THE GENERALIZED TOWERS OF HANOI FOR SPACE-DEFICIENT COMPUTERS AND FORGETFUL HUMANS

Timothy R. Walsh, Department of Computer Science, UQAM
P.O. Box 8888, Station A, Montreal, Quebec, Canada H3C-3P8

Abstract: The Towers of Hanoi puzzle was generalized by F. Scarioni and M. G. Speranza, and independently by M. C. Er, to allow the initial position of the rings to be any legal position, and then further generalized by A. M. Hinz to allow both the initial and final position of the rings to be any legal position. Their solutions require the storing of at least one size-n array of data, where n is the number of rings, in addition to whatever data is used to represent the problem (the current position of the rings and, in the most general case, the destination peg for each ring). We present a solution in which both the amount of extra data that has to be stored and the time needed to decide on each move except the first one are independent of the number of rings. For the sake of forgetful humans we also present a version in which the extra data has to be updated only O(n) times so that it can be written down and consulted when needed.

0. Legal positions and legal moves

In the classical Towers of Hanoi puzzle and all of the generalizations we consider here, there are three pegs, labelled A, B and C, and n rings of different sizes, labelled 1,2,...,n in increasing order of size. In a *legal position* of the rings, the rings are stacked on the three pegs so that no ring sits on top of a ring smaller than itself. In the notation we will be using here, the legal position for 6 rings in which the odd-numbered rings are on peg A and the even-numbered rings are on peg B will be denoted by the following line:

```
1 3 5 A|       2 4 6 B|              C|
```

A *legal move* takes the topmost ring of some non-empty peg and puts it on a peg which does not contain any ring smaller than the ring which is being moved. In the notation we will be using here, the move which takes the topmost ring of peg A and puts it on peg B will be denoted by AB. Starting from the position denoted by the above line, AB is a legal move, as are AC and BC, but BA is not, because this move would put ring 2 on top of ring 1, nor is CA or CB, because peg C is empty.

1. From a single peg to a single peg

In the original Towers of Hanoi problem, all the rings are initially stacked on a single peg, called the *source peg*, and have to be stacked on another peg, called the *destination peg*; the remaining peg is called the *spare peg*. The classical non-recursive solution to this problem, which was first proved to be correct by P. Buneman and L. Levy [1], can be represented by the following set of rules for moving the rings.

First-move rule (source-to-destination version): *If n is odd, then move ring 1 onto the destination peg; otherwise move ring 1 onto the spare peg.*

Next-ring-to-move rule: *Never move the same ring twice in a row. Thus, if you have just moved ring 1, then either all the rings are now stacked on the destination peg and you are done or else there is one legal move to make: move the second-smallest topmost ring onto the peg not containing ring 1. And if you have not just moved ring 1, move it.*

Cyclic-peg-order rule: *Ring 1 always moves from peg to peg in the same cyclic order (established by the first-move rule).*

To solve the puzzle by hand, follow the first-move rule and write down the cyclic order of the pegs, and then apply the next-ring-to-move rule and the cyclic-peg-order rule (when it is applicable) until all the rings have been stacked on the destination peg. The second-smallest topmost ring is easily identified in O(1) time by examining the topmost ring on each peg. The only piece of information which must be updated and is not obtainable by examining the topmost ring on each peg is *whether you have just moved ring 1*. It is infuriatingly easy to forget this bit of information when you're tired or interrupted, and frustratingly wasteful to update it on paper after every move.

Fortunately there is another rule which can either replace the cyclic-peg-order rule or be used in conjunction with it to make it possible to decide in O(1) time whether ring 1 should next be moved [2].

