00(s), iteration = 2


# penalty = 0/44 (hard/soft)
# cpu time = 0.00/0.00(s)
# iteration = 2/2

[best solution]
x[0]: 0
x[1]: 1
x[2]: 1
x[3]: 0

penalty: 0/44 (hard/soft)

[Violated constraints]
object: 44
```

この問題例は小規模なので，数回の反復で最適解が求まるが，SCOPでは既定の実行時間が600秒なので，計算結果が得られるまで時間を要する．その際には，ペナルティ逸脱の目標値（target） 44 を，SCOPのオプションとして入力することによって，一瞬で解を得ることがでいる．具体的には，以下のように実行を行う（目標値 44 と< の間には，必ず空白を入れる）.

```
scop -target 44 <mkp-scop.dat
```

3.2 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 3.2）．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？

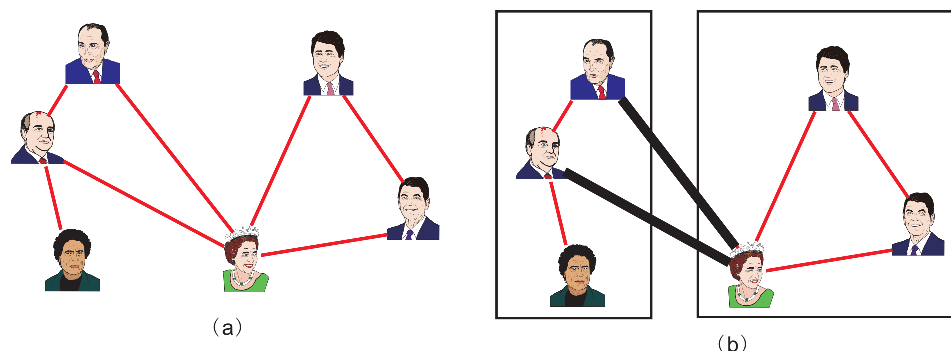

図 3.2: グラフ分割問題の例．(a) 線の引いてある人同士は仲良しであることを表すグラフ．(b) 仲良しが別のチームになることを最小にする等分割．違うチームに所属する仲良しのペアは太線で表されており，2 本である．よって，この等分割の目的関数値は 2 となる．

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．

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

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．一方，SCOP は 2 次の制約式を直接記述することができるので，自然なモデル化が可能である．

また，グラフ分割問題の定式化を市販の数理計画ソルバーにかけても，$n = 100$ 程度の問題例の厳密解を短時間で得ることはほぼ絶望的である．一方，SCOP を使えば，$n = 1000$ 程度の問題例でも短時間で良好な近似解を得ることができる．

例題の 6 点のグラフの 2 分割を求める SCOP のデータは，以下のように書ける．

```
variable  x[0] in {0,1}
variable  x[1] in {0,1}
variable  x[2] in {0,1}
variable  x[3] in {0,1}
variable  x[4] in {0,1}
variable  x[5] in {0,1}
constraint1 : weight=inf  type=linear 1(x[0],1) 1(x[1],1) 1(x[2],1) 1(x[3],1) 1(x[4],1) 1(x[5],1) =3
object: weight=1  type=quadratic  1 (x[0],1)(x[1],0) 1 (x[0],0)(x[1],1) 1 (x[0],1)(x[4],0) 1 (x[0],0)(x[4],1) 1
 (x[1],1)(x[0],0) 1 (x[1],0)(x[0],1) 1 (x[1],1)(x[2],0) 1 (x[1],0)(x[2],1) 1 (x[1],1)(x[4],0) 1 (x[1],0)(x[4],1
 1 (x[2],1)(x[1],0) 1 (x[2],0)(x[1],1) 1 (x[3],1)(x[4],0) 1 (x[3],0)(x[4],1) 1 (x[3],1)(x[5],0) 1 (x[3],0)(x[5]
 1 (x[4],1)(x[0],0) 1 (x[4],0)(x[0],1) 1 (x[4],1)(x[1],0) 1 (x[4],0)(x[1],1) 1 (x[4],1)(x[3],0) 1 (x[4],0)(x[3]
 1 (x[4],1)(x[5],0) 1 (x[4],0)(x[5],1) 1 (x[5],1)(x[3],0) 1 (x[5],0)(x[3],1) 1 (x[5],1)(x[4],0) 1 (x[5],0)(x[4]
```

上のデータを生成するための Python プログラムは，以下のようになる．

```
1  nodes=range(6)
2  adj=[[1,4],[0,2,4],[1],[4,5],[0,1,3,5],[3,4]]
3  n=len(nodes)
4  f = open("gpp-scop.dat", 'w')
5  for i in range(n):
6      f.write("variable  x["+str(i)+"] in {0,1}"+"\n")
7  f.write( "constraint1 : weight=inf  type=linear ")
8  for j in range(n):
9      f.write( "1(x["+str(j)+"],1) " )
10 f.write( "="+str(n/2)+"\n")
11 f.write( "object: weight=1  type=quadratic ")
12 for i in range(n):
13     for j in adj[i]:
14         f.write( " 1 (x["+str(i)+"],1)" + "(x["+str(j)+"],0)" )
15         f.write( " 1 (x["+str(i)+"],0)" + "(x["+str(j)+"],1)" )
16 f.write( "<="+str(0) +"\n")
17 f.close()
```

1,2 行目では，グラフのデータ構造を組み立てている．グラフは点のリスト `nodes` と，隣接する点のリストのリスト `adj` で与える．

SCOP で実行した結果は，以下のようになる．

```
# reading data ... done: 0.00(s)

penalty = 0/10 (hard/soft), time = 0.00(s), iteration = 0
# improving the initial solution greedily
```

```
penalty = 0/10 (hard/soft), time = 0.00(s), iteration = 0
# start tabu search
penalty = 0/4 (hard/soft), time = 0.00(s), iteration = 2


# penalty = 0/4 (hard/soft)
# cpu time = 0.00/0.00(s)
# iteration = 2/2

[best solution]
x[0]: 0
x[1]: 0
x[2]: 0
x[3]: 1
x[4]: 1
x[5]: 1

penalty: 0/4 (hard/soft)

