資 料

最小総和法に対する2つの $O(n \log n)$ アルゴリズム*

高 岡 忠 雄**

Abstract

As a method for floating point summation which reduces truncation errors, in this paper, a new technique, minimal summation, is presented. This technique sums up floating point numbers from small to large. An ordinary algorithm for this problem takes $O(n^2)$ run time where n is the size of data, while two algorithms presented take both $O(n \log n)$ run time. One applies so-called Heapsort method, the other utilizes the data structure of queues. Finally it is shown that the inherent complexity of this problem is also $O(n \log n)$.

1. まえがき

多くの浮動小数点数を加え合わす場合，たとえばFORTRAN で書いて

```
      S=0.
      DO 10 I=1,N
   10 S=S+X(I)
```

とすると，$X(I)$ が正の場合，S が大きくなるにつれて，$X(I)$ の加算において，いわゆる積み残し現象による丸め誤差が発生し，N が大きいとき，この誤差の累積は無視できないものになる，この丸め誤差の累積を純粋ソフトウェア的に，すなわち，ハード的に語長を長くしたりしないで，減らすプログラミング法がいくつか提案されている．1つは，Wolfe[1]，Kahan[2]らによる，多くのアキュミュレータを利用する方法，もう1つは Linz[3] によるトーナメント算法等である．

本論では，トーナメント算法を改良して，与えられた浮動小数点数のリストの中の常に最小の2つを加算するという条件で計算を進める方法を提案し，そのためのいくつかのアルゴリズムについて比較検討する，この総和法を最小総和法と呼ぶ．誤差解析は別の機会に譲り，本論では，アルゴリズム解析に重点を置いて議論を進める．

2. アルゴリズム HEAPSUM

まえがきで述べた本論の問題をアルゴル風に述べると，L を浮動小数点数のリストとして

```
while |L|>1 do
begin
    m1:=Min L;
    L:=L−Min L;
    m2:=Min L;
    L:=L−Min L;
    m:=m1+m2;
    L:=L∪{m}
end
```

(Min はリスト中の最小値をとる関数)

最終的に m に求める総和が得られる．このアルゴリズムを忠実に実行すると $O(n^2)$ の時間がかかる．ここで n はリストのサイズである，

そこで，有名な Heapsort (例えば Aho 他[4]参照) を応用して，$O(n \log n)$ ランタイムのアルゴリズム Heapsum を考案した．それを以下に述べる．

HEAP というのは **Fig. 1** に示すようなラベル付の2進木であり，各節点と葉には数が付されている．そして親の数は2つの子の数より大きくない．**Fig. 1**

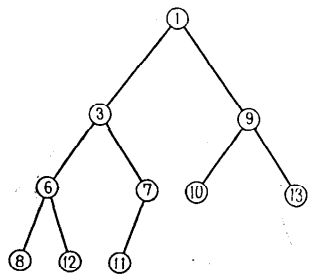

Fig. 1 HEAP

* Two $O(n \log n)$ Algorithms for Minimal Summation by Tadao TAKAOKA (Department of Information Science, Faculty of Engineering, Ibaraki University)

** 茨城大学工学部情報工学科

において1と11を入れ替える．そうすると1を除いた新しい木は HEAP になっていないから 11 をその子3と9の小さい方3と入れ替え，さらに11を6と入れ替え，最後に8と入れ替えると，**Fig. 2** に示すような1を除いた HEAP が出来上る．次に3と12を入れ替えて同様なことを行い，これらの過程を繰り返すと，根からとり出したもの，すなわち，木の同一のレベルでは右から左へ，そしてレベルは下から上へというように並んだものは，小さい順にソートされている．これが Heapsort の概要である．

さて，HEAP では最小のものが根に，その次に小さいものがその子のどちらかにあることに着目し，それらの2つ間で加算を行い，その和（最小和という）を一旦根に置き，加算に用いた子の節には大きな数（ここでは∞と記す）を置く．そしてその子から下の木を HEAP につくり直し，その後，根から下，すなわち全体の木を HEAP につくり直す．この操作，すなわち，1回の加算と2回の HEAP のつくり直し(大きな数が下に落ちて行くような過程なのでここでは SHIFT と呼ぶ) を $n-1$ 回やれば，最小総和を実行したといえる．

このためのプログラムを次に掲げる．（言語は ALGOL)

```
procedure HEAPSUM (A,N)
  array A; integer N;
  begin
      procedure SHIFT (I,J);
        value I,J; integer I,J;
        begin integer K;
          LOOP: K:=2*I;
          if K>J then go to EXIT;
          if A[K]>A[K+1] then K:=K+1;
          if A[K]<A[I] then
            begin real COPY;
                  COPY:=A[I];
                  A[I]:=A[K];
                  A[K]:=COPY;
                  I:=K;
                  go to LOOP
            end
          EXIT:
        end of SHIFT;
      procedure BUILPHEAP (N);
        integer N;
        begin integer I;
          for I:=N step-1 until 1
            do SHIFT (I,N)
        end of BUILDHEAP;
        integer I,K;
        A[N+1]:=∞;
        BUILDHEAP (N);
        for I:=1 step 1 until N-1 do
          begin
          K:=2;
          if A[2]>A[3] then K:=K+1;
          A[1]:=A[1]+A[K];
          A[K]:=∞;
          SHIFT (K,N);
          SHIFT (1,N)
          end
        end of HEAPSUM;
```

