メタヒューリスティクスアルゴリズムのための計算グラフを用いた関数評価の効率的実装

田 辺 隆 人†

離散変数による数理計画問題をメタヒューリスティクスによるアルゴリズムで解く場合，近傍探索と呼ばれる過程のために，変数のコンポーネントの一部を変化させた場合の関数（目的関数・制約式）の値の変化分の計算が必要となる．本稿では関数が計算グラフの形で表現されている場合，すなわち要素的な関数の代入の連鎖によって表現されている場合に，変数の変動に対する関数の変動を効率的に算出するアルゴリズムについて述べる．典型的な組合せ問題に対する計算機実験の結果，関数をまず評価して差をとるナイーブな方法に比べ最大 167 倍の高速化がなされることが示された．

An Efficient Implementation of Computational Graph Based Function Evaluation for Meta-heuristics Algorithms

Takahito Tanabe†

Meta-heuristic based solvers of mathematical programming problems with discrete variables will perform so-called “neighborhood search”. In the process, the evaluation of the difference of constraints/objective when changing a few (typically one) of variables, is required. In this paper, we present an efficient algorithm to evaluate the difference when the constraints/objective are expressed as a computational graph, in other words, a sequence of substitution of the result of elementary operations. A numerical experiment on some typical combinatorial problems shows that our algorithm compute the difference by order of magnitude (up to 167 times) faster than naive algorithm that evaluate the constraints/objective first and take the difference afterward.

1. はじめに

n 個の変数と m 個の制約式からなる数理計画問題：

$$\begin{aligned} &\text{最小化 } g_0(x_1,\cdots,x_n)\\ &\text{制約 } g_i(x_1,\cdots,x_n)=0,\ i=1,\cdots,m\\ &x_j\in W_j,\ j=1,\cdots,n \end{aligned} \tag{1}$$

を何らかのアルゴリズムを用いて解く処理系の実装を考える．W_j は x_j の範囲を示す集合である．設計としては，単体法や内点法などの数理計画法アルゴリズムを実行して $x=(x_1,\cdots,x_n)$ を決定する部分と目的関数と制約式（総称して関数と呼ぶ）：

$$g(x)=\{g_0(x),g_1(x),\cdots,g_m(x)\} \tag{2}$$

の情報を蓄えて評価を行う部分を分離するのが適切である．ここで便宜上前者を「ソルバ」，後者を「モデラ」と呼ぶ．モデラが x を受けて $g(x)$ の値や必要ならば微係数を返すというインタフェースを提供すれば，ソルバは $g(x)$ の表現の詳細に立ち入らなくてもアルゴリズムの実行が可能である．現に両者は独立したソフトウエアコンポーネントとして実装されており，線形計画法の実務においては，マトリクスジェネレータと呼ばれる形のモデラが古くより利用されてきた．近年では，より定式に近く直観的な定式記述が可能で，データと定式情報の分離が可能なモデリング言語処理系がモデラとして広く利用されている．モデリング言語処理系が数理計画法アルゴリズムの実行に必要なインタフェースを実装していれば，ユーザは用途に応じてモデラとソルバの組合せを自在に試すことができる．

本稿では，離散変数に対する数理計画問題，すなわち上記の W_j が値の離散集合

$$W_j=\left\{a_{j1},a_{j2},\cdots,a_{jn_j}\right\},\quad n_j=|W_j| \tag{3}$$

である問題にメタヒューリスティクスによるソルバを適用する際のモデラの実装について述べる．

メタヒューリスティクスによるソルバ，たとえば wcsp [1),2)] は，ある解 x が与えられたとき，その解が与える関数値と，その解の近傍 $N(x)$ における関数値の差

† 株式会社数理システム
Mathematical Systems Inc.

$$\Delta g(x, x^{'}) = g(x^{'}) - g(x) \quad x^{'} \in N(x) \tag{4}$$

を要求する．近傍 $N(x)$ は現在の解 x からコンポーネントの 1 つを変化させて得られる解の集合の全体で，$V \equiv \{1, \cdots, n\}$ を変数の添字の集合とすると，

$$N(x) \equiv \bigcup_{j \in V} N_j(x)$$
$$N_j(x) \equiv \{x^{'} | x^{'}_j \neq x_j, x^{'}_{j'} = x_{j'}, j' \in V \backslash \{j\}\} \tag{5}$$