[Violated constraints]
object: 4
```

解は $0,1,2$ （$3,4,5$）を同じグループにする分割であり，目的関数値は，破ったソフトな制約の逸脱量 4 に相当する．グループを跨いでいる枝の本数は 2 本であるが，SCOP によるモデル化では枝を 2 回カウントしているので，目的関数値はその倍の 4 となる．

3.3 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 3.3 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？

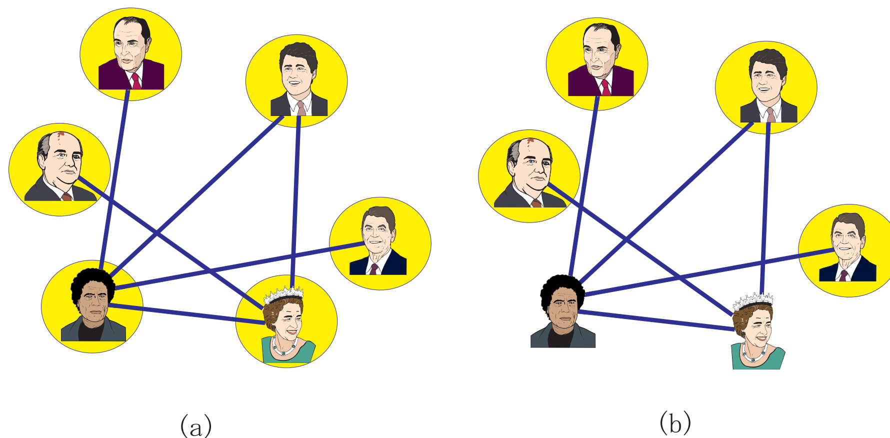

図 3.3: 最大安定集合問題の例．(a) 線の引いてある人同士は仲が悪いことを表すグラフ．(b) 仲が悪い同士を連れて行かないでピクニックに行くときの最大人数．丸で囲んだ人を連れて行くと目的関数値は 4 となり，これが最適解になる．

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．ここでは，最大安定集合問題に対するメタヒューリスティクスを設計する．

最大安定集合問題は，次のように定義される問題である．

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．

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

例題のグラフを最大安定集合を求める SCOP のデータは，以下のように書ける．

```
variable  x[0] in {0, 1}
variable  x[1] in {0, 1}
variable  x[2] in {0, 1}
variable  x[3] in {0, 1}
variable  x[4] in {0, 1}
variable  x[5] in {0, 1}
constraint02 : weight=inf  type=linear 1(x[0],1) +1(x[2],1)<=1
constraint13 : weight=inf  type=linear 1(x[1],1) +1(x[3],1)<=1
constraint23 : weight=inf  type=linear 1(x[2],1) +1(x[3],1)<=1
constraint24 : weight=inf  type=linear 1(x[2],1) +1(x[4],1)<=1
constraint25 : weight=inf  type=linear 1(x[2],1) +1(x[5],1)<=1
constraint35 : weight=inf  type=linear 1(x[3],1) +1(x[5],1)<=1
object: weight=1  type=linear 1(x[0],1)1(x[1],1)1(x[2],1)1(x[3],1)1(x[4],1)1(x[5],1)>=6
```

ここで，最大値は不明であるので，点の数 6 以上の安定集合を求めるというソフトな制約を最後に加え，これを目的関数としている．最小化されるペナルティは，安定集合に入れない点の数となる．

上のデータを生成するための Python プログラムは，以下のようになる．グラフは点のリスト `nodes` と，隣接する点のリストのリスト `adj` で与える．

```
1  nodes=range(6)
2  adj=[[2],[3],[0,3,4,5],[1,2,5],[2],[2,3]]
3  n=len(nodes)
4  f = open("ssp-scop.dat", 'w')
5  for i in range(n):
6      f.write("variable  x["+str(i)+"] in {0,1} \n")
7  for i in range(n):
8      for j in adj[i]:
9          if i<j:
10             f.write( "constraint"+str(i)+str(j)+" : weight=inf  type=linear ")
11             f.write( str(1)+"(x["+str(i)+"],1) +"+ str(1)+"(x["+str(j)+"],1)" "<="+str(1)+"\n")
12 f.write( "object: weight=1  type=linear ")
13 for i in range(n):
14     f.write( str(1)+"(x["+str(i)+"],1)")
```

```
15  f.write( ">="+str(n) +"\n")
16  f.close()
```

上で生成したデータを SCOP の入力として実行すると，以下の結果を得る．これは，点 $0, 1, 4, 5$ の安定集合を表す．逸脱したソフト制約のペナルティは 2 であり，これは安定集合に含まれなかった点の数を表す．

```
scop <ssp-scop.dat
# reading data ... done: 0.00(s)

penalty = 2/3 (hard/soft), time = 0.00(s), iteration = 0
# improving the initial solution greedily
penalty = 0/3 (hard/soft), time = 0.00(s), iteration = 0
# start tabu search
penalty = 0/2 (hard/soft), time = 0.00(s), iteration = 1


# penalty = 0/2 (hard/soft)
# cpu time = 0.00/0.00(s)
# iteration = 1/1

[best solution]
x[0]: 1
x[1]: 1
x[2]: 0
x[3]: 0
x[4]: 1
x[5]: 1

penalty: 0/2 (hard/soft)

[Violated constraints]
object: 2
```

3.4 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 3.4 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？

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．

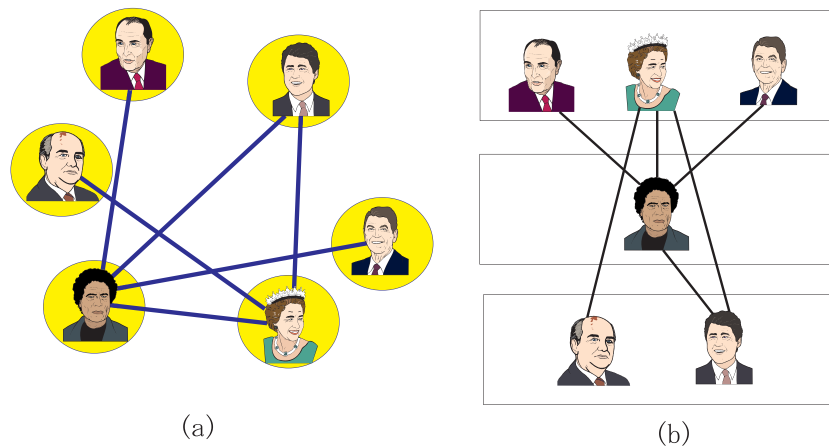

図 3.4: グラフ彩色問題の例．(a) 線の引いてある人同士は仲が悪いことを表すグラフ．(b) 3 つのクラスに分けると仲の悪い友達は同じクラスに入らない．目的関数値（クラス数）は 3 となり，これが最適解になる．

例題のグラフを 3 彩色を求める SCOP のデータは，以下のように書ける．

```
variable  x[0] in {0, 1, 2}
variable  x[1] in {0, 1, 2}
variable  x[2] in {0, 1, 2}
variable  x[3] in {0, 1, 2}
variable  x[4] in {0, 1, 2}
variable  x[5] in {0, 1, 2}
constraint02 : weight=1  type=alldiff x[0] x[2] ;
constraint13 : weight=1  type=alldiff x[1] x[3] ;
constraint23 : weight=1  type=alldiff x[2] x[3] ;
constraint24 : weight=1  type=alldiff x[2] x[4] ;
constraint25 : weight=1  type=alldiff x[2] x[5] ;
constraint35 : weight=1  type=alldiff x[3] x[5] ;
```

上のデータを生成するための Python プログラムは，以下のようになる．グラフは点のリスト `nodes` と，隣接する点のリストのリスト `adj` で与える．

```
1  K=3
2  nodes=range(6)
3  adj=[[2],[3],[0,3,4,5],[1,2,5],[2],[2,3]]
4  n=len(nodes)
5  f = open("gcp-scop.dat", 'w')
6  colorset=str(range(K))[1:-1]
7  for i in range(n):
8      f.write("variable  x["+str(i)+"] in {"+colorset+"}"+"\n")
9
10 for i in range(n):
11     for j in adj[i]:
12         if i<j:
13             f.write( "constraint"+str(i)+str(j)+" : weight=1  type=alldiff ")
14             f.write( "x["+str(i)+"] "+ "x["+str(j)+"] ; \n" )
15 f.close()
```

上のプログラムの 6 行目では，`str(range(K))` で生成された文字列 $[0,1,...,K-1]$ の両端の [] を除いた文字列 $0,1,...,K-1$ を生成している．Python のバージョン 3.0 以降では，この部分は，以下のように変更する必要がある．

```
colorset=str(set(range(K)))
```

上で生成したデータを SCOP の入力として実行すると，以下の結果を得る．これは，点 $0, 1, 5$ を色 0 で，点 2 を色 1 で，点 $3, 4$ を色 2 で塗る 3 彩色を表す.