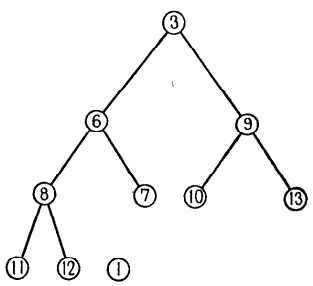

Fig. 2 NEXT HEAP

[プログラムの説明]

手続 HEAPSUM をサイズ N の配列に N 個の浮動小数点数を入れて実行すると A[1] に所期の結果が得られる，2進木は A[1] が根，A[2], A[3] が子，一般に，A[K] の子は A[2∗K] と A[2∗K+1] という形で表現されている．

手続き BUILDHEAP は手続 SHIFT を木の下から順次用いることによって HEAP をつくり上げる．HEAPSUM において，BUILDHEAP 呼び出しの前に A[N+1]:=∞ としてあるのは，N が偶数のとき，未定義の数が下から上ってくるのを防ぐためである．HEAP をつくり上げた後，$N-1$ 回以下のことを行う．すなわち，根とその子の小さい方とで加算を行い，その子のところに∞を代入して，その子から下の木で SHIFT を行い，その後，根から下で SHIFT を行う．

[アルゴリズムの解析]

4)によると，BUILDHEAP のランタイムは $O(n)$ である．つぎに，SHIFT (K, N) と SHIFT (1, N) は $\log_2 N$ のオーダのランタイムでおさえられるから，そしてループは $N-1$ 回であるから全体として $O(n \log n)$ のランタイムでおさえられる．(n は N と同じ)

3. アルゴリズム MINSUM

本章では，すでに小さい順にソートされたリストに対する最小総和法を考える．配列要素 A[1], A[2], … がこの順でソートされた浮動小数点数を保持しているとする．A[1]+A[2] でできた数を A[3] 以後の適当

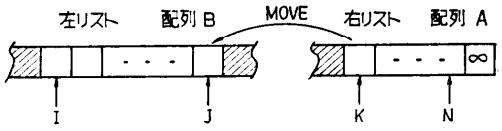

Fig. 3 Left List and Right List

な場所にマージしたとすると，つぎの最小な2つの和は前回マージしたところより後に来るという性質を利用する．マージすることによって多くの数を移動すると時間がかかるので，**Fig. 3** に示すように左リストと右リストに分け，左リストは Queue のデータ構造とし，左リストの最左端の2つの数の和をどこに入れるかを考える．前段階での和が左リストの右端に入っているとすると，右リストの左端の要素と大小を比較し，右リストの左端の要素の方が小さければ，それを左リストの右端に送る．この過程を繰り返し，現在の和の方が小さくなれば，それを左リストの右端に置く．それから，次に左リストの左端の2要素の和を求めてこの過程を繰り返すことになる．最小の2数の和を求める加算が $n-1$ 回実行されると終ることになる．

このためのプログラムを ALGOL で以下に示す．

```
real procedure MINSUM (A, N);
  array A; integer N;
  begin
    array B(1 : 2*N);
    procedure MOVE (J, K);
      integer J, K;
      begin
        J:=J+1;
        B[J]:=A[K];
        K:=K+1;
      end of MOVE;
      integer I, J, K;
      real SUM;
      B[1]:=A[1];
      A[N+1]:=∞;
      I:=J:=1;
      K:=2;
    LOOP 1: if I=J then MOVE (J, K);
            SUM:=B[I]+[I+1];
            I:=I+2;
    LOOP 2: if SUM>A[K] then
              begin
                MOVE (J, K);
                go to LOOP 2
              end;
            J:=J+1;
            B(J):=SUM;
            if I≤2*N−3 then go to LOOP 1;
  MINSUM:=SUM;
end of MINSUM;
```

[プログラムの説明]

まず変数 I, J, K について説明する，I, J はそれぞれ左リストの左端と右端へのポインタ，Kは右リストへのポインタである，左リストは配列Bで確保する．手続 MOVE は右リストから要素が1個左リストへ移動してくることを実行する．関数手続 MINSUM で最初にやることは与えられたデータを保持している右リストの配列Aから1個 A[1] をけずって B[1] にもってくる．そしてポインタ I, J, K をそれぞれ 1, 1, 2 に初期設定する．また A[N+1] に ∞ を代入する．

つぎに LOOP 1 のところでやることは，もし $I=J$ なら，左リストでの加算ができないので1個 MOVE する．そして SUM に最小和を代入してポインタ I を2つ右へ移動 (Fig. 3 参照)．

