SETL NEWSLETTER #214

March 27, 1980
MICHA SHARIR

## A Strange Sorting Algorithm Inspired by Formal Differentiation

In this note we will describe an application of the metarules given in section 4 of [Sh] in a generalized situation, where the object being constructed is not a set. This application leads (by informal intuitive steps) to a new sorting algorithm.

The problem considered is sorting a given sequence of integers. Let A=(A1,A2...An) be this sequence. A specification of the meaning of sorting might look like

> find a sequence of integers B such that
> B is a permutation of A and B is sorted

We can write a skeleton program to solve this problem as follows: Consider the first conjunct "B is a permutation of A" as an invariant property to be preserved throughout the program, and the second conjunct "B is sorted" as the program's goal. Then we can write

```
B := A;
(while not sorted(B))
     B := f(B);
end while;
```

where f(B) is a 'small' change in B which preserves the invariant conjunct.

For the moment let us ignore the problem of how to split the predicate defining B into an 'invariant' subpart and 'goal' subpart, and how to find a family of invariance-preserving transformations f on B, though these are problems deserving substantial study. (Note that in the set-valued case considered in [Sh] the function f was simply addition of elements to the set object being constructed.) I.e. suppose that we somehow find out that the transformations f can be taken to be swapping of any two elements of B. This put us into a situation quite similar to the one considered in [Sh], namely a complicated predicate

```
P(B) = not sorted(B)
```

is repeatedly computed in a loop in which B changes 'slightly' (although B is not a set). This is precisely the set-up needed for the application of formal differentiation and the other related techniques of [Sh] which aim at making the selection of the transformation f to be applied more intelligent.

```
Rewrite P(B) as
P(B) = exists  (i,j) : i in [1...n], j in [1...n], i<j and Bi>Bj
     =    {(i,j) :....}/= { }
```

Let K(B) denote this set.
Aining to apply the first transformation of section 4 of [Sh] and not bothering for the time being to verify its applicability), we want

to rewrite the program as

```
B := A;
(while K(B) /= { })
    B := f(B) where f is a swap such that
                    K(B) * DK(B,f)/= { }
end while;
```

writing K(B) in the form {x : Q(x,B)}
we can write

DK(B,f) = {x : Q(x,B) & ~ Q(x,f(B)) or ~ Q(x,B) & Q(x,f(B))}

Hence

```
K(B)*DK(B,f) = {x : Q(x,B) & ~ Q(x,f(B))}
    = {(i,j) : i<j & Bi > Bj & f(Bi) <=f(Bj)}
```

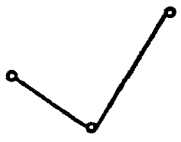
In other words, we want to perform a swap which will cause an inverted pair to become sorted.

Now of course there is always a way to select a swap to satisfy the above selection criterion (K(B) * DK(B,f) = { }), i.e. swap the inverted pair (i,j) itself. However, there also exist other choices for the swap. For example, suppose that there exists an inverted pair i<j & Bi > Bj. We could then swap j with any k for which Bi < = Bk, or swap i with any k for which Bk < = Bj. Although at first sight these swaps seem to be counter-productive it is nevertheless interesting to follow the lead and see what algorithms such a choice might yield.

To do this we will abandon the course of formal considerations and continue in an intuitive informal mode.

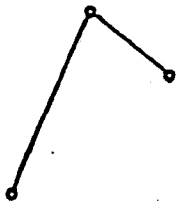
To push things to the extreme, let us consider only swaps which remove at least one inverted pair but which are not swaps of inverted pairs. It is easily seen that there are two such cases

(a) swap j and k, where exists i < j < k s.t. Bj<Bi<=Bk



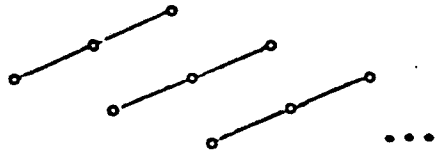
i j k

(b) swap k and i, where k < i < j s.t. Bk<=Bj<B:



k i j

Note that swaps (a) push a conflict 'to the right' whereas swaps (b) push it to the left. This suggests that we try only swaps (a) first, and when there are no more such swaps possible, try only swaps (b). It is an easy exercise to show that after swaps (a) are exhausted, swaps (b) cannot generate a triple (i,j,k) which makes a swap of the (a) type possible. Hence when the two phases of our algorithm terminate, there are no more swaps (a) or (b) possible, and it follows that the sequence being manipulated must now have the following structure:



i.e. a concatenation of non-decreasing subsequences, such that each subsequence has all its elements smaller than those of the preceding subsequence. Such a sequence can be sorted by a linear pass through its elements.

