Data Structures for Event Lists

Norm Matloff

February 12, 2008

Contents

1 Introduction

Any event- or process-oriented discrete-event simulation package will store all pending events in some kind of **event list**.[1] Let's illustrate that with the SimPy language.

Let's review how a **yield hold** works in SimPy. The main loop in SimPy's **simulate()** function consists of removing the earliest event from the events list **_e.events**, updating the simulated time to the time of that event, and then calling the iterator for that event.

So we see that in a discrete-event simulation package, there are two main operations on the event list:

- Insertion of a new entry into the list, with the position in the list determined by the time of this new event relative to the others.
- Deletion of the earliest item in the list.

There may be other operations needed as well:

- In SimPy, there is a **cancel()** operation. Here we must remove some entry in the event list, and it may not necessarily (and probably won't be) the earliest one.

[1] The reader should take care not to confuse the word *list* here with the Python type **list**.

In simple applications, the event list typically has only a handful of entries. But in some applications there could be thousands of events at a time. Since event list operations comprise the major portion of execution time in a simulation program, it is sometimes worthwhile to use a complex data structure for the event list. Over the years, many such structures have been used. We will get a glimpse of this area here.

2 The Basic Approach: a Linear Linked List

The most basic data structure for storing an event list would be a linked list. The earliest event is at the head of the list, with the second-earliest next, and so on.

This makes deletion of the earliest event very quick. But insertion, plus deletion of general events, can take a long time.

Much research has been done on how to make this approach faster. For example, consider the issue of insertion. With a plain linked list, we must search for the insertion point sequentially, but should we start our search at the front of the list or at the back? Research has shown that it depends on the distribution of **hold times**, meaning the times which are the third operands in the **yield hold** statements. It turns out that the key factor seems to be the **coefficient of variation** of the distribution, which is the ratio of the standard deviation to the mean. If the coefficient of variation is smal, it's better to start at the back of the list. This makes sense, if you think of the extreme case in which the hold times are constant; in such a case, the newest event should always be inserted at the end of the list.

SimPy uses the linear linked list approach.

3 Heaps

Many readers have studied the data structure known as a **heap**. This is a complete binary tree in which each child node has a value greater than or equal to that of its parent. This is well-suited for discrete-event simulation. Say there are n items in the event list. Then there are only $O(\log_2 n)$ levels in the tree, so an insertion can be done in that time, compared to $O(n)$ for a linked list.

Deletion of the earliest event is quick, since it is at the root of the tree, but then the tree must be reorganized, which takes longer than the corresponding operation for a linked list.

4 Calendar Queues

The last data structure we'll look at in this brief introduction is that of **calendar queues**. The name is a metaphor alluding to a personal appointment calendar, with one page for each day, and each page listing the appointments one has on that day.

To illustrate the idea, let's look at a simple example. We'll divide time into four multiples of 0.5: 0.0-0.5, 0.5-1.0, 1.0-1.5 and 1.5-2.0. (This will be relative time, e.g. with 0.5 meaning 0.5 time from now.) This would be considered a "year" consisting of four "days." Say our event list currently consists of events at times 0.12, 0.39, 0.58, 1.22, 1.34, 1.49 and 1.56. The calendar would look like this:

(relative) time	events
0.0-0.5	0.12
0.5-1.0	0.58
1.0-1.5	1.22, 1.34, 1.49
1.5-2.0	1.56

We first delete the 0.12 event, which itself will probably result in the addition of a new event, say at time 1.86.[2] The new calendar will look like this:

(relative) time	events
0.0-0.5	
0.5-1.0	0.58
1.0-1.5	1.22, 1.34, 1.49
1.5-2.0	1.56, 1.86

Note that the simulated clock time has now advanced by 0.12, so the labels on the "days" are a bit misleading, but it's all right as long as we understand this. (Obviously, a precise mathematical formulation could be given.)

We next delete the 0.58 event. Say it then generates an event at time 2.17. Where do we place it in the calendar? The problem here is that this event, in the calendar metaphor, is set to occur "next year." We solve that problem by wrapping around:

(relative) time	events
0.0-0.5	0.17
0.5-1.0	0.58
1.0-1.5	1.22, 1.34, 1.49
1.5-2.0	1.56, 1.86

Our 2.17 event is listed as 0.17. Since we are already past that time—we're in the midst of handling the 0.58 event—it is recognized that this is time 0.17 of the next 2.0 cycle, i.e. it's time relative to the beginning of the current cycle is 0.17 + 2.0 = 2.17.

What if the time had been 2.67 instead of 2.17? Our calendar would look like this:

(relative) time	events
0.0-0.5	
0.5-1.0	0.58, 0.67a
1.0-1.5	1.22, 1.34, 1.49
1.5-2.0	1.56, 1.86

with the 'a' signifying that this is 0.67 in the next cycle.

It's clear how the structure would be coded. In Python, for instance, we could do this as a list of lists. The "outer" list would be for calendar pages, and the "inner" lists would be events per page.

Theoretical and empirical research have shown this approach to work quite well.

[2]Recall why this occurs. When the **Run()** function returns from its **yield hold** which generated the 0.12 event, it will run a few lines and then encounter another **yield hold** (possibly the same statement), at which point it will add the 1.86 event to the list.