OPTIMUM REORGANIZATION POINTS FOR ARBITRARY DATABASE COSTS*

Raul J. Ramirez
Frank Wm. Tompa
J. Ian Munro

Research Report CS-79-29

Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1
Canada

---

* The research reported in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada under grants A8237 and A9292 and by the University of Waterloo.

ABSTRACT

The performance deterioration caused by updates to databases or data structures can be overcome by reorganizing the structure from time to time. In previous work, optimal reorganization intervals were determined for linearly increasing deterioration costs and linearly growing reorganization costs. To date only heuristics have been available for non-linear costs, and no work has been published on optimal solutions.

This paper extends previous results by identifying the reorganization points problem as a shortest route problem and by providing a dynamic programming algorithm to find optimal reorganization points when reorganization and deterioration costs are arbitrary. It is shown that our method uses $\Theta(T^{**}2)$ basic operations and $\Theta(T)$ space, where T represents the databases lifetime. Furthermore, we note that no algorithm can solve this problem in significantly less time or space. Examples involving linear and non-linear costs are presented and discussed. Finally, the algorithm is modified to find the optimal sequence of reorganizations to be applied in situations where partial reorganizations are possible.

Keywords and phrases: reorganization, data structure, database, file organization, shortest route, dynamic programming, recursive optimization, information retrieval.

CR categories: 3.73, 4.33, 4.34, 5.42

1. The problem.

When a database is created, its contents are typically organized in a manner convenient for efficient processing of queries and updates. As updates are made, however, it is quite often (indeed usually) the case that the performance of the system degrades to a point at which reorganization is required or at least justified. For example, if the information is stored in a large sorted table, and a few new items are to be added, it may be convenient to enter the new data in an unordered auxiliary list temporarily, since insertion into the primary table would force the movement of many elements in the system. Direct insertion into the primary table would not only be expensive, but is very likely to be completely prohibitive for an online system. Using an auxiliary list to collect a reasonable number of updates which can be merged into the primary table at some convenient time is a very attractive approach. An important question is, of course, "when should this merging occur?"

Although this example will be investigated further in subsequent sections, the main thrust of this paper is to consider the more general problem of deciding when an arbitrary structure should be reorganized. Consider the case, then, in which the demands to be made on a database are reasonably predictable, at least for some fixed period of time. This predictability does not preclude situations in which the volume of data stored and the number of queries and

updates vary widely in time. Such time-varying but predictable situations exist in practice, for example, in keeping track of the inventory of a holiday supplies shop or maintaining student records for an academic institution. As a consequence of the predictability, the cost of reorganizing a structure at any given point, as well as the cost of using it if it has been reorganized and if it has not, can be determined. The problem is, then, to determine the reorganization points so as to minimize the total cost of the operation. This total cost includes the operating cost of using the system (and thus implicitly the deterioration cost incurred by not restructuring) as well as the reorganization cost. Figure 1 illustrates some of these and related concepts.

Several specific cases of this problem have been investigated previously. Shneiderman presented a closed form solution for linearly increasing deterioration and reorganization costs assuming reorganization occurred at equidistant time intervals [3]. However, when the reorganization cost is not constant, the optimal intervals will not be equidistant. With this in mind, Tuel dropped the equidistant assumption and gave a closed form solution for arbitrary linear costs [4]. The example of a sorted table with an auxiliary unordered list, however, does not fit this linear criterion. Indeed, many "updating systems" are sub-linear in their deterioration cost. Unfortunately, when the reorganization or the deterioration costs are non-linear, no

