Assigning Coordinates to Events: Solving Combinatorial problems using Discrete Event Simulation

David Kitchin, John Thywissen, Jayadev Misra[*]
University of Texas at Austin

November 1, 2011

1 Introduction

This paper is inspired by the "event list" mechanism in discrete event simulations. We argue that descriptions of many combinatorial algorithms can be simplified by casting the solution in terms of processing events according to some order. We propose generalizations of the event list mechanism, and show their applications in problems from graph theory and computational geometry.

Discrete event simulation is used to mimic the real time behavior of a physical system. Typically, the physical system generates "events" at specific times, and the execution of an event may cause other events to happen in the future. Many combinatorial problems have the same structure as discrete event simulation in that they generate events, the events have to be processed in some order, and the processing of events may generate yet more events (to be processed later). Explicit real-time does not play a role in combinatorial problems, but processing of events imposes an order akin to the real-time order. Such problems can be solved by specifying (1) the order in which the events have to be processed, and (2) the steps needed for processing each event, where the steps may also generate new events. The management of the data structure for events, the *event list*, may be delegated to a standard run-time routine called the "event list manager". Structuring the solution in such a manner often simplifies the algorithm description by eliminating book keeping, as we show with several examples in this paper.

Structure of the paper We describe the allowable operations on event list in the next section. These operations are slightly more general than what is normally assumed in discrete event simulations. We show solutions to a number

[*]This author is partially supported by National Science Foundation grant CCF-0811536.

of combinatorial examples in Section 3. We conclude with some remarks on this approach in Section 4.

These ideas have been implemented in a concurrent programming language, Orc [8, 5], designed by the authors and their co-workers.

2 Event List

The combinatorial algorithms we consider consist of processing a sequence of events. At any point, there is a set of events scheduled for future processing; we call this set the *event list*. Initially, some set of events is scheduled. The scheduled events are processed according to some order as long as there is some event in the event list; a processing step may schedule additional events.

In discrete event simulations each event has an associated real time, the time of occurrence of the event in the physical world. For combinatorial problems, we dispense with the notion of "time". Instead, we associate a *coordinate* with each event. The coordinate may be non-numeric. The only requirement is that the coordinates be totally-ordered, so that we can identify an event of the smallest coordinate among the scheduled events. The type of the coordinate and a total order over that type has to be specified for each problem. During execution of an algorithm, the *current coordinate* is the coordinate of the last event whose processing has begun or been completed; it is undefined if no event has yet been processed.

There are three operations on event list.

1. *schedule*(E) where E is a finite, non-empty set of events, is added to the event list. Coordinate of every event in E is at least as large as the current coordinate whenever the current coordinate is defined.

2. *await*(x) removes and returns an event x of the smallest coordinate, according to the given total order, from the event list. The coordinate of x becomes the current coordinate. Additionally, the value of *await*(x) is itself boolean, *true* iff this is the first call to *await* or the coordinate of x differs from the last coordinate returned by *await*. Thus, *await*(x) is *false* if the value of current coordinate does not change. If the event list is empty, *await*(x) never completes.

3. *nonempty* returns a boolean, *true* iff there is some scheduled event.

The standard implementation of the event list uses a priority queue. There may be more efficient implementations for specific problems by exploiting the properties of the problem. For example, in breadth-first search (Section 3.5), we show how the event list can be maintained more efficiently.

3 Examples

Notation Henceforth, we write an event as a tuple whose first component is its coordinate. The coordinate may be the only component for events in some problems.

3.1 Minimum Spanning Tree

Most greedy algorithms can be described in terms of event processing. We consider Kruskal's algorithm [6] for the minimum spanning tree problem.

For a connected, finite, undirected graph, a spanning tree T is a subset of edges so that every node is incident on some edge in T. Further, given that each edge in the graph has a finite positive weight, it is required to find a spanning tree whose combined edge weight is minimum. A well-known algorithm, due to Kruskal [6], starts with T as an empty set, and then processes the edges in order of increasing weights, adding an edge to T provided it does not create a cycle. The algorithm terminates when T has $n-1$ edges where n is the number of vertices.

