Management Science/ Operations Research
D.F. Shanno Editor

An Efficient Data Structure for the Simulation Event Set

W.R. Franta and Kurt Maly
University of Minnesota

Recently algorithms have been presented for the realization of event scheduling routines suitable for general purpose discrete event simulation systems. Several exhibited a performance superior to that of commonly used simple linked list algorithms. In this paper a new event scheduling algorithm is presented which improves on two aspects of the best of the previously published algorithms. First, the new algorithm's performance is quite insensitive to skewed distributions, and second, its worst-case complexity is $O(\sqrt{n})$, where n is the number of events in the set. Furthermore, tests conducted to estimate the average complexity showed it to be nearly independent of n.

Key Words and Phrases: simulation, time flow mechanisms, event scanning mechanisms, multilinked lists

CR Categories: 3.34, 4.22, 5.5, 8.1

Authors' address: Department of Computer Science, University of Minnesota, Minneapolis, MN 55455.

1. Introduction

For discrete event digital simulation, changes of state (i.e. events) occur at times determined by random variates. When the time for a future event becomes known, it is scheduled, and a time flow mechanism is responsible for ensuring that it occurs at the scheduled time. Languages designed to support discrete event simulation maintain a record of future events in a data structure whose elements are keyed by future event times. We refer to the data structure as the event set and to the elements it contains as event notices. In contemporary simulation systems the physical data structure is generally a circular linked list ordered by event times. The scheduling of an event causes the generation of an event notice containing a record of the event time and event routine. The new notice is then inserted into the event set. To maintain proper time ordering the scheduling requires a scan of the notices in the set to determine the proper insertion point for the new notice. Since the computer time required to insert a notice is directly proportional to the number of notices scanned, the operation of scanning one notice is usually taken as the basic unit for any complexity analysis.

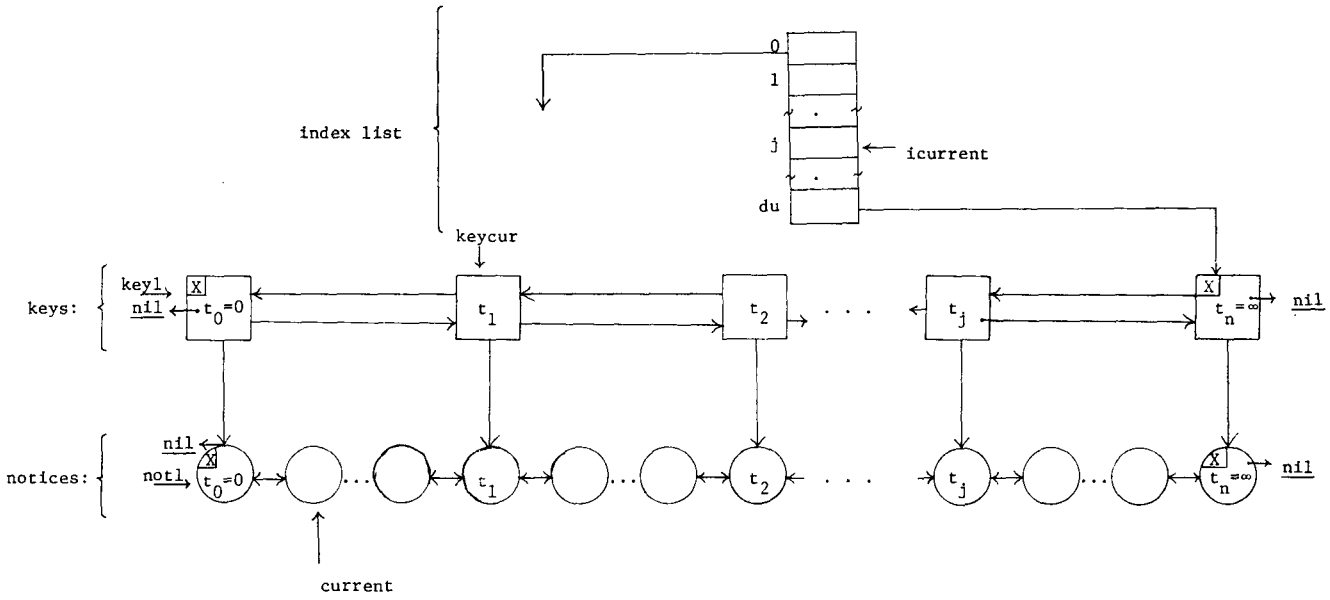
In recent years several researchers have proposed other physical data structures for the event set; see [4] and [5]. All are designed to reduce the number of scans required to locate the insertion point for a new notice. The best results to date have been recorded for a physical data structure based on so-called "dummy" notices and an index list.

