On space-efficient algorithms for certain NP-complete problems*

A. Ferreira
CNRS–Laboratoire de l’Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 69364 Lyon Cédex 07, France

Communicated by D. Perrin
Received July 1990

Abstract

Some recent results claimed the existence of a class of algorithms for certain NP-complete problems, with running time $O(n^{1/4} k n^{3/2})$ and storage requirements $O(k n^{2/3})$, for $2 < k < n$. In this note we show that those results do not hold, implying that an algorithm with time $O(n^{3/2})$ and space $O(2^{3/4})$ is still the best-known solution for such class of NP-complete problems.

1. Problem recall

Let $X$ and $Y$ be two vectors of $n$ elements. A matrix $A$ defined by the cartesian sum $X + Y$ is such that $a_{ij} = x_i + y_j$. If $X$ and $Y$ are sorted then $X + Y$ is a matrix with sorted rows and columns. Based on such a structure, Vyskoč [5] claimed the existence of a class of algorithms for certain NP-complete problems, with running time $O(n^{1/4} k n^{3/2})$ and storage requirements $O(k n^{2/3})$, for $2 < k < n$.

In [2] it was proved that the theorem presented in [5] in order to derive such a class of algorithms was not exact, invalidating Vyskoč’s claim. Another theorem was

*Correspondence to: A. Ferreira, LIP, CNRS, Ecole Norm. Supérieure de Lyon, 46, Allée d’Italie, 69364 Lyon Cedex 07, France. Email: ferreira@lip.ens-lyon.fr.

*This work was partially supported by the PRCs C3 and Math-Info of the French CNRS.

0304-3975/93/$06.00 © 1993—Elsevier Science Publishers B.V. All rights reserved
presented in a corrigendum [6], leading, according to Vyskoč, to a different algorithm that has the same time and space complexities. In this note we want to make it clear that even if the theorem in the corrigendum is exact, it does not help to derive such an algorithm. We therefore conclude, like in [2], that the algorithm in [5, 6] does not have the claimed time and space complexities. Thus the algorithm in [1, 4] (time $O(n 2^{n/2})$ and space $O(2^{n/4})$) is still the best-known solution for the class of NP-complete problems defined in [4].

2. A counterexample

Let $X$, $Y$, $R$, and $S$ be sorted vectors with $n$ elements. The basic idea behind the two-list four-table algorithm proposed in [4] is that the problem of searching a multiset of the form $X + Y + R + S$ can be solved through a reduction to searching a multiset formed by the sum of only two sorted vectors $(X + Y)$ and $(R + S)$ with $n^2$ elements each. Such a reduction is obtained through the successive generation of the elements of each new vector in a sorted order, with the help of a priority queue, which keeps the storage requirements in $O(n)$ (cf. [3] for a comprehensive study on search algorithms for sorted multisets of the form $\sum X_i$).

Since the class of algorithms proposed in [5, 6] is a generalization of a two-list six-table algorithm [5], we shall show through an example that this algorithm does not have the claimed time and space complexities. The theorem proved in the corrigendum states that the successive generation in sorted order of the elements of a set of the form $X + Y$ can be done in time $O(n^2 \log n)$. A careful study of [5, 6] reveals that the two-list six-table algorithm uses the same idea of the two-list four-table: a search is performed in a multiset $\sum X_i$, $1 \leq i \leq 6$, by generating successively the elements of $S_1 = X_1 + X_2 + X_3$ and $S_2 = X_4 + X_5 + X_6$, in sorted order, in claimed time $O(n^3 \log n)$ and space $O(n)$, where the sorted vectors $X_i$, $1 \leq i \leq 6$, have $n$ elements each.

The algorithm Multifoursearch presented below is the same as Algorithm 2 of [5], with the difference that Multifoursearch is used in the context of multisets instead of NP-complete problems.

**Algorithm Multifoursearch**

(input: vectors $X_i$, $1 \leq i \leq 4$, of cardinality $n$; an element $z$)
(output: yes/no, depending on whether $z$ belongs to $\sum X_i$, $1 \leq i \leq 4$);
(1) sort $X_2$ into increasing order and $X_4$ into decreasing order;
   let $Q'$ (or $Q''$) be the priority queue for pairs of elements from $X_1 + X_2$
   (or $X_3 + X_4$) so that the pair with the smallest (or largest) sum in $X_1 + X_2$
   (or $X_3 + X_4$) is accessible in $O(1)$ time;
   $\forall r \in X_1$ insert into $Q'$ all pairs $(r, \text{first}(X_2))$;
   $\forall r \in X_3$ insert into $Q''$ all pairs $(r, \text{first}(X_4))$;
(2) repeat until $Q'$ or $Q''$ becomes empty (in this case: no and halt):

$(u, v) \leftarrow$ pair with smallest sum in $Q'$;
$(r, s) \leftarrow$ pair with largest sum in $Q''$;
if $z = (u + v) + (r + s)$ then yes and halt;
if $z < (u + v) + (r + s)$ then delete $(u, v)$ from $Q'$
insert $(u, next(v))$ into $Q'$;
if $z > (u + v) + (r + s)$ then delete $(r, s)$ from $Q''$
insert $(r, next(s))$ into $Q''$;

This algorithm was proved to search $\sum X_i$, $1 \leq i \leq 4$, for a given element in time $O(n^2 \log n)$ [1, 3, 4]. We notice that $X_2$ and $X_4$ can also be seen as the cartesian sum of two vectors, what means that this algorithm can easily be extended to a Multisixsearch version to search $\sum X_i$, $1 \leq i \leq 6$, for a given element with storage requirements $O(n)$.

