生成順序を保存するコピー方式ガーベジコレクションについて

小出 洋† 野下浩平†

大容量の主記憶をもつ計算機における言語処理系では，コピー方式のガーベジコレクションアルゴリズム(GC)がよく使われている．コピー方式GCは，その実行時間が使用中オブジェクトの総容量に依存する時間で済み，記憶領域全体の大きさに依存しないため，実行時間が記憶領域の総容量に比例するスライディング圧縮方式などと比較して，相対的に速い．しかし，オブジェクトが生成された順序（生成順序）と，コピーされる順序が関係しない．本論文では，コピー方式GCに生成順序を保存する特性をもたせる方法を提示する．まず，生成順序を保存する基本的なアルゴリズム（各オブジェクトにマークビットをもつもの）を示し，次に，マークビットを持たない方法を示す．最初のものは，マークビットをコピー先領域にとる方法であり，2番目のものは，逆ポインタによりマークビットを省略する方法である．このアルゴリズムによって，コピー方式GCにおいて世代別の分類が自然に実現でき，スライディング圧縮方式で使うような効率化の方法も適用可能になる．さらに，オブジェクトの生成順序を保存すると都合の良い処理系のGCにも応用できる．

On the Copying Garbage Collector which Preserves the Generated Order

Hiroshi Koide† and Kohei Noshita†

A new algorithm for the copying garbage collection is presented, in which the generated order of objects in the heap space is always preserved. The basic idea is to sort the set of all pointers to active objects in the destination semispace. The amount of time of this sorting is relatively small if we use a fast sorting algorithm for the set of fixed-sized integers. The basic algorithm is further improved to eliminate the explicit marking bits. A simple bit-table suffices for this purpose, but it requires the cost proportional to the size of the heap space. Another method is presented, which replaces the marking bit with a pair of mutual pointers for each object. The cost of this method depends only on the number of active objects. Some applications of our algorithm are briefly discussed.

1. はじめに

コピー方式のガーベジコレクション（ゴミ集め，GCと略す）は，大容量の主記憶をもつ計算機上でよく用いられる記憶管理の方式である[3),4),9)]．コピー方式GCは，オブジェクト（レコード）の大きさが一定の固定容量の場合にも可変容量の場合にも用いられるが，本論文の方法は，いずれの場合にも適用できる．

コピー方式の概要は次のとおりである．オブジェクトを割り付ける記憶領域（heap）を2つの半領域(semispace)に分け，いつも片方の半領域で記憶領域を消費し，GCが起動されるとリスト構造をたどって使用中オブジェクト（ゴミではないオブジェクト）をもう一方の半領域であるコピー先領域にコピーする．コピーされた使用中オブジェクトはそのコピーを指すポインタで置き換えておく．このポインタは転送先ポインタ（forwarding pointer）と呼ばれる．すべてのオブジェクトをコピーした後で，オブジェクトの中の転送先ポインタを参照して，必要ならばコピー先領域のオブジェクトへのポインタに修正する．こうして，すべての使用中オブジェクトをコピー先領域の連続した領域にコピーする．この時点で2つの半領域の役割を入れ換え，GCが完了する．

コピー方式GCの計算時間は使用中のオブジェクトの総容量に比例し，それと対照的な他の方式であるスライディング圧縮方式[3),7),9)]などとは違い，記憶領域の総容量に関係しない．したがって，特に大容量の主記憶を用いるような応用ではGCに要する時間が相対的に少ない．また，最近ではLISP系言語の処理系のためのGCの方式として注目を集めているジェネレーショナル（generational）GCも部分的にコピー方式を使用している[2),5),8),9)]．

ところで，従来のコピー方式GCはコピーを行うとき，リスト構造をたどり，オブジェクトをコピーしていくため，そのたどり方が，例えば，先行順序（pre-

† 電気通信大学電気通信学部情報工学科
Department of Computer Science, The University of Electro-Communications

order）であっても，その他の順序であっても，もともと生成された順序（生成順序，generated order[3),7)]）に並んでいたオブジェクトは並べ換えられる．すなわち，一般にはオブジェクトの生成順序は保存されない．

