31. ファイル間の相違検査法†

角 田 博 保††

1. はじめに

2つのファイルを比較して，その内容のどこがどのように違っているかを見つけ出すアルゴリズムについて解説する．

テキストエディタなどで修正されたファイルがあった場合，元のファイルと比べてどこがどう変ったのかを知りたいことがよく起る．プログラムの修正がミスなくできたかを確認したかったり，英文テキストの修正サマリーを印刷してみたかったり，また，似かよったデータファイルからその類似箇所を取り出してみたいこともある．変ったところでは，二つのファイルを比較して，一方を他方に変換するエディタのコマンド列を作り出したいこともある．

以上のようなことをおこなうユーティリティプログラムが開発され，実用に供されている．たとえば，DEC 系の FILECOM, UNIX の diff, あるいは FCMP[1),2)]などである．これらのプログラムは比較の結果を印刷するしかたは様々ではあるが，内部では一律にファイル間の相違検査アルゴリズムが使われているのである．

アルゴリズムを解説する前に，用語の定義をすることにしよう．比較対象となるファイルを A, B とし，各々を $A=a_1a_2\cdots a_m$, $B=b_1b_2\cdots b_n$ と表わすことにする．ここで a_i, b_j はファイルの項目を表わす．ここで取り扱うファイルは行の列，単語の列，文字の列のように，項目の列であれば何でもよい．なお，i と j はそれぞれAとBの項目を表わす添字，m と n はそれぞれAとBの項目数を指し示すために使うことにする．

次に，項目 a_i と b_j が対応していることを (i,j) という対で書くことにする．以下の2条件を満たす対の集合Sのことを**対応づけ**と呼ぶことにする．

① $(i,j)\in S$ ならば $a_i=b_j$

② $(i_1,j_1),(i_2,j_2)\in S$ ならば $i_1\neq i_2$ は $j_1\neq j_2$ と同値

対応づけが次の条件③を満たすならば，**単調対応づけ**と呼ぶ

③ $i_1<i_2$ かつ $(i_1,j_1),(i_2,j_2)\in S$ ならば $j_1<j_2$

実際に求めようとする対応づけはSの要素数をある規準で最大化するようなものである．それを**最適対応づけ**と呼ぶことにすると，ファイル間の相違検査問題は，

「ファイルA, B間の最適対応づけSを求めること」

と表現することができる．

Sは要素数を単純に最大化したのでは意味のある対応づけとはならない場合が多い．今までのところ，2種の最適対応づけ規準が与えられている．一つはA, B間の**最長共通部分列** (Longest Common Subsequence 略して LCS) を求める対応づけであり[3)~5)]，もう一つはA, Bファイルに唯一の要素は対応し，それに隣接する同一項目は対応するという **Heckel** による対応づけ[6)]である．各々を S_{LCS}, S_{Heckel} と書くことにすると，その定義は以下の通りである．

定義 S_{LCS} とは次の2条件を満たす対応づけ S のことである．

① S は単調対応づけである．

② 任意の単調対応づけ S' に対し，$|S'|\leqq|S|$

S_{LCS}は単調であるから，要素の入れ換えといった状況は把握しない．ファイルAをBに変換するのに必要な挿入，削除数を最小にすることに S_{LCS} は対応している．S_{LCS}は与えられたファイルA, Bに対し，ただ一つ定まるものではない．ここでは，そのうちの一つを探すアルゴリズムがあれば十分である．Aとして「program」Bとして「algorithm」をとると，S_{LCS}は $\{(4,3),(5,5),(7,9)\}$ である．対応した項目を並べた「grm」が LCS である．また，LCS 長は3である．

a_i と b_j がそれぞれA, B中の唯一の項目であり，$a_i=b_j$ であるとき，(i,j) は**唯一**であると呼ぶことにすると，S_{Heckel} は次のように定義できる．

† Algorithms for Isolating Differences Between Files by Hiroyasu KAKUDA (Department of Computer Science, The University of Electro-Communications).

