JOURNAL OF COMPUTATIONAL PHYSICS **94**, 255–283 (1991)

How to Simulate Billiards and Similar Systems

BORIS D. LUBACHEVSKY

AT & T Bell Laboratories, Murray Hill, New Jersey 07974

Received October 23, 1989; revised February 14, 1990

An N-component continuous-time dynamic system is considered whose components evolve independently all the time except for discrete asynchronous instances of pairwise interactions. Examples include colliding billiard balls and combat models. A new efficient serial event-driven algorithm is described for simulating such systems. Rather than maintaining and updating the global state of the system, the algorithm tries to examine only essential events, i.e., component interactions. The events are processed in a non-decreasing order of time; new interactions are scheduled on the basis of the examined interactions using preintegrated equations of evolutions of the components. If the components are distributed uniformly enough in the evolution space, so that this space can be subdivided into small sectors such that only $O(1)$ sectors and $O(1)$ components are in the neighborhood of a sector, then the algorithm spends time $O(\log N)$ for processing an event which is the asymptotic minimum. The algorithm uses a simple strategy for handling data: only two states are maintained for each simulated component. Fast data access in this strategy assures the practical efficiency of the algorithm. It works noticeably faster than other algorithms proposed for this model. © 1991 Academic Press, Inc.

1. INTRODUCTION

Many continuous time dynamic systems can be accurately approximated by models whose components evolve independently all the time except for discrete asynchronous instances of pairwise interactions. A typical example is a set of chaotically colliding billiard balls. Each ball moves along a straight line until it collides with another ball or an immobile obstacle. Only pairwise ball collisions are considered, since the probability is zero that more than two balls are involved in the same collision.

Such "billiard" or "hard sphere" models have been in use among computational physicists since the pioneering work [1]. Recently these models have attracted the attention of simulationists [4]. The task of simulation of such a model is the reconstruction of the history of each component. Many models, even as far from billiards as models of combat [11], are conceptually similar to billiards. The similarity is in the techniques for handling spatial combinatorics of multitude of asynchronous pairwise interactions. Processing an interaction (two-ball collision) or an autonomous evolution of a component (moving a billiard ball along a straight line) depends on the specific model in hand.

Most recent attention has been drawn to speeding up billiard-type simulations [4, 8] on parallel computers. However, it is still not obvious how to write such simulations which are efficient in practice on a common serial computer. A "naive" serial algorithm advances the global state of the billiards from collision to collision. The states of all N balls are examined and updated at times $t_0 \leqslant t_1 \leqslant t_2 \leqslant \dots$, where t_0 is the initialization time and t_{i+1} is the nearest next collision time seen at time t_i. The naive scheme is inefficient for large N because

(a) the same collision is repeatedly scheduled an order of N times until it occurs,

(b) at a typical cycle, most balls are not participating in collisions; still, they are examined by the algorithm.

Aside from problems (a) and (b), there exists problem (c) of finding an inexpensive method of determining the nearest collision for a chosen ball. A straightforward method is to compare the chosen ball with $N-1$ others. The standard improvement in this method is the division of the pool table into an order of N sectors. Only balls in the neighboring sectors have to be checked to determine the immediate collision which reduces the work from $O(N)$ to $O(1)$ per one collision scheduled.

A natural idea for improvement in (a) and (b) is to postpone examining and updating the state of a ball until its collision. Implementing this idea does not appear as easy as it might seem. As the simlation progresses, a scheduled collision of a given ball may require rescheduling. The need for such rescheduling and the desire not to lose information about already planned collisions led in [1] to a complicated data structure and update scheme called "time-table" in [3]. Observe that, with all its inefficiency, the naive scheme has an attractively simple double-buffering data structure. The structure consists of only two copies of the global state vector, the old and the new, so that the new vector is computed on the basis of the old one and, in turn, becomes the old one during the next cycle.

We propose a new serial algorithm for simulations like billiards. The attraction of this algorithm is that it utilizes a simple and easy to hand double-buffering data structure, while avoiding problems (a) and (b). Problem (c) is handled in our algorithm using the standard technique of sectoring. In most cases the algorithm examines and processes only the events whose processing is unavoidable, e.g., ball collisions and boundary crossings. Sometimes, like the naive algorithm, it also processes events whose examinining is not necessary. However, the fraction of such overhead events is small and does not grow with N, while the speed-up due to simplicity of data handling is substantial. The proposed algorithm achieves the same theoretical optimal performances as other published algorithms, i.e., $O(\log N)$ instructions per one porcessed event with sectoring and $O(N)$ without. But its practical speed for the billiard case is at least an order of magnitude higher than that of other algorithms. (We compare, of course, computer-independent algorithmic speeds.) We were able to handle millions of collisions on a non-supercomputer

VAX8550 using FORTRAN. Using languages better adapted for the computer should result in additional speed-ups.[1] The computer handled 2000 balls in most experiments (see, e.g., Fig. 9.2 in Section 9). If needed this number could be easily increased to 10^4.