本論文は，コピー方式GCでオブジェクトの生成順序を保存する方法を提示し，その基本的な性質を考察するものである．生成順序を保存する方法は，関数型言語のある種の処理系（例えば文献6)）のように生成順序の保存を前提にするものに直接的に応用できる．また，ジェネレーショナルGC[2),9)]で行われているオブジェクトの世代別の分類は生成順序を保存することによって自然に実現できる．さらにこの分類の応用例として，スライディング圧縮方式で考案された生成順序を利用した最適化技法（例えば文献7)）がコピー方式にも利用できる．

次章では，コピー方式GCに生成順序を保存する特性をもたせる方法について，まず基本となるアルゴリズムを提示する．このアルゴリズムの計算量については，使用中オブジェクトの総容量Sに比例する時間（通常のコピー方式GCの時間と同じ）と，n個の整数（ポインタデータ）の列の整列（ソート）のための時間の和になる．ここでnは使用中オブジェクトの個数である．

さて，この基本的な方法では，通常のコピー方式で不要である各オブジェクトについてのマークビット（印付け用ビット）が必要になる．そこで4章では，通常のコピー方式と同様に各オブジェクトにマークビットをもたない方法を考案する．最初のものは，マークビットをコピー先領域にとる方法である．この方法では，コピー先領域のマークビットを最初にすべて初期設定しなくてはならないし，（普通のやり方では）記憶領域全体の語数に比例するビットを調べる必要がある．そのため，GCの計算時間は，記憶領域の総容量に比例するものになる．2番目の方法は，逆ポインタを用いてマークビットは全く用いない方法である．この部分の計算時間は，Sに関係せず，使用中のオブジェクトの個数nに比例する．

計算量の見地から見ると，nがSと比較して大きくなると整列の手間が相対的に大きくなる．nがSと比較して十分小さい例としては，オブジェクト指向言語のオブジェクト，配列，多倍長数，文字列，クロージャなどをもつ言語処理系があげられる．

2. 生成順序を保存するコピー方式 GC

まず生成順序を保存する基本的なアルゴリズムについて説明し，その計算量について考察する．**図1**に記憶領域全体の様子を描く．図では，上側が下位アドレス，下側が上位アドレスとする．ここで，言語処理系は，2つの半領域のうち，コピー元領域の連続した自由領域を最下位アドレスから順に領域を確保して，オブジェクトを割り付けていくものとする．また処理系のデータ構造は，ポインタタグ方式（オブジェクトを指す側でそのオブジェクトの型をもつ方式）であり，オブジェクトのポインタは1語に格納できるものとする．さらに，本章のアルゴリズムでは各オブジェクト1語につきマークビットが1ビット使用できるものとする．

基本となるアルゴリズムの概略は次のとおりになる．

段階1 言語処理系のレジスタで示されるオブジェクトから順にリスト構造をたどって，使用中オブジェクトすべてをマーク付けする（traverse）．マーク付けする際，コピー先領域の最上位アドレスから最下位アドレスに向かって順に，マークする使用中オブジェクトへのポインタを入れていく．この場所をA領域と呼ぶ．

段階2 A領域にあるコピー先領域の使用中オブジェクトへのポインタの集まりを（整数値データとみ

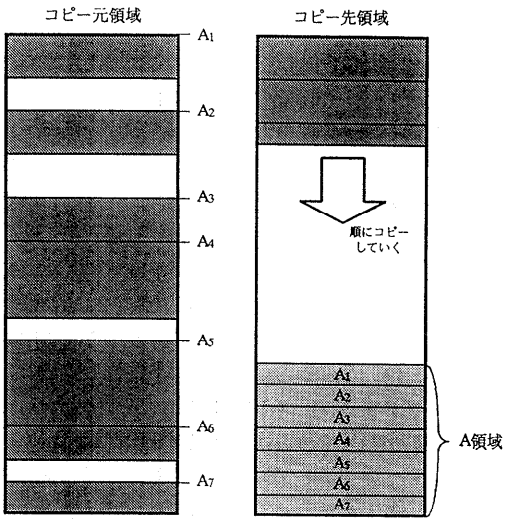


図1 基本的アルゴリズムにおける記憶領域割り当て（A領域の整列の直後）
Fig. 1 Memory allocation for the basic algorithm (immediately after sorting the A-area).

なして）上昇順に整列する（sort）.