のように定義される．

式 (4) の計算はモデラが解 x を受け取って関数値 $g(x)$ を与えるインタフェースを用意することにより実現可能であるが，近傍の大きさは

$$|N(x)| = \sum_{j \in V} (n_j - 1) \tag{6}$$

であるので，解 x を更新するたびにモデラが関数 $g(x)$ の各コンポーネントの計算を $\sum_{j \in V} (n_j - 1)$ 回行うこととなり，特に大規模問題においてはここが計算上のボトルネックとなることは明らかである．

ここでは，モデラが関数を計算グラフ[3] の形式で保持しているとき，すなわち関数を要素的な関数の代入の連鎖として表現しているときに $g(x)$ を介さず直接 $\Delta g(x, x')$ を求めるアルゴリズムとその効率化について述べる．自動微分アルゴリズムの一部は計算グラフ上の操作として記述されるため，計算グラフによる内部表現は非線形計画法に基づくソルバに微係数を提供する機能を持つモデラに広く用いられている．

ここで紹介するアルゴリズムはその同じ内部表現を用いてメタヒューリスティクスによるソルバ向けの効率的なインタフェースの実装方法を与えるものである．

2. 差分評価アルゴリズム

以降では関数 $g(x)$ の 1 つのコンポーネントを $f(x) = f(x_1, \cdots, x_n) \in \mathbf{R}$ とし，解 x を前章で定義した近傍 $N(x)$ に移動させた場合の変化分

$$\Delta f(x, x^{'}) = f(x^{'}) - f(x),\ x^{'} \in N(x) \tag{7}$$

の計算について考察する．ここで，$f(x)$ は計算グラフ形式で表現されているとする．具体的には $n+p+1$ 個の中間変数（最初の n 個は変数，最後の 1 個は関数 $f(x)$ に相当）$v_k, k \in \{1, \cdots, n+p+1\}$ と要素的な関数 $\varphi_k(v_l)_{l \in P_k}$ を用いて次のような連鎖的な代入関係によって表されているものとする☆．

$$\begin{aligned} v_k &= x_k, k \in V \\ v_k &= \varphi_k(v_l)_{l \in P_k}, \\ &\quad k \in \{n+1, \cdots, n+p+1\} \\ v_{n+p+1} &= f(x) \end{aligned} \tag{8}$$

要素的関数 $\varphi_k(v_l)_{l \in P_k}$ は $|P_k|$ の長さの引数ベクトルを入力とする．P_k は v_k が依存している変数・中間変数の添字，すなわち対応する要素的関数 $\varphi_k(v_l)_{l \in P_k}$ の引数の添字の集合であり，定義より

$$P_k \subseteq \{1, \cdots, n+p\} \tag{9}$$

である．要素的関数の具体例としては和，差演算を一般化した多項の線形演算

$$\varphi_k(v_l)_{l \in P_k} = \sum_{l \in P_k} c_{kl} v_l \quad c_{kl} : \text{定数} \tag{10}$$

あるいは $|P_k| = 2$ の例として

$$\varphi_k(v_l)_{l \in \{l_1, l_2\}} = v_{l_1} @ v_{l_2} \quad @ = \{\times, /, pow \cdots\} \tag{11}$$

さらに $|P_k| = 1$ であるような一般の関数

$$\varphi_k(v_l)_{l \in \{l_1\}} = \{\log, \exp, \text{bool}, \cdots\}(v_{l_1}) \tag{12}$$

などがある．ここで bool は離散値を変数とする最適化問題に頻出する関数で，引数と，与えられた定数に関する条件が満たされていたら 1，満たされていなかったら 0 を返す関数である．

さらに各要素的関数の値を格納する $p+1$ 個の中間変数 $v_{n+1}, v_{n+2}, \cdots, v_{n+p+1}$ の番号付けは依存関係の順に行われていることを仮定する．すなわち

$$l \in P_k, k \in \{n+1, \cdots, n+p+1\} \quad \text{ならば } k > l \tag{13}$$

とする．このとき関数 f の評価は v_k に対応する作業領域を用意して次のような手順で行うことができる．

Algorithm EVAL(f, x):
1. $v_k \leftarrow x_k \quad k = 1, \cdots, n$
2. $v_k \leftarrow \varphi(v_l)_{l \in P_k} \quad k = n+1, \cdots, n+p+1$
3. $f \leftarrow v_{n+p+1}$

一方，手続き EVAL によって各 v_k が設定されていると仮定し，変数 x_j を Δx_j だけ変化させたときの関数 f の変化分 Δf は以下のような手続きで計算される．

