

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 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	16
3.3	グラフ彩色問題	17
3.4	2次割当問題	19
第4章	応用例	21
4.1	時間割作成	21
4.2	スタッフスケジューリング	22

第1章 はじめに

SCOP (Solver for COntstraint 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より便利である.

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とりえる値の集合。

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、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切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」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, 3.4 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、以下ようになる。

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。

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

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 quadratic) 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。

-display

ログの出力レベルを設定する.

- 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.

-iteration

最大反復回数を設定する.

-seed

乱数系列の種を設定する.

-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.

-time

最大計算時間 (秒) を設定する.

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、絶対制約と見なされる)。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)からとったものである．

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 (10ℓ) 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？

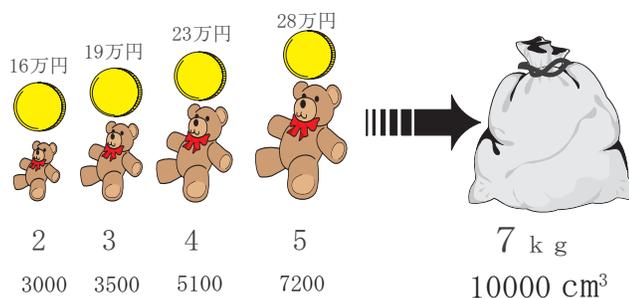


図 3.1: クマさん人形のラインアップと愛用のナップサック．

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．

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

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，市販の数値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)>=42
```

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．ただし，入力データは，制約行列を表す辞書 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 で与えられているものとする．

```
1 f = open("mkp-scop.dat", 'w')
2 for i in range(n):
3     f.write("variable x["+str(i)+"] in {0,1}"+ "\n")
4
5     for i in range(m):
6         f.write("constraint"+str(i)+" : weight=inf type=linear ")
7         for j in range(n):
8             f.write(str(a[i,j])+"(x["+str(j)+"],1) ")
9             f.write("<="+str(b[i])+"\n")
10
11     f.write("object: weight=1 type=linear ")
12     for i in range(n):
13         f.write(str(v[i])+"(x["+str(i)+"],1) ")
14
15     f.write(">="+str(sum(v)) + "\n")
16     f.close()
```

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？

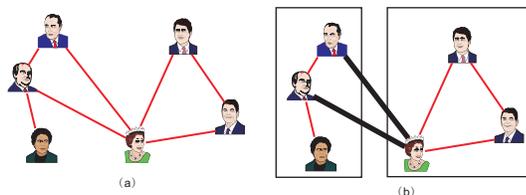


図 3.2: グラフ分割問題の例。(a) 線の引いてある人同士は仲良しであることを表すグラフ。(b) 仲良しが別のチームになることを最小にする等分割。違うチームに所属する仲良しのペアは太線で表されており、2本である。よって、この等分割の目的関数値は2となる。

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, equipartition) (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、 $L \cap R = \emptyset$ (共通部分がない)、 $L \cup R = V$ (合わせると点集合全体になる) を満たす非順序対 (L, R) を分割 (partition) もしくは 2 分割 (bipartition) とよぶ。分割 (L, R) において、 L は左側、 R は右側を表すが、これらは逆にしても同じ分割であるので、非順序対とよばれる。点 i が、分割 (L, R) の L 側に含まれているとき 1、それ以外の (R 側に含まれている) とき 0 の 0-1 変数 x_i を導入する。このとき、等分割であるためには、 x_i の合計が $n/2$ である必要がある。枝 (i, j) が L と R をまたいでいるときには、 $x_i(1 - x_j)$ もしくは $(1 - x_i)x_j$ が 1 になることから、以下の定式化を得る。

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

市販の数値計画ソルバーは、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値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。

ランダムに生成した 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[5],0) 1 (x[0],0)(x[5],1) 1 (x[1],1)(x[3],0) 1 (x[1],0)(x[3],1)
1(x[1],1)(x[5],0) 1 (x[1],0)(x[5],1) 1 (x[2],1)(x[4],0) 1 (x[2],0)(x[4],1) 1 (x[3],1)(x[1],0)
1 (x[3],0)(x[1],1) 1 (x[3],1)(x[5],0) 1 (x[3],0)(x[5],1) 1 (x[4],1)(x[2],0) 1 (x[4],0)(x[2],1)
1 (x[5],1)(x[0],0) 1 (x[5],0)(x[0],1) 1 (x[5],1)(x[1],0) 1 (x[5],0)(x[1],1)
1 (x[5],1)(x[3],0) 1 (x[5],0)(x[3],1)<=0
```

上のデータを生成するための Python プログラムは、以下ようになる。ただし、グラフは点のリスト `nodes` と、隣接する点のリスト `adj` で与えられているものとする。

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

3.3 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 3.3 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。

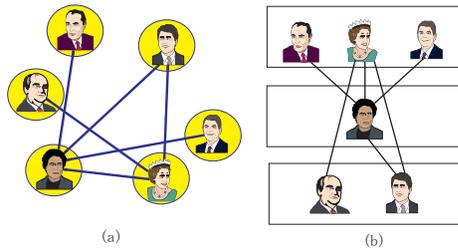


図 3.3: グラフ彩色問題の例 . (a) 線の引いてある人同士は仲が悪いことを表すグラフ . (b) 3 つのクラスに分けると仲が悪い友達は同じクラスに入らない . 目的関数値 (クラス数) は 3 となり , これが最適解になる .