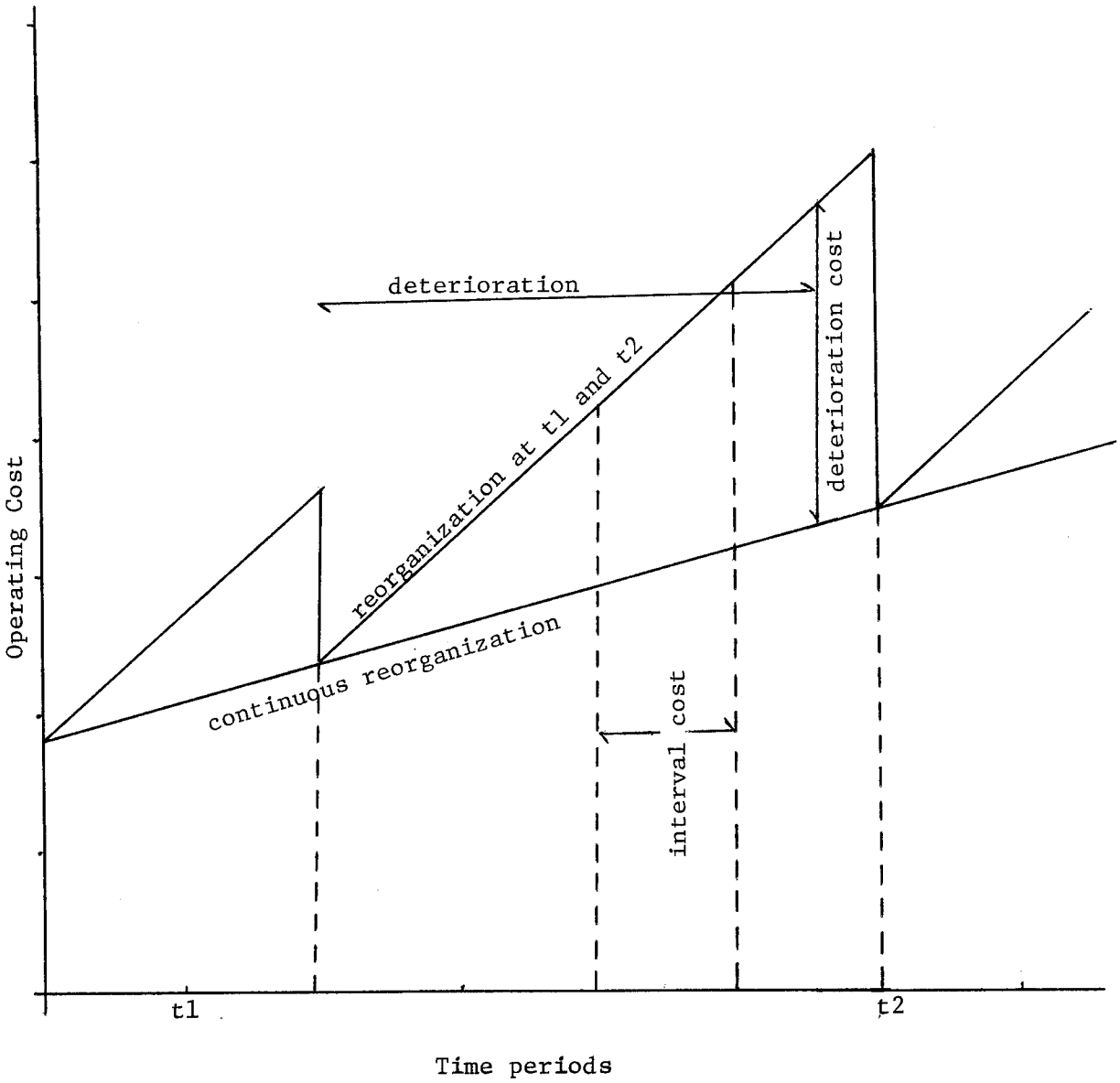


Figure 1 Linear deterioration cost.

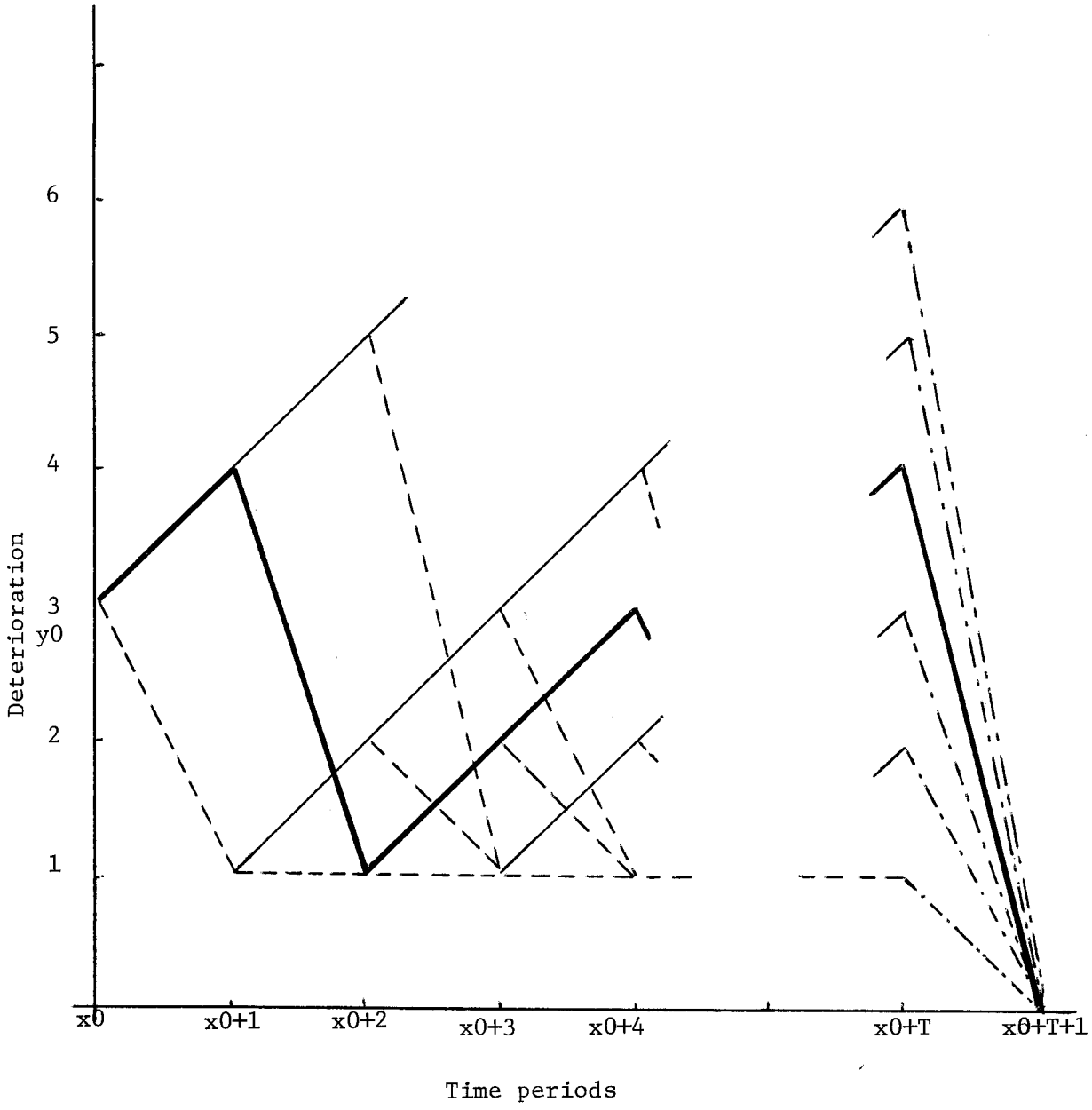
closed form solution is known for most cases. Yao, et al. [5] have presented a heuristic that is near optimal for constant reorganization cost and have claimed it to be "superior" for increasing reorganization costs.