†† 電気通信大学計算機科学科

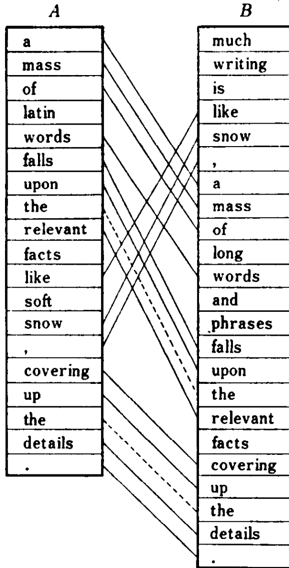

図-1 SHeckel の対応づけ (文献[6]による)
(実線は対応が唯一
点線は補間された対応)

定義 SHeckel とは次の2条件を満たす対応づけ S のことである.

① (i, j) が唯一であれば $(i, j)\in S$

② $(i, j)\in S$ に対し, $i_1\leqslant i\leqslant i_2$, $j_1\leqslant j\leqslant j_2$, $i_2-i_1=j_2-j_1$ なる i_1, i_2, j_1, j_2 が存在し,

ⅰ) $0\leqslant n\leqslant i_2-i_1$ なる任意の n に対し, $(i_1+n,\ j_1+n)\in S$ であり,

ⅱ) $(i_1-1,\ j_1-1)\notin S$ かつ, $(i_2+1,\ j_2+1)\notin S$ であり,

ⅲ) $0\leqslant n\leqslant i_2-i_1$ なる n が存在して, $(i_1+\mathrm{n}, j_1+n)$ は唯一である.

SHeckel には単調という条件がついていない. A=program, B=algorithm に対する SHeckel は $\{(3, 4), (4, 3), (6, 1), 7, 9)\}$ である. SHeckel はブロックで入れ替わった対応も把握しようとしている. **図-1** に典型的な例を示す.

以上の2つの対応づけ, SLCS と SHeckel, を求めるアルゴリズムについて, 次章以降で解説する.

2. LCS 法

LCS を求めるアルゴリズムは「2文字列間の距離を求めるアルゴリズム (文字列の訂正問題)[7]」の特殊ケースに対応している. ここでは開発された順に従って, **Wagner** と **Fischer** による2次法[7], **Hirschberg** による1次領域法[3], **Hunt** と **Szymanski** による高速法[4], および Hirschberg による pn 法[5]について述べる.

			B ファイル 1	2	3	4	5	6	7	8	9 → j
			c	b	a	c	b	a	a	b	a
A ファイル	1	a	0	0	①	1	1	1	1	1	1
	2	b	0	①	1	1	②	2	2	2	2
	3	c	①	1	1	②	2	2	2	2	2
	4	d	1	1	1	2	2	2	2	2	2
	5	b	1	②	2	2	③	3	3	3	3
	6 ↓ i	b	1	2	2	2	3	3	3	④	4

図-2 L 行列 (A=abcdbb, B=cbacbaaba)

いくつかの共通に使われる用語を定義する.

定義 (部分列) $Ai_1i_2=(a_{i_1}, a_{i_1+1}\cdots a_{i_2})$, $Bj_1j_2=(b_{j_1}\ b_{j_1+1}\cdots b_{j_2})$, とくに $A_i=A_{1i}$, $B_j=B_{1j}$.

定義 (LCS 長) L_{ij}=(A_i と B_j の LCS 長). とくに $P=Lmn$.

定義 (LCS 長行列, 略して L 行列) L=(L_{ij} を i, j 要素とする行列)

定義 (重複度) r=($a_i=b_j$ なる対 (i, j) の総個数)

定義 (種類数) s=(B 中の異なった項目数)

定義 (k 候補) (i, j) が k 候補=($a_i=b_j$ かつ $L_{ij}=k$).