One problem with these recent approaches is that worst-case complexity is not reduced, and furthermore, for skewed distributions (which in real situations occur quite frequently), the performance rapidly deteriorates with an increase in n, the number of notices in the structure. In this paper, we present a new algorithm (which we choose to call the TL (two-level) algorithm) for inserting an event into the event set. Its physical data structure employed is based in part on indexed lists and in part on the balanced terminal node structure of 3-2 trees; see [1]. The worst-case complexity of the two-level algorithm is $O(\sqrt{n})$. Further, its expected complexity is quite insensitive to the distribution of event times and, even more important, nearly independent of n, as is shown by the test results presented in Section 3.

2. The Two-Level Structure

Stripped of its simulation terminology, we require an efficient solution to the following problem: devise a physical data structure for a *dynamically changing* collection of records which supports retrieval of the record with the minimally valued key. In the simulation environment, the records are event notices, each containing

Fig. 1. Data structure for the two-level algorithm.



X denotes a dummy key or event notice

information identifying an event, and the key is the scheduled time for its occurrence. The scheduled time is known as the event time, and the collector of records is called the event set. The efficiency of the physical data structure will be measured in terms of the number of key comparisons (notice scans) the associated access routine (notice insertion) executes.

The key aspect in the above problem statement is communicated by the words "dynamically changing." When an event is scheduled to occur at a future time t_0, a notice with t_0 as the key is created and added to the event set. It remains in the set until t_0 becomes the minimal key value in the set. At this point the system time is advanced to t_0, the notice is removed, and the identified event routine is executed. Thus the range of keys in the set changes dynamically. Furthermore, the distribution of key values can seldom be described analytically. For example, assume that event times are calculated as current time plus delay, with the delay value given by a stochastic drawing. Even when all drawings are from a single uniform distribution, the distribution of the keys in the event set defies simple description.[1]

Most current simulation systems use a linear circular doubly linked list as the physical data structure, with the notices ordered by increasing event times. Thus removal of the notice with minimal event time is an $O(1)$ operation and the worst-case complexity of the insertion operation is $O(n)$, n being the number of notices in the list.

In [4] the average complexity for the insertion algorithm is improved by the following means. The range of event times represented in the set is estimated and divided into a number (heuristically determined) of equal-sized intervals. The intervals are bounded by "dummy keys (notices)" which are pointed to by the elements of an index list. Insertions are made by identifying the correct interval and then linearly scanning the notices associated with that interval for the proper insertion point. A single overflow interval is included for event times which fall outside the estimated range. The worst-case complexity for insertion remains $O(n)$ and occurs when all scheduled times fall outside the estimated range. Since the average complexity could not be determined analytically (owing to the difficulty of determining the distribution of event times in the set), it was estimated experimentally in [4]. The results confirm its superiority over the simple linear list algorithm.

In [5] several additional physical data structures are proposed. The most promising of the collection, labeled the "adaptive" structure, attempts to place an upper bound on the number of comparisons necessary for an insertion, and it does so by judiciously placing dummy (auxiliary) keys to delimit segments of the event notice list. Insertions are made by scanning the auxiliary keys to locate the appropriate segment and then linearly searching the segment for the insertion point. Auxiliary keys are created and destroyed as necessary.

The TL algorithm and its associated physical data structure is motivated in part by the indexed list (IL) algorithm and in part by the notion of balanced trees or, more specifically, 3-2 trees (see [1]). The basic idea

[1] *If* we assume that the distribution of keys in the event set is uniform and that the drawing is from the same uniform distribution, then analysis is possible; such analysis has been reported in [5].

Fig. 2. Details of the TL algorithm data structure.