In the program below, an event corresponds to processing an edge, and the associated coordinate is the edge weight.

```
$schedule(\{(w,e) |$ $e$ is an edge and $w$ is its weight$\})$;
$T := \{\}$;
while $|T| \neq n-1$ do
   $await(w,e)$;
   if $T \cup \{e\}$ has no cycle then $T := T \cup \{e\}$
   else $skip$
od
```

No event is added to the event list in this example; so, the only benefit of the given program is that it avoids an explicit sorting step by delegating that responsibility to the event list manager.

3.2 Huffman Coding

Given a finite non-empty set of positive weights, consider a binary tree where each weight is associated with a terminal node. The weighted path length to a terminal node is the length of the path times the associated weight, and the weight of the tree is the sum of weighted path lengths over all terminal nodes. Huffman's algorithm constructs a tree of the smallest weight, as follows.

Let W be the set of weights. Assign each weight to a distinct tree node. As long as W has more than one element, remove the two smallest elements, x and y, create a node with weight $x+y$ that is the parent of the nodes corresponding to x and y, and add $x+y$ to W.

In the program below, we do not construct the tree, but in each step output (x,y), the weights of the nodes that acquire a common parent in the step. The event list contains the set of weights that do not yet have a parent. Let n be the size of the event list; the program terminates when $n = 1$, i.e., there is a single node, root, without parent.

```
$schedule(W)$; $n := |W|$;
while $n \neq 1$ do
   $await(x)$; $await(y)$; output($x$,$y$); $schedule(\{x+y\})$;
   $n := n-1$
od
```

The algorithm avoids all book-keeping associated with the traditional algorithm. It has a property that can be exploited for slight improvement in performance. The successive weights that are scheduled in the loop are monotonic. Therefore, the weights can be grouped into two sets, the original set W corresponding to the terminal nodes of the tree, and set W' corresponding to the internal nodes which are scheduled in the loop. Elements of W' are produced in order. If W is sorted initially, then removing the elements in order from W and W' amounts to merging these two sorted lists. Merge of two sorted lists can again be structured as processing events from an event list, as shown next.

3.3 Merging sorted sequences

Given are two sequences sorted in ascending order. It is required to merge the sequences and output the elements in ascending order. We use processing an element as an event and the value of the element as its coordinate. The elements in the sequences need not be numeric. In the first version, all elements from both sequences are output, thus, possibly, producing duplicates. Below, $read(x,i)$ reads the next value from sequence i, $1 \leq i \leq 2$, and stores it in variable x.

```
$read(x,1)$; $read(y,2)$; $schedule(\{(x,1),(y,2)\})$;
while $nonempty$ do
   $await(t,i)$; $output(t)$; $read(x,i)$; $schedule(\{(x,i)\})$
od
```

Note that the event list length never exceeds two, because at most one element from either list is scheduled at any point. The program is easily modified to accommodate merging any finite number of sorted sequences.

The following variation outputs only distinct elements, avoiding duplicates.

```
$read(x,1)$; $read(y,2)$; $schedule(\{(x,1),(y,2)\})$;
while $nonempty$ do
   if $await(t,i)$ then $output(t)$; $read(x,i)$; $schedule(\{(x,i)\})$
   else $skip$
od
```

3.4 A Problem attributed to Hamming

The following problem appears in Dijkstra [3]; he attributes the problem to Hamming. It is required to output integers of the form $2^i \times 3^j \times 5^k$ in increasing order, for all non-negative integer values of i, j and k. Dijkstra's solution uses a variable length array in which the outputs are stored, and which can be used to compute the next number to be output. Here, we eliminate the required book-keeping using the event list to store the values that are yet to be output. For every value x that is output, $2 \times x$, $3 \times x$ and $5 \times x$ are scheduled for future output. Duplicate values are not output, using a technique employed in merging sorted sequences (Section 3.3). The coordinate of an event is its magnitude.

```
$schedule(\{1\})$;
**while** $true$ **do**
  **if** $await(x)$ **then** $output(x)$; $schedule(\{2 \times x, 3 \times x, 5 \times x\})$
  **else** $skip$
**od**
```