定義 (最小 k 候補) (i, j) が最小 k 候補=((i, j) が k 候補かつ $L_{ij}=L_{i-1,j}+1=L_{i,j-1}+1$).

A=abcdbb, B=cbacbaaba の場合の L 行列を **図-2** に示す. ○で囲んだ場所に対応する要素対が最小候補である. 最小 k 候補を1から $p(=Lmn)$ まで順にとれば, それが LCS となる.

2.1 2 次 法

Wagner と Fischer による方法[7]で, L 行列の全要素を計算し, それに基づいて LCS を求める.

アルゴリズム (2次法)

```
1.  Li0←0(i=0…m); L0j←0(j=0…n);
2.  for i←1 to m do
      for j←1 to n do
        if ai=bj then Lij←Li-1,j-1+1
              else Lij←max(Li,j-1, Li-1,j)
3.  i←m, j←n; k←p {p=Lmn}
4.  while (i≠0) & (j≠0) do
      if ai≠bj then
```

if $L_{ij-1}=L_{ij}$ **then** $j\leftarrow j-1$ **else** $i\leftarrow i-1$
else begin
S_{LCS} に (i,j) を追加；$i\leftarrow i-1$；$j\leftarrow j-1$；
$k\leftarrow k-1$ **end**

ステップ2で L を求め，ステップ4で逆順に LCS に対応する対を求めている．時間計算量，領域計算量ともに O(mn) である．

2.2 1次領域法

Hirschberg は領域計算量を1次にするアルゴリズムを開発した[3]．これは，L行列を全部作って配列として蓄えておくのではなく，必要に応じた2行分だけを蓄えることによって，領域計算量を2次から1次にしているのである．1次領域法アルゴリズム (ALGC) はそれ自身および下請けとなるアルゴリズム ALGB を再帰的に呼び出すことによって LCS を求めている．

アルゴリズム ALGB (m, n, A, B, LL)

1. $k_{1j}\leftarrow 0 (j=0\cdots n)$
2. **for** $i\leftarrow 1$ **to** m **do begin**
3. $k_{0j}\leftarrow k_{1j}$ $(j=0\cdots n)$
4. **for** $j\leftarrow 1$ **to** n **do**
   if $a_i=b_j$ **then** $k_{1j}\leftarrow k_{0j-1}+1$
   else $k_{1j}\leftarrow \max(k_{1j-1}, k_{0j})$
   end
5. $LL_j\leftarrow k_{1j} (j=0\cdots n)$

ALGB は与えられた A, B, m, n に対して，L 行列の一部，つまり L_{mj} $(1\leqslant j\leqslant n)$ なるベクトルを LL に返すものである．

アルゴリズム (1次領域法) ALGC (m, n, A, B, C)

1. Cを空にする；
   if$(n\geqslant 1)\&(m=1)$ **then begin**
   if $a_1=b_j$ なる j がある **then** Cに $(1,j)$ を追加
   end
2. **else degin**
   $i\leftarrow \lfloor m/2 \rfloor$；
3. ALGB $(i, n, A_{1i}, B_{1n}, L1)$;
   ALGB $(m-i, n, A_{n,i+1}, B_{n1}, L2)$;
4. $k\leftarrow \min j$ s. t. $L1_j+L2_{n-j}=\max_{0\leqslant j\leqslant n}(L1_j+L2_{n-j})$
5. ALGC $(i, k, A_{1i}, B_{1k}, C1)$;
   ALGC $(m-i, n-k, A_{i+1,m}, B_{k+1,n}, C2)$;
6. $C\leftarrow C1$ と $C2$ の和
   end