Algorithm DIFF_EVAL$(f, j, \Delta x_j)$:
1. $\Delta v_k \leftarrow 0 \quad k = 1, \cdots, n+p+1$
2. $\Delta v_j \leftarrow \Delta x_j$
3. $\Delta v_k \leftarrow \Delta \varphi_k(v_l, \Delta v_l)_{l \in P_k}\ k \in U(\{j\}) \backslash \{j\}$

☆ 計算グラフ表現では v_k が節点に対応し，P_k の要素に対応する節点と v_k に対応する節点とが有向枝で結ばれる．

4. $\Delta f \leftarrow \Delta v_{n+p+1}$

ここで，Δv_k は v_k に 1 対 1 対応する作業領域とし，要素的関数の変化分

$$\Delta\varphi_k(v_l, \Delta v_l)_{l\in P_k} = \varphi_k(v_l + \Delta v_l)_{l\in P_k} - \varphi_k(v_l)_{l\in P_k} \quad (14)$$

を蓄える．ステップ 3 は k の番号の若い順に行われることを仮定している☆．集合 $U(T)$ は，集合 T が変数，中間変数，関数の添え字からなるとき，それらに依存する中間変数，関数の添え字の集合であり，具体的には次のように再帰的に定義される．

$$i)\ T \subseteq U(T)$$
$$ii)\ \exists l \in P_k, l \in U(T) \text{ ならば } k \in U(T) \quad (15)$$

$U(T)$ の定義より DIFF_EVAL のステップ 3 は変数 x_j に無関係な中間変数については行われない．

集合 S，T，$U(S)$，$U(T)$ については一般に

$$T \supseteq S \rightarrow U(T) \supseteq U(S) \quad (16)$$
$$U(T) \supseteq S \rightarrow U(T) \supseteq U(S) \quad (17)$$

が成り立つ．また式 (16) と (17) より

$$T \subseteq S, U(T) \supseteq S \rightarrow U(T) = U(S) \quad (18)$$

がいえる．

EVAL における要素的関数の評価回数は $p+1$ 回と固定であるのに対して，DIFF_EVAL の評価回数は $\Delta\varphi_k(v_l, \Delta v_l)_{l\in P_k}$ の計算に必要な評価を 1 回と数えると（DIFF_EVAL の前提条件として $\varphi_k(v_l)_{l\in P_k}$ は v_k に蓄えられていることより関数評価は $\varphi_k(v_l + \Delta v_l)_{l\in P_k}$ の 1 回のみ必要である），$|U(\{j\})\backslash\{j\}|$ 回．一般に

$$|U(\{j\})\backslash\{j\}| \leq p+1 \quad (19)$$

であることから，DIFF_EVAL は EVAL に比べ高速化できる可能性がある．次章では関数の構造や変数が離散変数であること，さらには要素的関数の性質を用いて，DIFF_EVAL の要素的関数の評価回数を削減する方策について述べる．

3. 差分評価アルゴリズムの効率化

3.1 構造の利用

すべての変数 x_j，$j \in V$ は離散変数である．その値を表現するために $D_j \equiv \{1, \cdots, n_j\}$ と定義し，次を満たすような 0-1 変数 x_{js} を導入する．

$$\sum_{s\in D_j} x_{js} = 1, \quad j \in V \quad (20)$$

このとき x_j は式 (3) で定義されている W_j の要素 a_{js} を用いて次のように表すことができる．

$$x_j = \sum_{s\in D_j} a_{js} x_{js} \quad (21)$$

x_{js} は x_j に対応する値変数[1),2)] と呼ばれる．値変数 x_{js} は x_j が値 a_{js} をとるときに 1 になる．

中間変数 v_k が単一の変数 x_j のみに依存するならば v_k も x_j の値変数によって

$$v_k = \sum_{s\in D_j} a_{ks} x_{js} \quad (22)$$

と表すことができる．たとえば変数 $x(=v_1) \in \{1,2,3\}$ に関して，中間変数 v_2，v_3，v_4 と関数 $f(=v_5)$ が

$$\begin{aligned} v_2 &= \mathrm{bool}(v_1 = 3) \\ v_3 &= \mathrm{bool}(v_1 = 1) \\ v_4 &= v_2 + v_3 \\ v_5 &= \mathrm{bool}(v_4 \leq 1) \end{aligned} \quad (23)$$