3.5 Breadth-first Search

Consider a finite directed graph that has a special node called called *root*. The *level* of a node is defined as the length of the shortest path from root to the node; thus, root is at level 0, and any other node has level one higher than its lowest-level predecessor. Breadth-first search visits the nodes by their *levels*. We show a program that schedules nodes for visit based on their levels. The invariant of the program is: (1) every scheduled node is marked, and (2) no node is scheduled more than once. An event here is of the form (h, u) where u is a node and h is its level; processing of (h, u) visits u and schedules its unmarked successors.

```
mark $root$; $schedule(\{(0, root)\})$;
**while** $nonempty$ **do**
  $await(h, u)$;
  visit $u$;
  for each unmarked successor $v$ of $u$:
    mark $v$ and $schedule(\{(h + 1, v)\})$;
**od**
```

It can be shown that the coordinates of nodes in the event list differ by at most 1; the assertion holds vacuously at the start, and for a step in which node u of coordinate h is processed, any scheduled node has level h or $h + 1$, and only nodes of level $h + 1$ are added to the event list. Therefore, the event list can be maintained as a simple queue where $await(h, u)$ removes the element at the head of the queue and $schedule\{(h + 1, v)\}$ adds $(h + 1, v)$ to the rear of the queue. Using a standard event list manager has the limitation that such optimizations can not be exploited.

3.6 Depth-first Search

As in breadth-first search, assume that we have a finite directed graph with a special node called *root*. The program for depth-first search has the same structure as that for breadth-first search in that the scheduled events are of the form (h, u), where u is a node, but the coordinate h is more elaborate. Assume that the label of a node is a unique symbol from an ordered alphabet. An event corresponds to visiting a node, and the coordinate of event corresponding to node u is a string consisting of concatenation of node labels along some path to u. The coordinate of the root is the empty string, ϵ. For node u with coordinate h, the coordinate of a successor node v is hi (that is h concatenated with i), where i is the label of v. Comparing nodes by lexicographic order of their coordinates arranges them in a depth-first search tree. Note the following properties of lexicographic order: for string s and symbols x, y and z,

$$s < sx,\ x < y \Rightarrow\ sx < sy,\ x < y \Rightarrow\ sxz < sy$$

As in breadth-first search, the invariant of the program below is: (1) every scheduled node is marked, and (2) no node is scheduled more than once.

```
mark $root$; $schedule(\{(\epsilon, root)\})$;
**while** $nonempty$ **do**
  $await(h, u)$;
  visit $u$;
  for each unmarked successor $v$ of $u$ with label $i$:
    mark $v$ and $schedule(\{(hi, v)\})$;
**od**
```

3.7 Shortest Path

We first show a concurrent solution to the well-known shortest path problem. Dijkstra's algorithm [2] is a sequential simulation of this algorithm.

It is required to find a shortest path from a *source* node to a *sink* node in a finite directed graph in which each edge has a positive length. We will merely record the length of the shortest path from *source* to *sink*; determining the shortest path itself is a small extension of this scheme. We describe a real-time concurrent algorithm in which each node records the shortest path length to it.

Imagine that the length of an edge is the amount of time taken by a light ray to traverse that edge. First, *source* records 0, the length of the shortest path to itself. Then it sends rays along all its outgoing edges simultaneously at time 0. On receiving a light ray along any of its incoming edges, a node records the time and sends rays along all its outgoing edges immediately. It can be shown that every path length to a node is eventually recorded by the node in order of (non-decreasing) length. A node may stop its computation after recording its first value, because only its shortest path may be included in the shortest path to any other node. The computation is terminated as soon as *sink* records its first value, which is the shortest path length to it.

The Shortest Path Program We implement the scheme described above. An event corresponds to a light ray reaching a node, and the coordinate is the associated path length to the node. A node gets *marked* when a light ray reaches it for the first time. Then its shortest path is recorded and its neighbors are scheduled. This is same as Dijkstra's algorithm except that the book-keeping is delegated to the event list manager. Below, $d(u, v)$ is the length of the edge from u to v.

```
all nodes are unmarked;
$schedule(\{(0, source)\})$;
$await(h, u)$;
**while** $u \neq sink$ **do**
  **if** $u$ unmarked **then** mark $u$;
    for every unmarked successor $v$ of $u$: $schedule(\{(h + d(u, v), v)\})$
  **else** $skip$;
  $await(h, u)$
**od** ;
$output(h)$
```

It is simpler to write this algorithm in a concurrent style that mimics the behavior of the light rays reaching the nodes concurrently; we have a very concise description in the programming language Orc [7].