1次領域法のアルゴリズム ALGC はまずステップ1で単純な場合を処理する．$n\neq 0$ かつ $m>1$ の場合は A を $(i=\lfloor m/2 \rfloor$ で) 2分して，A_{1i} と B, $A_{n,i+1}$ と B_{n1} の比較をする．ここで A の後半の比較は列を逆転しておこなっている．ステップ4で求められる位置kによってBを分割すると，A_{1i} と B_{1k} の LCS に $A_{i+1,m}$ と $B_{k+1,n}$ の LCS を加えれば全体の LCS となるのである．部分列に対して再帰的に ALGC を呼び出すことで全体の LCS が求められている．**図-3** に分割の状況を示す．

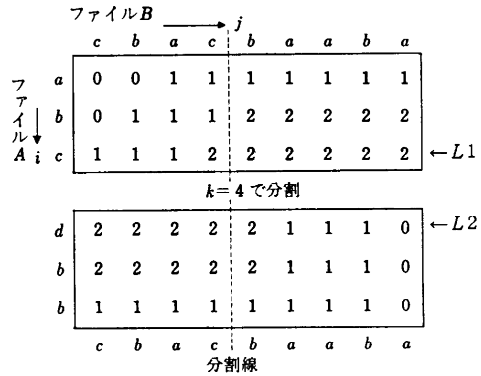

図-3 Lの分割

計算量についての細かな検討は原文[3]に譲るとして，大まかに述べることにする．領域計算量は ALGB が $O(m+n)$ であり，ALGC からの ALGB 呼び出しは独立におこなわれている．ALGC の再帰呼び出し回数は $O(m)$ であるので，ALGC の領域計算量は $O(m+n)$ である．時間計算量はステップ3で ALGB が何度も呼ばれるので，$O(mn)$ を越えそうにみえるが，再帰的に呼ばれるごとに対応する列が 1/2 で小さくなっていくので結局 $O(mn)$ でおさまる．したがって，ALGC の時間計算量は $O(mn)$ となる．

2.3 高 速 法

Hunt と Szymansky は i に対応する最小候補を順次求めるように工夫した[4]．

アルゴリズム (高速法)

1. **for** $i\leftarrow 1$ **to** m **do**
   $MATCHLIST_i\leftarrow <j_1, j_2, \cdots, j_p>$ s. t. $j_1>\cdots>j_p$ かつ $a_i=b_{j_q} (1\leqslant q\leqslant p)$
2. $THRESH_0\leftarrow 0$; $THRESH_i\leftarrow n+1 (i=1\cdots n)$;
   $LINK\leftarrow$nil
3. **for** $i\leftarrow 1$ **to** m **do** {時点1}
   for j **on** $MATCHLIST_i$ **do begin**
   $THRESH_{k-1}<j\leqslant THRESH_k$ となる k を探す．

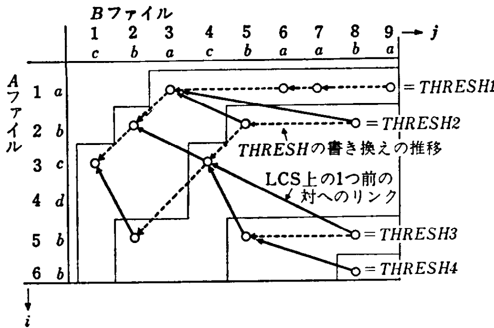

図-4 *THRESH* の使われかた

if $j<THRESH_k$ **then begin**
$THRESH_k \leftarrow j$; $LINK_k \leftarrow$ newnode $(i, j, LINK_{k-1})$ **end**;
{newnode は新しいリストノードを生成しそこへのポインタを値として返す関数である}

4. $k \leftarrow (THRESH_k \neq n+1$ なる最大の $k)$;
$PTR \leftarrow LINK_k$;
while $PTR \neq$ nil **do begin**
S_{LCS} に PTR で指される対 (i, j) を追加;
PTR を進める **end**

