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

テクニカル ドキュメント

LOG OPT Co., Ltd.

ご注意

目 次

第 1 章　問題定義 **3**

第 2 章　アルゴリズム **4**

第 3 章　例題 **6**

第 4 章　プログラムの構成 **7**

第 5 章　プログラムの使用法 **8**

5.1　テキストファイルからの入力 8

5.2　C++ クラスライブラリとしての使用法 12

5.3　AMPL からの実行 19

第1章　問題定義

本ドキュメントでは，制約計画ソルバー SCOP (Solver for Constraint Programing) で対象とする制約充足問題と，それを解くためのアルゴリズムについて解説する．

制約充足問題 (Constraint Satisfaction Problem, CSP) とは，n 個の**変数** X_i $(i = 0, 1, \ldots, n-1)$ と各変数 X_i の値集合 D_i，m 個の**制約** C_l $(l = 0, 1, \ldots, m-1)$ が与えられたとき，すべての制約を満たすよう各変数 X_i に値 $j \in D_i$ を割当てる問題である．ここで，各制約 C_l は変数 $X_{l_1}, X_{l_2}, \cdots, X_{l_{t_l}}$ に対する t_l-項制約であり，それらの変数が同時にとることのできる値の組の集合で与えられる．一般に，各制約 C_l の表現方法は一意ではなく，等式や不等式，非等式，論理式など，問題に応じて適切な表現方法を自由に用いることができる．

以下では，変数 X_i とその値 j $(\in D_i)$ の組それぞれに対し**値変数** x_{ij} を

$$x_{ij} = \begin{cases} 1, & \text{変数 } X_i \text{ が値 } j \text{ をとる}, \\ 0, & \text{その他}, \end{cases}$$

と定義し，割当てを 0-1 ベクトル $\boldsymbol{x} = (x_{ij} \mid i = 0, 1, \cdots, n-1, j \in D_i)$ で表す．ここで，各変数にはちょうど 1 つ値が割当てられるため，

$$\sum_{j \in D_i} x_{ij} = 1, \qquad i = 0, 1, \ldots, n-1, \tag{1.1}$$

でなくてはならない．等式 (1.1) を満たす 0-1 ベクトル $\boldsymbol{x}$ を CSP の解と呼び，さらに，すべての制約を満たす解を実行可能解と呼ぶ．CSP の目的は実行可能解を求めることであるが，とくに現実の応用において必ずしも実行可能解が存在するとは限らない．このような状況に対応するため，本研究では CSP を制約違反最小化問題として扱う．すなわち，各制約 C_l に制約違反度を表す**ペナルティ関数** p_l を導入し，これを最小化する．ただし，解 $\boldsymbol{x}$ が C_l を満たすとき $p_l(\boldsymbol{x}) = 0$，満たさないとき $p_l(\boldsymbol{x}) > 0$ であるとする．また，制約間の重要度の違いを考慮するため，各制約 C_l は**ペナルティ重み** $w_l > 0$ を持つものとする．これにより，本研究で考える問題は

$$\begin{aligned} &\text{minimize} && p(\boldsymbol{x}) = \sum_l w_l p_l(\boldsymbol{x}) \\ &\text{subject to} && (1.1) \end{aligned}$$

となる．以下ではこの問題を**重み付き制約充足問題 (Weighted CSP, WCSP)** と呼ぶ．

第2章　アルゴリズム

本アルゴリズムは, 代表的なメタヒューリスティクスの一つであるタブー探索法に基づいている. 探索空間は解集合全体 (制約 (1.1) を満たす 0-1 ベクトル集合) であり, 解 $\boldsymbol{x}$ の近傍 $N(\boldsymbol{x})$ は, $\boldsymbol{x}$ から, ある一つの変数の値を変更することで得られる解の集合とする. すなわち, 解 $\boldsymbol{x}$ に対し, 変数 X_i の値を現在の値 j' から別の値 j へ変更した解を $\boldsymbol{x}(x_{ij} \leftarrow 1)$ で表せば, $N(\boldsymbol{x}) = \{\boldsymbol{x}(x_{ij} \leftarrow 1) \mid x_{ij} = 0,\ j \in D_i,\ i = 0, 1, \ldots, n-1\}$ である.