Even-target rule: *Ring 1 should be moved on top of the second-smallest topmost ring if and only if this ring has an even label.*

For example, suppose the source peg is A, the destination peg is B, and the current position of the rings is

```
1 2 A|          3 6 B|          4 5 C|.
```

Since n=6, which is even, the first move was AC, establishing the cyclic peg order ACBA. Moving ring 1 according to the cyclic-peg-order rule would put it on peg C, in accordance with the even-target rule; so you know that you should make this move. The rings are now in the position

```
2 A|          3 6 B|        1 4 5 C|.
```

Moving ring 1 according to the cyclic-peg-order rule would put it on peg B, violating the even-target rule; so by the next-ring-to-move rule the next move is AB. The rings are now in the position

```
A|        2 3 6 B|        1 4 5 C|.
```

Once again, the cyclic-peg-order rule and the even-target rule both say to move ring 1 onto peg B. Continuing in this way, you can complete the solution without ever having to write down anything more than once (the cyclic peg order) or remember anything but the rules.

Of course, a computer has no problem storing in a single variable the information as to whether it is ring 1 which is to be moved; so either the cyclic-peg-order rule or the even-target rule can be used. The former has the advantage that the second-smallest topmost ring need be located only every second move; the latter has the advantage that the cyclic peg order need not be stored. In order to be able to locate the peg containing the second-smallest topmost ring in O(1) time, the topmost ring of each peg is stored in a size-3 array Top, and the ring beneath each ring is stored in a size-n array Beneath [3]; in either case, an empty peg is represented by n+1. For the last position above, Top would be (7,2,1) and Beneath would be (4,3,6,5,7,7). Examining the array Top one can determine in O(1) time that ring 1 is on peg C and that the second-smallest topmost ring is ring 2, on peg B, and then the correct next move - CB - can be made in O(1) time, changing Top to (7,1,4) and Beneath to (2,3,6,5,7,7). If the second-smallest topmost ring is greater than n - and this too can be determined in O(1) time - then all the rings are stacked on a single peg.

2. From an arbitrary legal position to a single peg

The first generalization - starting from an arbitrary legal position and ending on a designated destination peg - was made by F. Scarioni and M. G. Speranza [4], and independently by M. C. Er [5]. Recursive solutions appear in both of these works; non-recursive solutions were found by Er [3], T. R. Walsh [6] and A. M. Hinz [7].

All the non-recursive solutions depend upon the calculation of target pegs for each of the rings, using the target-calculation algorithm below.

Target-calculation algorithm: *The target for ring n is the destination peg. For each ring n-1,...,2,1: if ring i+1 is on its target peg, then this peg is also the target peg for ring i, and otherwise the target peg for ring i is the peg other than the peg containing ring i+1 and the target peg for ring i+1.*

For example, if the destination peg is B and the current position of the rings is

```
A|       2 3 6 B|       1 4 5 C|,
```

then the target pegs for all the rings are calculated as follows:

ring	target	location
6	B	B
5	B	C
4	A	C
3	B	B
2	B	B
1	B	C

The non-recursive algorithm in [7] for stacking the rings on the destination peg is essentially the one shown in Figure 1. It uses the array Target to store the target pegs and Peg to store the current peg containing each ring.

```
Find Target[i] for each i;
FOR i:=1 TO n DO
  IF Peg[i]≠Target[i] THEN
    Peg[i]:=Target[i];     (* that is, move ring i onto its target peg *)
    IF i>1 THEN            (* the smaller rings are all stacked on Target[i-1] *)
      Stack all the smaller rings on Peg[i]
                                        (* using the cyclic-peg-order rule *)
    END IF;
  END IF
END FOR.
```

Figure 1
Er-Hinz iterative algorithm for stacking n rings on the destination peg from any legal position.

The iterative algorithm in [3] is essentially the same, except that an explicit implementation is given, using the arrays Top and Beneath as well as Peg and Target so that each move can be made in O(1) time. The array Target is constantly updated to keep ring 1 moving in the correct cyclic direction; it would have sufficed to update the cyclic direction for ring 1 for each new value of i.