高速法では i に対応する候補を $MATCHLIST_i$ に格納している．時点1では，$THRESH_k$ には最小 k 候補 $\{(i', j')\}$ の内で $i'<i$ なる最大の i' に対応する j' が格納されている．時点2では，i に対応する最小 k 候補 (i, j) が存在する場合には，$THRESH_k$ が j に変更されている．このとき，LCS 上の (i, j) に先立つ対応対は，$THRESH_{k-1}$ に対応する対であるので，それに対してリンクづけをおこなっている．ステップ3終了後，リンクを逆にたどれば，LCS に対応する対を取り出すことができる．この状況を図-2と同じ例に対して示すと**図-4**の通りである．ここで点線は $THRESH_k$ が書き換った状況を示し，実線は LCS の一つ前の対へのリンクを示している．点線をたぐっていけば，$THRESH_1=(3, 1)$，$THRESH_2=(5, 2)$，$THRESH_3=(5, 5)$，$THRESH_4=(6, 8)$ がわかる．また，$THRESH_4$ からリンクをたぐれば，(6, 8)，(5, 5)，(3, 4)，(2, 2) が S_{LCS} の全要素であることがわかる．

まず，時間計算量を求めよう．ステップ1で $MATCHLIST$ を作るには，A と B をそれぞれ整列して対応づければよいのであるから，$O(n\log n+m\log m)$ の計算量が必要となる．ステップ2は $O(m)$ である．ステップ3は $THRESH$ 内を探す作業を合計で r 回（2節の定義参照）おこなうので $O(r\log p)$，その他の作業を m 回くり返すので，合計で $O(m+r\log p)$ となる．ステップ4では $O(p)$ である．以上合計して，$O(n\log n+m\log m+r\log p)$ となる．m と p は n とほぼ等しいとすれば，時間計算量は $O((r+n)\log n)$ である．

領域計算量は $MATCHLIST$ の構成が $O(m+n)$，$LINK$ のノードが高々 r であることから，$O(m+n+r)$ となり，同様の仮定によって，$O(r+n)$ となる．

2.4 pn 法

Hirschberg による新しい方法は時間計算量が $O(pn+n\log s)$ である[5]．この方法の基本的アイディアは，高速法と同様に，最小候補のみを求めようとする点にある．ここでは，最小 k 候補を $k=1$ から順次，みつからなくなるまで求めるという手順をふんでいる．

アルゴリズム（pn 法）

1. **for all** θ **in** Σ **do**
{$\Sigma=\{c|c$ が A あるいは B の項目$\}$}
$NB_\theta \leftarrow (b_j=\theta$ なる b_j の個数$)$;
$PB_\theta \leftarrow \langle j_1, j_2, \cdots, j_k\rangle$ s. t. $j_1<j_2<\cdots<j_k$ かつ $\theta=b_{j_h}(1\leqq h\leqq k)$
2. $D0_i \leftarrow 0(i=0\cdots m)$; $lowcheck \leftarrow 0$;
3. **for** $k \leftarrow 1$ **step** 1 **do begin**
4. $N_\theta \leftarrow NB_\theta$ for all θ in Σ; $Found \leftarrow$ false; $low \leftarrow D_{k-1}$, $lowcheck$; $high \leftarrow n+1$;
5. **for** $i \leftarrow lowcheck+1$ **to** m **do begin**
6. **while** $PB_{a_i, N a_i - 1}>low$ **do** $N_{a_i} \leftarrow N_{a_i}-1$;
7. **if** $high>PB_{a_i, N a_i}>low$ **then begin**
$high \leftarrow PB_{a_i, N a_i}$; $D_{k_i} \leftarrow high$;
if not $Found$ **then begin**
$lowcheck \leftarrow i$; $Found \leftarrow$ true **end**
end
else $D_{k_i} \leftarrow 0$;
8. **if** $D_{k-1, i}>0$ **then** $low \leftarrow D_{k-1, i}$
end;
9. **if** not $Found$ **then goto** step 10;
end {step 3};
10. $p \leftarrow k-1$; $k \leftarrow p$;
for $i \leftarrow m+1$ **to** 0 **do**
if $D_{k_i}>0$ **then begin**
S_{LCS} に (i, D_{k_i}) を追加; $k \leftarrow k-1$ **end**