近傍探索において近傍内の解 $\boldsymbol{x}(x_{ij} \leftarrow 1) \in N(\boldsymbol{x})$ は, ペナルティ関数値の変化量

$$p(\boldsymbol{x}(x_{ij} \leftarrow 1)) - p(\boldsymbol{x}) = \sum_l \left(p_l(\boldsymbol{x}(x_{ij} \leftarrow 1)) - p_l(\boldsymbol{x})\right)$$

によって評価される. この評価を効率よく行うため, 制約 C_l および値変数 x_{ij} それぞれに対し, 探索中, $p_l(\boldsymbol{x}(x_{ij} \leftarrow 1)) - p_l(\boldsymbol{x})$ の値を保持しておく必要がある.

現在のバージョンでは, 基本的な制約として以下の 2 種類を用意している:

- 線形等式・不等式制約 C_l

  $$\sum_{ij} a_{lij} x_{ij} \sim b_l$$

  ここで, $\sim \in \{=, \leq, \geq\}$ であり, 係数 a_{lij} および b_l は整数値であるとする. ペナルティ関数は, 左辺を $f_l(\boldsymbol{x})$ として

  $$p_l(\boldsymbol{x}) := \begin{cases} |f_l(\boldsymbol{x}) - b_l| & f_l(\boldsymbol{x}) = b_l \text{のとき}, \\ \max\{f_l(\boldsymbol{x}) - b_l, 0\} & f_l(\boldsymbol{x}) \leq b_l \text{のとき}, \\ \max\{b_l - f_l(\boldsymbol{x}), 0\} & f_l(\boldsymbol{x}) \geq b_l \text{のとき}, \end{cases}$$

  で与えられる.

- 2 次等式・不等式制約 C_l

  $$\sum_{ij} a_{lijkh} x_{ij} x_{kh} \sim b_l$$

  ここで, $\sim \in \{=, \leq, \geq\}$ であり, 係数 a_{lijkh} および b_l は整数値であるとする. ペナルティ関数は, 左辺を $f_l(\boldsymbol{x})$ として

  $$p_l(\boldsymbol{x}) := \begin{cases} |f_l(\boldsymbol{x}) - b_l| & f_l(\boldsymbol{x}) = b_l \text{のとき}, \\ \max\{f_l(\boldsymbol{x}) - b_l, 0\} & f_l(\boldsymbol{x}) \leq b_l \text{のとき}, \\ \max\{b_l - f_l(\boldsymbol{x}), 0\} & f_l(\boldsymbol{x}) \geq b_l \text{のとき}, \end{cases}$$

  で与えられる.

- all_different 制約 C_l

  $$\text{all\_different}(V_l)$$

与えられた変数集合 $V_l \subseteq \{X_0, X_1, \ldots, X_{n-1}\}$ に対し, V_l に含まれる変数すべてが互いに異なる値をとらなくてはならない. ペナルティ関数は

$$p_l(\boldsymbol{x}) := |V_l| - |\{j \mid \exists X_i \in V_l, x_{ij} = 1\}|$$

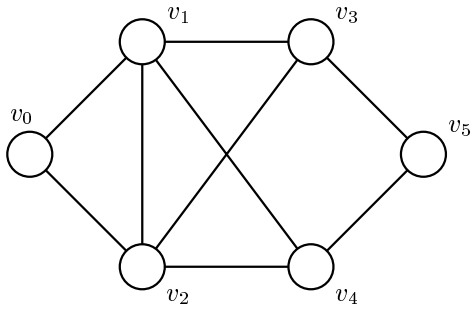
で与えられる.

この他, ユーザが新たに制約を定義することも可能である.