段階3 整列した（使用中オブジェクトへの）ポインタが指すオブジェクトすべてを，（A領域で並べられたアドレスの順に）コピー先領域の最下位アドレスから順にコピーしていく（copy）．その際，コピー元のオブジェクトの先頭に，コピー先オブジェクトへの転送先ポインタをいれておく．

段階4 コピーされたすべてのオブジェクトのポインタを表す部分に関して，転送先ポインタを参照して修正をする（adjust）.

段階5 コピー先領域とコピー元領域を入れ換える（switch）.

図2に，この基本アルゴリズムを具体的に記述する．図2のアルゴリズムは言語処理系のレジスタ

```
traverse (Object *p)
{/* LIST 構造をたどって印付けする */
 if (markp (p)) return;
 /* 構造を持つ型なら，型に応じて再帰的に呼び出す.
    次のものは，CONS セルの例（一般化は容易である）*/
 if (!atomp (p)) {traverse (car (p)), traverse (cdr (p)); }
 コピー元のオブジェクトに印付けをし，A領域に
 順にそのオブジェクトのポインタを書いていく;
}

copy ( )
{Object *p;
 /* top はコピー先領域のインデックス */
 top=to;
 for（A領域のすべてのポインタpに対して）{
  /* pの指しているオブジェクトは，いつも
     コピー先領域への転送先ポインタではない */
  印を消して top の指す場所にオブジェクトをコピー;
  /* 転送先ポインタをセット */
  ポインタpの指しているオブジェクト
  の先頭に top の値を書き込む;
  /* オブジェクトの大きさ分だけ top を更新する */
  top+＝コピーしたオブジェクトの大きさ;
}}

adjust ( )
{Object *p;
 for（A領域のすべてのポインタpに対して）
  if (vectorp (p)) /* 補正が必要な場合 */
   ポインタ *p の指しているオブジェクトの先頭にある
   転送先ポインタを新しいポインタ *p として設定する;
}

sort ( ) {A領域のすべてのポインタを整列する;}
switch ( ) {コピー先領域とコピー元領域を入れ換える;}

gc ( )
{traverse (ROOT); sort( ); copy ( ); adjust ( );
 switch ( ); }
```

図2 生成順序を保存するコピー方式 GC
Fig. 2 The copying GC for preserving the generated order.

ROOT から，使用中オブジェクトがたどれ，マーク付けできることを前提とした．言語処理系でレジスタが複数あるときは，そのレジスタ分だけ，手続き traverse を呼び出せば，使用中オブジェクトをすべてマーク付けすることができる．言語処理系でのレジスタとは，具体的には大域変数や実行時のスタックなどが挙げられる．

さて，この方法は，コピーする順序を段階2で明示的に整列しているので，GC の前後でオブジェクトの生成順序は保存されていることは明らかである．また，全体として使用中オブジェクトが圧縮されてコピーされている．

最小の大きさであるオブジェクト1語（例えば32ビット）の大きさを記憶領域の単位の大きさとする．1つのポインタは1語で表現されるもの（最小のオブジェクトより大きくない）と仮定する．本稿では次のような量を用いる．

m：半領域全体の大きさ（ヒープ全体では $2m$ の大きさ）
n：使用中のオブジェクトの個数
s_{min}：最小のオブジェクトの大きさ
s_i：各使用中のオブジェクトの大きさ
$(s_i \geq s_{min} \geq 1)$
S：使用中のオブジェクト全体の大きさ
$\left(S=\sum_{i=1}^{n} s_i\right)$
L：記憶領域の単位のビット長（1語の大きさ）

本アルゴリズムでは，整列のためのA領域をコピー先領域中にとるが，次の考察により，段階3（copy）の作業中に記憶領域が不足することはないことがわかる．copy において，i 番目のオブジェクトがコピーされた時点を考える $(1 \leq i \leq n)$．この時点では，A領域の中でその後で必要なものは最上位アドレスにある $(n-i)$ 個のポインタである．それよりも下位アドレスにある i 個のポインタは，コピーされた使用中オブジェクトによって上書きされても，すでにコピーされたオブジェクトであるので，不要である．またその後の段階のポインタ補正ではコピー元オブジェクトの転送先ポインタを参照して行い，A領域は使用しない．したがって，任意の時点 i で必要な記憶領域の大きさは次のようになる．

$$\sum_{j=1}^{i} s_j + (n-i) \leq \sum_{j=1}^{n} s_j = S \leq m$$

さて，このアルゴリズムの全計算時間は，段階2

(sort) における n 個の整数の整列時間とその他の段階全体の和である．後者は通常のコピー方式 GC と同様に S に比例する時間で実行できる．つぎに，整列の時間について考える．この整列は，n 個の整数（例えば 32 ビットの1語）を大きさの順序に並べる単純なものである．それで，(作業領域を要しない) ヒープ整列法や ($2\log_2 n$ 語の作業領域で済む) クイック整列法のように高速の整列法を用いると，$n\log n$ に比例する計算時間がかかるが，その定数係数が小さいので，S と比較して n が小さい場合には，相対的に短時間で済む．

