Event driven simulation

1 Introduction

Event-driven simulation is a way to simulate continuous time random processes where the state changes only at discrete random times. Rather than moving forward in small time increments, it jumps from event to event using an *event handler* and an *event list*. The event list is a collection of events that have been scheduled to happen in the future. The event handler is the part of the code that processes events as they happen. Handling an event probably means changing the state of the model and scheduling new future events.

To introduce some notation, at any time the *system* (or the *model*) is in some definite state $S(t) \in \mathcal{S}$. There are *event times* $T_1 < T_2 < \cdots$. The event times $T_k < t$ are "in the past", while times $T_k > t$ are in the future. The state S changes only at event times. An event is a pair $E = (T, A)$, where T is the time of the event and A is the *action*. The action is to change the state and possibly create one or more new future events $E' = (T', A')$, with $T' > T$. There are probability distributions, which depend on $S(T)$ and A that determine how the state changes and what new events are created. There is an *event list*, $L(t)$, which is a collection of events $E = (T, A)$ with $T > t$. One step of the event-driven simulation algorithm is to find the event from the event list with the smallest T, handle that event and remove it from the event list, then move the time to that T, and continue in that way.

2 Heap data structure

In Compute Science, *heap* is a data structure that stores objects that are indexed by *keys*. Keys are anything that has a natural order relation, such as numbers or words (in alphabetical order). We will suppose that the key is a floating point number, but the same algorithms work with any other ordered set of keys. The word "heap" in English refers to a disordered pile. You can think of a heap as having no structure except that you can always throw objects onto a heap and you always know which object is at the top of the heap. The Wikipedia page: `https://en.wikipedia.org/wiki/Heap_(data_structure)` has a clear explanation of heaps. Any good computer science algorithms book should cover heaps.

A *heap* allows two or three basic operations:

- **deletemax**: Return and remove the (key,object) pair in the heap with the largest key.
- **insert**: Put a new (key,object) pair into the heap.

- **remove**: (optional) Remove a specific (key,object) from the heap.

The **remove** capability is important in many practical uses of heaps but we describe the heap operations without it. Including **remove** uses other algorithm/data structure ideas that a person trained in that area would be able to supply (pointers and/or hash tables, depending on the implementation). The heap used in event driven simulation typically is upside-down, which means it has a **deletemin** operation that removes the object with the smallest key rather than **deletemax**.

The heap algorithms (see references) have the property that if there are n items in the heap, any one of the operations can be done using at most $\log_2(n)$ simple operatons. For a heap with n = one million, that is at most twenty operations for a **deletemax** or **insert**. This seems good, but the simple versions of heap algorithms can result in in irregular and unpredictable pattern of memory access, which can make them slow in practice. There is fancy high-performance heap software that works better.

The basic heap data structure is a binary branching tree folded into an array. In a binary branching tree, each *internal node* has at most two children. A node with no children is a *leaf*. There is a unique node that is not the child of another node, which is the *root* of the tree. Each node has a key. The heap-tree is in *heap order* if the key of each node is larger than the keys of either of its children. This implies that the root is the node with the largest key. The **deletemax** operation just removes this root. More operations are necessary to re-root this into a heap/tree.

The keys in the heap are stored in a one index array, `A[0], A[1],...,A[n]`. The root is `A[0]`. The children of `A[k]` are `A[2k+1]` and `A[2k+2]`. For example, the children of `A[0]` are `A[1]` and `A[2]`. The children of `A[1]` are `A[3]` and `A[4]`, etc. If there are n objects in the heap, the tree/heap operations make sure that each array element from `0` to `n-1` contains a tree node (internal or leaf). This guarantees that the tree is *balanced* in the sense that the *depth* of the tree is at most $\log_2(n) + 1$. The clever part of the heap algorithm (see references) is a way to insert a new element into the heap so that the new tree is still in heap order and the number of operations (comparisons and exchanges) is at most the depth of the tree. After a **deletemax** operation, the root that was removed is removed by inserting `A[n-1]` into the smaller tree. This way, the array gets shorter by one and there is no "hole" where the root used to be.

Most programming languages have heaps integrated, often under the name *priority queue*. The Python demo code that goes with these notes uses the module `heapq` (presumably for "heap queue"). This simple implementation seems to lack a **remove** feature, which means it cannot be used to implement fancy event-driven simulations.

In event-driven simulation, we typically use a heap with **deletemin** rather than **deletemax**.