本アルゴリズムでは, 探索能力を高めるため, およびパラメータ調整に伴うユーザの手間を軽減するために, タブー期間 (tabu tenure) とペナルティ重みの自動調整機能を組み込んでいる.

第3章　例題

コスト付きのグラフ彩色問題を考える. 以下のグラフの各節点に「赤」, 「黄」, もしくは「青」の色を塗りたい. ただし, 隣接する節点は異なる色でなくてはならない (彩色条件). また, 「赤」, 「黄」, 「青」にはそれぞれコスト 1, 2, 3 がかかり, 彩色条件の下, 総コストを最小化することが目的である.



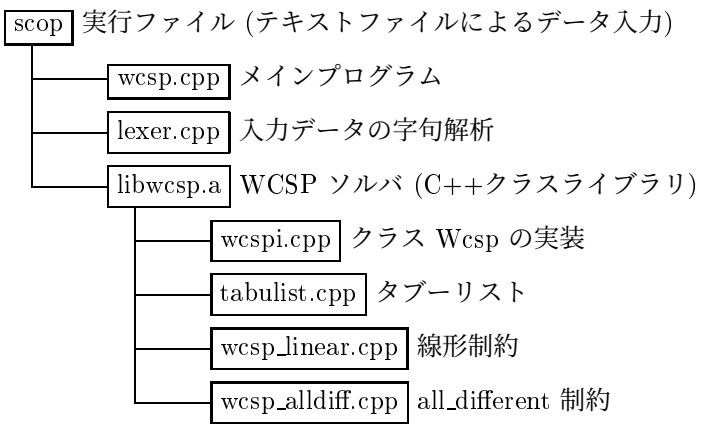
この問題は, 節点 v_i を変数 X_i $(i = 0, 1, \ldots, 5)$ に対応させ, それぞれの値集合を $D_i = \{0, 1, 2\}$ (0, 1, 2 はそれぞれ赤, 黄, 青に対応) とすることで WCSP に定式化できる. 彩色条件は, 各枝 (v_{i_1}, v_{i_2}) に対し制約 all_different$(\{X_{i_1}, X_{i_2}\})$ を課すことで記述できる. (節点集合 $\{v_0, v_1, v_2\}$ などクリークに対して all_different$(\{X_0, X_1, X_2\})$ を用いることで制約数を減らすことができる.) これらの制約は必ず満たさなくてはならない制約 (絶対制約) であるため, 重みを ∞ とする. 一方, 目的関数 (総コスト) は, 以下の線形不等式制約 (重み 1) で記述することができる:

$$\sum_{i=0}^{5} (x_{i0} + 2x_{i1} + 3x_{i2}) \leq 0. \tag{3.1}$$

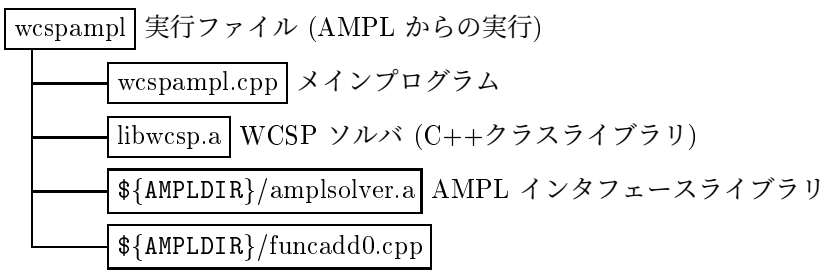
制約 (3.1) のペナルティ関数値が総コストを表すため, ペナルティ最小化により総コスト最小化が実現される.

第4章　プログラムの構成

プログラムの構成を以下に示す.