$k-1$ までの最小候補がわかっているときに，最小 k 候補を求めるには，次の補題が役に立つ．

補題 (i, j) が最小 k 候補 $(k\geqq 1)$ であるための必

要十分条件は，$a_i=b_j$ かつ $low<j<high$ であり，$low<j'<j$ なる j' に対して $a_i \neq b_{j'}$ であることである．ここで，$high=(i_0<i$ なる k 候補 (i_0, j_0) があればその中で最大の i_0 に対応する j_0，なければ $n+1)$，$low=(i_0<i$ なる $k-1$ 候補 (i_0, j_0) の中で最大の i_0 に対応する j_0，なければ 0) である．

この補題に基づいて上記アルゴリズムは *low*, *high* を計算していく．*NB*, *PB* は高速法の *MATCHLIST* に対応している．D_{ki} は (i, j) が最小 k 候補であれば j をさもなくば 0 を値として持つ．

時間計算量は以下のようにして求められる．ステップ1は高速法と同様に考えて $O(n\log s)$ である．問題はステップ6であるが，N_{ai} を減らしていくことからステップ5の繰り返し当り高々 n 回であるので，ステップ3の繰り返しと合せても高々 pn 回である．ステップ7，8はステップ3と5の繰り返しの総数で pm 回である．したがって，$O(n)=O(m)$ の仮定により，時間計算量は $O(pn+n\log s)$ である．

領域計算量は，D_{ki} を表わすために $O(pm)$, *PB* を表わすために $O(n)$ であることから，$O(pn+n)$ である．

2.5　その他のアルゴリズム

Hirschberg は論文[5]の中で時間計算量 $O(p\varepsilon \log n)$ のアルゴリズムを示している．ここで ε はあらかじめ与えられた $m-p$ より大きい値である．このアルゴリズムは p がほとんど m に等しいとわかっている状況を想定している．

最悪のケースで2次より低い時間計算量を持つものが **M. S. Paterson** によって，開発されているという報告がある[5]．その時間計算量は $O(n^2\log\log n/\log n)$ である．L を適当な大きさ k の箱に分割し，その各々についての L 行列の値をあらかじめ計算しておき，それをもとにして全体の LCS を求めるといった方法である．k を $\log n/2(1+\log s)$ に選べば上記の計算量を得る．問題点は n が相当に大きく，s が十分に小さいという状況でなければ分割の効果が現れない点にある．

同一項目が重複して現れない場合（つまり $s=m$）には $O(n\log n)$ で LCS が求まることが知られている．この場合は**最長増加列**(longest increasing sequence)を求める問題[8]に帰着できる．

2.6　検　討

4アルゴリズムの時間計算量と領域計算量を一般の場合と s が n に近い2つの場合（$p \fallingdotseq n \fallingdotseq r$ と $r \fallingdotseq p \fallingdotseq 0$）とについて求めたものを**表-1** に示す．よく似たファイルを比較する場合は $r \fallingdotseq n$ であることが多いので高速法が時間，領域ともに優れていることがわかる．

3. Heckel 法

LCS 法は時間がかかり，期待される対応づけに必ずしも一致しない場合がある．**図-5** に S_{LCS} と S_{Heckel} の例を示す．Heckel 法では唯一な対応は必ず採用されるので，この例のように LCS 法よりうまい対応づけとなる場合もあある．

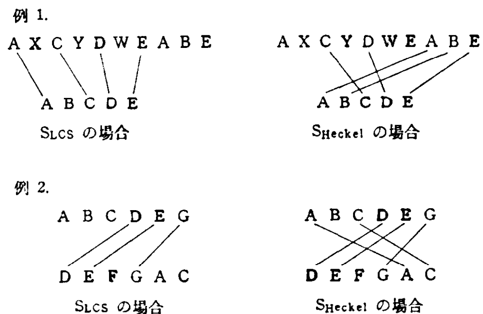

図-5　S_{LCS} と S_{Heckel} の違い

アルゴリズム (Heckel 法)