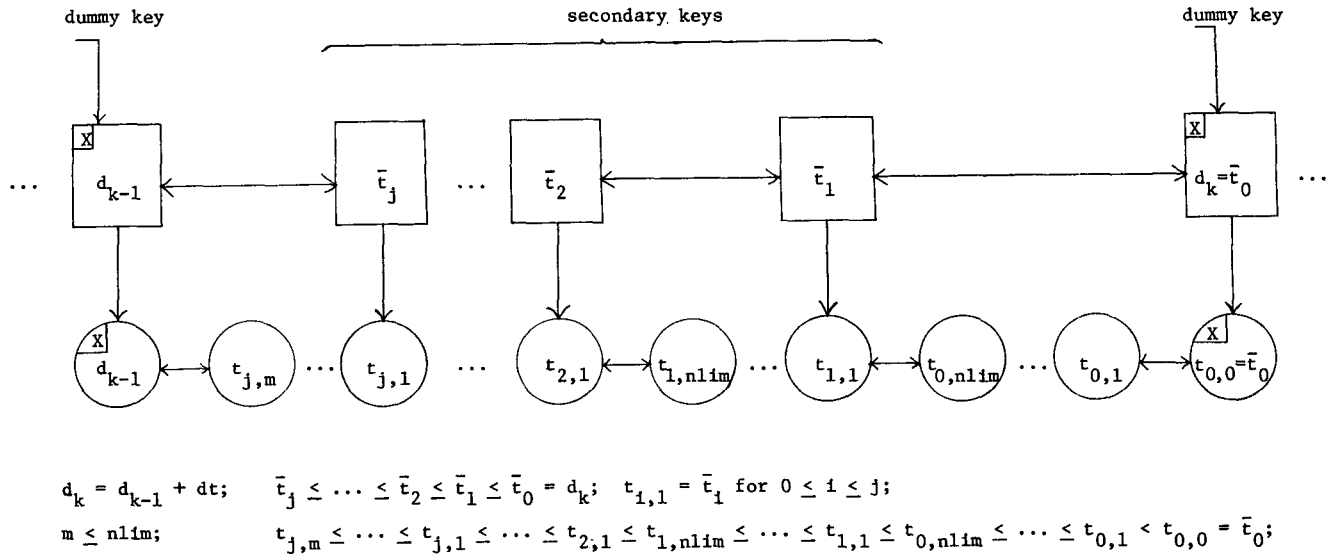
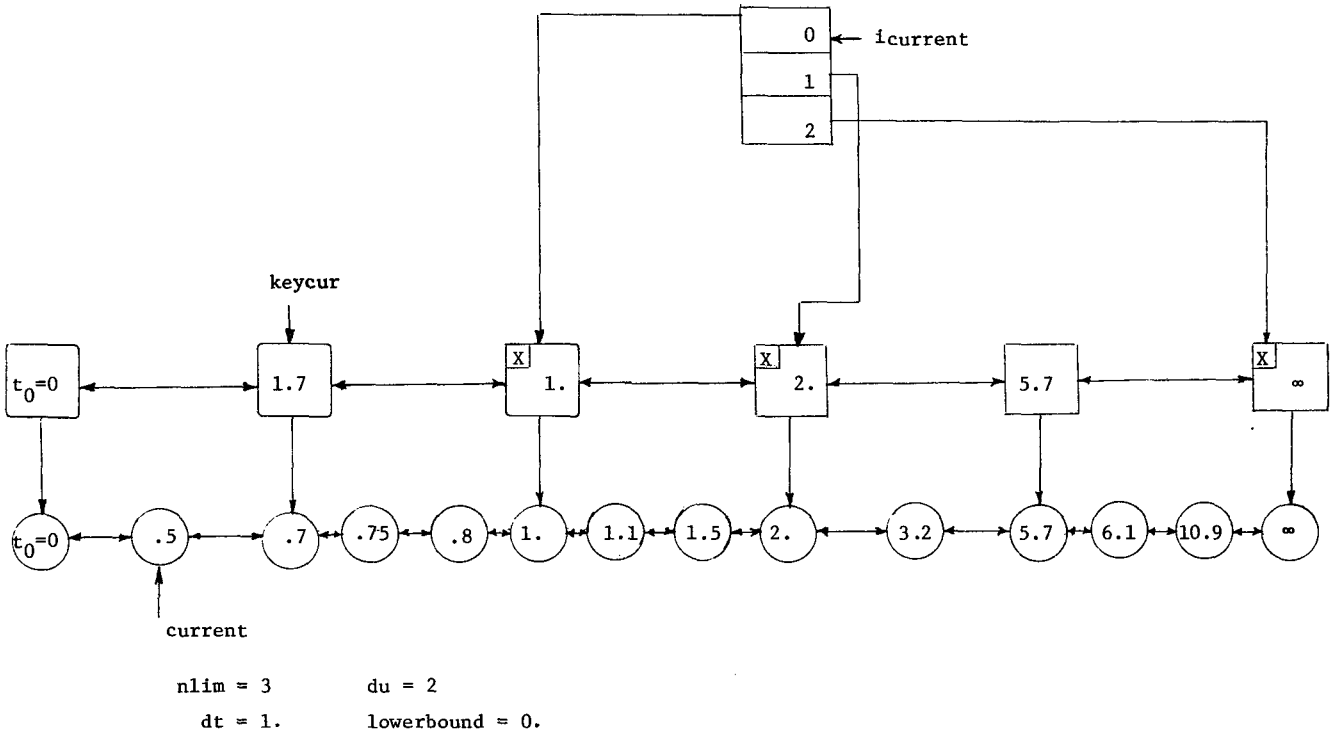