本プログラムの核となる WCSP ソルバは C++ クラスライブラリであり, ユーザは自分のプログラムから直接これを呼ぶことができる. プログラム scop は, テキストファイルからデータ入力できるよう WCSP ソルバに簡易インタフェースを加えたものである.



wcspampl は, AMPL (A Modeling Language for Mathematical Programming, 数理計画用モデリング言語)[1] から WCSP ソルバを呼び出せるようにしたものであり, AMPL, WCSP ソルバの他に, AMPL インタフェースライブラリをインストールする必要がある. 詳細は, 文献 [2] もしくは `http://www.ampl.com/hooking.html` を参照されたい.

第5章　プログラムの使用法

5.1　テキストファイルからの入力

make コマンドでコンパイル後, 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.

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.

問題例のデータは標準入力から読み込まれる. 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. 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

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.

例として, 3 章のグラフ彩色問題を記述した入力ファイルを以下に示す (example_gcp.dat).

```
variable vertex[0] in {red,yellow,blue}
variable vertex[1] in {red,yellow,blue}
variable vertex[2] in {red,yellow,blue}
variable vertex[3] in {red,yellow,blue}
variable vertex[4] in {red,yellow,blue}
variable vertex[5] in {red,yellow,blue}

target = 10

Clique0_1_2: weight = inf type = alldiff     vertex[0] vertex[1] vertex[2] ;
Clique1_2_3: weight = inf type = alldiff     vertex[1] vertex[2] vertex[3] ;
Clique1_2_4: weight = inf type = alldiff     vertex[1] vertex[2] vertex[4] ;

Edge3_5: weight = inf type = alldiff     vertex[3] vertex[5] ;
Edge4_5: weight = inf type = alldiff     vertex[4] vertex[5] ;

Objective: weight = 1 type = linear
1 (vertex[0],red) +2 (vertex[0],yellow) +3 (vertex[0],blue)
1 (vertex[1],red) +2 (vertex[1],yellow) +3 (vertex[1],blue)
1 (vertex[2],red) +2 (vertex[2],yellow) +3 (vertex[2],blue)
1 (vertex[3],red) +2 (vertex[3],yellow) +3 (vertex[3],blue)
1 (vertex[4],red) +2 (vertex[4],yellow) +3 (vertex[4],blue)
1 (vertex[5],red) +2 (vertex[5],yellow) +3 (vertex[5],blue) <= 0
```

これを実行したときの出力例を以下に示す. この場合, 11 回目の反復で目標値を達成する解が求まり, プログラムは終了している. なお, ペナルティ値は絶対制約, 考慮制約それぞれ別に計算・出力される (`hard` と `soft`).

```
% ./scop < example_gcp.dat
# reading data ... done: 0.00(s)

penalty = 1/11 (hard/soft), time = 0.00(s), iteration = 0
# improving the initial solution greedily
penalty = 0/13 (hard/soft), time = 0.00(s), iteration = 0
```

```
# start tabu search
penalty = 0/10 (hard/soft), time = 0.00(s), iteration = 11



# penalty = 0/10 (hard/soft)
# cpu time = 0.00/0.00(s)
# iteration = 11/11

[best solution]
vertex[0]: red
vertex[1]: blue
vertex[2]: yellow
vertex[3]: red
vertex[4]: red
vertex[5]: yellow

penalty: 0/10 (hard/soft)
```

5.2 C++ クラスライブラリとしての使用法

3 章のグラフ彩色問題を解くプログラム (example_gcp.cpp) を例に, 概略を説明する.

```
#include <iostream>
#include <string>
#include "wcsp.h"
#include "wcsp_linear.h"
#include "wcsp_alldiff.h"
```

クラス宣言をインクルードする. wcsp_linear.h, wcsp_alldiff.h は必要に応じてインクルードすればよい.

```
int main(){
  Wcsp *wcsp = Wcsp::makeWcsp();
  vector<unsigned int> numValues(6,3);
  wcsp->initialize(6, numValues);
```