Examining this algorithm one can see that it obeys the next-ring-to-move rule: it is obeyed during the execution of each instruction 'Stack all the smaller rings on Peg[i]', and the moves of ring 1 that end one execution of this instruction and begin the next one are separated by a move of a larger ring made by the instruction 'Peg[i]:=Target[i]'. All the moves of ring 1 except 'Peg[1]:=Target[1]', which would necessarily be the first move made, are made during the execution of the instruction 'Stack all the smaller rings on Peg[i]' for some i, and so they obey the even-target rule as well. It follows that these two rules determine every move except the first one! The first move is 'Peg[i]:=Target[i]' for the smallest ring i which is not on its target peg, if there is one, leading to a new first-move rule:

First-move rule (general-to-destination version): *If ring 1 is not on its target peg, then move it there; otherwise follow the next-ring-to-move rule for moving a larger ring.*

As an example of this rule, we recall that from the position

```
A|       2 3 6 B|       1 4 5 C|
```

the next move required to stack the rings on peg B (assuming they began on peg A) was CB. As we have shown above, the target-calculation algorithm puts the target for ring 1 on peg B, so that the first-move rule would lead to the same move CB.

If the initial position happens to be the one with all the rings stacked on the source peg, then the target-calculation algorithm puts the target for ring 1 on the destination peg if n is odd and on the spare peg otherwise, so that the source-to-destination version of the first-move rule falls out as a special case of the general-to-destination version.

To stack the rings on the destination peg from any legal position, find the target for ring 1 using the target-calculation algorithm, then determine the first move using the general-to-destination version of the first-move-rule, and then apply the next-ring-to-move rule and the even-target rule (when applicable) until all the rings are stacked on the destination peg. This is the algorithm which appears in [6], where it was derived from the recursive algorithm of [4]; a correctness proof that is independent of any other algorithm appears in [8].

This algorithm can be implemented using only the arrays Top and Beneath. It does not require storing the array Target: during the application of the target-calculation algorithm, a single variable (which contains the target peg of the current ring i) is initialized to the destination peg, is updated for each new value of i, and ends up being the target peg for ring 1, which is all that is required to apply the first-move rule. Since the rings are scanned from largest to smallest by the target-calculation algorithm, their position cannot be initially represented by the arrays Top and Beneath; however, the array Beneath can do double duty, playing the role of the array Peg until the target for ring 1 has been determined. For example, for the initial position

```
A|       2 3 6 B|       1 4 5 C|,
```

the array Beneath would initially be (C,B,B,C,C,B), with the pegs A, B, and C coded by convenient numbers like 0, 1 and 2. Then, once the target peg for ring 1 has been

established, another pass through the rings from largest to smallest suffices to convert the representation of the position of the rings to agree with the meaning of the words top and beneath: Top becomes (7,2,1) and Beneath becomes (4,3,6,5,7,7). In this way, only O(1) extra space is needed and every move except the first one can be executed in O(1) time.

Of course, this is all well and good for a computer which can remember whether it has just moved ring 1, but not for a forgetful human. Examining the algorithm of Figure 1 one can see that before the instruction 'Peg[i]:=Target[i]' has been executed, only rings smaller than ring i have ever been moved. Keep track of one number: the label on the largest ring which has so far been moved (the variable i in the algorithm of Figure 1). This number increases whenever the instruction 'Peg[i]:=Target[i]' gets executed. If i>1, then the smaller rings which were stacked on one peg get stacked on another, so that you can use the memory-avoidance tricks outlined for that case. The first move following 'Peg[i]:=Target[i]' is a move of ring 1 which obeys the even-target rule; this move establishes a cyclic order on the pegs which will be valid during the execution of 'Stack all the smaller rings on Peg[i]', after which either all the pegs are stacked on the destination peg or else a ring larger than ring i becomes free to move. Write down this cyclic order; then the cyclic-order rule can be compared with the even-target rule to determine whether ring 1 is to be moved next - *until a ring larger than ring i is free to move*. At this point, this larger ring gets moved, i increases, and a new cyclic order is established by the following move. The amount of information that needs to be written down is greater than that which would be required to record whether ring 1 has just been moved, *but it need be done only n times*!