なお，整列の作業中は，コピー先の半領域全体を利用できるので，(作業領域の大きさに関する緩い仮定を置くだけで) n に比例する時間しか要しない整列法（例えば，基数整列法）を用いることができる．これにより，理論的には，オーダの意味で全体の計算時間は $O(S)$ になる．

3. マークビットをもたないアルゴリズム

通常，コピー方式 GC では，使用中のものとゴミとを区別する印付けの段階においては，マークビットを使用しないで行う．コピーされて回収された使用中オブジェクトは，回収されたことを，マークビットに印を付ける代わりにコピー元オブジェクトからコピー先オブジェクトへの転送先ポインタを代入する．また，マークビットを調べる代わりに，コピー先領域への転送先ポインタが代入されているかどうかを調べることによって，使用中であると判定されたオブジェクトであるかどうかわかる．

本章では，前章のアルゴリズムを改良して，通常のコピー方式 GC と同様にマークビットを使用しないで，生成順序を保存するコピー方式 GC のアルゴリズムについて述べる．

まず，マークビットをコピー先領域にとる単純な方法を示す（**図3**参照）．前章ではマークビットを各オブジェクトに1ビットずつとっていたが，これをまとめてコピー先領域の下位アドレス部分にビットマップ方式でとる．この領域をM領域と呼ぶことにする．M領域の大きさは，ビットごとにオブジェクトに対応するのでコピー先領域全体 m の $1/L$ (例えば，1/32) でよい．しかし，マーキング段階の最初にM領域全体を初期設定する必要がある．したがって，マークビットをコピー先領域にとる方法では，全体の計算時間は，($O(m/L)$ で済むとはいえ) スライディング圧縮方式

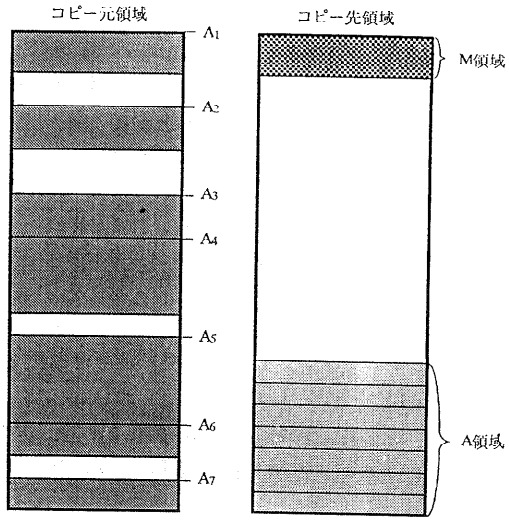


図 3 コピー先領域にマーク領域を取る方法における記憶領域配置
Fig. 3 Memory allocation for the method with the marking area in the destination space.

などと同様に領域の大きさに比例するものになる．

なお，コピーされたオブジェクトは，M領域を参照することによって知ることができるので，M領域とコピーされたオブジェクトに置かれた転送先ポインタによって，A領域を使用しなくても生成順序を保存するコピー方式の GC を行うことができる．しかし，この場合には，M領域の各ビットを1ビットずつ走査していくことが，2章で述べた基本的アルゴリズムでの，コピー元半領域全体を走査していくことに対応している．そのため，この部分に必要な計算時間は，全体で m に比例するものになる．

次に，逆ポインタを用いる第2の方法を説明する．まず，アルゴリズムの概略を次に示す（**図4**参照）．

段階1 言語処理系のレジスタで示されるオブジェクトから順にリスト構造をたどって，すべての使用中オブジェクトに関して，コピー先領域の最下位アドレスから順に最上位アドレスに向かって，そのオブジェクトの第1語の内容を保存する（B領域と呼ぶ）．また，コピー先領域の最上位アドレスから順に最下位アドレスに向かって，そのオブジェクトを指すポインタを入れていく（A領域）．またそのオブジェクトの第1語にはそこへの逆ポインタ（たったいまオブジェクトを指すポインタを入れた場所（A領域内）へのポインタ）を入れる（nomark-traverse）．

