Short Note

A New HEAPSORT Algorithm and the Analysis of Its Complexity

Heapsort algorithm HEAPSORT is widely used for its high efficiency and well-defined data structure. A new heapsort algorithm is given in this paper that makes the constant factor of the complexity smaller. And it does a comparative analysis of this algorithm with those that have been designed.

Received April 1988, revised February 1989.

INTRODUCTION

The sorting problem of n elements $a_1, a_2, \ldots\ldots, a_n$ drawn from a set having a linear ordering R is to find a permutation π of n elements $1, 2, \ldots\ldots, n$ such that $a_{\pi(i)} R a_{\pi(i+1)}$ for $1 <= i < n$.

It is theoretically proved that any algorithm for sorting n elements requires at least $\log_2(n!)$ comparisons. $\log_2(n!) \doteq n \log_2 n - 1.44n + \log_2\sqrt{n} + 1.325$. So we can consider it as the theoretical lower bound of complexity.[1],[2]

Heapsort algorithm HEAPSORT was created by Williams in 1960s, and it was improved by Floyd.[3] It is impossible to improve HEAPSORT algorithm to reduce its bound of complexity. The better way to have performance is to make the constant factor of the complexity smaller (only can be changed ranging from 2 down to 1). Surely, that would be of more significance.

ALGORITHM

1. The way of designing the algorithm

It is evidently effectiveless to revise the procedure of setting up the original heap. Now, let us reconsider the procedure of rearranging the heap.

It is vacant, in fact, when the root node is removed during rearranging the heap. The new root node is either its leftson or its rightson. Floyd used this fact and proposed the improvement of HEAPSORT algorithm as follows:

(1) By comparing leftson with rightson once, the larger node can move up one level. Repeat this action until the vacant node (say Y in the following) appears on the bottom level.
Then the deepest rightmost leaf will be put in the position of the vacant node.

(2) It is possible for node Y to move up. This is simply done by one comparison with its parent node each time until the parent node is larger.

It is obvious that the best case will happen when node Y won't move up during rearranging the heap each time. Therefore, the best case time complexity becomes $n \log_2 n + 0(n)$. On the other hand, under the worst case, node Y will move up $h - 1$ times when rearranging the heap (h is the height of the heap with $j - 1$ elements, $h = \log_2(j - 1)$), so that the complexity will be $\Sigma_{j=2}^{n} 2(\log_2(j-1) - 1)$, i.e. $2n \log_2 n + 0(n)$, which is that of the original HEAPSORT algorithm. In general, node Y will not move up too high, thus Floyd's algorithm has better average performance.

Based on the above ideas, now let us consider the improvement of the algorithm as follows.

First, we use (1) to let the process, comparing leftson with rightson once and the larger moving up one level, stop at level $(2/3)h$ (h is the height of the current heap). Then the deepest rightmost leaf will be put in the position of the vacant node with level $(2/3)h$. When node Y moves up, we use (2). Otherwise, in the necessary condition, we use original HEAPSORT algorithm to let Y move down along certain path.

2. Algorithms

(1) Set up the original heap [1]

```
procedure HEAPFIFY(i, j);
  {Arrange elements A[i]–A[j] of array
   A into a heap}
  if i is not a leaf and a son of i contains
     an element which is larger than i
  then begin
       Let k be a son of i with the larger
         element;
       interchange A[i] and A[k];
       HEAPFIFY(k, j)
     end;
procedure BUILDHEAP;
  for i: = ⌊n/2⌋ step −1 until 1 do
    HEAPFIFY(i, n);
```

(2) Rearrange the heap

```
procedure UPORDOWN(i, j);
  if i < d
    then begin
         Let k be a son of i with the larger
           element;
         A[i]: = A[k];
         UPORDOWN(k, j)
       end
    else begin
         A[i]: = A[j + 1];
         if A[i] < =A[⌊i/2⌋]
           then HEAPFIFY(i, j)
           else while A[i] > A[⌊i/2⌋]
             do
             begin
               interchange A[i] and
                 A[⌊i/2⌋];
               i: = ⌊i/2⌋
             end
       end;
```

(3) Heap sorting

Input: array of elements $A[i]$ ($1 <= i <= n$) to be sorted;
Output: the sorted array A.

```
procedure NEWheapsort;
  BUILDHEAP;
  for j: = n step −1 until 2 do
    begin
      d: = 2^⌊(2/3)log2(j − 1)⌋;
      B: = A[1];
      UPORDOWN(1, j − 1);
      A[j]: = B
    end;
```

Analysis of the Complexity

The process of rearranging the heap is obviously correct. And the correctness of the algorithm *NEWheapsort* can be proved by induction on the times that **'for'** loop has been executed.