のように表現されているとき，変数 x の値変数を x_1，x_2，x_3 とすると

$$v_1 = x_1 + 2x_2 + 3x_3 \quad (24)$$

さらに，v_2，v_3，v_4 も x のみに依存していることから x の値変数である x_1，x_3 を用いて

$$v_2 = x_3,\ v_3 = x_1,\ v_4 = x_1 + x_3 \quad (25)$$

と表すことができる．特に $f(=v_5)$ が直接依存している v_4 が $x(=v_1)$，v_2，v_3 を介さず，値変数 x_1，x_3 から直接計算できることに注意する．たとえば変数 x が 2 から 3 に変動した際には変数 x の値変数 (x_1, x_2, x_3) が $(0,1,0) \rightarrow (0,0,1)$ と変化していることから，$(\Delta x_1, \Delta x_2, \Delta x_3) = (0,-1,1)$ より

$$\Delta v_4 = \Delta x_1 + \Delta x_3 = 0 + 1 = 1 \quad (26)$$

と計算できる．この場合の $f(=v_5)$ を求める手続きを DIFF_EVAL の各ステップの操作に対応させて記述すると以下のようになる．

1. $\Delta v_k \leftarrow 0 \quad k = 1, \cdots, 5$
2. $\Delta v_4 \leftarrow 1\ (= \Delta x_1 + \Delta x_3)$
3. $\Delta v_5 \leftarrow \Delta\varphi_5(v_4, \Delta v_4)$
4. $\Delta f \leftarrow \Delta v_5$ (27)

Δv_1，Δv_2，Δv_3 は他の中間変数・関数の値に寄与していないため，ステップ 1 によって 0 と設定されているのみで，元になる変化分の設定を行うステップ 2 と中間変数の変化分の計算であるステップ 3 からは除外されている．

単一の変数 j にのみ依存している中間変数・関数の添字の集合を E_j とする．E_j は次により再帰的に定義される．

$$i)\ \{j\} \subseteq E_j$$
$$ii)\ \forall l \in P_k, l \in E_j \text{ならば } k \in E_j \quad (28)$$

このとき，v_k は定数 a^j_{ks}（$j \in V, k \in E_j, s \in D_j$）を

☆ 以降現れる DIFF_EVAL の改良手続きに関しても同様と約束する．

用いて

$$v_k = \sum_{s\in D_j} a^j_{ks} x_{js}, \quad k \in E_j \tag{29}$$

と表される．a^j_{ks}（$j \in V, k \in E_j, s \in D_j$）は変数 x_j が $s \in D_j$ に対応する値をとったときの v_k の値であり，式 (21) および (22) に表れる定数から計算される．ここから v_k の変化分 Δv_k は x_j の値変数 x_{js} の変化分 Δx_{js} により記述される．したがって，E_j に属する添字に対応する変数・中間変数の中で，その値が寄与している中間変数の添字がすべて E_j に属するような中間変数 v_k については Δv_k の計算は不要である．このような中間変数の添字の集合を E_j から除いた E_j の部分集合 ∂E_j を次のように定義する．

$$\partial E_j \equiv \{k \in E_j | \exists l \notin E_j, k \in P_l\} \tag{30}$$

∂E_j を用いて，x_j が値 $a_{js_{out}}$ から $a_{js_{in}}$ に変化した際の関数値の変動を求める次のようなアルゴリズムを導くことができる．

Algorithm DIFF_EVAL*(f, j, s_{in}, s_{out})：

1. $\Delta v_k \leftarrow 0 \quad k = 1, \cdots, n+p+1$
2. $\Delta v_k \leftarrow a^j_{ks_{in}} - a^j_{ks_{out}} \quad k \in \partial E_j$
3. $\Delta v_k \leftarrow \Delta\varphi_k(v_l, \Delta v_l)_{l\in P_k} \; k \in U(\partial E_j)\backslash\partial E_j$
4. $\Delta f \leftarrow \Delta v_{n+p+1}$

ステップ 2 の右辺は式 (29) に現れる係数の差である．E_j の定義より，$\{j\} \subseteq E_j$ かつ $U(\{j\}) \supseteq E_j$ であることから式 (18) より

$$U(\{j\}) = U(E_j) \tag{31}$$

ここで，$\hat{E}_j \equiv E_j\backslash\partial E_j$ と定義すると E_j，∂E_j の定義より

$$\begin{aligned} U(E_j) &= U(\partial E_j \oplus \hat{E}_j) \\ &= U(\partial E_j) \oplus \hat{E}_j \\ &= \{U(\partial E_j)\backslash\partial E_j\} \oplus \partial E_j \oplus \hat{E}_j \\ &= \{U(\partial E_j)\backslash\partial E_j\} \oplus E_j \end{aligned} \tag{32}$$