```
scop <gcp-scop.dat
# reading data ... done: 0.00(s)

penalty = 0/0 (hard/soft), time = 0.00(s), iteration = 0
# improving the initial solution greedily
penalty = 0/0 (hard/soft), time = 0.00(s), iteration = 0
# start tabu search


# penalty = 0/0 (hard/soft)
# cpu time = 0.00/0.00(s)
# iteration = 0/0

[best solution]
x[0]: 0
x[1]: 0
x[2]: 1
x[3]: 2
x[4]: 2
x[5]: 0

penalty: 0/0 (hard/soft)

[Violated constraints]
```

3.5　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 3.5 (a) のようになっている．家の間の移動距離は，図 3.5 (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, 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

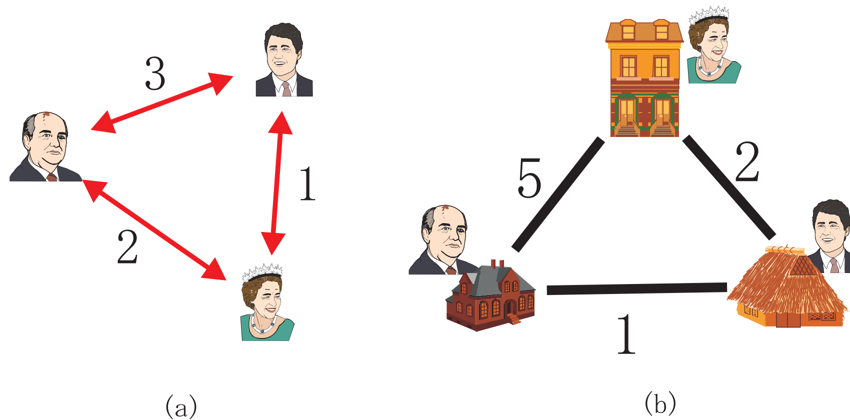

図 3.5: 2 次割当問題の例．(a) 3 人のお友達が打ち合わせをする頻度．枝の上の数字は週に何回打ち合わせをするかを表す．(b) 3 箇所の家とお友達の割り当ての例．枝の上の数字は地点間の距離を表す．お友達同士の行き来する頻度と割り当てられた家の間の距離の和は，$2 \times 5 + 3 \times 1 + 1 \times 2 = 15$ となり，この割り当ての目的関数値は，その 2 倍（お友達同士は往復するから）で $2 \times 15 = 30$ となる．

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．

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．

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

市販の数理計画ソルバーは，通常，上のような（下に凸でない）2 次の関数を含んだ最小化問題には対応していない．しかし，SCOP では，2 次の関数を直接記述できるため，簡単にモデル化を行うことができる．

例題の解を求める SCOP のデータは，以下のように書ける．

```
variable  x[0] in {0, 1, 2}
variable  x[1] in {0, 1, 2}
variable  x[2] in {0, 1, 2}
constraint: weight=inf  type=alldiff x[0] x[1] x[2] ;
obj: weight=1  type=quadratic 10(x[0], 0) (x[1], 1) 15(x[0], 0) (x[1], 2)
 10(x[0], 1) (x[1], 0) 5(x[0], 1) (x[1], 2) 15(x[0], 2) (x[1], 0)
...
4(x[1], 1) (x[2], 0) 2(x[1], 1) (x[2], 2) 6(x[1], 2) (x[2], 0) 2(x[1], 2) (x[2], 1) <=0
```

上のデータを生成するための Python プログラムは，以下のようになる．ただし，行列は 2 つのリストのリスト `f`, `d` で与えられているものとする．なお，以下のプログラムでは，行列は対称であると仮定している．

```
1 n=3
2 d=[[0,2,3],[2,0,1],[3,1,0]]
```

```
3  f=[[0,5,1],[5,0,2],[1,2,0]]
4  fi = open("qap-scop.dat", 'w')
5  for i in range(n):
6      fi.write("variable  x["+str(i)+"] in {"+str(range(n))[1:-1]+"}"+"\n")
7  fi.write( "constraint: weight=inf  type=alldiff ")
8  for i in range(n):
9      fi.write( "x["+str(i)+"] " )
10 fi.write("; \n")
11
12 fi.write( "obj: weight=1  type=quadratic ")
13 for i in range(n-1):
14     for j in range(i+1,n):
15         for k in range(n):
16             for ell in range(n):
17                 if k != ell:
18                     fi.write( str(f[i][j]*d[k][ell]) + "(x[" +str(i) +"], " +str(k) +") "+
                          "(x[" +str(j)+ "], " +str(ell) +") ")
19 fi.write("<=0")
20 fi.close()
```

上のプログラムの 6 行目では，`str(range(n))` で生成された文字列 $[0, 1, ..., n-1]$ の両端の [] を除いた文字列 $0, 1, ..., n-1$ を生成している．Python のバージョン 3.0 以降では，この部分は，以下のように変更する必要がある．

```
fi.write("variable  x["+str(i)+"] in str(set(range(n))) +"\n")
```

上で生成したデータを SCOP の入力として実行すると，以下の結果を得る．これは，順列 $(2, 1, 0)$ の解が目的関数値 12 を持つことを示している．実際にこれは最適解である．

```
scop < qap-scop.dat
# reading data ... done: 0.00(s)

penalty = 1/6 (hard/soft), time = 0.00(s), iteration = 0
# improving the initial solution greedily
penalty = 0/20 (hard/soft), time = 0.00(s), iteration = 0
# start tabu search
penalty = 0/19 (hard/soft), time = 0.00(s), iteration = 2
penalty = 0/15 (hard/soft), time = 0.00(s), iteration = 8
penalty = 0/13 (hard/soft), time = 0.00(s), iteration = 18
penalty = 0/12 (hard/soft), time = 0.01(s), iteration = 86


# penalty = 0/12 (hard/soft)
# cpu time = 0.01/0.01(s)
# iteration = 86/86

[best solution]
x[0]: 2
x[1]: 1
x[2]: 0

penalty: 0/12 (hard/soft)