This paper presents an algorithmic solution to the problem for arbitrary reorganization and deterioration costs. The basic concern is a database whose performance degrades with the number of updates and for which there is associated a reorganization cost. The problem is to determine when to reorganize it to minimize the expected overall cost. It is assumed that the remaining lifetime of the database (i.e. the amount of time until the database is no longer used, measured in hours, days, months, etc.) is T time periods (not necessarily of equal length) and that if the database is to be reorganized, the process will take place discretely at the beginning of a time period. Enumeration of all possible sets of reorganization points requires the computation of 2**T costs, since at each period the decision whether or not to reorganize can be made independently. Such a computation is, of course, infeasible for T greater than 20 or 30.

The problem of determining optimal reorganization points for a database can be identified with the problem of finding the shortest route in an acyclical network (i.e. finding the route of minimal cost from source to sink). Consider a grid of vertices at the non-negative integral

points in a portion of the plane (Figure 2). The x and y coordinates of each vertex denote respectively the time and the database deterioration deterioration, that is, the number of periods since the last reorganization. The source of the network is vertex (x0,y0), where x0 is the time at which the study begins (perhaps when the database is formed), and y0 is the time since the last reorganization (y0 may well be 0). The decision not to reorganize the structure in condition (x,y) is denoted by an edge from (x,y) to vertex (x+1,y+1). The weight of this edge is the operational cost of running a system from time x to x+1 starting with deterioration y. The decision to reorganize corresponds to an edge from (x,y) to (x+1,1), with weight equal to the sum of the reorganization cost of a database of deterioration y in period x and the operating cost of a newly reorganized database in period x. If T is the database lifetime, then every vertex with x-coordinate x0+T is connected by a zero-valued edge to the sink vertex (x0+T+1,0). As can be seen from the above description, the network is acyclic (i.e., no cycles are formed since time always moves to the right), and the solution is represented by the minimal cost route from vertex (x0,y0) to vertex (x0+T+1,0).

Section 2 contains a dynamic programming formulation of the solution for this shortest route problem and a discussion of its optimality. In Section 3 the algorithm is applied to a non-linear example. Section 4 contains proofs



Optimal path

No reorganization ⇒ time and deterioration increase by 1

Reorganization ⇒ time increases by 1, deterioration become 1

Fictitious ⇒ cost 0

Figure 2 Time vs Deterioration

of the optimality of the time and space required for this algorithm, and Section 5 extends all the results to applications in which at the beginning of each period the database can be partially reorganized to any of several levels at various costs. The appendix contains PASCAL code for the algorithms discussed.

2. An Efficient Solution.

Dynamic programming is a mathematical technique used to make a sequence of interrelated decisions that maximizes (or minimizes) some function [1, 2]. Dynamic programming is also called recursive optimization because of the convenience of viewing the optimization process in a recursive manner. The database reorganization problem can be expressed in terms of dynamic programming as follows:

Let $d(x,y)$ denote the operating cost from period x to period $x+1$ of a database with deterioration y at the beginning of the period,

$r(x,y)$ denote the reorganization cost at the beginning of period x of a database of deterioration y,

and $F(x,y)$ denote the minimum cost to get to the state in which the database has deterioration y at the beginning of period x, given that the process began period $x0$ with a database whose deterioration was $y0$.