さらに $U(\{j\})$，E_j がともに $\{j\}$ を含むことと，式 (31)，(32) より

$$U(\{j\})\backslash\{j\} = \{U(\partial E_j)\backslash\partial E_j\} \oplus (E_j\backslash\{j\}) \tag{33}$$

ここから DIFF_EVAL および DIFF_EVAL*のステップ 3 の要素的関数評価回数について

$$|U(\{j\})\backslash\{j\}| - |U(\partial E_j)\backslash\partial E_j| = |E_j| - 1 \tag{34}$$

がいえ，$|E_j|$ が大きいほど，DIFF_EVAL* は DIFF_EVAL に比べて効率化されることが分かる．$k \in \hat{E}_j$ なる v_k の変化分 Δv_k については $\Delta v_k = 0$ とのみ設定され，計算からは除外されている．

3.2 線形演算子の特性の利用

数理計画問題の記述には $\varphi_k(v_l)_{l\in P_k}$ として多項の線形演算

$$\varphi_k(v_l)_{l\in P_k} = \sum_{l\in P_k} c_{kl} v_l$$
$$c_{kl}：定数 \tag{35}$$

が頻出するが，そのとき

$$\Delta\varphi_k(v_l, \Delta v_l)_{l\in P_k} = \varphi_k(\Delta v_l)_{l\in P_k} \tag{36}$$

すなわち演算結果の変動が，引数ベクトルの変化分にのみ依存する．このことから DIFF_EVAL* (j, s_{in}, s_{out}) において $k \in E_j$ であるような v_k の線形関数と見なすことのできる中間変数・関数の変化分は，x_j の値変数の変化分によって表現することができる．たとえば変数 $x(= v_1) \in \{1, 2, 3\}$，$y(= v_2) \in \{1, 2, 3\}$，x，y が値変数 $\{x_1, x_2, x_3\}$，$\{y_1, y_2, y_3\}$ を用いて，$x = x_1 + 2x_2 + 3x_3$，$y = y_1 + 2y_2 + 3y_3$ と表現されているとき，中間変数 v_3，v_4，v_5 と関数 $f(= v_5)$ が

$$\begin{aligned} v_3 &= \mathrm{bool}(v_1 = 2) \\ v_4 &= \mathrm{bool}(v_2 = 1) \\ v_5 &= 5v_3 + 7v_4 \end{aligned} \tag{37}$$

と定義されているのならば，

$$\Delta v_5 = 5\Delta v_3 + 7\Delta v_4 = 5\Delta x_2 + 7\Delta y_1 \tag{38}$$

と表せる．前項で定義した E_j を拡張した C_j を次のように定義する．

$$\begin{aligned} &i)\ E_j \subseteq C_j \\ &ii)\ k \in U(\{j\}) \text{で} \varphi_k \text{が線形，かつ} \\ &\quad \forall l \in P_k \cap U(\{j\}) \text{について} l \in C_j \\ &\quad \text{ならば} k \in C_j \end{aligned} \tag{39}$$

C_j は E_j を含み $k \in E_j$ である v_k に関して線形な中間変数・関数の添字を含む集合である．C_j の要素を添え字として持つ中間変数 v_k は次のように表すことができる．

$$\begin{aligned} v_k = &\sum_{l\in P_k\cap U(\{j\})} c_{kl} v_l \\ &+ \sum_{l'\in P_k\backslash\{P_k\cap U(\{j\})\}} c_{kl'} v_{l'}, \quad k \in C_j \end{aligned} \tag{40}$$

2 項目は x_j に依存しない（$k \in E_j$ ならば 2 項目全体が存在しない）ことから変数 x_j の変化にともなう v_k の変化 Δv_k は

$$\Delta v_k = \sum_{l\in P_k\cap U(\{j\})} c_{kl} \cdot \Delta v_l, \quad k \in C_j \tag{41}$$

と記述できる．

右辺の和がとられている項の添字 l はすべて C_j に含まれること，および，$k \in E_j$ ならば式 (29) が成り立つことを用いれば a^j_{ks}（$k \in C_j, s \in D_j$）を定数として

$$\Delta v_k = \sum_{s\in D_j} a^j_{ks}\Delta x_{js},\quad k \in C_j \tag{42}$$

がいえる．a^j_{ks} は $k \in E_j$ ならば，式 (29) の右辺の係数と一致し，$k \in C_j \backslash E_j$ ならば式 (29) の右辺の係数と式 (41) の右辺の係数 c_{kl} を用いて計算される．

