

# DYNAMIZATION OF DECOMPOSABLE SEARCHING PROBLEMS \*

Jan VAN LEEUWEN

*Department of Computer Science, University of Utrecht, Utrecht, The Netherlands*

Derick WOOD

*Unit for Computer Science, McMaster University, Hamilton, Ontario, Canada L8S 4K1*

Received 19 November 1979

Full dynamization, computational complexity

## 1. Introduction

In the complexity theory of geometric configurations (and other areas of algorithmic endeavor) one encounters a fair number of problems for which a very efficient static solution is known, but no alternative to complete reconstruction seems to come to mind when we wish to insert or delete even a single point. Bentley [1] has recognized a large class of problems (which he called decomposable searching problems) for which there is hope that a reasonably efficient dynamic solution can be attained.

Briefly, a searching problem is said to be decomposable if its solution can be synthesized at only nominal extra cost from the solutions of the very same problem for all distinct parts of some arbitrary partition of the original point-set. The question to determine which point of a given set is closest to some (varying) point  $x$  is a typical example of a decomposable searching problem. Bentley's primary technique of dynamization for these problems consists of finding a partition of the set into pieces, each statically organised, such that insertions at the low end (see Fig. 1) most often require a repartition of

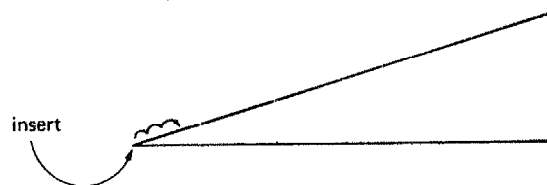


Fig. 1.

points at the same low end only. He has given a specific technique which usually adds a factor of  $\log n$  to both the processing and query times of the static structure. A number of other techniques have been explored in [6].

Deletions are harder to handle in Bentley's framework, because points to be deleted need no longer be at the low end and may have migrated into larger blocks to the right (see Fig. 2). It is suggested to tag such points as being deleted, but to keep them in the structure for a while longer. By the time about  $\frac{1}{2}$  of the current points have been tagged, a clean-up procedure is called into action which eliminates deleted points and rebuilds the dynamic structure from scratch out of present points only. The cost for reconstruction may be high, but usually comes to an affordable charge on the average per (past) deletion.

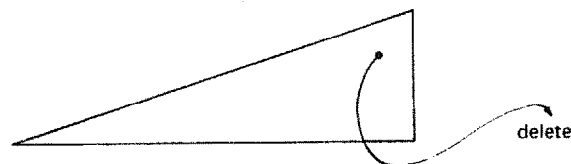


Fig. 2.

\* This work was carried out while the second author was visiting the University of Utrecht, sponsored by a grant from the Netherlands Organisation for the Advancement of Pure Research (ZWO). The second author was also supported by a Natural Sciences and Engineering Research Council of Canada Grant No. A-7700.

The quest for a dynamization which keeps its structure clean was recently taken up by Maurer and Ottman [5]. They observed that, when a limit on the largest set-size to occur over time is known, one can fix the number of blocks and maintain their contents such that the worst case bounds on processing and query time for the largest set-sizes are optimized. The method will do well if set-sizes do not vary by an awful lot, yet one must recognize that (by fixing the number of blocks) performance can degrade when the set shrinks or grows in size beyond the limit originally anticipated.

In this paper we shall expand on the method of [5] to arrive at a technique which will adapt the size-limits on (and the number of) blocks dynamically at no additional cost. The technique is perfectly general and guarantees optimized response times no matter how the set-size varies. For the technique to work, it is crucial that our programming environment provides an unrestrictive dynamic storage allocation facility.

In Section 2 we describe the technique of full dynamization and its many degrees of freedom. The method requires that a size-count is kept with every block. In a ramification of the technique, we will show how this can be avoided and that only 3 counters are needed. In Section 3 a number of examples, all known decomposable searching problems, will be discussed to show how they can be fully dynamized.

## 2. A technique to achieve full dynamization

Suppose we know an efficient static solution of some decomposable searching problem  $P$ , which involves a 'static' datastructure  $S$ . Suppose that for a current set of  $n$  points,  $S$  enables us to answer admissible queries in time  $Q(n)$  and to process updates (insertions and deletions) in time  $U(n)$ . Usually  $Q(n)$  is small and  $U(n)$  is large, often equal to the time required to build  $S$  from scratch. We shall assume, as we almost always can, that  $Q$  and  $U$  are nondecreasing.