When writing this algorithm special attention was paid to an often overlooked trade-off between the complexity of data organization and the amount of computations the algorithm is willing to abandon risking to incur them again. For example, to determine the next collision for a ball A we have to try to have it collide with any other ball (within its neighborhood, in the presence of sectoring) and then choose the collision closest in time, say it is a collision with ball B. However, the tentative parner B, in its turn, may choose an even better party C, $C \neq A$. As a result, later in the computations, A might figure out that the party $B1$, which was previously rated second to best, is to be considered the best.

Rating potential candidates costs computations: the algorithm tries to simulate a potential collision of ball A with a candidate X in order to rate this X. Should the algorithm retain the results of these preliminary simulations which correspond to the second, third, ..., best parties $B1$, $B2$, ... when the best candidate B is being chosen? Or is it more economical to abandon the information obtained during the rating and, if later needed, simulate these collisions again? The answer determines the data organization strategy which crucially affects algorithm efficiency. In the billiard case, a "pack rat" strategy entailing a search through dumped items to find the needed one incurs too high a cost. In a general case, the best trade-off depends on the relative cost of the basic operations, e.g., the amount of computing needed for repeated scheduling versus that needed to retrieve the same data.

Another scale of strategies and the associated trade-off is that of the "aggressiveness" of precomputation. In a more aggressive strategy, when the next collision for ball A is being scheduled, not only the existing states of other balls are taken into account but also their possible future states which might result from their as yet unprocessed collisions. The degree of aggressiveness might be measured in how many future collisions with the other balls are considered ahead. The two scales are correlated: a more aggressive precomputation requires a more complicated data structure and encourages the choice of a more "pack rat" data handling strategy.

In the described two trade-off scales, the strategy used in the proposed algorithm is close to the "wasteful" and "lazy" ends of the scales, the opposite of the "pack rat" and "aggressive" ones. Both the storage of not immediately needed data and precomputation lookahead are reduced. The candidates for the next collision for a

[1] In the UNIX environment, which is standard for AT & T computers, C-language is a better choice. The author tried to exploit automatic FORTRAN $\rightarrow$ C translator (named F2C), recently developed in AT & T Bell Laboratories. The created C-code worked about 10% faster than the original FORTRAN code. If a manually produced C-code was used, one would expect 20–30% speed-up. Profiling the code suggests that additional, perhaps, 50% speed-up is achievable by the usual lower level optimization, such as the reduction of the number of subroutine calls and indirect array references. These suggestions have not been vigorously tried yet, because the code "as is" seems ot be sufficiently fast for the considered applications.

particular ball which are rated below the winner are abandoned once the winner is chosen. The future collision of a ball is predicted based only on the existing states of the other balls, not on their future states after possible future collisions.

For a reader who is not familiar with simulation terminology, it is worth adding that the proposed is an *event-driven* simulation algorithm in which the state of the simulated system is examined by the computer only at the times of the *events*, e.g., ball collisions. A physicist may be more familiar with the *time-driven* simulation algorithms. Such algorithms (see, e.g., [5]) are usually employed in the many-body problems in which the components, say particles, rather than evolving almost always independently, are continuously interacting by exerting short and/or long range forces. The two computational approaches are radically different and each has its own difficulties.

A time-driven algorithm would maintain the snapshot of the states of all simulated components at a time t and would advance the time from t to $t + \Delta t$ by modifying all these states. To assure sufficient precision Δt should be rather small. As a result, the time-driven algorithm would be tremendously slower than the proposed event-driven algorithm. Event-driven algorithms are the best (and often the only practical) choice for models where discrete instantaneous events occur asynchronously. In the proposed algorithm, if an event involving ball A is processed for simulated time t, only the state of A is examined and explicitly modified. The states of most other balls need not be known at t and are not examined by the algorithm. In fact, the global state is explicitly known at no time t, except $t=0$. However, if we wish to know the global state, say, if we wish to known the location of each ball at a particular time t, then additional computations of "projecting" the motions of all balls into time point t are required.

The rest of the paper is organized as follows: In Sections 2 to 6, a definition of the basic operations, the data organization, the formulation of the algorithm with examples of its run, and some comments on the experience of its implementation are given. These sections should be sufficient for a reader who wishes to understand and write a simulation algorithm for a billiard-like system. Sections 7 and 8 introduce, explain, and analyze the conditions under which this algorithm works correctly. Section 9 presents an application example for the algorithm: a disk packing problem; Section 10 compares the performance of this algorithm with other published proposals, and the Conclusion discusses variants of the billiard simulation and other simulation models like billiards, including combat models.

2. Basic Operations

Assume that a basic function *interaction_time* is available which, given *state*1 of component 1 at *time*1 and *state*2 of component 2 at *time*2, computes the *time* of the next potental interaction while ignoring the presence of other system components:

$$time \leftarrow interaction\_time\ (state1, time1, state2, time2), \tag{2.1}$$

where $time \geqslant \max(time1, time2)$. If *interaction_time* cannot find such finite *time*, e.g., when two billiard balls are moving away from each other, we assume that $+\infty$ is returned.

