Oracle Cyclic Sort: improved sorting with minimum number of writes

Matteo Olivato* Romeo Rizzi† Massimo Cairo‡

University of Verona University of Verona University of Verona

matteo.olivato@studenti.univr.it romeo.rizzi@univr.it massimo.cairo@univr.it

May 8, 2017

Abstract

The CyclicSort *algorithm takes $\Theta(n^2)$ comparisons in order to sort an array v of n integers.*

Even in the best case, when given an already sorted array v, CyclicSort *still employs at least $n(n-1)$ comparisons.*

Yet, until now, CyclicSort *was the only sorting algorithm that, assuming $O(1)$ working memory, could guarantee the minimum possible number of writes on the array v.*

And, since this property may turn out to be the most significant factor in certain niche applications, Cyclic Sort *retains its own (though modest) place in the Olympus of sorting algorithms.*

We propose smarter and more performant algorithms with this property, and a first rough analysis of the issue of minimizing the number of write operations within the array v to be sorted.

One basic tool in our approach is an oracle subroutine which returns a position i such that element $v[i]$ is not at its correct place, or a check that array v is already sorted. This subroutine offers a nice decomposition opportunity for this problem.

I. Introduction

The problem of sorting with the minimum number of writes was studied for the first time by the Cycle Sort [2] algorithm. The core of the algorithm is based on the concept of in-place cyclic permutation [1] which allows to obtain the theoretical minimum number of writes.

The Cycle Sort provides a solution for this problem being able to identify every different permutation cycle to run to sort the array. The OCS algorithm is suitable for environments with little memory, as it uses costant space, and environments where memory writes have high cost, in terms of time, energy or memory life. The main problem of the algorithm in terms of complexity is that it can't recognize a global ordered element when it arrives on it, so it must compare the element with all its successors in the last part of the array. This involves a high number of unnecessary comparisons which increase the complexity of the algorithm.

Our goal is to pass quickly on elements already sorted and find an elements of a permutation cycle until the array is not sorted.

For simplicity our case study takes into consideration the sorting problem for arrays of distinct integers.

II. Preliminaries and notation

The task is to sort an array v of n elements taken from a totally ordered universe. To ease our exposition, we assume that all elements in v differ, so that the intended output vector $\overline{v}$ is unique. Also, the inverse maps v^{-1} and $\overline{v}^{-1}$ are well-defined. For $i = 1, \ldots, n$, position i and element $v[i]$ are called *fit* if $v[i] = \overline{v}[i]$, that is, if element $v[i]$ is already in its final position. The specificity of the Cycle Sort algorithm is

*A thank you or further information

†Corresponding author

‡Corresponding author

that is uses the minimum possible number of writes: no fit element gets ever moved and, when an unfit element $v[i]$ is moved, it must be placed directly in its final position $\overline{v}^{-1}[v[i]]$ and becomes fit. If we consider the graph having the set of positions $\{1,\ldots,n\}$ as its nodes, and with arc set $\{(i,\overline{v}^{-1}[v[i]]) : i = 1,\ldots,n\}$, then it comprises a family of directed cycles, with each node belonging to exactly one cycle of the family (the fit positions are those holding self-loops); this is the well know representation of a permutation as a family of node-disjoint covering cycles. Given our commitment to minimize the number of writes, i.e., to achieve precisely one write/movement for every unfit element, then, when we are about to move an unfit element $v[i]$, we are due to settle all the elements of its cycle.

We delegate this task to the following procedure, called FixCycle, which encapsulates the heart of the previous Cycle Sort algorithm (but will be managed more subtly and efficiently in our new proposal).

The FixCycle procedure is given the index of an element to place in its final position. To find its final position, say i, it counts all the smaller elements present in the array (spending $n-1$ comparisons). Then, it places the element in position i, and it recurs to place the element previously in position i.

The procedure stops when it finds an element which must be moved into the initial position, closing the cycle. The pseudo-code of this procedure is provided for reference in Algorithm 1.

Only procedure FixCycle is allowed to write elements in the array, so the final number of writes is precisely the number of writes made by this procedure, which is known to be optimal for the purpose of sorting an array. Our purpose is to improve on the number of comparisons.

Algorithm 1 FixCycle: Ciclic permutation sorting procedure

```
1:  procedure FixCycle(i)
2:      val ← v[i]
3:      do
4:          k ← 1
5:          for j ← 1 to n do
6:              if j ≠ i and v[j] < val then
7:                  k ← k + 1
8:              end if
9:          end for
10:         temp ← v[k]
11:         v[k] ← val
12:         val ← temp
13:     while k ≠ i
14: end procedure
```

Lemma II.1. *A sorting algorithm performs the minimum number of writes within vector v if all its writes are performed within calls to the* FixCycle *procedure.*

III. FixCycle version of Cycle Sort

Algorithm 2 CycleSort: Array sorting algorithm producing the minimum number of array writes

```
1: procedure CycleSort
2:     for i ← 1 to n do
3:         FixCycle(i)
4:     end for
5: end procedure
```

IV. The Oracle Cyclic Sort algorithm

Our goal is to minimize the number of redundant comparison when there are few elements which are unfit. Specifically, since the FixCycle procedure performs $n-1$ comparison when it is given a position which is already fit, we want to avoid calling FixCycle at all on fit elements. To this end, we devise a technique to find an unfit element in the array.