Fig. 3. Snapshot of TL data structure.



of the TL algorithm is to limit the number of notices which must be scanned for an insertion. This requirement is met by dynamically creating for each interval a list of secondary keys and associating them with the right boundary dummy key. Each secondary key points to the beginning of a sublist of notices within an interval; see Figures 1 and 2. Whenever one of these sublists becomes unbalanced, i.e. too large, the structure is adjusted by either moving the notice to the adjacent list or creating a new sublist with its associated secondary key. For clarification, a very simple snapshot of the TL structure, with *du* = 2, *nlim* = 3, *dt* = 1, is given in Figure 3. The snapshot depicted contains ten event notices (with event times .5, .7, .75, .8, 1.1, 1.5, 3.2, 5.7, 6.1, and 10.9) and reflects a simulated time of .5.

As in [4], we select as the representative access routine the scheduling primitive 'hold'. It removes the first event notice in the list, changes its event time, and

reinserts it into the list. The revised time is given by the old value plus an increment which is passed as a parameter to the hold routine. The TL algorithm, given below, uses the following global variables:

du	number of intervals
dt	width of interval
current	points to first event notice
keycur	points to "key" whose sublist contains current
icurrent	points to the element of the index list whose interval contains current
lowerbound	lower bound of interval containing *current* (initially 0)
interval	sequential representation of the index list
nlim	maximum number of notices per sublist

The field identifiers for notice nodes are 'suc', 'pred', and 'evtime', with their obvious interpretation, and those for dummy and secondary key nodes are 'left', 'right', 'sec' (a pointer to its notice node), 'number' (indicating the current size of a sublist), 'dummy' (differentiating secondary and dummy keys), and 'evtime'.

Algorithm (for hold operation[2]).

```
procedure hold(t)
begin
  /*update current notice*/
  evtime(current) ← evtime(current) + t; h ← evtime(current);
  if evtime(suc(current)) > h then return;
  /*update event set*/
  p ←current; delete(current); /*delete removes a node from a list*/
    /*calculate pointer to index list*/
  i ← [(t − lowerbound)/dt];
  if i ≥ du then i ← du else i ← mod(icurrent + i, du);
  p1 ← left(interval(i));
    /*search secondary keys*/
  while(evtime(p1) > t)p1 ← left(p1);
  p1 ← right(p1); /*p1 points to key of proper sublist
    /*search sublist*/
  p2 ← sec(p1);
  while(evtime(p2) > t)p2 ← pred(p2);
  insert(p,p2); /*insert notice after p2*/
    /*check balance*/
  if number(p1) < nlim then number(p1) ← number(p1) + 1
                       else /*check for dummy on left*/
               if dummy(left(p1)) then newkey(p1)
                                  else adjust(p1);
                 /*newkey generates a new secondary key, adjust
                   rebalances the sublists*/
    /*update key lists*/
  number(keycur) ← number(keycur) − 1;
  if number(keycur) = 0  then if dummy(keycur) then movedummy
                                               else delete
                                                    (keycur)
                         /*movedummy moves dummy key(s)
                           and associate pointer to the end of the
                           interval range and adjusts lower-
                           bound*/
end/*hold*/
```

[2] For a complete Pascal implementation see [2].

Table I. Test Design.

Stochastic distributions, representative of various simulation problems, tested:

A. Unimodal continuous
   1. negative exponential (average of 1)
   2. uniform distribution over the interval [0 to 2]
   3. uniform distribution over the interval [0.9 to 1.1]

B. Bimodal continuous
   4. 90 percent probability—uniform over the interval [0 to S]
      10 percent probability—uniform over the interval [100S to 101S] where S is chosen to give the mixed distribution an average value of unity
   5. T is constant with value of unity
   6. T = 0, 1, or 2 with equal probabilities

Determination of estimated average complexity:

initialization:	(i) schedule n events at time 0 "hold" each notice twice according to distribution
computation:	(i) average of 3000–6000 hold operations
	(ii) average of 300–600 hold operations using antithetic t
	(iii) average of (i) and (ii)