WCSP ソルバを使用するためには, まず `Wcsp::makeWcsp()` によって Wcsp オブジェクトを生成する必要がある. その後, `Wcsp::initialize(unsigned int, vector<unsigned int>&)` によって変数の数, および各変数の値数を指定する.

```
unsigned int clique[3][3] = { {0, 1, 2}, {1, 2, 3}, {1, 2, 4} };
for (WcspId l = 0; l < 3; ++l){
  WcspAlldiff *alldiff = new WcspAlldiff(wcsp, Wcsp::inf);
  alldiff->addVariable(clique[l][0]);
  alldiff->addVariable(clique[l][1]);
  alldiff->addVariable(clique[l][2]);
}
unsigned int edge[2][2] = { {3, 5}, {4, 5} };
for (WcspId l = 0; l < 2; ++l){
  WcspAlldiff *alldiff = new WcspAlldiff(wcsp, Wcsp::inf);
  alldiff->addVariable(edge[l][0]);
  alldiff->addVariable(edge[l][1]);
}
```

各制約は, WcspConstraint の派生クラスのオブジェクトでなくてはならない. 制約は, コンストラクタによる初期化の際, 指定された Wcsp オブジェクトに追加される. コンストラクタの第 2 引数は制約の重みを表し, `Wcsp::inf` は ∞, すなわち制約が絶対制約であることを意味する.

```
WcspLinear *linear = new WcspLinear(wcsp, 1);
for (WcspId i = 0; i < wcsp->getNumVariables(); ++i){
  for (WcspId j = 0; j < wcsp->getNumValues(i); ++j){
    linear->addTerm(j+1, i, j);
  }
}
linear->setUb(0);
```

目的関数 (重み 1 の制約) の追加. `WcspLinear::addTerm(j+1, i, j)` で項 $(j+1)x_{ij}$ を線形関数 $f(\boldsymbol{x})$ に追加し, `WcspLinear::setUb(0)` で制約が $f(\boldsymbol{x}) \leq 0$ という形の不等式であることを指定する.

```
wcsp->setTarget(10);
wcsp->solve();
```

目標値を 10 に設定し, 解を求める.

```
  string color[3] = { "red", "yellow", "blue" };
  for (WcspId i = 0; i < wcsp->getNumVariables(); ++i){
    cout << "node " << i << ": " << color[wcsp->getValue(i)] << endl;
  }
```

タブー探索で得られた最良解を出力する.

```
  delete wcsp;
  return 0;
}
```

Wcsp オブジェクトを解放し (このとき, 制約オブジェクトも自動的に解放される), 終了. 以下は, このプログラムファイル example_gcp.cpp をコンパイル, 実行した例である (libwcsp.a がライブラリファイル).

```
% g++ -Wall -O2 -o example_gcp example_gcp.cpp libwcsp.a
% ./example_gcp
penalty = 1/11 (hard/soft), time = 0.00(s), iteration = 0
# improving the initial solution greedily
penalty = 0/13 (hard/soft), time = 0.00(s), iteration = 0
# start tabu search
penalty = 0/10 (hard/soft), time = 0.00(s), iteration = 11



# penalty = 0/10 (hard/soft)
# cpu time = 0.00/0.00(s)
# iteration = 11/11
node 0: red
node 1: blue
node 2: yellow
node 3: red
node 4: red
node 5: yellow
```

次に, 各クラスの説明を行う.

WcspId (型定義)

```
typedef unsigned int WcspId;
```

により定義. 変数, 値, および制約のインデックス番号 (非負整数) を表す.

Wcsp (プロトコルクラス)

WCSP を解くためのクラス. オブジェクトは `Wcsp::makeWcsp` により生成される. `Wcsp::solve` の前に各種パラメータを設定し, `Wcsp::initialize` を実行する必要がある. オブジェクトの解放時, 追加されたすべての制約オブジェクトも解放される.

静的定数

`static const int inf`

無限大 ∞ を表す定数. 絶対制約の重みを指定するときなどに使用.

公開メンバ関数

`static Wcsp* makeWcsp()`

Wcsp クラスオブジェクトを生成し, そのポインタを返す.

`virtual ~Wcsp()`

デストラクタ. 自身に追加された制約オブジェクトも解放する.