Theorem IV.1. *If* $v[0] \leq v[1] \leq ... \leq v[i]$ *and* $v[i] > v[i+1]$ *then* $v[i]$ *is not fit.*

Proof. Suppose by contradiction $\overline{v}^{-1}[v[i]] = i$. Then, there exist exactly i elements in v strictly smaller than $v[i]$. However, we have $i+1$ such elements, i.e., $v[0], \ldots, v[i-1], v[i+1]$. □

This Theorem always applies for some i, unless v is already sorted. Indeed, it is sufficient to scan the array from left to right, and return the first index i such that v [i] > v [i + 1] . This sweep is conducted by

Observe that Theorem IV.1 always applies for some i, unless v is already sorted. Indeed, it is sufficient to scan the array from left to right, and return the first index i such that $v[i] > v[i+1]$. This sweep is conducted by the function UnfitOracle, which either returns the index of an unfit element, or determines that the array is already sorted, in which case it returns -1. The pseudo-code of this function is given in Algorithm 3.

Algorithm 3 UnfitOracle: Oracle able to find the index of an unfit array element

```
1: function UnfitOracle
2:     for i ← 1 to n − 1 do
3:         if v[i] > v[i + 1] then
4:             return i
5:         end if
6:     end for
7:     return −1
8: end function
```

Function UnfitOracle, combined with the FixCycle procedure, yields our sorting algorithm. This algorithm runs UnfitOracle repeatedly, and, as long as there is an unfit element in the array, it fixes the position of that element along with its permutation cycle by calling FixCycle. This algorithm is realized by the procedure OracleCyclicSort, whose pseudo-code is given in Algorithm 4.

Algorithm 4 OracleCyclicSort: Array sorting algorithm producing the minimum number of array writes with improved comparisons

```
1: procedure OracleCyclicSort
2:     i ← UnfitOracle()
3:     while i ≠ −1 do
4:         FixCycle(i)
5:         i ← UnfitOracle()
6:     end while
7: end procedure
```

V. Complexity and number of writes

We now analize:

i. Total number of array writes.

ii. Complexity of the algorithm.

i. Total number of array writes

Only the FixCycle procedure writes elements into the array v. Moreover, whenever it performs such a write, the element was unfit before the writing and becomes fit with the writting. The total number of writes into the array v is hence $|U(v)|$, which is clearly optimal.

ii. Complexity of the algorithm

The OracleCyclicSort algorithm has a simple pseudocode made of a main loop that calls the UnfitOracle and FixCycle.

The FixCycle performs $n-1$ comparisons in order to find the correct position of an element k. To sort the array v, FixCycle have to find the correct position of every unfit element in the array. Therefore FixCycle have to resolve every array's permutation cycle. Additional comparisons may be necessary just in case, a fit element k would be passed to the procedure FixCycle. Nevertheless, according to the UnfitOracle property and the OracleCyclicSort algorithm, this case will never occur. We can conclude that, if we call u the total number of unfit elements, we have $u * (n-1)$ comparisons.

If we call m the number of ciclic permutations in an array v of size n, we can say that $0 \leq m \leq u/2 \leq n/2$, because one ciclic permutation have to be made at least of two unfit elements then there are at most $n/2$ differents permutation cycles in v. We need less then $(n-1)$ comparison to find a permutation cycle, then a valid upperbound for UnfitOracle procedure is $(u/2) * (n-1) + (n-1)$ comparisons.

The UnfitOracle will ever perform globally less comparisons than FixCycle, so the final complexity is in order of $O(u * n)$. In the best case we have an array already sorted, in fact the OracleCiclicSort main loop ends after the first UnfitOracle execution, so the lowerbound complexity is $\Omega(n)$.

References

[1] AJW Duijvestijn. "Correctness proof of an in-place permutation". In: *BIT Numerical Mathematics* 12.3 (1972), pp. 318–324. URL: http://doc.utwente.nl/85477/1/Duijvestijn72correctness.pdf.

[2] Bruce K. Haddon. "Cycle-sort: a linear sorting method". In: *The Computer Journal* 33.4 (1990), pp. 365–367. URL: http://comjnl.oxfordjournals.org/content/33/4/365.full.pdf+html.