[Violated constraints]
obj: 12
```

3.6 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 3.6 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?

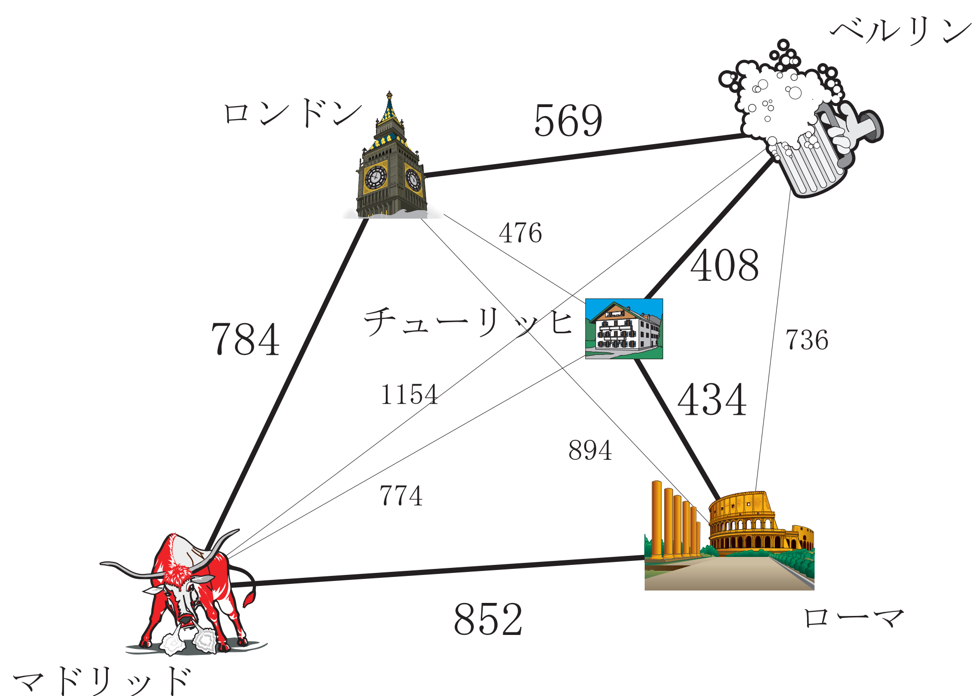

図 3.6: ヨーロッパ旅行のグラフ表現（枝上の数値は距離で単位はマイル）と最適巡回路（太線）．

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 $D : E \to \mathbb{R}$ 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よぶ）を最小にするものを求める問題.

上の問題は，無向グラフ上で定義されていたので，対称巡回セールスマン問題とよばれる．一方，有向グラフ上で定義される（行きと帰りの枝の距離が異なる）問題を，非対称巡回セールスマン問題とよぶ.

対称巡回セールスマン問題は良く研究された問題であるので，専用の解法を使うべきであるが，非対称巡回セールスマン問題については良いアルゴリズムは少ないが，SCOP を用いれば容易に求解することができる．

巡回セールスマン問題は 2 次割当問題の特殊形であるので，同じように順列を解をすることによって SCOP でモデル化できる．距離は，順列で隣り合う順番にあるときにだけ目的関数値に加えられるので，例題は，以下のような SCOP のデータに変換できる．

例題の解を求める SCOP のデータは，以下のように書ける．ここで，点の番号は $0, 1, 2, 3, 4$ とし，順にチューリッヒ，ロンドン，マドリッド，ローマ，ベルリンとしている．

```
variable  x[0] in {0, 1, 2, 3, 4}
variable  x[1] in {0, 1, 2, 3, 4}
variable  x[2] in {0, 1, 2, 3, 4}
variable  x[3] in {0, 1, 2, 3, 4}
variable  x[4] in {0, 1, 2, 3, 4}
constraint: weight=inf  type=alldiff x[0] x[1] x[2] x[3] x[4] ;
obj: weight=1  type=quadratic 476(x[0], 0) (x[1], 1) 476(x[0], 1) (x[1], 2) 476(x[0], 2)
...
 569(x[4], 2) (x[3], 3) 569(x[4], 3) (x[3], 4) 569(x[4], 4) (x[3], 0) <=0
```

上のデータを生成するための Python プログラムは，以下のようになる．ただし，行列は 2 つのリストのリスト `f`, `d` で与えられているものとする．なお，以下のプログラムでは，行列は対称であると仮定している．

```
1  n=5
2  d=[[0,476,774,434,408],[476,0,784,894,569],[774,784,0,852,1154],
3                          [434,894,852,0,569],[408,569,1154,569,0]]
4  f = open("tsp-scop.dat", 'w')
5  for i in range(n):
6      f.write("variable  x["+str(i)+"] in {"+str(range(n))[1:-1]+"}"+"\n")
7  f.write( "constraint: weight=inf  type=alldiff ")
8  for i in range(n):
9      f.write( "x["+str(i)+"] " )
10 f.write("; \n")
11
12 f.write( "obj: weight=1  type=quadratic ")
13 for i in range(n):
14     for j in range(n):
15         if i!=j:
16             for k in range(n):
17                 if k ==n-1:
18                     ell=0
19                 else:
20                     ell=k+1
21                 f.write( str(d[i][j]) + "(x[" +str(i) +"], " +str(k) +") "+ "(x[" +str(j)+ "], "
                       +str(ell) +") ")
22 f.write("<=0")
23 f.close()
```

上のプログラムの 5 行目では，`str(range(n))` で生成された文字列 $[0, 1, ..., n-1]$ の両端の [] を除いた文字列 $0, 1, ..., n-1$ を生成している．Python のバージョン 3.0 以降では，この部分は，以下のように変更する必要がある．

```
    fi.write("variable  x["+str(i)+"] in str(set(range(n))) +"\n")
```

上で生成したデータを SCOP の入力として実行すると，以下の結果を得る．これは，順列 $(1, 3, 4, 0, 2)$ の解が目的関数値 3047 を持つことを示している．これは，点を $3, 0, 4, 2, 3$ の順でまわる解，すなわち

ローマ ⇒ チューリッヒ ⇒ ベルリン ⇒ ロンドン ⇒ マドリッド ⇒ チューリッヒ

の順でまわることを表している．実際にこれは最適解であり，最適値はソフトな制約の逸脱量 3047 と一致する．

```
scop < tsp-scop.dat
# reading data ... done: 0.00(s)

penalty = 2/4311 (hard/soft), time = 0.00(s), iteration = 0
# improving the initial solution greedily
penalty = 0/3867 (hard/soft), time = 0.00(s), iteration = 0
# start tabu search
penalty = 0/3784 (hard/soft), time = 0.00(s), iteration = 2
penalty = 0/3674 (hard/soft), time = 0.00(s), iteration = 4
penalty = 0/3485 (hard/soft), time = 0.00(s), iteration = 8
penalty = 0/3047 (hard/soft), time = 0.00(s), iteration = 10


# penalty = 0/3047 (hard/soft)
# cpu time = 0.00/0.00(s)
# iteration = 10/10

[best solution]
x[0]: 1
x[1]: 3
x[2]: 4
x[3]: 0
x[4]: 2

penalty: 0/3047 (hard/soft)