3 An example

The model involves things (molecules, phone calls, whatever) that arrive independently and stay "in the system" a random amount of time. In a time dt there is a probability λdt that a new thing arrives. Each thing in the system leaves with probability μdt, with all decisions made independently. This can be a model of the number of some large molecule in a cell. New copies are created with rate λ and existing copies decay (disappear) with rate μ. Both λ and μ are rate parameters for exponential random variables.

For event-driven simulation we need to figure out how to "schedule" future events when they are generated. Suppose a new thing arrives at time T_k. We need to schedule it to leave. If it is still in the system at time $t > T_k$, the probability to leave in the next dt time interval is μdt. Let S be the random time it spends in the system. That μdt model implies that S is an exponential random variable with rate parameter μ. Therefore (as we saw in class), it is possible to generate a sample of this distribution using

$$S = -\frac{1}{\mu}\log(U)\ , \quad U \sim \text{unif}[0,1]\ . \tag{1}$$

The departure time will be $T_k + S$. Thus, the event handler can schedule the departure to happen at time $T_k + S$ when it handles the new arrival at time T_k.

The arrivals may be scheduled in a similar way. If an arrival happens at time T, the next arrival may be scheduled for time $T + S$, where S is an exponential random variable with rate parameter λ. A sequence of arrival times separated by independent exponentials (exponential random variables with a common rate parameter) is called a *Poisson process*, or Poisson *arrival* process.

4 The code

Please download the Python module `EventDrivenLimulation.py` and open it in a code editor that shows line numbers. It is an implementation of the simulation described in Section 3. The following comments are indexed by the line numbers in the code they refer to.

11. `heapq` (probably for "heap queue") is a Python module that implements the priority queue. It came with my Python installation. If you don't have it, you should be able to install it using `pip`.

15. You instantiate the random number generator at the beginning of the program to avoid the mistake of re-instantiating it repeatedly and in that way repeating the same "random number" sequence more than once. Setting the seed makes the code give identical results every time you run it. If you run this one, your graph will be exactly the same as the one posted as `ArrivalDecayProcess.pdf`.

20, 21. The formula (1) for generating exponential random variables uses the inverses of the rate parameters.

31 *heapify* is the name of an algorithm that puts an un-ordered list into *heap order*. I was surprised that the `eventlist` object is not an instance of some sort of *heap* class.

34. An event is a Python two element tuple. The first element is the event time. The second is an object that describes the event. In this simulation, there are three types of event, called `observation`, `arrival`, and `decay`. A more complex simulation might call for us to store more information concerning the event.

35. The operation `heappush` (for "heap push") adds the event to the event list. The terminology comes from the *stack* data structure. In a *stack*, there are operations *push* and *pop*. A *push* adds a new item to the stack and a *pop* returns the most recently pushed item and removes it from the stack. You are supposed to think of a stack of plates, with "push" meaning put a new plate (object) on top of the stack and "pop" meaning take off the top plate. This is called *LIFO* access (Last In First Out). A *queue* is a similar collection, but with *FIFO* (First In First Out) access. In a queue, new plates go on top and places are removed from the bottom (don't try this with breakable dishes!). The authors of this priority queue module chose to use "push" to mean adding an event to the priority queue and "pop" to mean **deletemin**. The "push" and "pop" terminology for stacks of plates is motivated by spring loaded plate dispensers (rarely seen these days). You take a plate off the top and the next one pops up. Click here for an image.

38. This implements the formula (1).

39. You have to put at least one simulation event into the event list or the event handler will have nothing to do.

42. This is a potential *infinite loop*. It's bad programming practice. There should be some "escape valve" such as quitting when the event count gets too high. I didn't put that into this code to keep this code simple.

44. `heappop` (for "heap pop") returns the item in the event list with the smallest `t` and removes it from the priority queue.

46. `break` is a keyword that means: exit from the `while` loop and go to the command on line 68.

48. The event handler does different things depending on what kind of event is being handled.

49. n is the number of things in the system. It goes up when there is an arrival.

51 to 53. Schedule this thing to depart. The time to departure is exponential with rate constant μ and $\beta = \frac{1}{\mu}$. Create a tuple with the time $t + S$ and event type `depart`. Push it to the priority queue.

59 and 60. A decay doesn't create any new events that need to be entered into the event list.

62 to 64. Record n at this time, for plotting. This is not part of the simulation but it is convenient to put these times into the event list. Try to think of a better way to record $n(t)$ at uniformly spaced times.

plot code. Please comment if you think this does not represent good coding standards or add features that make the plot better. Observe (see the file `ArrivalDecayProcess.pdf`) that $n(t)$ has a *transient* where it goes from zero to somewhere between 30 and 50, then it fluctuates approximately around 40. Can you explain why a number around 40 is the long time average value?