In the billiard simulation, the state of a ball is the pair of vectors *state* = (*position*, *velocity*). If the velocities of the balls are constant between the collisions, and all balls are of the same constant diameter D, then (2.1) is of the form

$$\begin{aligned}
&time \leftarrow \max(time1, time2) + t \\
&\text{where} \\
&t = \begin{cases} (-b - \sqrt{b^2 - ac})/a, & \text{if } b \leqslant 0 \text{ and } b^2 - ac \geqslant 0 \\ +\infty, & \text{if } b > 0 \text{ or } b^2 - ac < 0 \end{cases} \\
&\text{and} \\
&a = |velocity2 - velocity1|^2, \\
&b = (position20 - position10) \cdot (velocity2 - velocity1), \\
&c = |position20 - position10|^2 - D^2, \\
&position10 = position1 + velocity1\ (\max(time1, time2) - time1), \\
&position20 = position2 + velocity2\ (\max(time1, time2) - time2),
\end{aligned} \tag{2.2}$$

where $u \cdot v$ denotes the dot product of vectors u and v, and $|v|$ denotes the length of vector v: $|v| = (v \cdot v)^{1/2}$. The expression for t in (2.2) is the least real solution $t = t_-$ of the equation $at^2 + 2bt + c = 0$ which is derived from $|p + vt|^2 = D^2$, where $p = position20 - position10$ and $v = velocity2 - velocity1$. The meaning of the latter equation and of both its solutions $t = t_-$ and $t = t_+$ is obvious from Fig. 2.1. Note that c may be 0 in which case $t = t_- = 0$ and $time = \max(time1, time2)$. This means that *interaction_time* is applied when one ball is already at the site of the scheduled collision.

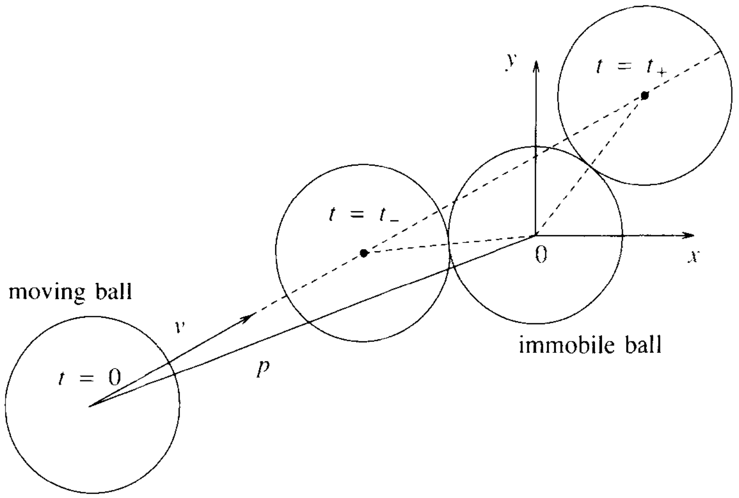


FIG. 2.1. The geometrical meaning of the two solutions of equation $|p + vt|^2 = D^2$.

Two components 1 and 2 with *state*1 and *state*2 are said to be *interacting*, if

$$\textit{interaction\_time}\ (\textit{state}1, \textit{time}, \textit{state}2, \textit{time}) = \textit{time} \tag{2.3}$$

holds for any value of *time*. For example, billiard balls i and j of diameter D each with velocities and positions v_i, p_i and v_j, p_j, respectively, are interacting (i.e., colliding), if

$$|p_j - p_i| = D,\ (v_j - v_i)\cdot(p_j - p_i) \leqslant 0. \tag{2.4}$$

Assume that a basic function *jump* is available which, given *state*1 and *state*2 of interacting components 1 and 2, computes *new_state*1 and *new_state*2 of these components immediately after the interaction:

$$(\textit{new\_state}1, \textit{new\_state}2) \leftarrow \textit{jump}\ (\textit{state}1, \textit{state}2). \tag{2.5}$$

When billiard balls 1 and 2 with vector velocities v_1^{old} and v_2^{old} collide, only their velocities experience jumps, not positions. Assuming the energy and momentum are conserved, the tangential components of the initial velocities are not changed, but the normal components are switched as depicted in Fig. 2.2. The velocities after the collision are v_1^{new} and v_2^{new}.

A ball bouncing off a boundary of the pool table, in principle, needs not be examined by the algorithm. It may be considered as an ordinary point on the autonomous interval of the trajectory. For example, given positions $(x0, y0)$ and the velocity vector v of a ball at time $t = 0$, we can construct functions $X(v, x0, y0, t)$ and $Y(v, x0, y0, t)$ which would return the position $x = X$ and $y = Y$ of the ball at time t without explicitly processing intermediate boundary reflections.

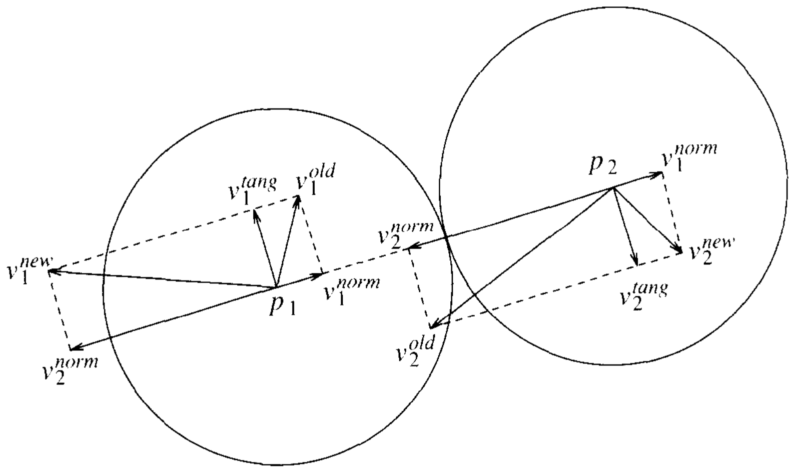


FIG. 2.2. Change of velocities of two billiard balls at their collision.