[Violated constraints]
obj: 3047
```

3.7　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 9 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 $(6,6,5,5,5,4,4,4,4,2,2,2,2,3,3,7,7,5,5,8,8,4,4,5)$ とする（図 3.7）．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．

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．たと

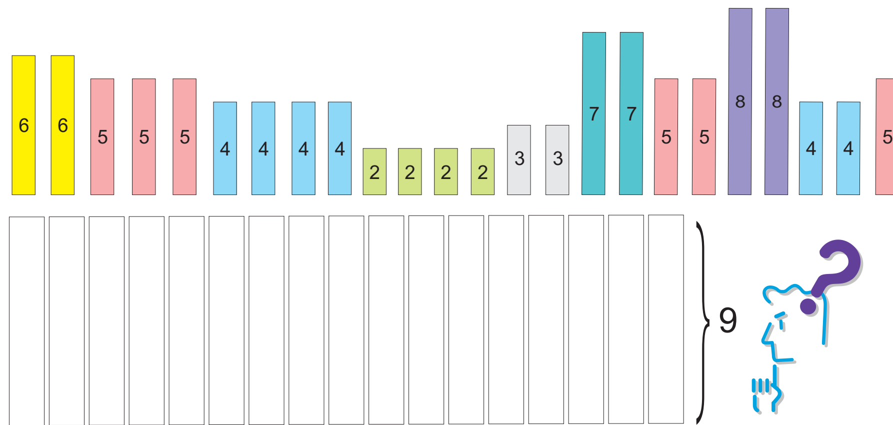

図 3.7: 箱詰め担当部長のビンパッキング問題の問題例．$n = 24$, $N = \{6,6,5,5,5,4,4,4,4,2,2,2,2,3,3,7,7,5,5,8,8,4,4,5\}$, $B = 9$.

えば，4.3 節で述べる実際問題は，3.4 節で解説したグラフ彩色問題とビンパッキング問題の融合した問題と考えられるので，SCOP によって容易に解決可能できる．

例に示した問題例の解を求める SCOP のデータは，以下のように書ける．

```
variable  x[0] in {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
...
variable  x[23] in {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
constraint0 : weight=1  type=linear 6 (x[0],0) 6 (x[1],0) 5 (x[2],0) 5 (x[3],0) 5 (x[4],0) 4 (x[5],0) 4 (x[6],0
 4 (x[7],0) 4 (x[8],0) 2 (x[9],0) 2 (x[10],0) 2 (x[11],0) 2 (x[12],0) 3 (x[13],0) 3 (x[14],0) 7 (x[15],0)
7 (x[16],0) 5 (x[17],0) 5 (x[18],0) 8 (x[19],0) 8 (x[20],0) 4 (x[21],0) 4 (x[22],0) 5 (x[23],0)  <=9
...
```

上のデータを生成するための Python プログラムは，以下のようになる．

```
1  Items=[6,6,5,5,5,4,4,4,4,2,2,2,2,3,3,7,7,5,5,8,8,4,4,5]
2  B=9
3  num_bins=sum(Items)/B+1
4  n=len(Items)
5  f = open("bpp-scop.dat", 'w')
6  binset=str(range(num_bins))[1:-1]
7  for i in range(n):
8      f.write("variable  x["+str(i)+"] in {"+binset+"}"+"\n")
9  for i in range(num_bins):
10     f.write( "constraint"+str(i)+" : weight=1  type=linear ")
11     for j in range(n):
12         f.write( str(Items[j])+" (x["+str(j)+"],"+str(i) +") " )
13     f.write(" <="+ str(B)+" \n")
14 f.close()
```

上のプログラムの 2 行目では，`str(range(num_bins))` で生成された文字列 $[0, 1, ..., \mathrm{num_bins} - 1]$ の両端の [] を除いた文字列を生成している．Python のバージョン 3.0 以降では，この部分は，以下のように変更する必要がある．

```
    bin_set=str(set(range(num_bins)))
```

SCOP による実行結果は，以下のようになる．また，結果を図示したものを図 3.8 に示す．

```
scop <bpp-scop.dat
# reading data ... done: 0.00(s)
```

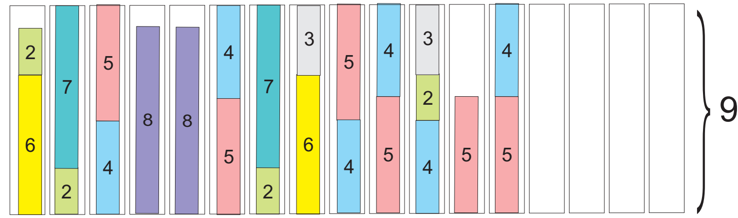

図 3.8: SCOP によるビンパッキング問題の解.

```
penalty = 0/24 (hard/soft), time = 0.01(s), iteration = 0
# improving the initial solution greedily
penalty = 0/11 (hard/soft), time = 0.01(s), iteration = 0
# start tabu search
penalty = 0/8 (hard/soft), time = 0.01(s), iteration = 1
penalty = 0/6 (hard/soft), time = 0.01(s), iteration = 2
penalty = 0/5 (hard/soft), time = 0.01(s), iteration = 3
penalty = 0/4 (hard/soft), time = 0.01(s), iteration = 5
penalty = 0/2 (hard/soft), time = 0.01(s), iteration = 7
penalty = 0/1 (hard/soft), time = 0.01(s), iteration = 9
...
penalty = 0/0 (hard/soft), time = 0.05(s), iteration = 123

# penalty = 0/0 (hard/soft)
# cpu time = 0.05/0.05(s)
# iteration = 123/123

[best solution]
x[0]: 0
x[1]: 7
x[2]: 5
x[3]: 9
x[4]: 12
x[5]: 2
x[6]: 10
x[7]: 5
x[8]: 8
x[9]: 10
x[10]: 1
x[11]: 6
x[12]: 0
x[13]: 10
x[14]: 7
x[15]: 6
x[16]: 1
x[17]: 2
x[18]: 11
x[19]: 4
x[20]: 3
x[21]: 9
x[22]: 12
x[23]: 8

penalty: 0/0 (hard/soft)

[Violated constraints]
```

3.8　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．

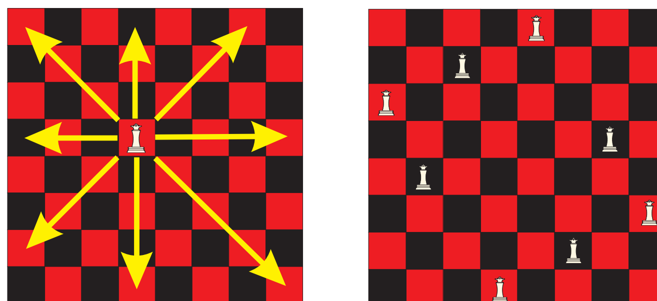

図 3.9: クイーンの動きと 8 クイーン問題の解の一例．

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 $\{1, 2, ..., 8\}$ 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．SCOP の入力データは，以下のようになる．