`void setMaxIteration(unsigned int iteration)`

最大反復回数を設定する.

`void setMaxCpuTime(double time)`

最大計算時間 (秒) を設定する.

`void setTarget(int soft)`

目標とするペナルティ値を設定する. `setTarget(0, soft)` と同じ.

`void setTarget(int hard, int soft)`

目標とするペナルティ値を, 絶対制約, および考慮制約に対するペナルティの組で設定する. (辞書式順序の意味で) 目標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.

`void setRandomSeed(int seed)`

乱数系列の種を設定する.

`void setDisplay(int display)`

ログの出力レベルを設定する (0〜3).

`void setInterval(unsigned int interval)`

解移動情報を何反復ごとに出力するか設定する.

`void setAdjustment(bool adjustment)`

ペナルティ重み自動調整機能を有効 (`true`) もしくは無効 (`false`) にする.

`void initialize(unsigned int numVariables, vector<unsigned int> &numValues)`

変数の数 n, および各変数 X_i の値数 $|D_i|$ を設定する.

`unsigned int getNumVariables() const`

変数の数 n を返す.

`unsigned int getNumValues(WcspId variable) const`

指定された変数の値数を返す.

`unsigned int getNumConstraints() const`

制約の数 m を返す.

`void solve()`

ランダムな初期解からタブー探索を開始する.

`void solve(const vector<WcspId> &initial)`

指定された初期解からタブー探索を開始する.

`WcspId getValue(WcspId variable) const`

指定された変数の値を返す.

`int getHardPenalty() const`

絶対制約の総ペナルティ値を返す.

`int getSoftPenalty() const`

考慮制約の総ペナルティ値を返す.

`int getPenalty(WcspId constraint) const`

指定された制約の (重み付けされていない) ペナルティ値を返す.

WcspConstraint (抽象クラス)

すべての制約クラスの基底クラス. これを継承したクラスを作成することで, 新たな制約タイプを定義することができる. `preprocess` と `calculate` は純粋仮想関数であり, 派生クラスにおいて実装しなくてはならない.

限定公開メンバ関数

`WcspConstraint(Wcsp *wcsp, int weight = 1)`

コンストラクタ. 呼び出し時に, 第 1 引数として指定された Wcsp オブジェクトに制約として追加される. 第 2 引数を省略すると, 重み 1 と見なされる.

`void involve(WcspId variable, WcspId value)`

preprocess 内で呼ばれる. 指定された値変数が制約に関わっていることをソルバに通知する (preprocess 参照).

`void involve(WcspId variable)`

preprocess 内で呼ばれる. 指定された変数に対応するすべての値変数が制約に関わっていることをソルバに通知する (preprocess 参照).

`void setDiffPenalty(WcspId variable, WcspId value, int diff)`

calculate 内で呼ばれる. 指定された変数の値を指定された値に変更したときのペナルティ変化量をソルバに通知する (calculate 参照).

`const Wcsp *getWcsp()`

自身が追加された Wcsp オブジェクトへのポインタを返す.

公開メンバ関数

`virtual ~WcspConstraint()`

デストラクタ. 制約オブジェクトの解放は Wcsp オブジェクトの解放時に行われる.

`virtual void preprocess() = 0`

近傍探索効率化のためタブー探索実行前に呼ばれる関数. 制約に関わっている値変数を関数 involve を用いてソルバに通知する.

ここで, 「値変数 x_{ij} が制約 C_l に関わっている」とは, ある解 (0-1 ベクトル) $\boldsymbol{x}$ が存在し, x_{ij} の値を反転したベクトル $\boldsymbol{x}'$ が $p_l(\boldsymbol{x}) \neq p_l(\boldsymbol{x}')$ を満たすことである.