The complexity of computations by functions $X(\ )$ and $Y(\ )$ would not depend on the number of bounces. (This holds at least for a rectangular pool table.)

In most applications, however, such elaboration would be of little practical use, because the ball would usually collide with another ball after at most one boundary reflection. Besides, an explicit examination of the reflection event might be needed anyway for statistical purposes and for convenience of data update. Thus we will treat a reflection from an immobile obstacle as a separate event.

A boundary crossing may be considered under a periodic boundary condition model, wherein a ball, rather than bouncing off, disappears at a boundary and reappears at the opposite side (see Fig. 2.3). This may be treated as the same type of event as boundary reflection. If the pool table is divided into sectors, a similar type of event constitutes a ball moving from one sector to another. Such an event should be examined by the algorithm in order to update the membership in the sectors. We will treat all such events as one-component interactions.

We will assume that basic functions with the same names *interaction_time* and *jump* represent one-component interactions

$$time \leftarrow interaction\_time\ (state1,\ time1,\ obstacle) \tag{2.6}$$

$$new\_state \leftarrow jump\ (state,\ obstacle), \tag{2.7}$$

where *obstacle* is the identification of a boundary or an immobile obstacle or a demarcation line. To apply *jump* in (2.7), the component, whose *state* is represented in (2.7), must be *interacting* with *obstacle*. The condition which defines such one-component interaction is

$$interaction\_time\ (state1,\ time,\ obstacle) = time \tag{2.8}$$

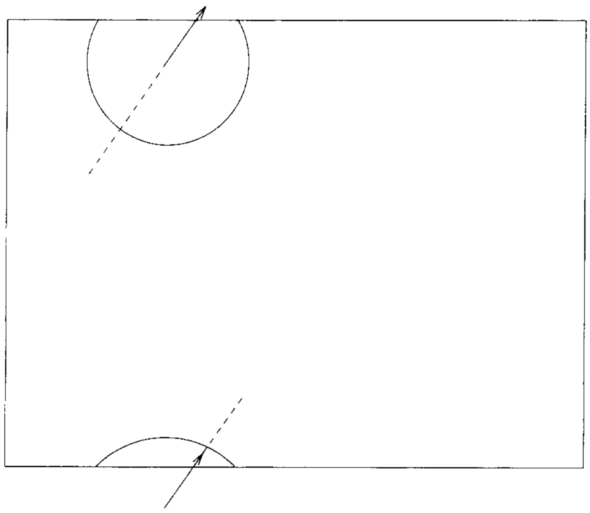


FIG. 2.3. A ball is disappearing at a boundary and reappearing at another side.

holds for any *time*. This is similar to (2.3). Capital K will be reserved for the number of obstacles so that *obstacle* in (2.6), (2.7), and (2.8) is an integer in the interval from 1 to K. The one-component versions of functions *interaction_time* and *jump* will be easily distinguished by context from their two-component synonyms in (2.1) and (2.5).

We also assume the availability of a basic function *advance* which, given *state*0 of a component at *time*0 and a value *time*1 $\geqslant$ *time*0, returns *state*1 this component would have at *time*1 ignoring possible interactions with other components or obstacles on the interval [*time*0, *time*1]:

$$\mathit{state}1 \leftarrow \mathit{advance}\ (\mathit{state}0, \mathit{time}0, \mathit{time}1). \tag{2.9}$$

In a frictionless billiard, (2.9) is of the form

$$\mathit{position}1 \leftarrow \mathit{position}0 + (\mathit{time}1 - \mathit{time}0)\, \mathit{velocity}0 \tag{2.10}$$

$$\mathit{velocity}1 \leftarrow \mathit{velocity}0$$

which simply says that the ball moves with *velocity*0 along a straight line starting from *position*0 at *time*0.

Note that in particular cases, specific calculations for *interaction_time*, *advance*, and *jump* may be not as simple as in the billiard case. The assumption that they are *basic* saves us from the burden to detail them in the general discussion.

3. Data Organization

The basic data unit of the algorithm is called *event* and has the format

$$\mathit{event} = (\mathit{time}, \mathit{state}, \mathit{partner}), \tag{3.1}$$

where *time* is the time to which *state* of a component corresponds. Note that *state* is the new state of the component *immediately after* the event, e.g., if a ball has experienced a collision at *time*, the velocity-coordinate of the *state* is the new velocity vector after the collision; *partner* identifies the other component, if any, involved in the event. If there is no partner in the event, the program assigns a special "no-value" symbol Λ to the *partner* coordinate. If *time* $= +\infty$, then the other two coordinates in the *event* have no value; i.e., *state* = *partner* = Λ.