Algorithm Multisixsearch

(input: vectors $X_i$, $1 \leq i \leq 6$, of cardinality $n$; an element $z$)
(output: yes/no, depending on whether $z$ belongs to $\sum X_i$, $1 \leq i \leq 6$);

(1) sort $X_2$ into increasing order and $X_4$ into decreasing order;
let $Q'$ be the priority queue for pairs of elements from $X_1 + (X_2 + X_3)$
so that the pair with the smallest sum in $X_1 + (X_2 + X_3)$ is accessible
in $O(1)$ time;
let $Q''$ be the priority queue for pairs of elements from $X_4 + (X_5 + X_6)$
so that the pair with the largest sum in $X_4 + (X_5 + X_6)$ is accessible
in $O(1)$ time;
\forall r \in X_1$ insert into $Q'$ all pairs $(r, \text{first}(X_2 + X_3))$;
\forall r \in X_4$ insert into $Q''$ all pairs $(r, \text{first}(X_5 + X_6))$;

(2) repeat until $Q'$ or $Q''$ becomes empty (in this case: no and halt):
$(u, v) \leftarrow$ pair with smallest sum in $Q'$;
$(r, s) \leftarrow$ pair with largest sum in $Q''$;
if $z = (u + v) + (r + s)$ then yes and halt;
if $z < (u + v) + (r + s)$ then delete $(u, v)$ from $Q'$
insert $(u, next(v))$ into $Q'$;
if $z > (u + v) + (r + s)$ then delete $(r, s)$ from $Q''$
insert $(r, next(s))$ into $Q''$;

Its time complexity depends on the crucial operation $\text{next}$. Instead of computing $\text{next}$ in a sorted set (which can be done in $O(1)$), now we are obliged to find the successor of a given element in a matrix of the form $X + Y$. In [2] it was proved that this operation is $\Theta(n)$ time. Hence Multisixsearch would have time complexity $O(n^4)$.

In order to improve the performance of Multisixsearch, it was proposed in [6] that the successive generation of the elements of $(X_2 + X_3)$ and $(X_5 + X_6)$ could be used to
implement the operation `next`, with an overall complexity of $O(n^2 \log n)$. However, such a strategy of generation is not enough to guarantee that the elements of $X_1 + (X_2 + X_3)$ and $X_4 + (X_5 + X_6)$ will be generated successively in a sorted order.

For instance, suppose that we just inserted a pair composed by $v_i$, the $i$th element of $X_2 + X_3$, into $Q'$. If the new smallest element in $Q'$ is not composed by $v_i$ then `next` ($v_i$) cannot be correctly generated as it was proposed in [6], since the elements in $X_2 + X_3$ are generated successively and in sorted order. Unfortunately, the correct behavior of the operation `next` is exactly what is required for the correctness of the algorithm.

Below we give an example of $X_1$, $X_2$, and $X_3$, for which the successive generation of the elements of $X_1 + (X_2 + X_3)$ cannot be done as it was proposed in [6]. We remark that more complex examples can be easily shown to exist.

Let $X_1 = \{0, 5\}$, $X_2 = \{0, 1\}$, $X_3 = \{0, 2\}$; then:

$$X_2 + X_3 = \begin{bmatrix} 0 & 2 \\ 1 & 3 \end{bmatrix}, \quad X_1 + (X_2 + X_3) = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 5 & 6 & 7 & 8 \end{bmatrix}.$$ 

With this example we can see that the operation `next` has to generate all the elements in $X_2 + X_3$ in order to generate the first row of $X_1 + (X_2 + X_3)$. After the last element of the first row of $X_1 + (X_2 + X_3) - the 3 - is selected from the heap $Q'$, the first element of the second row of $X_1 + (X_2 + X_3) - the 5 - is the next element to be selected from $Q'$. The operation `next` is then called but it cannot generate any element since 3 was the largest element of $X_2 + X_3$. It is not difficult to see that, to generate all the elements of $X_1 + (X_2 + X_3)$ with this strategy, the algorithm should keep track of all the elements generated from $X_2 + X_3$, since different total orders can be embedded into $X_1 + (X_2 + X_3)$. Therefore, the storage requirements will increase and the algorithm in [5, 6] cannot have the claimed complexities.

3. Conclusion

An algorithm was proposed in [5] (and corrected in [6]) to solve some NP-complete problems that belong to a class defined in [4]. The complexities described were $O(n^{\log k} 2^{n/k})$ time and $O(k 2^{n/k})$ storage requirements, for $2 < k < n$.

We showed in this note that such an algorithm cannot match the claimed complexities. In order to be correct, the algorithm needs either more space – with the same time complexity –, or more time – keeping the same storage requirements. Hence, the best-known sequential algorithm for such a class of NP-complete problems is still the two-list four-table algorithm introduced in [1, 4].

We believe that the design of a better, space-efficient algorithm, if such exists, is closely related to the answer to an open question proposed in [3], concerning the complexity of searching the cartesian sum of three sorted vectors ($X_1 + X_2 + X_3$, for instance) for a given element.
References