Note that no assumptions (e.g. continuity, monotonicity, or even non-negativity) have been made for the above functions $r(x,y)$ and $d(x,y)$. In fact, the functions may be represented by arbitrary tables of discrete values (not necessarily equidistant in time) computed or estimated by monitoring, simulating, or analyzing the database under consideration.

At the beginning of each period there is the option to reorganize the database, and so, at the end of the period the deterioration of the database could be 1 or it could be one more than at the beginning. From Figure 2 it is apparent that the minimal cost expended from time x0 until some later time x is the minimal cost to reach period x-1 with deterioration y-1 plus either the reorganization cost plus the operating cost for the newly reorganized structure, or the operating cost in period x-1 of a database of deterioration y-1.

This leads to the following recurrence relation for F:

$$F(x, 1) = \min \{ F(x-1, y-1) + r(x-1, y-1) + d(x-1, 0) \}$$

over all choices of $y = 2, 3, .. x-1$ and $y0+x-1$

and

$$F(x, y) = F(x-1, y-1) + d(x-1, y-1)$$

for $y = 2, 3, .. x-1$ and $y0+x-1$

The boundary condition is:

$$F(x0, y0) = 0$$

It is relatively straightforward to write a program to determine $F(x0+T+1, 0) = \min F(x0+T, y)$ over all choices of $y=1,2, .. T$ and $y0+T$: simply use the above recursion to determine the optimal cost for each time step and state of deterioration based on the optimal cost up to the previous time step. Note that the value of $F(x, y)$ need not be retained after $F(x+1, y+1)$ and $F(x+1, 1)$ have been computed. Hence at most T storage locations are required to retain

these values. In fact, each arc in the network is inspected and used in some arithmetic computation once only, and thus if the values of r(x, y) and d(x, y) can be computed, only $\Theta(T)$ storage locations are needed to maintain the costs. Furthermore, since each arc is inspected only once, it immediately follows that $\Theta(T^{**}2)$ basic operations are performed. Note as well that the above recursion produces the shortest route from the source vertex (x0, y0) to every other vertex in the network.

However, the real problem is not to discover the cost of the optimal reorganization scheme, but rather to determine the reorganization points that lead to that cost.

Taking a closer look at Figure 2, it is realized that the only vertices that must store information regarding the optimal path are those with y-coordinate 1. The only way to reach the vertex (x, y) for $y \neq 1$ is through vertex (x-1, y-1), and so there is no need to store this information while determining the optimal path. With this observation in mind, and since there are only T vertices with y-coordinate 1, it follows that, if the values of r(x,y) and d(x,y) can be computed, only $\Theta(T)$ units of storage are required to determine the optimal reorganization scheme as well as its cost. (The appendix contains an implementation of this implied algorithm written in PASCAL.)

3. A Non-linear Application.

This section deals with the reorganization of a database whose deterioration and reorganization costs are non-linear. In particular the application's deterioration cost is logarithmic (i.e. sub-linear), and the reorganization cost is super-linear.

Recall the example in Section 1 in which one of the data structure maintained by the database under consideration is an ordered table (i.e. a set of consecutive locations each containing one element, the elements to be kept in sorted order). Using binary search, the number of comparisons required to access a randomly designated element is essentially $\log(n)$[1] , where n is the number of elements in the structure. Inserting a new element into the structure requires that a hole be created by shifting some elements down one position and then making the insertion into the newly vacated cell. Since this operation is expensive, it may be decided to keep the elements to be inserted in an unordered secondary list. If an element is not found in the primary table, the search continues with a sequential scan of the secondary list. Since, when looking for a randomly selected element, the primary table will always be searched and the secondary list will be searched in proportion to its relative size, the cost of accessing a particular element is $\Theta(\log(n-k) + k*k/n)$ where n is the total number of elements

[1] Logarithms are taken to base 2.

in both structures and k is the number of elements in the secondary list. In fact, for this example, the access cost used will be log(n-k+1)-1 + (1+k/2)*(k/n), the expected number of comparisons required.

If there are no deletions, the structure will grow. From time to time the elements in the secondary list may be merged with the ones in the ordered table (i.e. the structure will be reorganized) at a cost of Θ(k*log(k) + n) operations: the k*log(k) term accounts for the average time required to sort k elements (e.g. using Quicksort), and Θ(n) operations are used in merging the two sorted lists.

In summary, assuming that there are a total of n elements, (n-k) in the ordered structure and k in the secondary structure, the following costs apply:

```
access cost: log(n-k+1)-1 + (1+k/2) * (k/n)
insertion cost: 1                                    (**)
reorganization cost: C1 * (k*log(k) + n)
```

where the parameter C1 is introduced solely to illustrate the effects of various related costs.