At any stage of simulation, the algorithm maintains two events for each component: an old, already processed in the past event and a new, next scheduled event. This information is stored in array *event* [1:N, 1:2], where N is the number of components of the simulated system. Let us agree to understand a reference like *time*[3, 1] as the *time* coordinate of element *event*[3, 1] of this array.

Two arrays, *new*[1:N] and *old*[1:N], with elements equal to 1 or 2 are maintained. For component i, *new*[i] is the pointer to the new event and *old*[i] is the pointer to the old event. Thus the new event for component i is stored at *event*[i, *new*[i]] and the old event is stored at *event*[i, *old*[i]]. When *new*[i] is updated, *old*[i] is updated immediately afterward, so that the relation *new*[i] + *old*[i] = 3 remains invariant.

4. The Algorithm

In the algorithm pseudocode in Fig. 4.1, /" and "/ mark the beginning and the end of a comment, the minimum over an empty set of values is assumed to be $+\infty$. The following short-hand notations are used:

$P_{ij} \overset{\text{def}}{=}$

$\textit{interaction\_time}(\textit{state}[i, \textit{old}[i]], \textit{time}[i, \textit{old}[i]], \textit{state}[j, \textit{old}[j]], \textit{time}[j, \textit{old}[j]]),$

```
initially $current\_time \leftarrow 0$ and for $i = 1, 2, \ldots N$ :
$new[i] \leftarrow 1$, $old[i] \leftarrow 2$, $time[i,1] \leftarrow 0$, $partner[i,1] \leftarrow \Lambda$,
$state[i,1] \leftarrow$ initial state of component $i$, $event[i,2] \leftarrow event[i,1]$

1.  while $current\_time < end\_time$ do {
2.    $current\_time \leftarrow \min_{1 \le i \le N} time[i, new[i]]$ ;
      $i_* \leftarrow$ an index which supplies this minimum (i.e., $current\_time$) ;
3.    $new[i_*] \leftarrow old[i_*]$ ;  $old[i_*] \leftarrow 3 - new[i_*]$ ;
4.    $\mathbf{P} \leftarrow \min_{j \in A(i_*)} P_{i,j}$ , where $A(i_*) = \{j \mid 1 \le j \le N,\ j \ne i_*,\ time[j, new[j]] \ge P_{i,j}\}$ ;
      if $\mathbf{P} < +\infty$ then $j_* \leftarrow$ an index which supplies this minimum (i.e., $\mathbf{P}$) ;
5.    $\mathbf{Q} \leftarrow \min_{k \in B} Q_{i,k}$ , where $B = \{k \mid 1 \le k \le K\}$ ;
      if $\mathbf{Q} < +\infty$ then $k_* \leftarrow$ an index which supplies this minimum (i.e., $\mathbf{Q}$) ;
6.    $\mathbf{R} \leftarrow \min\{\mathbf{P}, \mathbf{Q}\}$ ; $time[i_*, new[i_*]] \leftarrow \mathbf{R}$ ;
7.    if $\mathbf{R} < +\infty$ then {
8.      $state1 \leftarrow advance\,(state[i_*, old[i_*]], time[i_*, old[i_*]], \mathbf{R})$ ;
9.      if $\mathbf{Q} < \mathbf{P}$ then {
10.       $state\,[i_*, new[i_*]] \leftarrow jump\,(state1, k_*)$ ;
11.       $partner\,[i_*, new[i_*]] \leftarrow \Lambda$ ;
        } /" end $\mathbf{Q} < \mathbf{P}$ close "/
12.     else {    /" case $\mathbf{Q} \ge \mathbf{P}$ "/
13.       $time[j_*, new[j_*]] \leftarrow \mathbf{R}$ ;
14.       $state2 \leftarrow advance\,(state[j_*, old[j_*]], time[j_*, old[j_*]], \mathbf{R})$ ;
15.       $(state\,[i_*, new[i_*]], state\,[j_*, new[j_*]]) \leftarrow jump\,(state1, state2)$ ;
16.       $m_* \leftarrow partner\,[j_*, new[j_*]]$ ;
17.       $partner\,[i_*, new[i_*]] \leftarrow j_*$ ; $partner\,[j_*, new[j_*]] \leftarrow i_*$ ;
18.       if $m_* \ne \Lambda$ and $m_* \ne i_*$ then { /" update third party $m_*$"/
19.         $state\,[m_*, new[m_*]] \leftarrow$
              $advance\,(state[m_*, old[m_*]], time[m_*, old[m_*]], time[m_*, new[m_*]])$ ;
20.         $partner\,[m_*, new[m_*]] \leftarrow \Lambda$ ;
          } /" end update third party "/
        } /" end $\mathbf{Q} \ge \mathbf{P}$ close "/
      } /" end $\mathbf{R} < +\infty$ close "/
    } /" end while loop "/
```

Fig. 4.1. The simulation algorithm.

where $1 \leqslant i, j \leqslant N$ and

$$Q_{ik} \overset{\text{def}}{=} \mathit{interaction\_time}(\mathit{state}[i, \mathit{old}[i]], \mathit{time}[i, \mathit{old}[i]], k),$$

where $1 \leqslant i \leqslant N$ and $1 \leqslant k \leqslant K$.