Determination of complexity:

elapsed CPU time between call to and return from hold procedure minus linkage overhead and random number generation time

3. Test Results

For the experiments described below, conducted to estimate the average complexity of the hold procedure, we used a revised definition of complexity. For a given configuration of the event set and a given hold time t, we take the processor time (in μsec) expended for the execution of hold(t) as the complexity of the algorithm for that input. We made this change in order to avail ourselves of results in [4]. There a comparative study was made of various data structures and algorithms for the problem under discussion. The language Pascal [3] was used as the test vehicle. Since it is supported by a standardized compiler and since the machine used there is the same (Cyber 74) as used here, the results of this paper and [4] are directly comparable. Actually this definition of complexity measure is more indicative of the real performance because it does take into account the frequency of, and the time needed for, rebalancing the structure and the overhead involved in maintaining the index list.

For reference, the distributions and the test environment are summarized in Table I. Our tests were conducted for n = 2, 5, 50, 100, 200, and 500, using, as in [4], du = 30 and $dt = (8/(n-1) + .07) * \bar{t}$ for $n > 1$ and $dt = 200 * \bar{t}$ for $n = 1$, where $\bar{t}$ is the mean of the hold time distribution. For comparison our version of Figure 3(b) of [4] depicting the results for the indexed list algorithm is given in Figure 4. The agreement is well within the limits of allowable statistical variation except for case 2. A possible explanation is that in [4] the authors inadvertently reported a result for a value of dt not given by the above formula. The corresponding results for the two-level algorithm are given in

Fig. 4. Indexed list algorithm: test results.

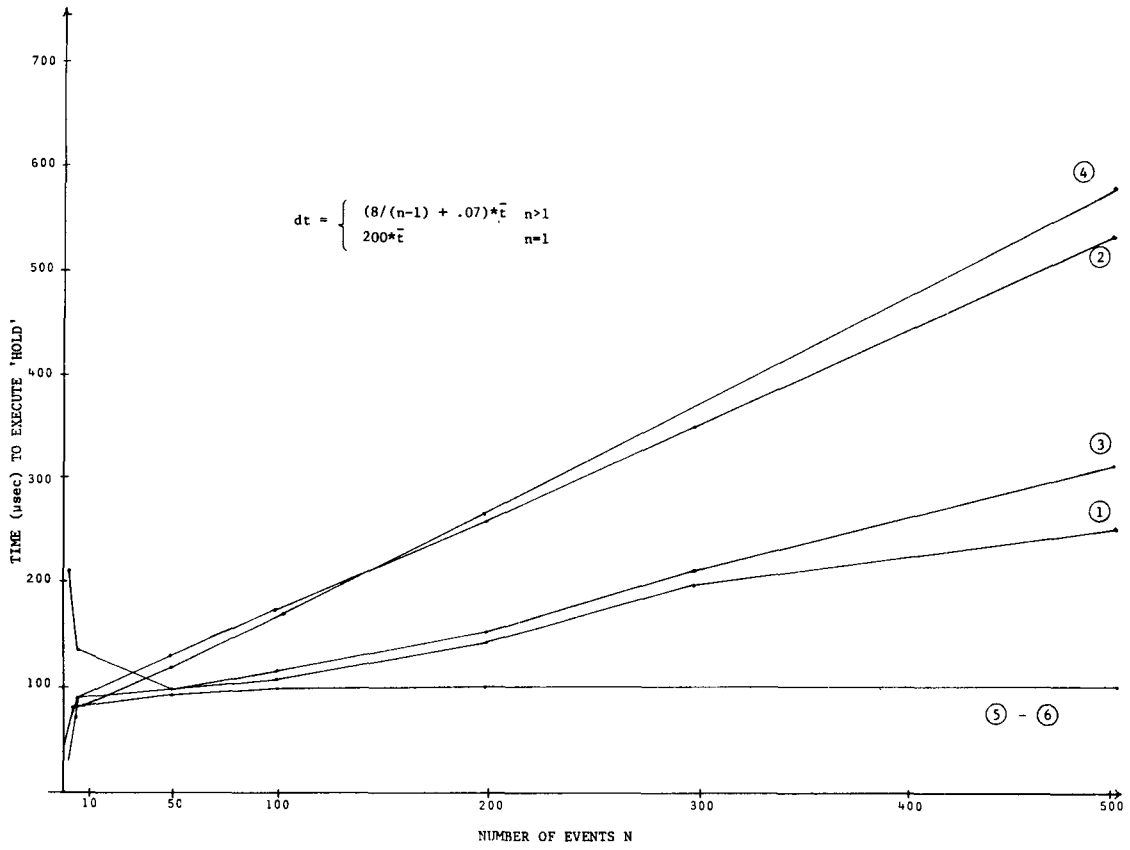


Fig. 5. Two-level algorithm: test results.