3.8 Plane-sweep algorithms in Computational Geometry

Plane-sweep is a powerful algorithmic technique in computational geometry. Algorithms using plane-sweep can always be described in the proposed manner. The advantage of such a description is that book-keeping aspects of plane-sweep can be completely ignored. We sketch the outline of an algorithm, due to Bentley and Ottman [1, 9], for enumerating the intersection points of a given set of line segments in a plane.

Line Segment Intersections We are given a finite set of line segments in a plane, where each segment is described by the pair of its end points. It is required to enumerate the points of intersection of all pairs of segments. The algorithm has the following salient features.

1. All end points and intersection points are processed in order of their x-coordinates.

2. The algorithm maintains a list, $\mathcal{A}$. The list is initially empty.

3. Processing a point may update $\mathcal{A}$ and create new intersection points.

We associate an event with each end point and intersection point. Point (x, y) has coordinate x; assume that x-coordinates of all points are distinct. Initially only end points are stored in the event list. At any step, the point with the smallest x-coordinate is removed from the event list and processed, which may cause intersection points to be added to the event list. The algorithm terminates, because there are a finite number of end points and intersection points, and each processing step terminates. The exact details of updating $\mathcal{A}$

are not germane to the discussion here; it can be shown that all intersection points are generated by the processing steps.

```
$schedule(\{(x,y) |\ (x,y)$ is the coordinate of an end point$\})$;
while $nonempty$ do
  $await(x,y)$
  compute intersection points $S$ from $\mathcal{A}$ and $(x,y)$;
  $schedule(S)$;
  update $\mathcal{A}$
od ;
```

4 Cancelling Events

Discrete event simulation allows removing events from the event list, i.e., cancelling a scheduled event. We propose a fourth operation on the event list, $cancel(h)$ that cancels all (possibly 0) events whose coordinate is h.

The Eratosthenes sieve for computing prime numbers from 2 to N can now be easily expressed.

```
$schedule(\{j |\ 2 \leq j \leq N\})$;
$h := 1$;
while $h \leq \sqrt{N}$ do
  $await(h)$; output$(h)$
  $cancel(\{k \times h |\ k > 1$ and $k \times h \leq N\})$
od
```

This example demonstrates one of the limitations of using a standard data structure for a variety of problems. The running time of this algorithms, assuming a typical priority queue implementation of the event list, is $O(n\ \log n)$, because each of the n items is removed from the event list either for output or cancellation. By contrast, a doubly-linked data structure over the elements will run in linear time because each removal then takes constant time.

5 Concluding Remarks

We have argued in this paper that casting certain combinatorial problems as problems of event processing, with the events scheduled according to their coordinates, often eliminates book keeping. The resulting programs have simpler descriptions. For some problems, explicit book keeping may be more efficient; for others, using a standard event list manager is adequate.

The algorithms we have described are all sequential. These algorithms may often be described even more simply in a concurrent style; then, explicit scheduling can be replaced by scheduling a thread in the future. We have successfully applied this technique with concurrent programs in the Orc programming language [8, 5, 4].

References

[1] J. L. Bentley and T. A. Ottman. Algorithms for reporting and counting geometric intersections. *IEEE Trans. on Computers*, 28:643–647, 1979.

[2] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:83–89, 1959.

[3] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[4] Jayadev Misra et. al. Orc language project. Web site. Browse at `http://orc.csres.utexas.edu`.

[5] David Kitchin, Adrian Quark, and Jayadev Misra. Quicksort: Combining concurrency, recursion, and mutable data structures. In A. W. Roscoe, Cliff B. Jones, and Ken Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing. Springer, 2010. Written in honor of Sir Tony Hoare's 75th birthday.

[6] Joseph. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):4850, Feb 1956.

[7] Jayadev Misra. Virtual time and timeout in client-server networks. http://www.cs.utexas.edu/users/misra/VirtualTime.pdf.

[8] Jayadev Misra and William Cook. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling (SoSyM)*, 6(1):83–110, March 2007.

[9] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.