The main cycle in Fig. 4.1 consists essentially of two steps:

(1) selecting the next component i_* to process its event (line 2),

(2) processing the event (the rest of the cycle).

Processing the event means scheduling next events for the chosen component and the other involved components, if any. **P** and **Q** are the nearest next interaction times. There are two main cases in such scheduling depending on the type of the future event:

(a) $\mathbf{Q} < \mathbf{P}$, when the scheduled interaction involves only one component i_* (lines 8, 10, and 11);

(b) $\mathbf{Q} \geqslant \mathbf{P}$, when the scheduled interaction involves i_*, a second party j_* (lines 8 and 13–17), and may involve a third party m_*, the previous partner, if any, of j_* (lines 19 and 20).

Section 5 further explains this algorithmic structure in examples. Now we discuss the aspects of the algorithm which are *not* represented in the aggregated pseudocode in Fig. 4.1, namely the way minimizations in lines 2, 4, and 5 are implemented. Since these techniques are well known (see, e.g., [6]), their discussion will be brief.

A straightforward method to find the minimum of

$$\mathit{time}[i, \mathit{new}[i]]$$

for i ranging from 1 to N in line 2 requires $O(N)$ operations per event. To reduce the cost, the algorithm instead organizes values

$$\mathit{time}[i, \mathit{new}[i]]$$

into an implicit *heap* structure. Two pointer arrays $\mathit{pht}[1{:}N]$ and $\mathit{pth}[1{:}N]$ are maintained so that

$$\mathit{time}[\mathit{pht}[m], \mathit{new}[\mathit{pht}[m]]]$$

is the value which is implicitly located at the mth position of the imaginary heap array and pth is the inverse map for pht, i.e., pht, i.e.,

$$\mathit{pth}[\mathit{pht}[m]] = m$$

for all m. In particular,

$$\mathit{time}[\mathit{pht}[1], \mathit{new}[\mathit{pht}[1]]]$$

corresponds to the heap tree root, i.e., the minimum value, so that line 2 can be simply rewritten as

$$i_* \leftarrow \mathit{pht}[1], \qquad \mathit{current\_time} \leftarrow \mathit{time}[i_*, \mathit{new}[i_*]].$$

This method requires updating the heap structure (arrays pht and pth) each time a value of

$$\mathit{time}[i, \mathit{new}[i]]$$

is changed in other sections of the algorithm. Including this updating, the total cost of finding the minimum next event time is $O(\log N)$ operations per one event.

The main difficulty in the direct method for minimization in lines 4 and 5 is the need to compute the $N-1$ values P_{i_*j} in line 4 and the K values Q_{i_*k} in line 5. The opportunity to decrease the $O(N+K)$ complexity burden of these computations depends on the topology of the evolution space and the uniformity of the component and obstacle distribution in this space. In the billiard case, the space is just the Euclidean plane and there is an independent of N maximum number of balls which can be located in a bounded vicinity of a given ball. To exploit this boundedness, the simulation space is divided into sectors and only the components or obstacles incident to the neighboring sectors are examined. The sector boundaries naturally become additional "obstacles" in this method and examining boundary crossings constitutes the method's overhead. The complexity in this method reduces to $O(1)$.

Among the available grids used for planar sectorization, the grid of equal squares is the most convenient. (The ratio of the area to the perimeter of a sector is larger for the hexagonal grid, though.) Specifically in the case of equal balls, we usually choose square sides larger than the ball diameter, and for each square we maintain the membership linked-list of balls whose centers project to this square. When a square boundary crossing is processed the two lists are updated. Only those P_{i_*j} are computed for which the center of ball j belongs to one of the nine sectors neighboring the one whose member is i_*. This small number of P_{i_*j} are subject to minimization in line 4.

5. Computational Examples

Two examples of the execution of the algorithm in Fig. 4.1 are reproduced in Figs. 5.2 and 5.3. Both simulate four-ball billiards on a square pool table. Unlike real billiards with hard wall boundaries, periodic boundary conditions are assumed (these conditions are explained in Section 2, see Fig. 2.3). In the example shown in Fig. 5.2, the table is subdivided into 3×3 equal square sectors. Figure 5.2 consists of three frames, 5.2a, 5.2b, and 5.2c. Each frame shows a snapshot of the simulation state at a particular *current_time* with the identification, position, and velocity vector of each ball at this time. Since the execution state usually does not contain the positions of all the balls at the same *current_time*, a picture-producing routine (not considered in this discussion) accepts $t =$ *current_time* as an input and interpolates between the *old* and the *new* positions of the ball as shown in Fig. 5.1. Note that while Fig. 5.1 shows a "general" case, with $time[i, old[i]] < t < time[i, new[i]]$, the snapshots in Fig. 5.2 and 5.3 have many "degenerated" cases, e.g., $time[i, old[i]] = t$. Also note that to simplify the pictures, times are rounded off to their integer parts while the computer manipulates them with the machine precision for representing real numbers.