Theorem. The worst case complexity of the algorithm *NEWheapsort* is $T(n) = (4/3)n \log_2 n + 0(n)$.

Proof. The complexity of the algorithm consists of two parts:

(1) Setting up the original heap by calling *BUILDHEAP*. It takes time $0(n)$;[4]
(2) The time it requires for the **for** loop to perform.

Once the **for** loop is performed, *UPORDOWN* rearranges the heap with $j - 1$ elements, the height of which is $h = \log_2(j - 1)$.[1] By one comparison the leftson with the rightson, the larger moves up one level. The process will stop at level $(2/3)h$ with $(2/3)h$ comparisons. The deepest rightmost leaf will be then put into the position of the vacant node. Under the worst case, node Y with $(2/3)h$ moves either up to the root with $(2/3)h$ comparisons or down to the leaf with $2*(1/3)h$ comparisons. Therefore the times of comparison of rearranging the heap each time is at most

$$(2/3)h + \begin{cases}(2/3)h \\ 2*(1/3)h\end{cases} = (4/3)h = (4/3)\log_2(j-1)$$

Here, we can obviously see why we choose $(2/3)h$ is that, whether node Y moves up or down, the times of comparison of rearranging the heap each time is not more than $(4/3)h$. Otherwise, no matter what the number we choose is greater or smaller than $(2/3)h$, the times of comparison will be greater than $(4/3)h$ when node Y moves up or down.

So, the time it requires for the **for** loop to perform is

$$\sum_{j=2}^{n} ((4/3)\log_2(j-1) \doteq (4/3)n \log_2 n.$$

Thus the time complexity of the algorithm *NEWheapsort* is $T(n) = 0(n) + (4/3)n \log_2 n$.

In general, node Y will move neither up too high nor down too low. At any case, the efficiency of it is two times as high as that of *HEAPSORT*. It is clear that it has better performance and runs at better case. Further, it fundamentally solves the problem that Floyd suggested.

G. XUNRANG* and Z. YUZHANG
Computer Science Department, Shanghai University of Science and Technology, Shanghai, PR China

* To whom correspondence should be addressed.

References

1. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass., 86–92 (1975).
2. Sara Baase, Computer Algorithms: Introduction to Design & Analysis. Addison-Wesley, Reading, Mass., 60–73 (1978).
3. D. E. Knuth, The Art of Computer Programming, Vol. 3, 'Sorting & Searching'. Addison-Wesley Publishing Company, Inc., 145–149, 158, 618 (1973).
4. E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms. Computer Science Press, Inc., 61–70 (1978).

Announcements

10–13 SEPTEMBER 1990

ETH ZURICH

Awards and computation race at CONPAR 90, VAPP IV

The Awards Committee will award one or two prizes and a Plaque of Recognition in each of four categories:
(1) Best technical contribution
(2) Excellent presentation (based on nomination by the audience)
(3) Best visually animated presentation of concurrency and/or parallelism
(4) The fastest computer solution to a (highly?) parallel problem.

The committee anticipates announcing the awards and the results of the competition during the last session of the conference on Thursday 13, September 1990. At that session, selected animations will be shown, and participants in the computation race will present selected descriptions of their projects.

The Awards Committee: J. Nievergelt (chair), M. Annaratone, J. Dongarra, I. Duff, W. Haendler, H. Jordan, P. Kropf, E. Rothauser, H. Simon, J. Staunstrup, P. Stucki.

We solicit contributions in the area of **Visualization of Concurrent Processes at CONPAR 90, VAPP IV** ETH Zurich, 10–13 September 1990.

The Awards Committee is looking for original research and development work in the area of Visualization of Concurrent Processes. Processes are said to be concurrent or simultaneous or parallel if they co-exist in time. Contributions should be animated, esthetically pleasing, provide 'aha-effects', and have their origin in a field of application such as parallel circuit and architecture design, parallel geometric algorithms, pattern recognition, artificial neural nets, particle motion simulation, cellular automata, etc. The expected length of the individual contributions should be in the range of 30 to 240 seconds of running video or real-time display on a workstation.