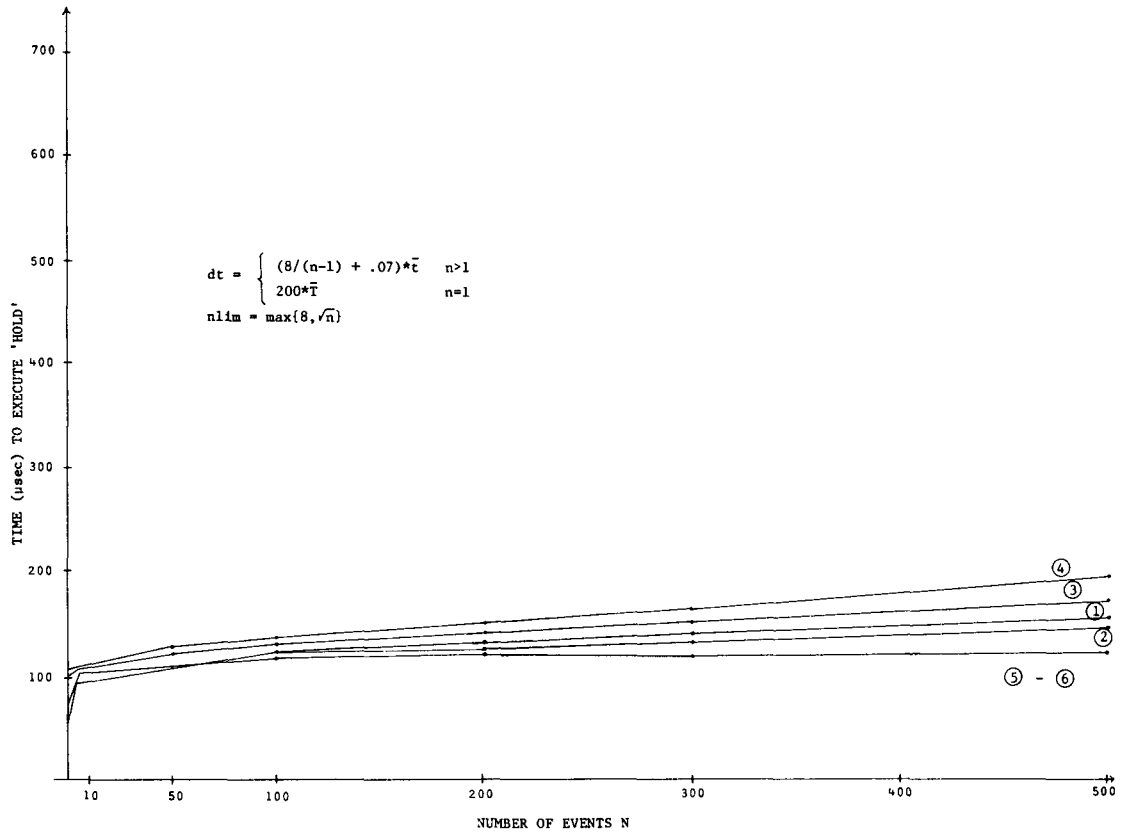


Fig. 6. Mixed distribution test results.