For example, suppose you are moving the rings onto peg B from the position

```
A|        2 3 6 B|        1 4 5 C|.
```

The first move has already been calculated: since the target peg for ring 1 is B, the first move is CB. Now i, the largest ring that has been moved, is 1; write it down, but no cyclic order need be written down. After you make this move, the position becomes

```
          A|     1 2 3 6 B|          4 5 C|.
```

Already a larger ring is free to move - ring 4. Do so and replace 1 by 4. The position is now

```
        4 A|     1 2 3 6 B|            5 C|.
```

By the next-ring-to-move rule, you must now move ring 1, and by the even-target rule, it moves to peg A:

```
        1 4 A|       2 3 6 B|            5 C|.
```

This establishes the cyclic order BACB on the pegs; write it down. Now, if the telephone rings and you forget whether you've just moved ring 1, observe that the cyclic-order rule is now incompatible with the even-target rule, so that ring 1 must not be moved yet.

3. From a single peg to an arbitrary legal position

Now suppose that the rings begin on a single peg - the source peg - and are to be moved to some arbitrary legal position. Hinz' solution to this problem [7] is: find the sequence of moves that would stack the rings on the source peg if they started in their designated final position, and then reverse that sequence of moves. The algorithm of Figure 2 performs the same sequence of moves, but its description is somewhat easier to program and to execute by hand.

```
FOR i FROM n BY -1 TO 1 DO
  IF ring i is not on peg Dest[i] THEN
    Stack all the smaller rings          (* which are now on ring i *)
    on the 'other' peg besides Dest[i];
    Move ring i to Dest[i]
  END IF
END FOR.
```

Figure 2
An algorithm for taking n rings which are stacked on a source peg
and putting each ring i on a specified peg Dest[i].

To prove that this algorithm works, we use the loop invariant: *all the rings from n down to i+1 are on their destination pegs, and all the rings from i down to 1 are stacked on a single peg*. At the beginning, when i=n, this invariant is true because all the rings begin

stacked on the source peg. Suppose it to be true for a given i. If ring i is on its destination peg, then all the rings from n down to i are on their destination pegs, and all the rings from i-1 down to 1 are stacked on a single peg; the loop invariant is retrieved when i decreases to i-1. If ring i is not on its destination peg, stacking all the smaller rings on the 'other' peg arranges that all the rings from i-1 down to 1 are stacked on a single peg and makes it possible to put ring i onto its destination peg, and again the loop invariant is retrieved when i decreases to i-1. The algorithm terminates when i drops to 0, at which point the loop invariant says that all the rings are on their destination pegs, as required.

The array Dest is part of the problem; so it cannot be considered extra storage. The arrays Top and Beneath are sufficient to represent the position of the rings: to determine whether ring i is on the peg Dest[i], we use the fact that all the rings from i down to 1 are on the same peg, so that all we have to do is examine the topmost rings on each peg to determine the location of ring 1 and then compare it with Dest[i]. The extra storage is thus O(1). It may happen that several rings in a row will turn out to be on their destination pegs, so that several decrements of i will be made before the next move; however, the failure of this algorithm to take only O(1) time per move is limited to O(n) operations performed throughout the entire execution of the algorithm - those which test whether each ring i is on Dest[i].

To execute this algorithm by hand, write down the final position of the rings, and then cross each ring off when it is either moved to its destination peg or turns out to be there already. Whenever you get to a ring i which has not yet been crossed off, if i=1 then move ring 1 to its destination peg and you are done, and otherwise the next move is to *move ring 1 to the destination peg of ring i if i is odd and to the 'other peg' if i is even* (this is the first move in the instruction 'Stack all the smaller rings on the 'other' peg besides Dest[i]'). This move establishes a cyclic order on the pegs; write it down and compare it with the even-target rule to determine the next move until ring i gets moved to its destination peg.

For example, suppose you want to move the rings from peg B to the position

```
1 4 A|        2 3 6 B|            5 C|.
```

The initial position is

```
          A| 1 2 3 4 5 6 B|              C|.
```

Ring 6 is already on its destination peg; cross it off the final position. Ring 5 is not on its destination peg, and since 5 is odd, ring 1 must now be moved to peg C. This establishes the cyclic order BCAB on the pegs, so that by comparing this cyclic order with the even-target rule you can move the pegs with nothing to remember until the rings reach the position

```
1 2 3 4 A|            6 B|            5 C|.
```

Now ring 5 has been moved to its destination peg and ring 4 turns out to be already there; so these rings are crossed off and you get to ring 3, which needs to be moved from peg A to peg B.