`virtual int calculate(const vector<WcspId> &solution) = 0`

近傍探索の際に呼ばれる関数. 指定された解 solution ($\boldsymbol{x}$ とする) に対し, 変数 X_i の値を j に変更したときのペナルティ変化量 $p_l(\boldsymbol{x}(x_{ij} \leftarrow 1)) - p_l(\boldsymbol{x})$ を関数 setDiffPenalty を用いてソルバに通知する. 返り値は $p_l(\boldsymbol{x})$.

setDiffPenalty は, 変数 X_i, 値 j のインデックス番号の辞書式順 (昇順) に呼び出す必要があり, さらに, 各 (X_i, j) に対して高々 1 度しか呼び出してはならない. なお, ペナルティ変化量が 0 である (X_i, j) の組に対して setDiffPenalty を呼ぶ必要はない.

`WcspId getId()`

自身のインデックス番号を返す.

`int getWeight()`

自身の重みを返す.

WcspLinear (WcspConstraint を公開継承)

線形等式・不等式制約クラス.

`WcspLinear(Wcsp *wcsp, int weight = 1)`

`~WcspLinear()`

`void addTerm(int coefficient, WcspId variable, WcspId value)`

指定された項を線形関数 $f(\boldsymbol{x})$ に追加する.

`void setLb(int lb)`

$f(\boldsymbol{x})$ の下限を指定する ($f(\boldsymbol{x}) \geq$ `lb`).

`void setUb(int ub)`

$f(\boldsymbol{x})$ の上限を指定する ($f(\boldsymbol{x}) \leq$ `ub`).

WcspQuadratic (WcspConstraint を公開継承)

2 次等式・不等式制約クラス.

`WcspQuadratic(Wcsp *wcsp, int weight = 1)`

`~WcspQuadratic()`

`void addTerm(int coefficient, WcspId variable1, WcspId value1 WcspId variable2, WcspId value2)`

指定された項を 2 次関数 $f(\boldsymbol{x})$ に追加する.

`void setLb(int lb)`

$f(\boldsymbol{x})$ の下限を指定する ($f(\boldsymbol{x}) \geq$ `lb`).

`void setUb(int ub)`

$f(\boldsymbol{x})$ の上限を指定する ($f(\boldsymbol{x}) \leq$ `ub`).

WcspAlldiff (WcspConstraint を公開継承)

all_different 制約クラス.

`WcspAlldiff(Wcsp *wcsp, int weight = 1)`

`~WcspAlldiff()`

`void addVariable(WcspId variable)`

指定された変数を変数集合 V に追加する.

5.3 AMPL からの実行

WCSP ソルバを AMPL から呼び出すには wcspampl を用いる. wcspampl のコンパイルは, AMPL インタフェースライブラリおよびヘッダファイルの場所を makefile 内の AMPLDIR で指定した後,

```
% make wcspampl
```

で行う.

まず, AMPL から WCSP ソルバを呼び出す際の注意点を以下に挙げる.

- AMPL 内で定義される変数 (`var`) はすべて 0-1 変数 (`binary`) でなくてはならない.
- 制約としては線形等式・不等式制約のみ扱うことができる.
- AMPL で記述するモデルに変数 X_i, 値集合 D_i という概念はなく, 制約 (1.1) (各変数 X_j はただ 1 つの値 j をとる) もユーザが陽に記述しなくてはならない.

  AMPL で記述されたモデル (問題) は, wcspampl 内部で WCSP に変換される. その際, $\sum_{k\in K} x_k = 1$ という形の制約は制約 (1.1) と見なされ, 値集合 K を持つ変数 X_i が定義される. どの変数 X_i にも対応しない 0-1 変数 x_k は, 値集合 $\{0,1\}$ をもつ変数 X_{i_k} として定義される.
- 目的関数 f の最小化 (`minimize f`) および最大化 (`maximize f`) は, それぞれ, 重み 1 の不等式制約 $f \leq 0$, $f \geq 0$ と見なされる (f は定数項を含んでいてもよい). 目的関数を複数個指定することも可能である.
- 制約 (`subject to`) の重みは, 目的関数が存在しないとき 1, 存在するときは ∞ (絶対制約) と見なされる.