We give an example of this algorithm as applied to the sequence

4 1 5 9 26 3 8 7

I. swaps (a) performed from left to right (the triple suggesting the swap is circled):

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 5 | 9 | 2 | 6 | 3 | 8 | 7 |
| 4 | 5 | 1 | 9 | 2 | 6 | 3 | 8 | 7 |
| 4 | 5 | 9 | 1 | 2 | 6 | 3 | 8 | 7 |
| 4 | 5 | 9 | 6 | 2 | 1 | 3 | 8 | 7 |
| 4 | 5 | 9 | 6 | 8 | 1 | 3 | 2 | 7 |
| 4 | 5 | 9 | 6 | 8 | 7 | 3 | 2 | 1 |

In the last row no more swaps (a) are possible.

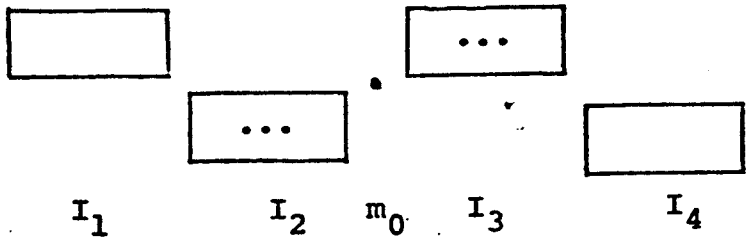
II. swaps (b) performed from right to left

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 5 | 9 | 6 | 8 | 7 | 3 | 2 | 1 |
| 4 | 5 | 9 | 8 | 6 | 7 | 3 | 2 | 1 |
| 4 | 8 | 9 | 5 | 6 | 7 | 3 | 2 | 1 |
| 9 | 8 | 4 | 5 | 6 | 7 | 3 | 2 | 1 |

We have now achieved the situation described above which decomposes the subsequences 9, 8, 4567, 3, 2, 1 and the sorted sequence can now be easily obtained. We will not digress here into a study of the complexity of this algorithm.

Another algorithm that suggests itself is as follows: pick any component Bmo of the sequence; apply swaps (a) from left-to-right to the collection of elements which lie to the right of using mo as the first index of each triple suggesting a swap. (I.e. swap j and k where mo <j<k and Bj < Bmo <= Bk). Similarly perform swaps (b) to the left of Bmo using mo as the last index of each swap-inducing triple. It

is easy to see that the sequence which results from this has the following structure:




where the blocks shown are unsorted but have elements which compare in the manner suggested by the figure. We can thus use a recursive approach which sorts each of the four subsequences I1, I2, I3, I4, and then obtain the final sorted order by

merge (sort(I2),sort(I4)) || [mo] || merge (sort(I1), sort(I3))

This will yield a variant of QUICKSORT.

In the preceding example, suppose that we pick Bmo = 6. Then applying swaps (a) to its right we get

6 3 8 7
6 8 3 7
6 8 7 3

and applying swaps (b) to its left gives us

4 1 5 9 2 6
4 1 9 5 2 6
4 9 1 5 2 6
9 4 1 5 2 6

We thus obtain the sections I1 = 9 I2 = 4 1 5 2 I3 = 8 7 I4 = 3 and the sorted order can be obtained by sorting I2 , I3 separately and then using the above formula. Note: A variant may be obtained by picking mo as the first (or last) index. That would leave only two subsequences to be sorted at each step, and would eliminate the need to merge.

The moral of this little Dijkstra-like happening is that interaction between a formal program-development system and its user could be very fruitful even if applied in a very loose sense, in which the formal rules of the system suggests ideas, or 'algorithm-fragments' which the system user must pick up and develop intuitively . As an example of this I note that the idea of using the above swaps did not come to my mind (and would have probably never done so) except thru application of the formal-differentiation-based transformations of [Sh].

[Sh] Sharir, Micha, "Some Observations Concerning Formal Differentiation of Settheoretic Expressions", N.Y.U. Computer Science Technical Report 16, 1979.