4. From one arbitrary legal position to another

The final problem we consider is moving the rings from one legal position to another. Hinz solves this problem by presenting two sequences of moves [7]. Let ring i be the largest ring that is not on its destination peg Dest[i]. Ring i is moved onto Dest[i] in either one move or two. Before ring i can be moved, all the smaller rings must be stacked on the appropriate 'other' peg. Before the first move of ring i, the smaller rings are in an arbitrary legal position; they must be stacked on a single peg. Before the second move of ring i, if there is one, the smaller rings are stacked on one peg and must be stacked on another one. Once ring i is on Dest[i], the smaller rings must be taken from the single peg on which they are now stacked and moved to their destination pegs. One of these two sequences of moves (moving ring i once or twice) is minimal, and since one can compute the length of each sequence of moves from the formula in [7] for the number of moves required to stack the rings from an arbitrary legal position to a single peg (which we published six years earlier

[6]), one chooses the shorter sequence of moves and then executes the necessary algorithms.

If each of the component parts of the Hinz algorithm is implemented using the algorithms proposed here, the entire algorithm uses O(1) extra storage and each move takes O(1) time, apart from the O(n) operations required to decrease i from n to 0 during the execution of the algorithm of Figure 2, and can be executed by hand with O(n) updating of extra information.

At this point it becomes necessary to discuss ways of computing in advance the number of moves that each of the component algorithms will make.

Moving n rings from one peg to another takes 2^n-1 moves ([1] and elsewhere).

The following formula for the number of moves required to move n rings from an arbitrary legal position to a destination peg was proved both in [6] and in [7]: it is the binary number $b_n...b_2b_1$ where $b_i=0$ if and only if Peg[i]=Target[i]. It is possible to compute this binary number while computing the target for ring 1 without storing the array Target: as soon as the single variable that contains Target[i] for the current value of i is updated it is compared with Peg[i] to determine if the next bit b_i is equal to 0 or 1. Working by hand, the entire binary number is written down and then evaluated. Working by computer, the bits are not stored. Instead, a variable is initialized to 0, doubled for each value of i and then increased by 1 if ring i is not on the current target peg.

To compute the number of moves required to move n rings from a source peg to an arbitrary legal position, Hinz [7] proposes computing the array Target which would result from moving the rings from their final position to the source peg and then computing the binary number from the formula. We propose a simpler algorithm. Examining Figure 2, we can see that if ring i is not on its destination peg, then $2^{i-1}-1$ moves will be made to stack the smaller rings, and 1 more to move ring i, for a total of 2^{i-1}, and otherwise no moves will

be made. The binary number of moves is thus $b_n...b_2b_1$, where b_i=0 if and only if ring i is already on its destination peg when the test is made. The algorithm of figure 2 can be modified so that it keeps track of the peg on which the smaller rings will be stacked instead of actually moving the rings and so that it writes down or evaluates the resulting binary number (see Figure 3).

```
StackPeg:=source; m:=0;     (* StackPeg is the destination for a stack of rings *)
FOR i FROM n BY -1 TO 1 DO
  m:=m*2;
  IF StackPeg≠Dest[i] THEN (* bi=1 else bi=0 *)
    StackPeg:=the peg other than StackPeg or Dest[i];
    m:=m+1
  END IF
END FOR.
```

Figure 3

An algorithm for finding m, the number of moves needed to execute Figure 2.