As in [5] our method of dynamization will be based on a partition of the current set of points into blocks, each block separately organized 'like  $S$ '. We will need a dictionary to keep track of what points are present and where they are located. One may think of a balanced tree or some type of extendible hashing for it. We shall assume that the dictionary is fully dynamic

and guarantees an (average) response time of  $D(n)$  for each transaction.

A very important ingredient of our method will be some (predefined) sequence of switchpoints  $\{x_k\}_{k \geq 1}$  which satisfies

- (a)  $x_k \in \mathbb{N}$ ,
- (b)  $k \mid x_k$ ,
- (c)  $x_{k+1}/(k+1) > x_k/k$ .

Observe that (c) implies that  $x_{k+1} > x_k$ . We leave it open what the switchpoints are, but note that their choice will be highly application-dependent. The  $x_k$  are typically given by means of some formula  $f(k)$  which is easy to compute. (It will be true for almost all examples.)

Given a fixed sequence of switchpoints  $\{x_k\}_{k \geq 0}$ , let  $y_k = x_k/k$ . The  $y_k$  are integers and  $y_{k+1} > y_k$ . From now on we shall always use  $n$  to denote the current set-size. Hence,  $n$  will vary as insertions and deletions take place. If  $n$  passes certain thresholds, then our dynamized structure will 'switch gears'. The thresholds will be our  $x_k$ , the switchpoints. In the following definition we understand  $x_0$  to be 0.

**Definition.** We are operating on level  $k$  when  $x_k \leq n < x_{k+1}$ .

On level 0 we shall operate the point-set manually. This can be justified by the observation that on this level set-size is bounded by  $x_1$ , a 'small' constant. On level  $k$ ,  $k \geq 1$ , the point-set will be partitioned into  $k$  blocks and a dump (see Fig. 3), with the following characteristics for each of the constituents. Each block (and the dump) is structured like  $S$  and is augmented with a size-counter. For blocks  $B$ , the size-counter  $s(B)$  will satisfy

- (d)  $y_k \leq s(B) \leq y_{k+1}$

and for the dump's size-counter  $s(D)$  we shall main-

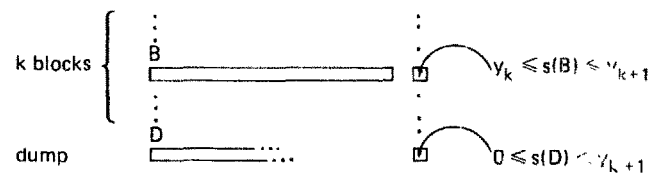


Fig. 3.

tain

$$(e) \quad 0 \leq s(D) < y_{k+1}.$$

Conditions (d) and (e) are the characteristic invariants of level  $k$ .

Blocks on level  $k$  are divided into three classes: low blocks (which have  $s(B) = y_k$ ), halfway blocks (which have  $y_k < s(B) < y_{k+1}$ ) and full blocks (which have  $s(B) = y_{k+1}$ ). Rather than keeping blocks sorted by size-count, it is sufficient just to maintain these three classes.

When all blocks on level  $k$  have become full, we will switch to level  $k + 1$  by the time the dump gets full too. This will be done by including the current dump in the collection of blocks (making for a total of  $k + 1$ ), renaming the 'full' class as the 'low' class (at no cost) and initializing a new dump with 0 elements. The reader should convince him/herself that we have correctly switched to the invariants of level  $k + 1$ . Observe that when the switch from level  $k$  to level  $k + 1$  is made, there are precisely  $(k + 1) \cdot y_{k+1} = x_{k+1}$  points in the set.

Another crucial moment on level  $k$  occurs when all blocks are low, the dump is empty and a deletion takes place (we will discuss how in a moment). Then we will have to switch to level  $k - 1$ , by 'degrading' the block that lost an element to be the dump (thereby replacing the currently empty dump) and renaming the 'low' class of remaining blocks as the 'full' class. Again the reader should convince him/herself that we have correctly switched to the invariants of level  $k - 1$ . Observe that the switch from level  $k$  to level  $k - 1$  was made just when the set-size went down from  $x_k$  to  $x_k - 1$ .