すなわち $k \in C_j$ ならば v_k の変化分 Δv_k は x_j の値変数である x_{js} の変化分 Δx_{js} により記述される．したがって，添え字が C_j に属する中間変数にしか寄与していない中間変数の変化分の計算は不要になるので，

$$\partial C_j \equiv \{k \in C_j | \exists l \notin C_j, k \in P_l\} \tag{43}$$

なる集合を定義して DIFF_EVAL*(f, j, s_{in}, s_{out}) の集合 ∂E_j を ∂C_j で置き換えた手続き DIFF_EVAL** (f, j, s_{in}, s_{out}) が構成できる．C_j の定義より，$E_j \subseteq C_j$，$C_j \subseteq U(\{j\}) = U(E_j)$ であることから式 (18) より

$$U(C_j) = U(E_j) = U(\{j\}) \tag{44}$$

さらに，式 (32) の E_j，∂E_j を C_j，∂C_j で置き換えた

$$U(C_j) = \{U(\partial C_j)\backslash \partial C_j\} \oplus C_j \tag{45}$$

がいえる．式 (44)，(45) の両辺から集合 $\{j\}$ を除いた式より

$$|U(\{j\})\backslash\{j\}| - |U(\partial C_j)\backslash\partial C_j| = |C_j| - 1 \tag{46}$$

この式と (34) および $E_j \subseteq C_j$ より DIFF_EVAL*と DIFF_EVAL**のステップ 3 の演算回数について

$$|U(\partial E_j)\backslash\partial E_j| - |U(\partial C_j)\backslash\partial C_j| \geq |C_j| - |E_j| \tag{47}$$

がいえる．すなわち式の線形性が高く，C_j が E_j に比べて大きいほど DIFF_EVAL**は DIFF_EVAL*よりも効率的であることが分かる．$k \in C_j \backslash \partial C_j$ なる v_k について Δv_k の計算が不要なのは DIFF_EVAL*の場合と同様である．

特に bool 関数と線形関数の組合せのみから定義されている関数については，j を任意の変数の添え字とするとき，関数そのものを示す添字 $n+p+1$ が C_j に含まれ，$U(\partial C_j) = \phi$ となる．この場合には任意の変数 j の変動に対する関数値の変動が値変数の変動から求められる．このような関数の例として一般化割当て問題

$$\begin{aligned} &\text{変数}\quad x_j \in A, j \in J \\ &\text{最小化}\quad \sum_{i\in A, j\in J} c_{ij}\cdot \mathrm{bool}(x_j = i) \\ &\text{制約}\quad \sum_{j\in J} a_{ij}\cdot \mathrm{bool}(x_j = i) \leq b_i, i \in A \end{aligned} \tag{48}$$

の目的関数および制約式がある．

ここで，J は仕事の集合，A は仕事の割当て先の集合，x_j は仕事 $j \in J$ の割当先を示す変数，a_{ij}，c_{ij} はそれぞれ仕事 $j \in J$ を割当先 $i \in A$ に割り振ったときのリソース消費，コストを示しており，b_i は仕事の割当て先 $i \in A$ の容量を示している．

類似の構造はナーススケジューリング問題[4] など，多くの数理計画問題の記述に現れる．

3.3 値変数の中間変数の寄与情報の利用

式 (42) に現れる係数 a^j_{ks} $(j \in V, k \in \partial C_j, s \in D_j)$ はすべてが非零ではない．特に bool 演算が多用される問題の場合，一部の値変数にしか依存しない中間変数・関数が増大するので，0 になる要素は多い．今，変数 x_j の値変数 x_{js} の変動によって影響される中間変数・関数の添字の集合 $\partial C_{js} (\subseteq \partial C_j)$ を

$$\partial C_{js} \equiv \{k \in \partial C_j | a^j_{ks} \neq 0\} \tag{49}$$

と定義する．この集合を使うと以下の手続きが構成できる．

Algorithm DIFF_EVAL***(f, j, s_{in}, s_{out}):