つぎに LOOP 2 のところでやることは SUM が A[K] より大なら A[K] を左リストへ MOVE という動作を繰り返す．SUM≤A[K] となれば J を1増やして SUM を左リストの右端におく．そして，$I \leq 2*N-3$ なら LOOP 1 にもどってこの過程を繰り返す．この I に関する条件は加算を $N-1$ 回実行することと等価である．$N-1$ 回の加算が終れば MINSUM に SUM の値を代入して終る．

[アルゴリズムの解析]

go to LOOP 1 による外側のループの中に go to LOOP 2 による内側のループがあり，それぞれ最大 $O(N)$ 回実行されるので，全体として $O(N^2)$ のランタイムになりそうであるが，実はそうではない．内側のループでの数の移動は外側のループで加算が1回実行されるごとに何回か起こるわけであるが，移動する数の個数は $N-1$ であり，A[N+1] には ∞ が入っているのでそれ以上の移動は起こらない．それ故，内側のループの総実行時間は $O(N)$ である．結局全体として $O(N)$ のランタイムであり，配列Aがすでにソートされたデータを含むときは，この procedure は線型時間で走ることになり，能率はよい．

配列Aがソートされていないときは，Aho 他[4]に掲載されている $O(n \log n)$ ランタイムの HEAPSORT (A, N) を一度呼び出し，然る後 MINSUM (A, N) を呼べば，全体のランタイムは $O(N \log N)$ となる．

4. 最小総和法の固有複雑度

サイズ n のデータを処理する問題において，いかにアルゴリズムを工夫しても，そのアルゴリズムのランタイムの上限が $O(f(n))$ に達する場合，そして $O(f$

(n)) ランタイムのアルゴリズムが存在する場合，この問題は $O(f(n))$ の上限固有複雑度 (upper inherent complexity) をもつという．また，いかにアルゴリズムを工夫しても，そのアルゴリズムのランタイムの平均値が $O(f(n))$ に達する場合，そして $O(f(n))$ の平均ランタイムのアルゴリズムが存在する場合，この問題は $O(f(n))$ の平均固有複雑度 (expected inherent complexity) をもつという．

明らかに，ある問題が $O(f(n))$ の平均固有複雑度をもてば，その問題は $O(f(n))$ 以上の上限固有複雑度をもつ．

さて，本論の問題に対しては次の定理が成り立つ．

定理．サイズ n のデータに対する最小総和法は $O(n \log n)$ の上限固有複雑度をもつ．

証明

$$b_i=2^{i-1} \quad (i=1,\cdots,n)$$

なる数列 $\{b_i\}$ を考える．

$$\sum_{i=1}^{k-1} b_i<b_k<b_{k+1}$$

なる関係が任意の k に対して成り立つから，リスト $(b_1, b_2, \cdots, b_n)$ に対する最小総和法は ALGOL 風に書けば

```
s:=0;
for i:=1,2,…,n do
  s:=s+b_i;
```

となる．すなわち，最初の2つはともかくとして，加算に参加する b_i は小さい順に，すなわち $b_1, b_2, \cdots, b_n$ の順である．今，配列 A[1], A[2], …, A[N] が $b_1, b_2, \cdots, b_n$ を適当に置換したものを含んでいるとする，($n=N$)．配列Aに対する最小総和法はやはり $b_1, b_2, \cdots, b_n$ の順で加算に各 b_i が参加することになる．すなわち，最小総和が完了したとき A[1], A[2], …, A[N] のソーティングもまたできたことになる．今，配列Aが $b_1, \cdots, b_n$ の $n!$ 個の置換をどれも等しい確率 $p=1/n!$ で含んでいるとする．最小総和法は最小値をみつけるために必ず大小の比較を行うのであるから，比較の2進木を考えると，その比較の2進木の葉に達したとき，最小総和が完了し，そして，1つの置換が発見されたことになる．$n!$ 個の葉をもつ2進木の高さはよくバランスしたときで $\log n!=O(n \log n)$ であるから，上限固有複雑度は $O(n \log n)$ より小さくない．一方前に述べたように，$O(n \log n)$ のランタイムの最小総和のアルゴリズムが存在するから，最小総和法の上限固有複雑度は $O(n \log n)$ であるといえる．

5. あとがき

最小総和法の平均固有複雑度も $O(n \log n)$ になると推測されるが，与えられた浮動小数点数の分布を考慮に入れねばならなず，解析は難しいものとなろう．

なお，本論文中のアルゴルプログラムは HITAC 8350 のアルゴルコンパイラによってデバッグされている．また，本研究は一部文部省科学研究費によるものである．

参考文献

1) J. M. Wolfe: Reducing Truncation Errors by Programming, CACM, Vol. 7, No. 6, 355～356 (1964)
2) W. Kahan: Further Remarks on Reducing Truncation Errors, CACM, Vol. 8, No. 1, p. 40 (1965)
3) P. Linz: Accurate Floating Point Summation, CACM, Vol. 13, No. 6, pp. 361～362 (1970)
4) A. V. Aho, J. E. Hopcroft and J. D. Ullman: The Design and Analysis of Computer Algorithms, Addison-Wesley (1974)

(昭和51年9月20日受付)
(昭和52年1月19日再受付)