The Awards Committee will review submissions and select pieces to show at the conference. The best contribution will be awarded a CONPAR 90/Vapp IV Animation Prize.

Contributions should be recorded on PAL or NTSC video tape (U-Matic low-band or VHS), or be ready for display on a workstation at the conference. Contributions must be submitted by 1, July 1990 to:
Prof. Peter Stucki, University of Zurich, Department of Computer Science, Multimedia Laboratory, CH-8057 Zurich, Switzerland. Tel: XX41-1-257-4350. Fax: XX41-1-257-4343. This address will also provide further information if required.

Computation race at CONPAR 90, VAPP IV

Parallel solution of a jigsaw puzzle.

Have you ever put together a complicated jigsaw puzzle with the help of friends? The first two or three help a great deal – one can start assembling the blue sky, another one the green forest, and you put together the macropieces they have prepared. But how many friends can be of help? There will be a point of diminishing return when those in charge of the sky and the sea start hunting for the same blue pieces. You can of course, let each one work on his own copy of the puzzle, but how much faster will they reach the goal than a single puzzle solver? Jigsaw puzzles are amenable to the standard techniques for exploiting parallelism, and appear to exhibit all the difficulties commonly encountered. They may be a good test problem for honing our speed-up skills.

We propose a kind of jigsaw puzzle built on numbers and equations, called a 'number-cross', shown in the following example. Reading from left to right along any row, or top to bottom along any column, we see strings separated by blanks. Each string represents a single-digit constant or a valid sequence of equations that involves single-digit constants. The picture below shows to puzzles on a 7 by 7 array. Each puzzle is given by prescribing 49 pieces; we exhibit one solution for each puzzle.

16 blanks

2 '+' 0 '0'
2 '-' 4 '1'
1 '*' 6 '2'
1 '/' 2 '3'
3 '=' 3 '4'
2 '5'
1 '6'
2 '7'
1 '8'
3 '9'

	1		4		9	
7	=	4	+	6	/	2
	2		5		3	
2	*	4	=	9	-	1
	3		7		1	
8	-	1	+	2	=	9
	5		2		2	

24 blanks

3 '+'
6 '='
8 '0'
8 '1'

			0			
		1	=	1		
	1	+	0	=	1	
0	=	0	+	0	=	0
	1	=	0	+	1	
		1	=	1		
			0			

In a programming notation, an N-cross might be given as follows:

```
constant = ..;                {n < 238}
type piece =' ', '+',         {types of pieces to
'-', '*', '/', '=','0' ..     fill in the puzzle}
'9';
var puzzle: array[1
..n, 1..n] of piece;
var pieces:                   {how many pieces of
array[piece] of               each type}
integer;
```

The array 'pieces' specifies how many pieces of each type there are: how many blanks, how many '+'s, how many digits '1', etc. Your program receives the initialized array 'pieces', starts a clock, fills 'puzzle' to represent a solution, and computes the puzzle-solving time by stopping the clock before outputting the results.

Let us call a string (of pieces) a 'max string' if it meets all the following requirements:
- it runs left-to right, or top-to-bottom,
- it is bounded by blanks or array boundaries,
- it contains no blanks.

A solution is characterized by the following requirements:
- Every piece (as counted in the array 'pieces') has been used to fill exactly one entry of 'puzzle',
- Every max string represents either an integer constant, or a sequence of correct integer equations that involve parentheses-free expressions over constants,
- The only constants are 0 .. 9 (unsigned, single decimal digit).

Conventions:
'/' is integer division where the remainder must be zero. (e.g. 7/3 is not allowed)
'+' and '−' are binary operators; they may not be used as a unary plus or minus.
Operators are evaluated left to right or top to bottom, with usual precedence.
Example of a sequence of equations: 3 = 1 + 2 = 9/3

A program to solve an N-cross accepts as input a data set D consisting of 17 non-negative integers: the puzzle size n, and the 16-element array 'pieces'. The program may not accept any other input.

This race is for real time. Everyone competes for the fastest solution regardless of alogrithm, software, or hardware used. At a conference on parallel computation, we would expect someone to do better on parallel hardware than can be done on a personal computer. We'll see, good luck.

Rules of participation:

1. An 'entry' to the computation race consists of a team of 1 or more people, a set of hardware (including distributed systems), one program, and one run only for each data set.
2. The same person may participate in at most 3 entries.
3. For each entry, we encourage participants to provide the following infor-