1. $\Delta v_k \leftarrow 0 \quad k = 1, \cdots, n+p+1$
2. $\Delta v_k \leftarrow a^j_{ks_{in}} - a^j_{ks_{out}} \quad k \in \partial C_{js_{in}} \cup \partial C_{js_{out}}$
3. $\Delta v_k \leftarrow \Delta\varphi_k(v_l, \Delta v_l)_{l\in P_k}$
   $k \in U(\partial C_{js_{in}} \cup \partial C_{js_{out}}) \cap \{U(\partial C_j)\backslash\partial C_j\}$
4. $\Delta f \leftarrow \Delta v_{n+p+1}$

ステップ 3 では値変数 $x_{js_{in}}$，$x_{js_{out}}$ の変動によって影響を受ける中間変数・関数のみに対して評価を行うこととなり，a^j_{ks} の零要素が多いほど，DIFF_EVAL** に比べ高速化されることが期待できる．

4. 近傍探索における関数評価の実際

式 (4) を計算するにはモデルに現れる関数 $g_i(x)$，$i \in \{0, \cdots, m\}$ について，変数 x_j のコンポーネント j を Δx_j だけ動かしたときの変化分を計算することを繰り返せばよい．その変化分を $\Delta g_i(x; j, \Delta x_j)$ と表し，$s_c \in D_j$ は x_j の現在の値に対応する D_j の要素と定義すると，式 (4) の計算は各 $i \in \{0, \cdots, m\}$，$j \in V$ について，

$$\Delta g_i(x; j, a_{js} - a_{js_c}),\quad s \in \{1, \cdots, n_j\}\backslash\{s_c\} \tag{50}$$

を計算することに帰着できる．計算には式 (50) は DIFF_EVAL***(g_i, j, s, s_c) を用いることができる．DIFF_EVAL***を実行するために必要な集合データ ∂C_j，∂C_{js}，$U(\partial C_j)$，$U(\partial C_{js})$ $(j \in V, s \in D_j)$ および式 (42) に現れる係数データ a^j_{ks} $(j \in V, k \in \partial C_j, s \in D_j)$ は式の構造が明らかになった時点であらかじめ前処理において作成しておけばよい．∂C_j，∂C_{js}，$U(\partial C_j)$，$U(\partial C_{js})$ は，変数・中間変数・関数

の依存関係の情報があれば，$n+p$ に比例するオーダの計算時間と作業領域で求めることができる．

DIFF_EVAL***（f, j, s_{in}, s_{out}）を実行すると，作業領域 Δv_k に各変数・中間変数・関数の変化分が設定されるので，

$$v_k \leftarrow v_k + \Delta v_k \tag{51}$$

とすれば，変数 x のコンポーネント j のみを変化させた x' について EVAL（g_i, x'）を行ったのと等価となる．このことを利用すれば，EVAL の実行は最初の 1 回のみとし，以降はすべて DIFF_EVAL***のみで，EVAL 相当の計算を行うことができる．

5. 数値実験

数理計画法パッケージ NUOPT7[5] 組み込みのモデラ SIMPLE と，タブーサーチによる制約充足問題解法ソルバ wcsp の組合せに関して，差分評価アルゴリズム導入の効果を検証した．実験には 5 種類の離散最適化問題をベンチマークとして用いた．SIMPLE はユーザが記述した問題の定式を内部では計算グラフ形式，すなわち要素的関数の代入の連鎖として表現している．式 (4) を計算するにあたって，EVAL を用いてのみ行う場合（「導入前」）と DIFF_EVAL***を導入した場合を比較した．（**表 1**）は wcsp が一定時間（60 秒）以内に行うことができた反復回数を示す．実験には PentiumM 1.4 GHz，メモリ 1.5 GBytes の WindowsPC を用いた．DIFF_EVAL***導入後の計算時間には，DIFF_EVAL***の実行のためのデータ構造を作成する前処理時間を含む．本実験で用いた問題はいずれも前処理時間が最大 1 秒以下で終了するため，DIFF_EVAL***適用のための前処理のオーバヘッドは無視できる程度といえる．

なおベンチマーク問題 4 は式 (42) で WWW☆から取得したデータを用いた，1 は通常の n-Queen 問題に駒ができるだけ対角線上に並ぶという考慮制約を付加したもの，2 は巡回セールスマン問題，3，5 はそれぞれ時間割作成問題[6]，ナーススケジューリング問題[4] を一部簡略化したものである．導入後一定時間に行える反復回数は導入前と比べて大幅に増大しており，より多くの解空間を探索できていることが分かる．**表 2** は上記の実行において 10 万回あたりの EVAL と DIFF_EVAL***の平均時間☆☆を比較したものである．

式の表現によってその効果は異なるが EVAL の代わりに DIFF_EVAL***を用いることで，式 (4) の計算は最大 167 倍程度高速化されていることが分かる．

表 1 60 秒間に可能な wcsp の反復回数
Table 1 Number of wcsp iterations available in 60 sec.