A structure that is continuously reorganized has an access cost of log(n+1) operations. Using these formulae and assuming that the primary table initially has 1000 elements and the secondary list is empty, that there are 5000 accesses and 100 insertions uniformly distributed in each interval, and that the lifetime (T) is 50 periods, the

following results may be obtained from the algorithm presented in Section 2:

C1	optimal reorganization points	optimal cost (in 10000's)
1	3,5,7,10,13,16,19,22,25 28,31,34,37,40,43,46	272
10	4,8,13,18,23,29,35,41	335
20	5,10,16,22,29,37	391
50	6,13,22,31	531
100	7,15,25	712
500	no reorganization	1225

Table 1.

In previous work, optimal reorganization points were determined for linear costs only. Thus one might be tempted to compute those points by approximating the costs by functions that are linear in k. A reasonable linear model of the behaviour of the system, derived by examining the optimal solution, is:

access cost: log(3500-K(C1)+1)-1 + (1+k/2)*(K(C1)/3500)

insertion cost: 1

reorganization cost: C1 * (k*log(K(C1)) + 3500)

where 3500 is the average value of n and K(C1) is the average number of probes for searching the secondary list when reorganization occurs under the optimal scheme.

The following results may be obtained under this linear model:

C1	K(C1)	reorganization points	actual cost (in 10000's)	ratio to optimal
1	147	1,2,3..49	281	1.033
10	278	2,4,6..48	383	1.143
20	357	3,5,7..47	495	1.266
50	50	3,6,9..45	704	1.326
100	625	4,8,12..44	1020	1.433
500	2500	5,10,14,18,22 26,30,34,38,42	3795	3.098

Table 2.

The column labeled "actual cost" indicates the cost charged according to the formulae in (**) and using the reorganization points suggested by this approximate solution. Comparing these results with the ones in Table 1, it is seen that, even when knowledge of the optimal solution is used to derive the approximations, results based on assumptions of linear costs can give solutions which differ substantially from the optimal. Therefore the algorithm that permits the removal of all assumptions regarding the costs is a more desirable tool for database administration.

4. Lower bounds.

In Section 2 it was demonstrated that the optimal reorganization scheme can be determined in time quadratic in the number of potential reorganization points and space linear in this parameter. Of course, it is of interest to find whether or not a better algorithm exists. It will be assumed throughout this Section that r(x,y) and d(x,y) are computable rather than stored in tables of discrete values; otherwise it is obvious that $\Theta(T^{**}2)$ space is required merely to store the algorithm's input. That the space bound cannot be appreciably improved follows from the fact that the output (number of reorganization points used) may be of length $\Theta(T)$[1].

Intuitively, the quadratic time bound seems optimal as well, since there are $\Theta(T^{**}2)$ potential situations for reorganization. The following theorem and its proof formalize this notion.

Theorem : $\Theta(T^{**}2)$ operations and $\Theta(T)$ storage locations are necessary and sufficient to determine the optimal reorganization points even if the operation costs and the reorganization costs are known to be monotonically increasing as functions in the deterioration.

Proof: The space bounds and the sufficiency of the time

[1]A more intricate argument shows that even ignoring the space required to store the results, this bound still applies.

bound follow from the algorithm presented and the observations above. To show that $\Theta(T^{**}2)$ basic operations are necessary, it suffices to exhibit a case in which it is more or less irrelevant which reorganizations are done, as long as a reorganization is performed at the particular time that allows the application to incur a "cheap" operation cost at a specific, but unknown node. That is, the shortest route must go through some unknown point (x, y) and any path through that point has the same cost. Since there are $\Theta(T^{**}2)$ potential "cheap" edges, finding the crucial one requires that all edges be inspected. A scheme which is not strictly monotonic is outlined first. It is then modified slightly to achieve monotonicity.

Consider an application for which the operation costs at each time step other than the last and each reorganization cost is 2. At the last time step the reorganization costs are also 2, but the simple operation costs are very large and all equal. Now alter an arbitrary operation cost of weight 2 to 1. Hence the optimal reorganization scheme must take advantage of this reduced cost; anything else which is done is irrelevant, proving the $\Theta(T^{**}2)$ lower bound without the monotonicity assumption.

The weights can be arranged to be monotonically increasing functions of y by setting the operating

costs to be d(x, y)=y, the reorganization costs for a database of deterioration 1 to be r(x, 1)=2x and all other reorganization costs to be r(x, y)=2xy-(3/2)y**2-y/2. Reducing one arbitrary operating cost by 1/2 results in a system that establishes the Θ(T**2) lower bound.

Repeated application of this proof technique also leads to a bound on the time required to determine near optimal solutions.

Corollary : Θ(T**2/k) basic operations are necessary to determine a reorganization schedule which is within a factor of (1+1/k) of the optimal. This lower bound holds even if the operation and reorganization costs are monotonically increasing as functions of the the deterioration.