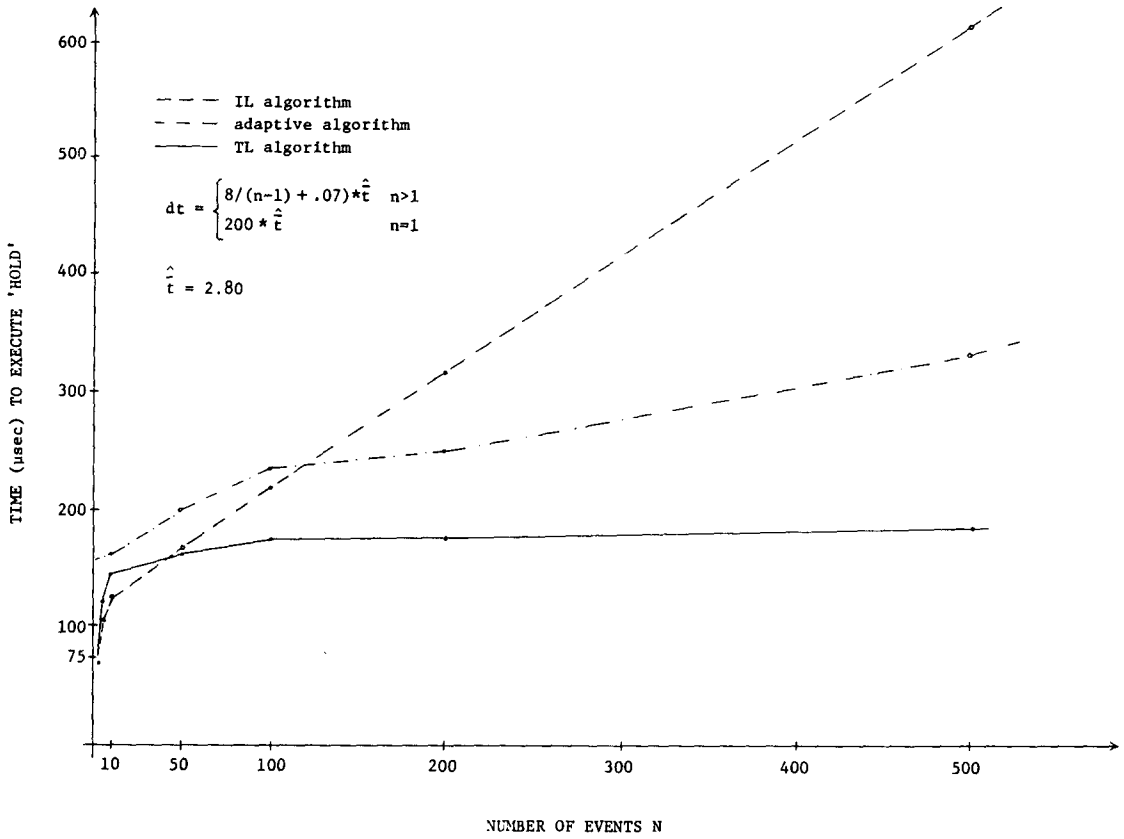


Figure 5. As Figure 5 indicates, the average complexity of the two-level algorithm is nearly independent of n and for practical purposes can be considered $O(1)$. It also is far more insensitive to the distribution of hold times than is the indexed list algorithm. The overhead of the TL algorithm incurred by the complex physical data structure can be ascertained by comparing the curves for distributions 5 and 6 in Figures 4 and 5. For these distributions, both the IL and TL algorithms have identical search times, and the difference in performance can be attributed to the additional maintenance of the data structure by the TL algorithm. Furthermore, we wish to point out the shape of the graph for distribution 4. As expected, the IL algorithm does not perform as well as for the other distributions as shown in Figure 4, but for the TL algorithm the performance is not markedly different.

As we hope is obvious, the two-level structure is designed to reduce the dependence of the search time on n. The value of *nlim* used was determined by analyzing the worst-case behavior of the TL algorithm, denoted by $c_w(n)$, by using key comparisons as basic units, which occurs when all n notices are associated with the same dummy key and the notice being inserted has an event time smaller than all other notices in the structure. That is,

$$c_w(n) = n/nlim + nlim. \quad (1)$$

Under these conditions, $c_w(n)$ is minimized by setting *nlim* to $\sqrt{n}$; thus $c_w(n)$ is $O(\sqrt{n})$. Obviously this selection does not account for the overhead incurred in maintaining a balanced structure.

Experimentally we confirmed that, for the distributions tested, the setting $nlim = \sqrt{n}$ does indeed result in best algorithm performance, except when n is very small. We therefore suggest the setting

$$nlim = max(8, \sqrt{n}). \quad (2)$$

We further note here that the algorithm is written in such a way that *nlim* can be changed dynamically. For instance, if n is not known, the scheduling procedure can keep track of the number of events in the system and increase *nlim* at appropriate times.

The specification of the adaptive structure as reported in [5] is sketchy; for example, it leaves unspecified the mechanism to select the size of the event notice segments. To investigate the performance of the adaptive scheme (as we understood it), we used (2) to determine segment lengths. For $n > 150$ and excluding distributions 5 and 6, the performance curves for the adaptive algorithm are spread between those for distributions 1 and 2 in Figure 4 and have slopes which vary with distribution and are all significantly greater than zero. The important point is that for all values of n and all distributions tested, the TL algorithm is superior to

the adaptive. For distributions 5 and 6, the adaptive algorithm has performance curves identical to those for 5 and 6 in Figure 5.