問題	#変数	#制約	導入前	導入後
1. n-Queen	100	5,050	333	4,110
2. TSP	52	2	1,324	19,624
3. 時間割	30	844	1,426	44,649
4. 一般化割当	100	6	1,349	233,883
5. ナース	750	11,178	452	160,077

表 2 関数評価時間平均（10 万回あたり）
Table 2 Average function evaluation time (per 10^5 calls).

問題	EVAL	DIFF_EVAL	比率
1. n-Queen	0.465 秒	0.074 秒	6.3
2. TSP	1.499 秒	0.102 秒	14.6
3. 時間割	0.373 秒	0.013 秒	27.7
4. 一般化割当	2.601 秒	0.017 秒	151.6
5. ナース	3.310 秒	0.020 秒	167.2

6. おわりに

連続変数による非線形数理計画法を内点法や逐次二次計画法などを用いて解く場合，モデラ側の計算時間の多くは微係数の計算に費やされ，それに比べて関数評価の時間にはほとんど時間を消費しない．一方，離散変数による最適化問題をメタヒューリスティクスに基づくアルゴリズムによって解く場合には，微係数を必要としないかわりに（もともと関数の微分可能性を仮定していない），現在の解の近傍における関数値の変化分を必要とするため，関数評価の計算がボトルネックとなる．ここでは，関数の変化分を直接求める算法を用いることにより，効率の向上が図れることを実際に示した．

ここで述べた算法 DIFF_EVAL は自動微分法[3] における forward mode による一階微係数の計算と似通っており，離散変数によって記述される系における微係数計算に相当するものであるという見方も成り立つ．関数を計算グラフとして表現して自動微分法を実装しているモデラならば，DIFF_EVAL を用いて，メタヒューリスティクスに基づくソルバ向けの効率的なインタフェースを実装することが可能であり，離散変数に基づく数理計画問題へとその適用範囲を広げることが可能である．

ここでは，近傍を現在の解のコンポーネント 1 つを変動させたところと定義したが，より一般的な近傍においても関数の変化分が計算できるように DIFF_EVAL を拡張することは容易である．

謝辞 制約充足問題解法エンジンの内部の構造に関

☆ http://people.brunel.ac.uk/~mastjjb/jeb/orlib/gapinfo.html

☆☆ 問題に現れるすべての関数（制約式，目的関数）の評価時間を合算しての平均である．

する情報をいただきました法政大学工学部システムデザイン学科の野々部宏司先生，ならびに初校の誤りに関して丁寧なご指摘をいただきましたレフェリーの方々にこの場を借りまして感謝申し上げます．

参 考 文 献

1) Nonobe, K. and Ibaraki, T.: A tabu search approach for the constraint satisfaction problem as a general problem solver, *European Journal of Operational Research*, Vol.106, pp.599–623 (1998).
2) Nonobe, K. and Ibaraki, T.: An improved tabu search method for the weighted constraint satisfaction problem, *INFOR*, Vol.39, pp.131–151 (2001).
3) Griewank, A.: Evaluating Derivatives, *SIAM* (2000).
4) Ikegami, A. and Niwa, A.: A Subproblem-centric Model and Approach to the Nurse Scheduling Problem, *Mathematical Programming*, Vol.97, No.3, pp.571–541 (2003).
5) (株) 数理システム：NUOPT マニュアル (2004).
6) Miyazaki, S., Iwama, K. and Kambayashi, Y.: Database Queries as Combinatorial Optimization Problems, *Proc. International Symposium on Cooperative Database Systems for Advanced Applications* (*CODAS'96*), pp.448–545 (1996).

(平成 17 年 1 月 24 日受付)
(平成 17 年 4 月 26 日採録)

田辺 隆人

昭和 42 年生．平成 2 年京都大学理学部卒業．同年（株）数理システム入社，現在数理計画室室長．大規模線形演算，数理計画法パッケージ，モデリング言語処理系の研究開発に従事．平成 14 年より法政大学工学部非常勤講師．平成 17 年中央大学大学院理工学研究科情報工学専攻博士課程修了．OR 学会会員．工学博士．