5. Partial Reorganization.

For some applications it is possible to have more than the simple choice of reorganizing or not at each stage. For example, there may be the option of several partial reorganization algorithms each transforming the database to a different level of operation cost as well as having different reorganization cost. It is now necessary to know not only when to reorganize the database but also which reorganization algorithm to use, in order to minimize the overall cost.

An extreme case is that in which it is possible to reorganize the database of deterioration y to a level equivalent to that of any deterioration represented by an integer in the range 0 to y. Of course the reorganization and operating costs are again assumed to be arbitrary. A simple modification of the previously discussed algorithm to evaluate the O(T) reorganization alternatives at each step will take $\Theta(T^{**}3)$ operations. In fact by using the same argument as in the previous case, it can be shown that $\Theta(T^{**}3)$ operations are necessary for any algorithm solving the problem.

Unfortunately, if the algorithm is implemented as outlined in Section 2, at least $\Theta(T^{**}2)$ storage cells are necessary to determine the reorganization points and levels even if the reorganization and operating costs are computable. From Figure 3, it is clear that virtually every node

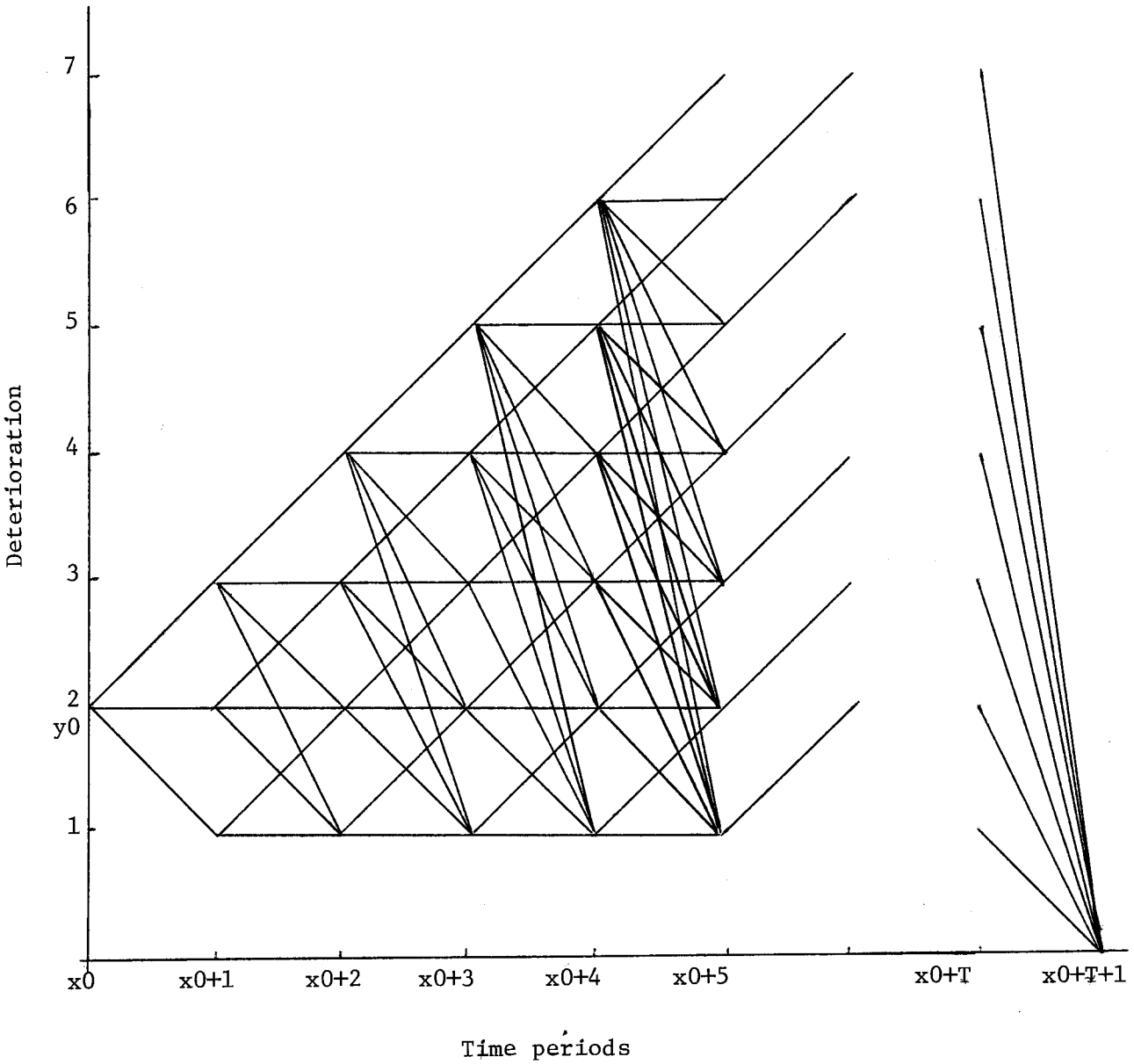


Figure 3 Partial reorganization

may be the target of one or more reorganizations in the previous time period. Thus, unlike the situation described in Section 2, for each of the O(T) values of F for a given time, O(T) reorganization points may have to be stored. In other words this algorithm may require $\Theta(T^{**}2)$ units of storage to determine the optimal reorganization scheme. When T is large this storage requirement is at best annoying and at worst prohibitive.