4. Discussion

It should be clear from Figures 4 and 5 (and the preceding remark) that the performance of the TL algorithm is superior to the IL and adaptive algorithms except for small values of n, and so by implication it is definitely superior to the linear list approach graphically depicted in Figure 8(a) of [4].

One factor contributing to the superior performance of the TL algorithm is its uniform handling of the overflow list (i.e. the rightmost sublist). Specifically the overflow portion is structured in the same way as the remainder of the subintervals so that as it migrates to the left it does not become an excessively long linear list and therefore does not require restructuring.

An interesting question is whether it is fruitful to continue the structuring of the event set towards the direction of balanced trees. Intuitively one would surmise that further levels of indexing will reduce the efficiency since the set is changing rapidly, and rebalancing becomes the major operation. Preliminary experiments indicate that, in fact, further structuring results in a deterioration in performance.

For parameter settings, we observed the same insensitivity to the choice of dt as reported in [4] as long as its value has the proper order of magnitude. Nevertheless, when information is known about the distribution of event times (e.g. approximations to its mean and variance through pilot runs) in the event set, we believe it advantageous to establish theoretically derived criteria for the selection of parameters. The most promising avenue seems to be a distribution dependent approach. That is, establish criteria as facts become known about a particular distribution. For instance, for distribution 5 of Table I, the choice

$$du = 2, \quad dt = \bar{t}, \tag{3}$$

is clearly optimal. Generally we wish to base the values of du and dt on $E(n)$ as well as the mean μ_s and variance σ_s^2 of the distribution of hold times contained in S.[3] We can use the TL structure to obtain estimates of these parameters from pilot runs. Specifically, for pilot runs when the distribution of hold times is unknown, we have found it effective to set $du = 1$, let *nlim* vary upward as a function of n, and collect data to obtain estimates $\hat{\bar{t}}$ of $\bar{t}$ and $\hat{E}(n)$ of $E(n)$. The estimates $\hat{\bar{t}}$ and $\hat{E}(n)$ are then used to parameterize subsequent runs. We applied this procedure in an experiment where the hold times were drawn from a uniform mix of distributions 1–6 (from Table I) with the parameters of those distributions modified in arbitrary ways. A pilot run consisting of 600 hold operations produced the estimate $\tilde{t} = 2.80$. Subsequent experiments using $\hat{t}$ resulted in the curves of Figure 6 and are representative of several such studies. They clearly show the TL algorithm to be superior, by a factor of 2 for $n \geq 400$, to both the adaptive and IL approaches. Additionally, note that the slope of the curve is near zero for the TL algorithm but not so for the others.

Thus, until additional theoretical results become available, we suggest:

1. $du = 30$ (the algorithm is quite robust in du if $20 \leq du < 50$),
2. obtain $\hat{t}$ from pilot runs with $du = 1$,
3. and use $du = 30$, $dt = 8/(n - 1) + .07)*\hat{t}$ in subsequent runs.

The theoretical determination of the functional relationship between μ_s, σ_s^2, $E(n)$ and the pair du, dt is for most distributions an open question and remains an area for further research.

Received November 1975; revised November 1976

References

1. Aho, A., Hopcroft, T.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.
2. Franta, W.R., and Maly, K. An event scanning algorithm of nearly constant complexity. Tech. Rep. 75-18, Univ. of Minnesota, Minneapolis, Minn. Nov. 1975.
3. Jensen, K., and Wirth, W. Pascal user manual and report. *Lecture Notes in Computer Science, Vol. 18*, Springer-Verlag, 1974.
4. Vaucher, J.G., Duval, P. A comparison of simulation event list algorithms. *Comm. ACM 18*, 4 (April 1975), 223–230.
5. Wyman, F.P. Improved event-scanning mechanisms for discrete event simulation. *Comm. ACM 18*, 6 (June 1975), 350–353.

[3] Such parameters are applicable only if the distribution on S becomes (is) stationery, a state often realized for simulations investigating steady state behavior.

Corrigendum. Reports and Articles

Orrin E. Taulbee and S.D. Conte, "Production and Employment of Ph.D.'s in Computer Science – 1976," *Comm. ACM 20*, 6 (1977), 372.

Due to a printer's error, part of the caption for Table VIII did not appear. The caption should read: Faculty Salary Distribution, in Thousands of Dollars* (update of last year's Table F).