We summarize these considerations into

**Lemma 2.1.** Level switching takes only constant time.

It remains to be shown that the structure as presented supports the complete repertoire of dynamization, without the worst case response times getting totally out of hand. Fortunately it can all be dealt with rather elegantly, as is spelled out in the proof of the following principal result. Recall that  $n$  stands for the current number of elements in the set.

**Theorem 2.2.** One can fully dynamize the static solution of a decomposable searching problem such that

queries can be answered in time  $O(k \cdot Q(y_{k+1}))$  and updates (insertions and deletions) can be processed in time  $O(D(n) + U(y_{k+1}))$  each, where  $k = k(n)$  is the level on which we operate for a set of  $n$  elements.

**Proof.** Consider the structure as presented. We show how to operate it, by discussing each of the allowable types of transactions in turn.

(i) *Querying.* Because  $P$  is decomposable, we can just query each of the blocks (and the dump) separately and assemble the final answer at no substantial extra cost. The query time per block can be estimated as  $Q(y_{k+1})$  and the total amount of work spend is certainly bounded by  $O(k \cdot Q(y_{k+1}))$ .

(ii) *Insertion.* Suppose we wish to insert a new point  $x$ . We first check the dictionary (at cost  $D(n)$ ) that  $x$  is not already present and see if there still are low or halfway blocks around. If there are, then pick one and insert  $x$  into it, update its size-counter and check its class. Otherwise, insert  $x$  into the dump (in the same way). If it causes the dump to become 'full' too, then switch to level  $k + 1$ . After all this is completed, enter  $x$  into the dictionary and record where it was placed in the structure (by a pointer).

(iii) *Deletion.* Suppose we wish to delete a point  $x$ . First we check the dictionary (at cost  $D(n)$ ) that  $x$  is present. If it is, then we pick up the pointer to its actual location in the structure and delete it from its block (or from the dump). Some care must be taken in doing so. If we deleted  $x$  from a halfway or full block (or from the dump), then we just decrement the size-count and adjust the classification of the block if needed. If we deleted  $x$  from a low block, then we have an 'underflow' situation. Borrow an element from the dump or, when it is empty, from any halfway or full block around, delete it and plug it into the underflowing block. Size-counts (and perhaps the classification of the block we borrowed from) must be updated accordingly. Note that it will restore the size-count of the underflowing block to  $y_k$ . If no element can be borrowed, then we cannot repair the underflow and must switch to level  $k - 1$  (which will effectively make the underflowing block into a dump). After all this has been done, delete  $x$  from the dictionary and, if an element got repositioned because we borrowed, make sure its location information is updated. Note that it all requires no more than a bounded number of dictionary accesses and a bounded number of inser-

tions and deletions for blocks of size  $\leq y_{k+1}$ .

The formulation of Theorem 2.2 is a bit awkward, because we have to refer to  $y_{k+1}$ . It can be rephrased if we assume that  $Q(n)$  and  $U(n)$  are 'smooth', in the sense that for all constants  $c$  the functions  $Q(cn)$  and  $U(cn)$  are still of the same order. This is almost always the case in practical examples.

**Theorem 2.3.** When  $Q$  and  $U$  are smooth and there is a constant  $c$  such that  $y_{k+1} \leq cy_k$  for all  $k$ , then one can fully dynamize the static solution of a decomposable searching problem such that queries can be answered in time  $Q(k \cdot Q(n/k))$  and updates can be processed in time  $Q(D(n) + U(n/k))$  each, where  $k = k(n)$  is the level on which we operate for a set of  $n$  elements.

**Proof.** Observe that  $n/k = O(y_{k+1})$  and hence  $Q(n/k) = O(Q(y_{k+1}))$  and  $U(n/k) = O(U(y_{k+1}))$ .

The condition that  $y_{k+1} \leq cy_k$ , again, will almost always be true in practical cases. It means that on a single level we allow the blocks to expand by at most a factor  $c$  from their original sizes. Theorem 2.3 also expresses rather succinctly what the dynamization has achieved. It cuts the update time from  $U(n)$  down to  $U(n/k)$ , which can be a lot when  $k = k(n)$  grows sufficiently fast. On the other hand, if we arrange for a large or fast growing  $k = k(n)$ , then the time to answer queries will get out of hand, because it is proportional to  $k \cdot Q(n/k)$ . It only shows that one must make a very judicious choice of the switchpoint sequence the structure is to operate with, to strike a desirable balance between query and update times for the application at hand.