```
variable  x[1] in {1, 2, 3, 4, 5, 6, 7, 8}
variable  x[2] in {1, 2, 3, 4, 5, 6, 7, 8}
variable  x[3] in {1, 2, 3, 4, 5, 6, 7, 8}
variable  x[4] in {1, 2, 3, 4, 5, 6, 7, 8}
variable  x[5] in {1, 2, 3, 4, 5, 6, 7, 8}
variable  x[6] in {1, 2, 3, 4, 5, 6, 7, 8}
variable  x[7] in {1, 2, 3, 4, 5, 6, 7, 8}
variable  x[8] in {1, 2, 3, 4, 5, 6, 7, 8}
noconflict: weight=inf type=alldiff x[1] x[2] x[3] x[4] x[5] x[6] x[7] x[8] ;
rightdown2: weight=inf type=linear
1 (x[7] ,1) 1 (x[8] ,2) <=1
rightdown3: weight=inf type=linear
1 (x[6] ,1) 1 (x[7] ,2) 1 (x[8] ,3) <=1
rightdown4: weight=inf type=linear
1 (x[5] ,1) 1 (x[6] ,2) 1 (x[7] ,3) 1 (x[8] ,4) <=1
rightdown5: weight=inf type=linear
1 (x[4] ,1) 1 (x[5] ,2) 1 (x[6] ,3) 1 (x[7] ,4) 1 (x[8] ,5) <=1
rightdown6: weight=inf type=linear
1 (x[3] ,1) 1 (x[4] ,2) 1 (x[5] ,3) 1 (x[6] ,4) 1 (x[7] ,5) 1 (x[8] ,6) <=1
rightdown7: weight=inf type=linear
1 (x[2] ,1) 1 (x[3] ,2) 1 (x[4] ,3) 1 (x[5] ,4) 1 (x[6] ,5) 1 (x[7] ,6) 1 (x[8] ,7) <=1
rightdown8: weight=inf type=linear
1 (x[1] ,1) 1 (x[2] ,2) 1 (x[3] ,3) 1 (x[4] ,4) 1 (x[5] ,5) 1 (x[6] ,6) 1 (x[7] ,7) 1 (x[8] ,8) <=1
rightdown9: weight=inf type=linear
1 (x[1] ,2) 1 (x[2] ,3) 1 (x[3] ,4) 1 (x[4] ,5) 1 (x[5] ,6) 1 (x[6] ,7) 1 (x[7] ,8) <=1
rightdown10: weight=inf type=linear
1 (x[1] ,3) 1 (x[2] ,4) 1 (x[3] ,5) 1 (x[4] ,6) 1 (x[5] ,7) 1 (x[6] ,8) <=1
```

```
rightdown11: weight=inf type=linear
1 (x[1] ,4) 1 (x[2] ,5) 1 (x[3] ,6) 1 (x[4] ,7) 1 (x[5] ,8) <=1
rightdown12: weight=inf type=linear
1 (x[1] ,5) 1 (x[2] ,6) 1 (x[3] ,7) 1 (x[4] ,8) <=1
rightdown13: weight=inf type=linear
1 (x[1] ,6) 1 (x[2] ,7) 1 (x[3] ,8) <=1
rightdown14: weight=inf type=linear
1 (x[1] ,7) 1 (x[2] ,8) <=1
leftdown2: weight=inf type=linear
1 (x[1] ,2) 1 (x[2] ,1) <=1
leftdown3: weight=inf type=linear
1 (x[1] ,3) 1 (x[2] ,2) 1 (x[3] ,1) <=1
leftdown4: weight=inf type=linear
1 (x[1] ,4) 1 (x[2] ,3) 1 (x[3] ,2) 1 (x[4] ,1) <=1
leftdown5: weight=inf type=linear
1 (x[1] ,5) 1 (x[2] ,4) 1 (x[3] ,3) 1 (x[4] ,2) 1 (x[5] ,1) <=1
leftdown6: weight=inf type=linear
1 (x[1] ,6) 1 (x[2] ,5) 1 (x[3] ,4) 1 (x[4] ,3) 1 (x[5] ,2) 1 (x[6] ,1) <=1
leftdown7: weight=inf type=linear
1 (x[1] ,7) 1 (x[2] ,6) 1 (x[3] ,5) 1 (x[4] ,4) 1 (x[5] ,3) 1 (x[6] ,2) 1 (x[7] ,1) <=1
leftdown8: weight=inf type=linear
1 (x[1] ,8) 1 (x[2] ,7) 1 (x[3] ,6) 1 (x[4] ,5) 1 (x[5] ,4) 1 (x[6] ,3) 1 (x[7] ,2) 1 (x[8] ,1) <=1
leftdown9: weight=inf type=linear
1 (x[2] ,8) 1 (x[3] ,7) 1 (x[4] ,6) 1 (x[5] ,5) 1 (x[6] ,4) 1 (x[7] ,3) 1 (x[8] ,2) <=1
leftdown10: weight=inf type=linear
1 (x[3] ,8) 1 (x[4] ,7) 1 (x[5] ,6) 1 (x[6] ,5) 1 (x[7] ,4) 1 (x[8] ,3) <=1
leftdown11: weight=inf type=linear
1 (x[4] ,8) 1 (x[5] ,7) 1 (x[6] ,6) 1 (x[7] ,5) 1 (x[8] ,4) <=1
leftdown12: weight=inf type=linear
1 (x[5] ,8) 1 (x[6] ,7) 1 (x[7] ,6) 1 (x[8] ,5) <=1
leftdown13: weight=inf type=linear
1 (x[6] ,8) 1 (x[7] ,7) 1 (x[8] ,6) <=1
leftdown14: weight=inf type=linear
1 (x[7] ,8) 1 (x[8] ,7) <=1
```

このデータは，以下の Python プログラムによって自動生成したものである．

```
1  n=8
2  f = open("queen-scop.dat", 'w')
3  for i in range(n):
4      f.write("variable  x["+str(i+1)+"] in {"+ str(range(1,n+1))[1:-1] +"} \n")
5
6  f.write ("noconflict: weight=inf type=alldiff ")
7  for i in range(n):
8      f.write (str("x["+str(i+1)+"] "))
9  f.write("; \n")
10
11 for k in range(2,2*n-1): #right down diagonal constraints
12     f.write ("rightdown"+str(k)+": weight=inf type=linear \n")
13     for i in range(1,n+1):
14         j=k-n+i
15         if j>=1 and j<=n:
16             f.write("1 (x["+str(i)+"] "+","+str(j)+") ")
17     f.write("<=1 \n")
18
19 for k in range(2,2*n-1): #left down diagonal constraints
20     f.write ("leftdown"+str(k)+": weight=inf type=linear \n")
21     for i in range(1,n+1):
22         j=k+1-i
23         if j>=1 and j<=n:
24             f.write("1 (x["+str(i)+"] "+","+str(j)+") ")
25     f.write("<=1 \n")
26
27 f.write("<=0 \n")
```