For example, suppose that the rings begin in the position

```
   1 4 5 A|          2 6 B|            3 C|
```

and are to end in the position

```
     3 6 A|        1 4 5 B|            2 C|.
```

Suppose you decide to move ring 6 from peg B to peg A in one step. You must first stack all the smaller rings on peg C. Here is a trace of the computation of the target for ring 1 together with the number of moves required.

```
ring target location bit  moves
  5      C        A      1      1
  4      B        A      1      3
  3      C        C      0      6
  2      C        B      1     13
  1      A        A      0     26
```

One more move puts ring 6 on peg A. After 27 moves the position will be

```
            6 A|              B|    1 2 3 4 5 C|.
```

Now you must move rings 1 through 5 to their final positions. A trace of the computation of the number of moves follows.

```
ring stackpeg destination bit  moves
  5       C          B       1      1
  4       A          B       1      3
  3       C          A       1      7
  2       B          C       1     15
  1       A          B       1     31
```

It will take 58 moves to put the rings on their destination pegs.

Now suppose you decide to move ring 6 from peg B to peg A in 2 steps. Before moving ring 6 from peg B to peg C you must stack all the smaller rings on peg A. The trace follows.

```
ring target location bit  moves
  5      A        A      0      0
  4      A        A      0      0
  3      A        C      1      1
  2      B        B      0      2
  1      B        A      1      5
```

One more move puts ring 6 on peg C. After 6 moves, the position will be

```
   1 2 3 4 5 A|              B|           6 C|.
```

It will take 31 more moves to stack rings 1 through 5 on peg B and one more move to put ring 6 on peg A. After 38 moves the position will be

```
            6 A|    1 2 3 4 5 B|             C|.
```

Finally the rings must be brought to their final position, which is

```
         3 6 A|       1 4 5 B|           2 C|.
```

```
ring stackpeg destination bit moves
```

```
5       B       B       0       0
4       B       B       0       0
3       B       A       1       1
2       C       C       0       2
1       C       B       1       5
```

It will take only 43 moves to put the rings in their final position, compared with 58 moves if you move ring 6 only once. The target for ring 1 has already been calculated - it's B; so the first move to make is AB.

I have written a computer game for the Macintosh 68000 series which implements the algorithms presented here. The only arrays used are Top, Beneath and Dest (Dest is used only when the rings are to be moved to an arbitrary legal position), and except for the display and the O(n) extra operations performed during the execution of the algorithm of Figure 2, each move is made in O(1) time. You get to choose between the four versions of the puzzle (single or multiple source, single or multiple destination), the number of rings (up to 10), whether you or the computer sets up the initial and final position and whether you or the computer moves the rings. If you let the computer set up the initial and final position and you move the rings yourself, then you can score points depending upon the puzzle version, the number of rings, the number of moves made in excess of the minimum and the number of illegal moves attempted. Send me a diskette and enough money to cover postage and you'll receive the executable file, the source code (about 900 lines, in C) and a copy of this paper. If you score the maximum 100 points, save the output file and send it to me and you'll be duly inducted into the Timewasters' Hall of Fame!

REFERENCES

[1]: P. Buneman and L. Levy, The towers of Hanoi problem, Information Processing Letters 10 (1980), p. 243-244.

[2]: T.R. Walsh, The towers of Hanoi revisited: moving the rings by counting the moves, Information Processing Letters 15 (1982), 64-67.

[3]: M.C. Er, An iterative solution to the generalized towers of Hanoi problem, BIT 23 (1983), 295-302.

[4]: F. Scarioni and M.G. Speranza, A probabilistic analysis of an error-correcting algorithm for the towers of Hanoi puzzle, Information Processing Letters 18 (1984), p. 99-103.

[5]: M.C. Er, The generalized towers of Hanoi problem, J. Inform. Optim. Sci. 5 (1984), p. 89-94.

[6]: T.R. Walsh, A case for iteration, Proceedings of the 14th Southeastern Conference on Computing, Graph Theory and Combinatorics, 1983, Congressus Numerantium 40 (1983), p. 38-43.

[7]: A.M. Hinz, The tower of Hanoi, L'Enseignement Mathématique 35 (1989), p. 289-321.

[8]: T.R. Walsh, A simple sequencing and ranking method that works on almost all Gray codes, Research Report No. 243, Department of Mathematics and Computer Science, University of Quebec in Montreal, April 25, 1995.