Observe from the proof of Theorem 2.2 that there still are many degrees of freedom in the routines for insertion and deletion. For instance, we suggested inserting a point into just any block that still had room. One might wish to keep blocks balanced and always insert into the currently smallest (as in [5]), or promote blocks from low to full as fast as one can by always inserting into the largest. Likewise, one might wish to carry out additional size-rebalancings among blocks when points get deleted. We imagine that variations of this sort will be highly application-dependent. As long as the invariants of a level remain

valid, one can do as one pleases provided the additional overhead in processing time is worth the trouble.

One objection to the structure as presented might be that apparently all blocks are required to have a size-counter. Strictly speaking we only need to maintain size-counts for the halfway blocks and for the dump, the other blocks all have  $y_k$  or  $y_{k+1}$  elements. Because size-counts do not change by more than 1 at a time, it is not hard to show that blocks can be kept ordered by size at only constant extra charge per update. We will show that, if desired, the need for keeping track of size-counts can be almost completely eliminated.

**Theorem 2.4.** The technique of dynamization presented can be modified such that there never are any halfway blocks, without affecting query and update times in order of magnitude.

**Proof.** The result will be shown by modifying the routines for insertion and deletion. Suppose we are operating on level  $k$  and assume there presently are no halfway blocks.

(i) *Insertion.* After passing the usual dictionary test, always insert  $x$  into the dump. If the dump gets full, then promote it to a full block (and have it join the full class) and pick a low block to replace the dump. If there is no low block left to do so, then switch to level  $k + 1$ .

(ii) *Deletion.* Here we must be careful again. Use the dictionary to find where  $x$  is located. If  $x$  belongs to the dump, then delete it without further ado. If  $x$  belongs to a block and the dump is (still) non-empty, then delete  $x$  from its block and borrow an element from the dump (and delete it) to plug back into the block (to keep it low or full). If the dump is empty, then we proceed as follows. If  $x$  must be deleted from a full block, then do so and make the block into the dump. If  $x$  must be deleted from a low block, then do so and borrow an element from a full block (and delete it from it) that we can plug in to restore the size of the low block. The full block borrowed from is made into the dump. If there was no full block left to borrow from, then just delete  $x$  from its (low) block and switch to level  $k - 1$ .

We leave the reader with the instructive task of verifying that the level invariants are fully obeyed. No halfway blocks are needed to let it work.

The modified technique of dynamization may require that in processing an update more (and larger) blocks may get broken than before, but it will only add a factor 2 or 3 to the worst case time estimates. Note how the manipulation and constant rôle-changing of the dump is crucial for the routines to work. An important conclusion is that full dynamization can be achieved without the need for an unbounded number of counters. Just maintaining  $y_k$  and  $y_{k+1}$  (and the classification of blocks as being low or full) and a single size-counter for the dump are sufficient.

### 3. Some applications

The pay-off from dynamization will depend a great deal on the sequence of switchpoints  $\{x_k\}_{k \geq 0}$  that is chosen. We shall discuss a number of choices one could make and their application to some common decomposable searching problems.

A typical sequence of switchpoints is obtained by defining

$x_k =$  'the first multiple of  $k$  that is  $\geq 2^k$ ',

which makes  $k = k(n)$  about  $\log n$ . Hence, by Theorem 2.3, one can fully dynamize all normal decomposable searching problems such that the following worst case estimates for individual transactions are guaranteed:

$$\begin{array}{ll} O(\log n \cdot Q(n/\log n)) & \text{for queries,} \\ O(D(n) + U(n/\log n)) & \text{for updates.} \end{array}$$

Let's apply it to an example also used by Bentley [1] for his technique of dynamization.

**Theorem 3.1.** There is a fully dynamic solution to nearest neighbour searching in the plane which takes no more than  $O(\log^2 n)$  for querying and  $O(n)$  for updates at any point in time, where  $n$  is the current set-size.