```
28 f.close()
```

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．

```
f.write("obj: weight=1 type=linear ")
for i in range(1,n+1):
    for j in range(1,n+1):
        f.write(str(i*j)+" (x["+str(i)+"] "+","+str(j)+") ")
```

SCOP の入力データを生成して求解してみると，以下のようになる．

```
scop <queen-scop.dat
# reading data ... done: 0.00(s)

penalty = 7/169 (hard/soft), time = 0.00(s), iteration = 0
# improving the initial solution greedily
penalty = 3/154 (hard/soft), time = 0.00(s), iteration = 0
# start tabu search
penalty = 2/126 (hard/soft), time = 0.00(s), iteration = 1
penalty = 2/124 (hard/soft), time = 0.00(s), iteration = 2
penalty = 1/164 (hard/soft), time = 0.00(s), iteration = 3
penalty = 1/161 (hard/soft), time = 0.00(s), iteration = 5
penalty = 1/143 (hard/soft), time = 0.00(s), iteration = 7
penalty = 0/166 (hard/soft), time = 0.00(s), iteration = 14
penalty = 0/150 (hard/soft), time = 0.00(s), iteration = 93

[incumbent solution]
x[1]: 5
x[2]: 3
x[3]: 8
x[4]: 4
x[5]: 7
x[6]: 1
x[7]: 6
x[8]: 2

penalty: 0/150 (hard/soft)

[Violated constraints]
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．

第4章　応用例

本章では，SCOP を用いた様々な応用例を示す.

4.1　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.

4.2 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 $0,1,2,3$ で表し，期は $0,1,...,6$ で表す．変数は，スタッフの番号 i と期の番号 t を添え字として $x[i][t]$ とし，その領域をシフトに休日を加えた集合 $\{0,1,2,3\}$ とする．SCOP のデータは以下のように書ける．

```
variable  x[0][0] in {0, 1, 2, 3}
...
variable  x[3][6] in {0, 1, 2, 3}

LB0: weight=1 type=linear 1 (x[0][0],1) 1 (x[0][0],2) 1 (x[0][0],3) 1 (x[0][1],1) 1 (x[0][1],2) 1 (x[0][1],3)
1 (x[0][2],1) 1 (x[0][2],2) 1 (x[0][2],3) 1 (x[0][3],1) 1 (x[0][3],2) 1 (x[0][3],3) 1 (x[0][4],1)
1 (x[0][4],2) 1 (x[0][4],3)  1 (x[0][5],1) 1 (x[0][5],2) 1 (x[0][5],3) 1 (x[0][6],1)
1 (x[0][6],2) 1 (x[0][6],3) >=5
...

ShiftUB0_1: weight=1 type=linear 1 (x[0][0],1) 1 (x[1][0],1) 1 (x[2][0],1) 1 (x[3][0],1) =1
...
Different0_0_1: weight=1 type=linear 1 (x[0][0],1) 1 (x[0][1],2) 1 (x[0][1],3) <=1
...
Consecutive0_0_2: weight=1 type=linear -1 (x[0][0],2) 1 (x[0][6],2) 1 (x[0][1],2) >=0
...
```

上のデータは，以下の Python のプログラムで生成したものである．

```
1  periods=7
2  staffs=4
3  shifts=3
4  #shift 0 means OFF
5
6  f = open("staff.dat", 'w')
7  for i in range(staffs):
8      for t in range(periods):
9          f.write("variable  x["+str(i)+"]["+str(t)+"] in {"+ str(range(shifts+1))[1:-1] +"}  \n")
10
11 for i in range(staffs):
12     f.write ("LB"+str(i)+": weight=1 type=linear ")
13     for t in range(periods):
14         for j in range(1,shifts+1):
15             f.write("1 (x["+str(i)+"]["+str(t)+"]," +str(j)+") ")
16     f.write(">=5 \n")
17
18 for t in range(periods):
19     for j in range(1,shifts+1):
20         f.write ("ShiftUB"+str(t)+"_"+str(j)+": weight=1 type=linear ")
21         for i in range(staffs):
22             f.write("1 (x["+str(i)+"]["+str(t)+"]," +str(j)+") ")
23         f.write("=1 \n")
24
25 #forbid two different shifts on two consecutive days
26 for i in range(staffs):
27     for t in range(periods):
28         for j in range(1,shifts+1):
29             f.write ("Different"+str(i)+"_"+str(t)+"_"+str(j)+": weight=1 type=linear ")
30             f.write("1 (x["+str(i)+"]["+str(t)+"]," +str(j)+") ")
31             for k in range(1,shifts+1):
32                 if k!=j:
33                     if t==periods-1:
34                         f.write("1 (x["+str(i)+"]["+str(0)+"]," +str(k)+") ")
35                     else:
36                         f.write("1 (x["+str(i)+"]["+str(t+1)+"]," +str(k)+") ")
37             f.write("<=1 \n")
38
39 #shifts 2 and 3 must do at least two consecutive days
40 for i in range(staffs):
41     for t in range(periods):
42         for j in range(2,shifts+1):
```

```
43                  f.write ("Consecutive"+str(i)+"_"+str(t)+"_"+str(j)+": weight=1 type=linear ")
44                  f.write("-1 (x["+str(i)+"]["+str(t)+"]," +str(j)+") ")
45                  if t==0:
46                      f.write("1 (x["+str(i)+"]["+str(periods-1)+"]," +str(j)+") ")
47                  else:
48                      f.write("1 (x["+str(i)+"]["+str(t-1)+"]," +str(j)+") ")
49                  if t==periods-1:
50                      f.write("1 (x["+str(i)+"]["+str(0)+"]," +str(j)+") ")
51                  else:
52                      f.write("1 (x["+str(i)+"]["+str(t+1)+"]," +str(j)+") ")
53                  f.write(">=0 \n")
54  f.close()
```

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．

```
scop <staff.dat

# reading data ... done: 0.03(s)

penalty = 0/37 (hard/soft), time = 0.03(s), iteration = 0
# improving the initial solution greedily
penalty = 0/17 (hard/soft), time = 0.03(s), iteration = 0
# start tabu search
penalty = 0/15 (hard/soft), time = 0.03(s), iteration = 1
penalty = 0/13 (hard/soft), time = 0.03(s), iteration = 2
penalty = 0/11 (hard/soft), time = 0.03(s), iteration = 3
penalty = 0/8 (hard/soft), time = 0.03(s), iteration = 4
penalty = 0/7 (hard/soft), time = 0.03(s), iteration = 5
penalty = 0/6 (hard/soft), time = 0.03(s), iteration = 6
penalty = 0/5 (hard/soft), time = 0.03(s), iteration = 7
penalty = 0/4 (hard/soft), time = 0.03(s), iteration = 16
penalty = 0/3 (hard/soft), time = 0.03(s), iteration = 19
penalty = 0/2 (hard/soft), time = 0.03(s), iteration = 22
penalty = 0/1 (hard/soft), time = 0.03(s), iteration = 26
penalty = 0/0 (hard/soft), time = 0.03(s), iteration = 626

# penalty = 0/0 (hard/soft)
# cpu time = 0.03/0.03(s)
# iteration = 626/626

[best solution]
x[0][0]: 2
x[0][1]: 2
x[0][2]: 0
x[0][3]: 1
x[0][4]: 0
x[0][5]: 2
x[0][6]: 2
x[1][0]: 1
x[1][1]: 1
x[1][2]: 1
x[1][3]: 0
x[1][4]: 1
x[1][5]: 1
x[1][6]: 1
x[2][0]: 3
x[2][1]: 0
x[2][2]: 2
x[2][3]: 2
x[2][4]: 2
x[2][5]: 0
x[2][6]: 3
```

```
x[3][0]: 0
x[3][1]: 3
x[3][2]: 3
x[3][3]: 3
x[3][4]: 3
x[3][5]: 3
x[3][6]: 0

penalty: 0/0 (hard/soft)

[Violated constraints]
```