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 .

ランダムに生成した 10 点のグラフを 4 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, 3}
variable x[1] in {0, 1, 2, 3}
variable x[2] in {0, 1, 2, 3}
variable x[3] in {0, 1, 2, 3}
variable x[4] in {0, 1, 2, 3}
variable x[5] in {0, 1, 2, 3}
variable x[6] in {0, 1, 2, 3}
variable x[7] in {0, 1, 2, 3}
variable x[8] in {0, 1, 2, 3}
variable x[9] in {0, 1, 2, 3}
constraint08 : weight=1 type=alldiff x[0] x[8] ;
constraint01 : weight=1 type=alldiff x[0] x[1] ;
constraint02 : weight=1 type=alldiff x[0] x[2] ;
constraint04 : weight=1 type=alldiff x[0] x[4] ;
constraint12 : weight=1 type=alldiff x[1] x[2] ;
constraint13 : weight=1 type=alldiff x[1] x[3] ;
constraint15 : weight=1 type=alldiff x[1] x[5] ;
constraint16 : weight=1 type=alldiff x[1] x[6] ;
constraint24 : weight=1 type=alldiff x[2] x[4] ;
constraint25 : weight=1 type=alldiff x[2] x[5] ;
constraint26 : weight=1 type=alldiff x[2] x[6] ;
constraint28 : weight=1 type=alldiff x[2] x[8] ;
constraint34 : weight=1 type=alldiff x[3] x[4] ;
```

上のデータを生成するための Python プログラムは , 以下ようになる . ただし , グラフは点のリスト `nodes` と , 隣接する点のリスト `adj` で与えられているものとする .

```
1 n=len(nodes)
2 f = open("gcp-scop.dat", 'w')
3 colorset=str(range(K))[1:-1]
4 print colorset
5 for i in range(n):
```

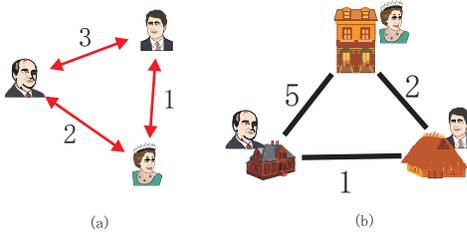


図 3.4: 2 次割当問題の例 . (a) 3 人のお友達が打ち合わせをする頻度 . 枝の上の数字は週に何回打ち合わせをするかを表す . (b) 3 箇所の家とお友達の割り当ての例 . 枝の上の数字は地点間の距離を表す . お友達同士の行き来する頻度と割り当てられた家との距離の和は, $2 \times 5 + 3 \times 1 + 1 \times 2 = 15$ となり, この割り当ての目的関数値は, その 2 倍 (お友達同士は往復するから) で $2 \times 15 = 30$ となる .

```

6         f.write("variable  x["+str(i)+"] in {"+colorset+"}"+ "\n")
7
8     for i in range(n):
9         for j in adj[i]:
10            if i<j:
11                f.write("constraint "+str(i)+str(j)+": weight=1  type=alldiff ")
12                f.write("x["+str(i)+"] "+ "x["+str(j)+"] ; \n" )
13
14     f.close()

```

3.4 2 次割当問題

いま, 3 人のお友達が 3 箇所の人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 3.4 (a) のようになっている . 家との移動距離は, 図 3.4 (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, 施設の配置を決定する応用から生まれた . n 個の施設があり, それを n 箇所の地点に配置することを考える . 施設 i, j 間には物の移動量 f_{ij} があり, 地点 k, ℓ 間を移動するには距離 $d_{k\ell}$ がかかるものとする . 問題の目的は, 物の総移動距離を最小にするように, 各地点に 1 つずつ施設を配置することである . 順列 π は施設 i を地点 $\pi(j)$ に配置することを表す .

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

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

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 .

ランダムに生成した $n = 5$ の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}
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
60(x[0], 0) (x[1], 1) 42(x[0], 0) (x[1], 2) 18(x[0], 0)
...
10(x[3], 4) (x[4], 2) 10(x[3], 4) (x[4], 3) <=0

```

上のデータを生成するための Python プログラムは , 以下ようになる . ただし , 行列は 2 つの辞書 f, d で与えら
れているものとする .

```

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

```

第4章 応用例

本章では、SCOP を用いて様々な実際問題を解決するためのコツについて解説する。

4.1 時間割作成

最初の実例は、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（専門的には解の対称性のため）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、通常は、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も、線形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。