Event Lists

1. Event Lists

Based on our algorithm for generating sample paths of a GSMP, it is clear that one way to implement the clock-setting mechanism is to maintain a vector of clock readings (one per event). This approach is not efficient if there are a large number of events, but only a few are scheduled at any time point.

Another approach (which works for GSMP’s with unit speeds) is to maintain a list of the scheduled occurrence times of the active events. This is called a future-event list. This approach is used in most commercial simulation packages. The topic merits some discussion, because event list processing can represent a substantial portion of the CPU time for a simulation---up to 40% by one estimate.

2. Data Structures for Event Lists

The major activities that are performed on the event list are:

1. insertion of events
2. deletion of events
3. determining the next event to occur

By keeping the events sorted by time of occurrence, the above tasks are simplified.

*Linked lists* are one way of accomplishing this.

Deletions always occur at the head of the linked list. Insertions must be done by searching from the head. This algorithm is optimal when there are < 10 events in the list, due to its low overhead.

Using *doubly linked lists*, deletions always occur at the head of the list. Insertions can be done by searching either from the head or the tail (depending on the “time stamp” for the record to be inserted).

*Indexed Lists*

To reduce search time, one can have a pointer to, for example, the middle record in the linked list. Thus, to insert a record, one can determine which half of the list needs to be searched by doing a couple of comparisons. This can save substantial search time, but there is additional overhead required to maintain the middle pointer. The record to which the middle pointer points must be changed dynamically.

A closely related idea is to use a *bucket* system in which events scheduled to occur in the interval \([iw, (i+1)w)\) are stored in bucket \(i\); here \(w\) is the bucket width. (An “overflow” bucket might be necessary to store events scheduled far in the future.) Events within the current bucket are sorted as the bucket is emptied, and newly scheduled events can quickly be assigned to the appropriate bucket.
Heaps

If a list of scheduled occurrence times $t_1, \ldots, t_n$ for events $e_1, \ldots, e_n$ satisfies the property that

$$t_{\lfloor j/2 \rfloor} \leq t_j \text{ for } 1 \leq \lfloor j/2 \rfloor < j \leq n$$

then the list is called a heap. Conceptually, one can think of the $t_i$'s arranged in a binary tree:

```
            t1
           /   \
          t2   t3
         /     \
        t4     t5
       /       \
      t6     t7
```

The heap property means that the time for a node is $\leq$ the time for each of its two child nodes.

It follows from the definition that $t_i \leq t_j$ for $1 < i \leq n$. Thus, the time of the next event can be taken from the “top of the heap”. After such a deletion, the rightmost node at the bottom level is placed at the top of the tree. Call this new top node “node x”. The children of node x are examined, say, from left to right. If the time of the node x is less than the times of all of the children nodes, then the deletion operation is complete. Otherwise, the top node is swapped with a child node that has a lesser time. The process of scanning the children of node x is repeated, until node x “sifts down” to its proper position in the tree. (By “proper”, we mean that the heap property holds at each node.) In an analogous manner, a new node is inserted into the tree and then is “sifted up” to its proper location. (Some care must be taken to deal with simultaneous events.)

Thus the worst-case time to delete or insert an event time is therefore $O(\log n)$, since a binary tree of $n$ nodes has $\log n$ levels. Heap-based methods are therefore well-suited to simulations with many events. (For linked lists or arrays of clock readings, insertions and deletions, respectively, are $O(n)$ operations.)

Further References

The above data structures are those most widely used in commercial discrete-event simulation languages. Actually, many commercial packages use hybrid algorithms that combine these data structures, for example, Henrickson’s algorithm as described in Section 5.3 of L&P. For more on data structures, see The Art of Computer Programming, Volume 1: Fundamental Algorithms by Donald Knuth. Also see the discussion of “priority queues” in Volume 3: Sorting and Searching, Section 5.3 in L&P, Section 2.8 in L&K and Section 1.7 of the book of Bratley, Fox and Schrage give some further references on the topic of event-list management.