以下, 3 章のグラフ彩色問題を解く AMPL ファイル (example_gcp.amp) の説明を行う (AMPL の使用法については [1] を参照).

```
param num_node;
set NODE = {0..num_node-1};
set EDGE dimen 2;
set COLOR;
param cost {COLOR} > 0;
var x {i in NODE, j in COLOR} binary;
```

変数はすべて `binary`.

```
minimize total_cost: sum {i in NODE, j in COLOR} cost[j] * x[i,j] ;
```

目的関数の設定. 重みが 1 の制約として扱われる.

```
subject to edge {(i1,i2) in EDGE, j in COLOR}: x[i1,j] + x[i2,j] <= 1;
```

彩色条件. all_different 制約が使えないので, 各枝に対し, 色の数だけ不等式制約を導入している. 目的関数が定義されているので, これらの制約は絶対制約として扱われる.

```
subject to one_color {i in NODE}: sum {j in COLOR} x[i,j] = 1;
```

すべての線形項の係数が 1 であり右辺値も 1 であるため, 制約 (1.1) と判断され値集合 COLOR をもつ変数 X_{i} (i $\in$ NODE) が定義される.
ここまでがモデルの記述である.

```
data;
param num_node := 6;
set EDGE := (0,1) (0,2) (1,2) (1,3) (1,4) (2,3) (2,4) (3,5) (4,5);
set COLOR := red yellow blue;
param: cost :=
        red     1
        yellow  2
        blue    3;
```

データの入力.

```
option solver wcspampl, wcspampl_auxfiles rc;
option wcsp_options 'target=10';
```

wcspampl をソルバに指定し, オプションを与える. オプションとしては target=n の他, adjust, display=n, interval=n, iteration=n, noadjust, seed=n, time=n が使用可能.

```
solve;
print "Solution";
printf {i in NODE, j in COLOR: x[i,j] = 1} "node %d: %s\n", i, j;
```

ソルバを呼び出し, 解を出力.

以下に, 実行結果を示す (wcspampl にパスが通るよう環境変数 PATH を設定する必要がある).

```
% ampl example_gcp.amp
ILOG AMPL 9.000, licensed to "xxx-xxx".
AMPL Version 20021031 (Linux 2.4.18-14)
WCSP 0.4: target=10
penalty = 1/11 (hard/soft), time = 0.00(s), iteration = 0
# improving the initial solution greedily
penalty = 0/13 (hard/soft), time = 0.00(s), iteration = 0
# start tabu search
penalty = 0/11 (hard/soft), time = 0.00(s), iteration = 9
penalty = 0/10 (hard/soft), time = 0.00(s), iteration = 29



# penalty = 0/10 (hard/soft)
# cpu time = 0.00/0.00(s)
# iteration = 29/29
penalty: 0/10 (hard/soft)

violated constraints
total_cost (soft): 10

Wcsp terminates.

Solution
node 0: red
node 1: blue
node 2: yellow
node 3: red
node 4: red
node 5: yellow
```

なお, 上の例で目的関数 (コスト) がない場合, 問題は, 彩色条件に違反した枝数 (両端点が同色である枝数) の最小化になる. これを 0-1 整数計画問題に定式化しようとすると, 枝が違反しているかどうかを表す 0-1 変数を新たに導

入する必要がある. このように AMPL + WCSP ソルバを用いることで, 0-1 整数計画問題と比べ, 問題をより簡潔に記述できる場合がある.

関連図書

[1] R. Fourer, D.M. Gay and B.W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, Duxbury Press, 2002.

[2] D.M. Gay, "Hooking Your Solver to AMPL," Technical report, Bell Laboratories, Murray Hill, NJ, 1993; revised 1994, 1997.

[3] K. Nonobe and T. Ibaraki, "An improved tabu search method for the weighted constraint satisfaction problem," INFOR, 39, pp.131-151, 2001.