Figure 5.2a shows the positions and velocities of the four balls at *current_time* $= 0$. These quantities are the initial values. Observe that no two balls

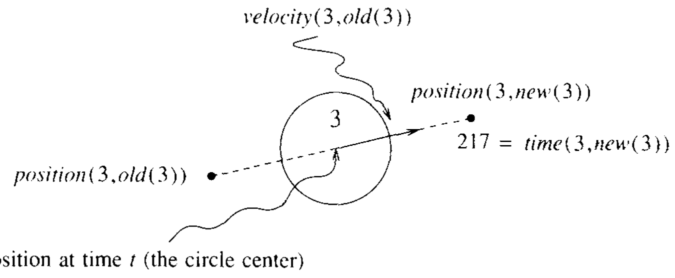


FIG. 5.1. Ball 3 at time *current–time* = *t* (a legend for Fig. 5.2 and 5.3).

overlap. (A method to define such initial positions is discussed in Section 9. Correct simulation should preserve this property.) As the initialization statement in Fig. 4.1, reads, the balls are initialized at the same zero *time* with identical *old* and *new* events. Succeeding the test in line 1, Fig. 4.1 (assuming *end_time* is sufficiently large), the algorithm is searching for a ball index i_* which yields the minimum to $time[i, new[i]]$. As Fig. 5.2a indicates, the algorithm has chosen ball 1. Observe that in the beginning of simulation, all four *new* events have the same *time* so the other three choices are correct. After switching the senses of *old* and *new* event storages for ball 1 in line 3 (here a redundant manipulation), in line 4 the algorithm tries to select the ball with which ball 1 will collide first. Since $time[i, new[i]] = 0$ for all i and all $P_{ij} > 0$, no j satisfies $time[j, new[j]] \geqslant P_{i_*j}$. This means that the set subject to minimization in line 4 is empty. Hence, $\mathbf{P} = +\infty$, and no j_* is selected. In line 5 the algorithm selects the boundary k_* which will be reached by ball 1 first. This boundary happens to be the lower side of the sector to which *position*[1, *old*[1]] belongs and the ball reaches it (in the absence of other balls) at time $\mathbf{Q} = 58$. In line 6, $\mathbf{R}$ and *time*[1, *new*[1]] are becoming this time. Tests in line 7 and 9 are succeeding and the rest of cycle 1 is spent on assigning the scheduled values in lines 8, 10, and 11 to the *new* coordinates. These new values will come into

---

FIG. 5.2. (a) Result of cycle 1; *current–time* = 0; ball 1 has scheduled a boundary crossing for time 58. (b) Result of cycles 2, 3, and 4; *current–time* = 0; ball 2 has scheduled a boundary crossings for time 124; ball 3 has scheduled a boundary crossing for time 150; ball 4 has scheduled a collision with ball 1 for time 25. (c) Result of cycles 5 and 6; *current–time* = 25; balls 1 and 4 have processed a collision for time 25; ball 1 has scheduled a boundary crossing for time 94; balls 2 and 4 have scheduled a collision for time 87.

FIG. 5.3. (a) Result of cycles 1 to 4; *current–time* = 0; balls 1 and 4 have scheduled a collision for time 25; balls 2 and 3 have scheduled a collision for time 388. (b) Result of cycle 5; *current–time* = 25; ball 1 has processed its collision with ball 4 for time 25; balls 1 and 2 have scheduled a collision for time 226; ball 2 canceled an earlier scheduled collision with ball 3 for later time 388 and this collision is turned into an advancement for ball 3. (c) Result of cycle 6; *current–time* = 25; ball 4 has processed its collision with ball 1 at time 25; balls 2 and 4 have scheduled a collision for time 87; ball 2 has canceled an earlier scheduled collision with ball 1 for later time 226 and this collision is turned into an advancement for ball 1.