The following modification of the basic scheme permits a solution using only $\Theta(T)$ space while (roughly) doubling the previous time bound. Again in discussing the possibility of linear space, it is assumed that the deterioration and reorganization costs are computable.

It has been noted in Section 2 that storing the optimal path to the point (x,y) is not essential for computing the length of the shortest path, but is used only to reconstruct the shortest path itself. Because of the optimality of the shortest route, if follows that that if the shortest route from (x0, y0) to (x0+T+1, 0) passes through points (x1, y1) and (x2, y2), then the shortest route from (x1, y1) to (x2, y2) must coincide with the shortest path for the complete problem in the [(x1, y1), (x2, y2)] interval. Consequently the optimal reorganization points for the subproblem are the same as those for the original problem (in the [x1, x2] interval).

First consider solving the original problem as if only

the cost of the shortest path were required and not the path itself. Now, at the midpoint (i.e. x0+T/2) label each node in the period by its y-coordinate, i.e. a value from {1, 2,..,y0+T/2}. From time period (x0+T/2)+1 to the end of the lifetime, record for each node the node through which the route passed at period x0+T/2. In fact the values can be recorded as they are carried forward during the computation of the cost of the shortest route. When time x0+T+1 is reached and the value of the shortest route computed, the mid-node, y', that this (optimal) route passed through will be known. Since at any time period there are only $\Theta(T)$ nodes, it follows that only $\Theta(T)$ space is required for forwarding mid-node identification.

Once the node (x0+T/2, y') is known, solve the following two subproblems (recursively):

(i) find the optimal reorganization points for a database starting period x0 with deterioration y0 and running until time x0+T/2, with the constraint that it must go through y' at the last time step

and,

(ii) find the optimal reorganization points for a database starting period x0+T/2 with deterioration y' and having a lifetime of T/2 periods.

The first subproblem is equivalent to finding the set of optimal partial reorganization points of a database

starting period x0 with deterioration y0 and terminating period x0+T/2 with deterioration y'. The second subproblem is identical to the original, except that it is half the size and starts with a database of deterioration y' at period x0+T/2, and thus the original algorithm can be applied without alteration. (The appendix contains an implementation of this method in PASCAL.)

The recursive application of this "divide-and-conquer" technique will produce the set of optimal reorganization points for the original problem. It remains to show that the application of this technique will conserve the $\Theta(T^{**}3)$ time bound. Let $c^{*}T^{**}3$ denote the number of basic operations required to find the cost of the optimal arrangement by the dynamic programming scheme first proposed, and let $\mathfrak{T}(T)$ represent the computation time required in the worst case for the above scheme. Then, ignoring the minor cost of recursive calls and some trivial pointer operations,

$$\mathfrak{T}(T) = c^{*}T^{**}3 + 2^{*}\mathfrak{T}(T/2) \quad \text{for } T \geq 2$$

and suppose

$$\mathfrak{T}(1) = 1$$

As a result $\mathfrak{T}(T) \simeq 2^{*}c^{*}T^{**}3$, that is, the time to find the optimal reorganization points is approximately twice that required to find the minimum cost alone and so is still $\Theta(T^{**}3)$. If the scheme maintained the 1/3 and 2/3 positions (rather than the midpoint) of the optimal path and carried them through on the first pass, the running time of the

algorithm would be roughly 3/2 that of the basic scheme, but the space requirement, although still $\Theta(T)$ would be noticeably greater than for the method outlined. It is straightforward to develop this time-space trade-off for maintaining any fraction of the points.

6. Conclusions.

We have shown that $\Theta(T^{**}2)$ basic operations and $\Theta(T)$ storage locations are necessary and sufficient to compute the reorganization points for arbitrary or for monotonic costs, where T is the database lifetime. Furthermore, we have shown that $\Theta(T^{**}3)$ basic operations and $\Theta(T)$ space are required to compute partial reorganization points. The space-saving divide-and-conquer technique presented in Section 5 is applicable to any shortest route path problem in which the weights of edges are computable and thus do not have to be stored explicitly. Since a common measure of "cost" is the product of time and space used, this trick is often very effective.

For some applications, reorganizing the database may imply that all users must be locked out during the reorganization period. A possible minor extension to the algorithm is to compute reorganization points optimally given a limit on the maximum number of reorganizations and/or the total reorganization time for a given time interval T (and thus guaranteeing a minimal availability for the database).