段階2 第1段階で保存した使用中オブジェクトの第

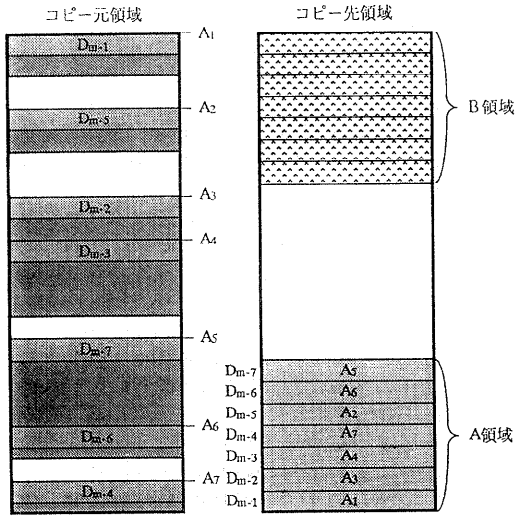


図 4 逆ポインタを使用する方法における記憶領域割り当て
Fig. 4 Memory allocation for the method with mutual pointers.

1語をB領域からもとに戻す．戻す場所へのポインタを格納した場所は，A領域とB領域のコピー先領域内のアドレスの対称性から簡単に算出できる（restore）．

段階 3, 4, 5, 6 これ以降の各段階，すなわち領域の整列と現役オブジェクトのコピーとポインタ修正はマークビットを使用する基本アルゴリズムと同様である（sort, copy, adjust, switch）．

図5にこのアルゴリズムを具体的に記述する．前章のマークビットを利用するGCとの違いはマークビットをセットする代わりに，（A領域のポインタとオブジェクトのポインタが互いに指し合うようにするため）A領域のポインタに対する逆ポインタを作ることである．逆ポインタを用いることによって，マークビットを省略することができる．つまり，第1段階でリスト構造をたどるとき，マークされたかどうかを調べる代わりに第1語をポインタとして見たとき，コピー先領域を指していて，その指しているものは自分自身への逆ポインタであるかどうか調べる．（例えば同様の技法は文献1)に述べられている．）詳しくは次のとおりである．

場合1 第1語のポインタをみて，A領域内を指していない場合は，このオブジェクトはまだマークされていないと判定する（明らか）．

場合2 第1語ポインタを見て，A領域内を指している場合には，指している先の語の中のポインタがそのオブジェクトを指しているかどうか調べる．

```
nomark_traverse (Object *p)
{
 if（pの指すオブジェクトがA領域で，指している先の
    オブジェクトがそのオブジェクトを指している）
  return;
 /* 構造を持つ型なら，型に応じて再帰的に呼び出す．
    次のものは，CONS セルの例（一般化は容易である）*/
 if（pの指すオブジェクトはコンス型）{
  traverse (car (p)); traverse (cdr (p));
 }
 pの第1語の値をB領域の最上位に保存;
 pの第1語にA領域の最上位アドレスへのポインタを代入;
 A領域の最上位にpの第1語へのポインタを代入;
}

restore ( )
{
  for（A領域のすべてのポインタpに対して）
    B領域に保存したpの第1語をもとに戻す;
}

gc ( )
{
  nomark_traverse (ROOT);
  restore ( ); sort ( ); copy ( ); adjust ( ); switch ( );
}
```

図 5 マークビットを使用しないで生成順序を保存するコピー方式 GC
Fig. 5 The improved copying GC having no marking bits.