4.3 枝重み容量制約付きグラフ彩色問題

ここで解説する実際問題は，3.4 節のグラフ彩色問題と，3.7 節ビンパッキング問題を融合した問題と考えられる．これは，資材置き場に製品を保管する実際問題から抽出したものであり，現実的な問題を SCOP を用いて解決する良い例題と考えられる．

同じ資材置き場に保管すると競合する可能性がある複数の製品を，幾つかの保管場所に割り当てることを考える．製品の厚さは所与であり，これらの製品を保管場所の高さ上限の下で，保管場所の数が最小になるように配置することが目的となる．

同じ場所に保管すると競合する 2 つの製品 i, j に対する重み w_{ij} が与えられている．製品を点とみなすことによって，枝に重み w_{ij} を付加したグラフ彩色問題に帰着する．また，点上には，製品の厚さを表す重み h_i が付加されている．積み付けの高さ上限を H とする．

色数を K に固定したときの定式化を考える．点 i に塗られた色が ℓ のとき 1，それ以外のとき 0 の 0-1 変数 $x_{i\ell}$，枝 (i,j) の両端点 i, j が同色に塗られたとき 1，それ以外のとき 0 の 0-1 変数 y_{ij} を用いると，グラフ $G = (V, E)$ 上の重み付きグラフ彩色問題は，以下のような 0-1 整数計画問題として定式化できる．

$$
\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{(i,j)\in E} w_{ij} y_{ij} & \\
\text{subject to} & \displaystyle\sum_{\ell=1}^{K} x_{i\ell} = 1 & \forall i \in V \\
& x_{i\ell} + x_{j\ell} \leq y_{ij} & \forall (i,j) \in E, \ell = 1, \cdots, K \\
& \displaystyle\sum_{i\in V} h_i x_{i\ell} \leq H & \forall \ell = 1, \cdots, K \\
& x_{i\ell} \in \{0,1\} & \forall i \in V, \ell = 1, \cdots, K \\
& y_{ij} \in \{0,1\} & \forall (i,j) \in E
\end{array}
$$

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，容量制約を表している．3 番目の制約は，変数 x と y の繋ぎ条件を表している．

この定式化を市販の数理計画ソルバーにかけても，玩具問題以外は解くことができなかった．これは，専門的に言うと，問題が対称性を含んでいるためであり，数理計画ソルバーの求解原理である分枝限定法の弱点と言える．一方，SCOP を用いることによって，問題は自然な定式化が可能であり，現実規模の問題でも容易に良好な解を得ることができる．

4.4 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 SCOP のデータは，以下のようになる．

```
variable x[0] in {A,B,C,D}
...
variable x[119] in {A,B,C,D}

constraint0 : weight=inf  type=linear 1(x[0],A) 1(x[1],A) 1(x[2],A) 1(x[3],A)
              ... 1(x[118],A) 1(x[119],A) = 30
...
constraint3 : weight=inf  type=linear 1(x[0],D) 1(x[1],D) 1(x[2],D) 1(x[3],D)
              ... 1(x[118],D) 1(x[119],D) = 40

constraint0_0 : weight=1  type=linear 1(x[0],B) 1(x[0],C) 1(x[1],B) 1(x[1],C)
             1(x[2],B) 1(x[2],C) 1(x[3],B) 1(x[3],C) 1(x[4],B) 1(x[4],C) <= 3
...
constraint0_115 : weight=1  type=linear 1(x[115],B) 1(x[115],C) 1(x[116],B)
1(x[116],C) 1(x[117],B) 1(x[117],C) 1(x[118],B) 1(x[118],C) 1(x[119],B) 1(x[119],C) <= 3

constraint1_0 : weight=1  type=linear 1(x[0],A) 1(x[0],C) 1(x[1],A) 1(x[1],C) 1(x[2],A) 1(x[2],C) <= 2
...
constraint1_117 : weight=1  type=linear 1(x[117],A) 1(x[117],C) 1(x[118],A) 1(x[118],C)
                  1(x[119],A) 1(x[119],C) <= 2
```

上のデータを生成するための Python プログラムは，以下のようになる．

```
f=open("car-scop.dat","w")
Type=["A","B","C","D"]
Number=[30,30,20,40]
n=sum(Number)
Need=[["B","C"],["A","C"]]
Length=[5,3]
Capacity=[3,2]

for i in range(n):
    f.write("variable x["+str(i)+"] in {A,B,C,D}"+"\n")

for i in range(len(Type)):
    f.write("constraint"+str(i)+" : weight=inf  type=linear ")
    for j in range(n):
        f.write( "1(x["+str(j)+"],"+Type[i]+") " )
    f.write( "= "+str(Number[i])+" \n" )

for i in range(len(Length)):
    for k in range(n-Length[i]+1):
        f.write("constraint"+ str(i)+"_"+str(k)+ " : weight=1  type=linear ")
        for t in range(k,k+Length[i]):
            for j in range(len(Need[i])):
                f.write( "1(x["+str(t)+"],"+Need[i][j]+") ")
        f.write( "<= "+str(Capacity[i])+" \n" )
f.close()
```

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．

```
scop <car-scop.dat
# reading data ... done: 0.05(s)

penalty = 42/63 (hard/soft), time = 0.05(s), iteration = 0
# improving the initial solution greedily
penalty = 0/21 (hard/soft), time = 0.06(s), iteration = 0
# start tabu search
penalty = 0/14 (hard/soft), time = 0.08(s), iteration = 2
penalty = 0/9 (hard/soft), time = 0.08(s), iteration = 4
penalty = 0/4 (hard/soft), time = 0.08(s), iteration = 6
penalty = 0/2 (hard/soft), time = 0.08(s), iteration = 8
penalty = 0/1 (hard/soft), time = 0.09(s), iteration = 10
penalty = 0/0 (hard/soft), time = 0.09(s), iteration = 12

# penalty = 0/0 (hard/soft)
# cpu time = 0.09/0.09(s)
# iteration = 12/12

[best solution]
x[0]: D
x[1]: A
x[2]: C
x[3]: D
x[4]: D
x[5]: A
...
x[117]: C
x[118]: D
x[119]: A

penalty: 0/0 (hard/soft)

[Violated constraints]
```