1. $LAi \leftarrow 0 (i=0 \cdots m+1)$; $LB_j \leftarrow 0 (j=0 \cdots n+1)$
2. **for all** 唯一な (i, j) **do begin** $LA_i \leftarrow j$; $LB_j \leftarrow i$ **end**;
3. **for** $i \leftarrow 1$ **to** m **do**
   if $LAi \neq 0$ **then begin**
   $j \leftarrow LAi$;
   if $(a_{i+1}=b_{j+1})$ & $(LA_{i+1}=0)$ & $(LB_{j+1}=0)$

表-1　4アルゴリズムの時間計算量と領域計算量

	一般の場合		$s \fallingdotseq n$　$r \fallingdotseq n$　$p \fallingdotseq n$		$s \fallingdotseq n$　$r \fallingdotseq 0$　$p \fallingdotseq 0$	
	時間計算量	領域計算量	時間計算量	領域計算量	時間計算量	領域計算量
2 次 法	$O(mn)$	$O(mn)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
1次領域法	$O(mn)$	$O(m+n)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$
高 速 法	$O((r+n)\log n)$	$O(r+n)$	$O(n\log n)$	$O(n)$	$O(n\log n)$	$O(n)$
pn 法	$O(pn+n\log s)$	$O(pn+n)$	$O(n^2)$	$O(n^2)$	$O(n\log n)$	$O(n)$

表-2 LCS 法と Heckel 法の比較結果
(時間値は Melcom COSMO 700 III Pascal 8000 による)

	例1	例2	例3	例4	例5	例6
ファイル A m	485	664	461	973	1406	1014
ファイル B n	492	664	476	983	1451	1777
唯一な項目 u	409	529	368	767	1034	458
対応した個数 p	478	662	444	851	1403	736*
LCS 法 (ms)	320	444	330	630	1002	1802
Heckel 法 (ms)	166	180	256	312	400	512

* Heckel 法では 732.

```
        then begin
            LA_{i+1}←j+1; LB_{j+1}←i+1 end
        end;
4.  for i←m to 1 do
        if LA_i≠0 then begin
        j←LA_i;
        if(a_{i-1}=b_{j-1}) & (LA_{i-1}=0) & (LB_{j-1}=0)
        then begin
          LA_{i-1}←j-1; LB_{j-1}←i-1 end
        end;
```

ステップ2での唯一要素の対応づけには名前表を使えばよい．2分探索木を使えば $O(n\log n)$，ハッシュ法を使えば $O(n)$ の時間計算量で処理できる．ステップ3と4は時間計算量が $O(m)$ であるから，全部あわせて，$O(m+n)$ となる．領域計算量は配列 LA, LB を使うことから $O(m+n)$ となる．

Heckel 法は唯一な項目をもとにして対応づけをおこなうので，似た項目が多くある場合には S_{Heckel} は S_{LCS} と比べてかなり要素数が少なくなってしまうであろう．

4. おわりに

LCS 法と Heckel 法のファイル比較プログラムを使った筆者の経験について述べたい[1),2)]．(注：Heckel 法では単調な対応づけのみに着目した)．筆者の手元にあるよく似たファイル同志を比較したところ，ほとんどの場合が両方式でその結果が一致した（**表-2**）．一致しなかったのは例6 ($n=1014$, $m=1777$, $p=736$) であった．Heckel 法での対応した対の数は732と4小さかった．Heckel 法がうまく働いているのは，唯一な項目数 (u) が LCS 長 (p) にかなり近いことによっている．逆にいえば，一般的に u は p に近いことを示しているといえよう．資源の消費量の点では Heckel 法が優れているのがみてとれる．この例では，Heckel 法の方が2～4倍速くなっている．領域の方も数分の一ですむ．