場合 2.1 指していなければ，まだマークされていないと判定する．（指しているものは，別のマークされたオブジェクトであるからである．）

場合 2.2 指していれば，すでにマークされていると判定する．（マークしたものに相当するオブジェクトのみ互いにポインタが指し合うからである．）

第2段階が終わった時点で，コピー先領域の様子は，マークビットを利用する GC の方式の概略を表した段階1が終わった時点と同じになっている．したがってこれ以降は同じ操作をすることで，GC が完了する．

この方法のポインタを扱う上記の部分の計算時間は，使用中オブジェクトの総数 n に比例する．よって，（整列の部分以外の）全計算時間は S に比例する．作業領域の大きさは $2n$ になる（A領域とB領域の和）．それで，この方法では，記憶領域に関する仮定 $(2n \leq m)$ を満たす必要がある．この仮定は，オブジェクト1個あたり，2語以上である $(s_{min} \geq 2)$ ことに相当する，実際の処理系では，この仮定は満たされるのが普通である．例えば，最近の言語処理系では，配列

型や多倍長型，複雑なアトム型など1オブジェクトの大きさが大きい傾向がある．また小さく単純なオブジェクトを数多く扱う場合は配列型などまとまった1つのオブジェクトとしてまとめて扱う場合がほとんどである．

4. おわりに

本論文ではコピー方式GCに，生成順序を保存する特性をとりいれる基本アルゴリズムを提示し，その計算時間と領域量に関する考察を行った．次に，基本的なアルゴリズムでは，普通のコピー方式とは違い最初に使用中のオブジェクトをマーク付けする必要があった．これを解消するために

1. コピー先領域の空いた部分を利用してビットマップ形式でマークするための領域をとり，その領域にマーク付けする方法
2. 転送先ポインタと逆ポインタの組を用いる方法

の2種類を考案した．コピー方式の特徴からみて，計算時間が m に依存しない後者の方法が推奨される．

言語処理系の起動時から徐々にたまってくる静的なオブジェクトは，処理系が実質的な仕事をするために必要なライブラリモジュールなどで，実際に利用者が処理系を使って有意義な仕事をするまでには，かなり大きなものになる．普通のコピー方式GCでは，このような大きくてほとんどゴミとはならないオブジェクトを頻繁に移動してしまう．しかし，本稿のコピー方式GCは，オブジェクトの生成順序を保存することが保証されているので，従来のコピー方式GCでは適用不可能であったスライディング圧縮方式GCなどで使われてきた最適化技法が使える．

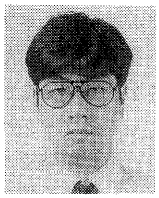
例えば，静的なコピーをしなくてもよい領域は2つの半空間の間で共有することが可能になる．ただし，コピーしなくてもよい領域から，コピーされた領域をさすポインタがある場合があるため，ポインタの補正は必要である．

今後の課題として，本稿の方法をジェネレーショナルGCの性能向上に応用できるか否かを調べることをあげておく．

参考文献

1) Aho, A., Hopcroft, J. and Ullman, J.: *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).
2) Appel, A.: Simple Generational Garbage Collection and Fast Allocation, *Softw. Pract. Exper.*, Vol. 19, No. 2, pp. 171-183 (1989).
3) Courts, R.: Improving Locality of Reference in Garbage Collecting Memory Management System, *Comm. ACM*, Vol. 31, No. 9, pp. 1128-1138 (1988).
4) Fenichel, R. and Yochelson, J.: A Lisp Garbage-Collector for Virtual-memory Computer Systems, *Comm. ACM*, Vol. 12, No. 11, pp. 611-612 (1969).
5) Lieberman, H. and Hewitt, C.: A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Comm. ACM*, Vol. 26, No. 6, pp. 419-429 (1983).
6) Noshita, K. and Hikita, T.: The BC-Chain Method for Representing Combinators in Linear Space, *New Generation Computing*, Vol. 3, pp. 131-144 (1985).
7) Terashima, M. and Goto, E.: Genetic Order and Compactifying Garbage Collectors, *Inf. Process. Lett.*, Vol. 7, No. 1, pp. 27-32 (1978).
8) Ungar, D.: Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm, *SIGPLAN Notices*, Vol. 19, No. 5, pp. 157-167 (1984).
9) Zorn, B.: Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection, *ACM Conference on Lisp and Functional Programming*, pp. 87-98 (1991).

(平成4月6月23日受付)
(平成5年9月 8 日採録)



小出 洋

1991年，電気通信大学計算機科学科卒業．1993年，同大学大学院情報工学専攻博士前期課程修了．現在，同博士後期課程在学中．記憶管理方式，並列分散処理に関する研究を行っている．

野下 浩平（正会員）

1943年生．東京大学工学部計数工学科卒業．現職：電気通信大学情報工学科教授．工学博士（東京工業大学）．専門：アルゴリズム解析，組合せゲームの理論と実験．ACM, EATCS, CSA 等の各会員．