The major limitation of this algorithmic approach is its dependence on a discretized, finite database lifetime. There exist some special cases for which the algorithm could be modified to handle unbounded lifetime, for example when the deterioration and reorganization costs are identical in every stage after some time T or when they are periodic

after T.

It should be noted that when the reorganization and deterioration costs are linear, Tuel's closed form solution is to be preferred to any algorithm since it requires virtually no computation. Similarly, if other closed forms can be found for particular cases, they should be preferred as well; unfortunalely no work has been reported other than for the linear case. Therefore, the simplicity, universality, and practicality of this reorganization algorithm make it a worthwhile tool for database or data structure designers.

APPENDIX

ALGORITHM 1

```
function shortest (x0, y0, t : integer) : real;
var f1, ftop, fopt : real;
       x, y, yopt, who : integer;
       from : array [0..T] of integer;
       f : array [1..T] of real;
   begin
   ftop := 0; from[x0]:=TOP;  {bounday condition}
   for x := 2 to t do
      begin
      f1 := r(x0+x-2,x+y0-2) + d(x0+x-2,0) + ftop;
      from[x0+x-1]:=TOP;
      ftop := d(x0+x-2,x+y0-2) + ftop;
      for y := x-2 downto 1 do
         begin
         f[y+1] := d(x0+x-2,y)+f[y];
         f1 := min(f1, r(x0+x-2,y)+d(x0+x-2,0)+f[y], who);
         if who = 2 then from[x0+x-1]:=y;
         end;
      f[1] := f1;
      end;
      { find optimal value }
      fopt:=ftop; yopt:=TOP;
      for y:=1 to t-1 do
         begin
         fopt:=min(fopt, f[y], who);
         if who = 2 then yopt:=y;
         end;
      shortest:=fopt;
      {retrace optimal path}
      while yopt <> TOP do
         begin
         t:=t-yopt;
         writeln('reorganize at stage',t);
         yopt:=from[x0+t+1];
         end
   end;
```

ALGORITHM 2

```
var   f : array [0..TY0] of real;
   half : array [0..TY0] of integer;
function shortest(x0, y0, t, yprime : integer; forced : boolean):real;
   var ft, oft : real;
       x, y, z, halft, ohalft, yp, who : integer;
   begin
   for y:=0 to t div 2 + y0 do half[y]:=y;
   for y:=0 to y0 do f[y]:=INFINITE;
   f[y0]:=0; oft:=0; halft:=0;
   for x:=x0+1 to x0+t do
      begin
      for y:=1 to x-x0+y0 do
         begin
         if y = 1 then ft:=INFINITE else ft:=d(x-1,y-1) +f[y-1];
         f[y-1]:=oft;
         if x-x0 > t div 2 then
            begin
            halft:=half[y-1]; half[y-1]:=ohalft;
            end;
         for z:=y to x-x0+y0-1 do
            begin
            ft:=min(ft,r(x-1,z,y)+d(x-1,y-1)+f[z],who);
            if (x-x0 > t div 2) and (who = 2) then halft:=half[z];
            end;
         oft:=ft; ohalft:=halft;
         end;
      f[x-x0+y0]:=ft;
      if x-x0 > t div 2 then half[x-x0+y0]:=halft;
      end;
   if forced then
      begin
      shortest:=f[yprime]; yp:=half[yprime];
      end
   else begin
      ft:=f[1]; yp:=half[1]; yprime:=1;
      for y:=2 to t+y0-1 do
         begin
         ft:=min(ft,f[y],who);
         if who = 2 then begin yp:=half[y]; yprime:=y end
         end;
      shortest:=ft;
      end;
   if t = 2 then
      begin
      writeln('stage',x0,' from',y0,' to',yp);
      writeln('stage',x0+1,' from',yp,' to',yprime);
      end
      else if t = 3 then writeln('stage',x0,' from',y0,' to',yp);
   if t >= 4 { t div 2 >= 2}
      then ft:=shortest(x0,y0,t div 2, yp, true);
   if t >= 3 { t div 2 >= 2}
      then ft:=shortest(x0+t div 2, yp,t - t div 2,yprime,true);
end;
```

REFERENCES

[1] Bellman R.
Dynamic Programming
Princeton University Press 1957.

[2] Dreyfus S. E. and Law A. M.
The art and theory of dynamic programming
Mathematics in Science and Engineering. Vol. 130
Academic Press 1977.

[3] Shneiderman B.
Optimum database reorganization points
Communications of the ACM
Vol. 16, No. 6 (June 1973), pp. 362-365.

[4] Tuel W.
Optimum reorganization points for linearly growing files.
ACM Transactions on Database Systems
Vol. 3, No. 1 (March 1978), pp. 32-40.

[5] Yao S. B., Das K. S. and Teorey T. J.
A dynamic database reorganization algorithm.
ACM Transactions on Database Systems
Vol. 1, No. 2 (June 1976), pp. 159-174.