ファイルの大きさが一度に主記憶上で処理できない位の場合のアルゴリズムについてはまだはっきりとは検討されていない．ある単位で分割して比較をおこなうことが考えられるが，筆者の実験によれば，20000位の項目を2000ずつ比較し，その中での最終対応 (i, j) によって分割して，次の2000個を比較していくという単純な方法では，たびたびうまく比較されない場合があることがわかっている．

ファイルの項目 a_i と b_j については，今までのアルゴリズムの中では比較がすぐできるようにみなしてあるが，項目が文字の場合はよいが行や単語である場合には工夫が必要となる．ファイルの項目は等しいかどうかの比較ができさえすればよいのであるから，適当な精度の整数に変換しておけばよい．ハッシュ法を使って 32 bit の整数に変換しておけば ($h(a_i)$ のように)，$a_i \neq b_j$ なのに $h(a_i)=h(b_j)$ となるようなことは実用上起こらないので，$h(a_i)$, $h(b_j)$ を新たに a_i, b_j とみることにすれば，a_i と b_j の比較は単位時間でおこなうことができる．

ファイル間の相違検査法における問題点は，どんな対応づけが最適なのかがはっきりとわかっていない点にある．LCS 法はある意味では最適な対応づけを与えるが，十分に満足がいっているとはいい難い．また，Heckel 法も同様である．ファイルの各項目間に対応のしやすさという測度を与えた場合の対応づけとか，ファイル内に階層（ブロック）を導入して，階層（ブロック）ごとの比較も考慮した対応づけといった，より現実のファイルの構造を反映した対応づけのしかたとそれを解くアルゴリズムの開発が待たれるのである．

参考文献

1) 角田博保，前野年紀，佐渡一広：ファイル比較技法を利用したソフトウェアツールについて，第20回全国大会講演論文集，情報処理学会，pp. 359-360 (1979).
2) 藤村直美：テキストファイルを比較するソフトウェアツール FCMP の使用について，東京大学大型計算機センターニュース，Vol. 12, No. 4 (1980).
3) Hirschberg, D. S.: A Linear Space Algorithm for Computing Maximal Common Subsequen

ces, Comm. ACM, Vol. 18, No. 6, pp. 341-343 (1975).

4) Hunt, J. W. and Szymanski, T. G.: A Fast Algorithm for Computing Longest Common Subsequences, Comm. ACM, Vol. 20, No. 5, pp. 350-353 (1977).
5) Hirschberg, D. S.: Algorithms for the Longest Common Subsequence Problem, J. ACM, Vol. 24, No. 4, pp. 664-675 (1977).
6) Heckel, P.: A Technique for Isolating Differences Between Files, Comm. ACM, Vol. 21, No. 4, pp. 264-268 (1978).
7) Wagner, R. A. and Fischer, M. J.: The String-to-String Correction Problem, J. ACM, Vol. 21, No. 1, pp. 168-173 (1974).
8) Fredman, M. L.: On Computing the Length of Longest Increasing Subsequences, Discrete Mathematics, Vol. 11, pp. 29-35, North-Holland (1975).

関連する文献

9) Aho, A. V., Hirschberg, D. S. and Ullman, J. D.: Bounds on the Complexity of the Longest Common Subsequence Problem, J. ACM, Vol. 23, No. 1, pp. 1-12 (1976).
10) Lowrance, R. and Wagner, R. A.: An Extension of the String-to-String Correction Problem, J. ACM, Vol. 22, No. 2, pp. 177-183 (1975).
11) Wong, C. K. and Chandra, A. K.: Bounds for the String Editing Problem, J. ACM, Vol. 23, No.1, pp. 13-16 (1976).
12) Masek, W. J. and Paterson, M. S.: A Faster Algorithm Computing String Edit Distances, J. Comput. Syst. Sci., Vol. 20, pp. 18-31 (1980).

(昭和57年12月14日受付)