**Proof.** There is a static solution to the problem due to Lipton and Tarjan [4], which has  $Q(n) = \log n$  and  $U(n) = n \log n$ . Dynamizing it w.r.t. the given switchpoint sequence yields the bounds stated.

Note that this result for nearest neighbour searching is an improvement over Bentley's, which only

allowed for insertions and could take up to  $O(n \log^2 n)$  for updates with an  $O(\log^2 n)$  time-bound for querying.

Another typical sequence of switchpoints  $\{x_k\}_{k \geq 0}$  is obtained by defining

$$x_k = k^\alpha$$

for some integer exponent  $\alpha > 1$ . It makes  $k = k(n)$  equal to about  $n^{1/\alpha}$ . Hence, one can dynamize all normal decomposable searching problems such that the following worst case time estimates for individual transactions are guaranteed

$$\begin{array}{ll} O(n^{1/\alpha} Q(n^{1-1/\alpha})) & \text{for queries,} \\ O(D(n) + U(n^{1-1/\alpha})) & \text{for updates.} \end{array}$$

Note that different choices for  $\alpha$  can lead to very different time estimates.

**Theorem 3.2.** There are fully dynamic solutions to range querying in the plane which

- (i) take no more than  $O(\log^3 n)$  for queries and  $O(n)$  for updates, or which
- (ii) take no more than  $O((n \log^4 n)^{1/2})$  for queries and  $O((n \log^2 n)^{1/2})$  for updates.

**Proof.** Results of Bentley and Shamos [2] can be rephrased to yield a static solution to range querying in the plane with  $Q(n) = \log^2 n$  and  $U(n) = n \log n$ . Dynamizing it w.r.t. the two distinct types of switchpoint sequences we now know leads to the two conclusions stated (using  $\alpha = 2$  for the second).

Theorem 3.2 shows the kind of trade-off one can achieve between query and update time, by varying the sequence of switchpoints that is in effect. In this example one could balance the cost for queries and updates perfectly by choosing the sequence of switchpoints  $\{x_k\}_{k \geq 0}$  defined by

$$x_k = k^2 \lceil \log k \rceil.$$

It makes  $k = k(n)$  about equal to  $(n/\log n)^{1/2}$ . Hence one can dynamize to guarantee

$$\begin{array}{ll} O((n/\log n)^{1/2} Q((n \log n)^{1/2})) & \text{for queries,} \\ O(D(n) + U((n \log n)^{1/2})) & \text{for updates.} \end{array}$$

Applying it yields

**Theorem 3.3.** There is a fully dynamic solution to range querying in the plane which requires no more

than  $O((n \log^3 n)^{1/2})$  time for both querying and updates, where  $n$  is the current set-size.

And so the applications go on without end. Full dynamization will be particularly fruitful for the many decomposable searching problems for sets of lines and hyperplanes treated in Dobkin and Lipton [3], which usually are 'fast' except for some polynomial preprocessing time.

A last question might be how efficient one can hope to get when balancing query and update times in an approach like this. An argument of Saxe and Bentley [6] shows that one will never get below the  $\sqrt{n}$  limit. The structure we presented to achieve full dynamization will never replace ingenuity in dynamizing a specific problem, but should be taken as a general method which is to be brought into action when everything else fails.

## References

- [1] J.L. Bentley, Decomposable searching problems, *Information Processing Lett.* 8 (1979) 244–251.
- [2] J.L. Bentley and M.I. Shamos, A problem in multivariate statistics: algorithm, datastructure and applications, Technical Report Department of Computer Science, Carnegie-Mellon University (1977).
- [3] D. Dobkin and R.J. Lipton, Multidimensional searching problems, *SIAM J. Comput.* 5 (1976) 181–186.
- [4] R.J. Lipton and R.E. Tarjan, Applications of a planar separator theorem, *Conference Records 18th Annual IEEE Symposium on Foundations of Computer Science* (1977) 162–170.
- [5] H.A. Maurer and Th. Ottman, Dynamic solutions of decomposable searching problems, Bericht 33, Institut für Informationsverarbeitung, TU Graz (1979).
- [6] J.B. Saxe and J.L. Bentley, Transforming static data structures to dynamic structures, Technical Report Department of Computer Science, Carnegie-Mellon University (1979).