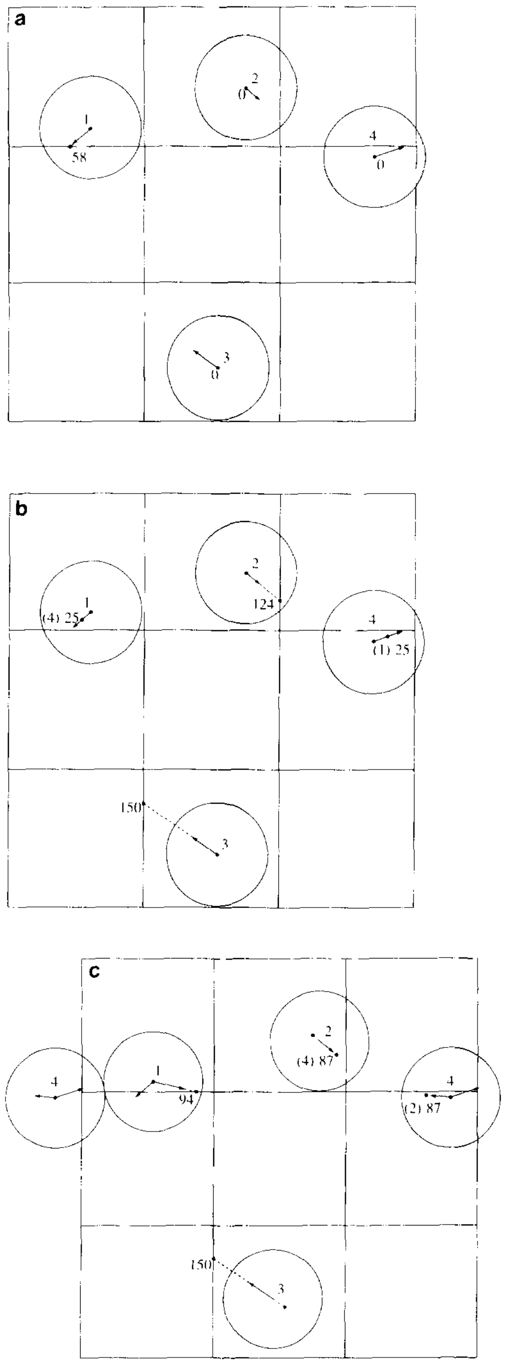


FIGURE 5.2

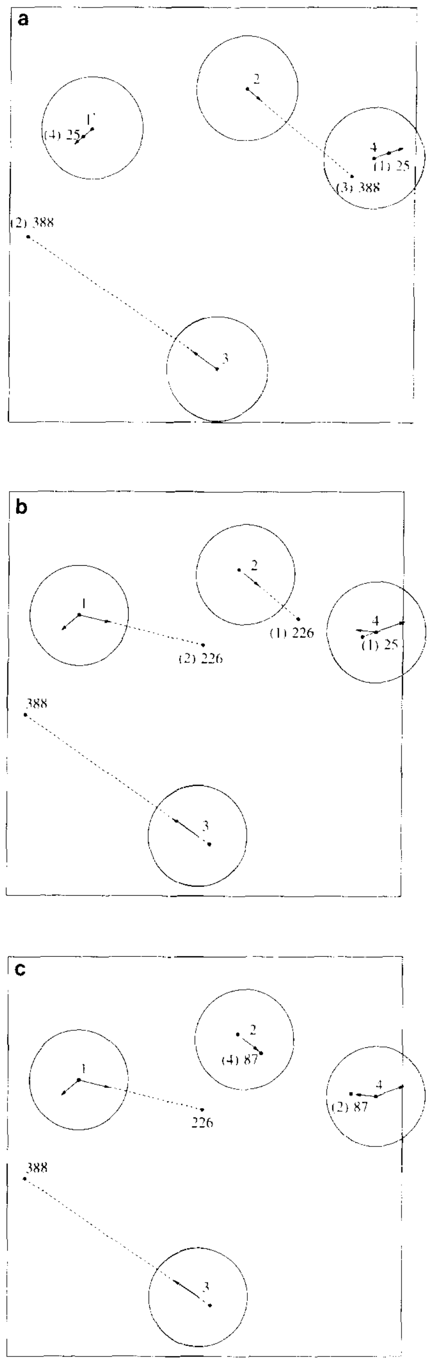


FIGURE 5.3

effect immediately after crossing the specified boundary. Note that if *obstacle* is a demarcation boundary between sectors, then *jump* is defined as an identical function: *jump* (*state*, *obstacle*) $\overset{\text{def}}{=}$ *state*. The algorithm then takes the snapshot of the situation (see Fig. 5.2a), after which cycle 2 is started. In the snapshot, ball 1 has a scheduled event at time 58, while the other three balls still have scheduled events at time 0 as indicated.

Cycles 2, 3, and 4 are spent scheduling future events with positive times for the remaining three balls. Update initiators i_* are chosen in the following order: ball 2 becomes i_* at cycle 2, ball 4 becomes i_* at cycle 3, and ball 3 becomes i_* at cycle 4. Figure 5.2b shows the progress made in this scheduling.[2] While *current_time* is still at 0 because no event with positive time has yet been processed, balls 2 and 3 have scheduled boundary crossings (case $\mathbf{Q} < \mathbf{P}$) and ball 4 has scheduled a collision at time 25 with ball 1 (case $\mathbf{Q} \geqslant \mathbf{P}$). When a scheduled collision is indicated on a picture, not only its time is given but also (in parentheses) the partner index. Thus, (4) 25 at the *new* position of ball 1 means that (the center of) ball 1 reaches this position at time 25 and when it does so, it collides with ball 4. (The dashed line which is supposed to indicate the future motion of ball 4 is overstricken by the arrow indicating the velocity.)

The algorithm schedules the collision of balls 1 and 4 at cycle 3 when balls 1 and 2 have already scheduled their next events, boundary crossings at times 58 and 124, respectively, but ball 3 has not been touched by the algorithm yet. This scheduling proceeds as follows. First (line 4), ball $i_* = 4$ finds out that the only P_{4j} which is not larger than *time*[j, *new*[j]] is $P_{41} = 25$ and $\mathbf{P}$ becomes 25. Then (line 5), it is determined that the nearest boundary crossing occurs at time $\mathbf{Q}$. The smallest of the two, $\mathbf{P}$ and $\mathbf{Q}$, becomes $\mathbf{R}$ and also *time*[4, *new*[4]] in line 6. Since $\mathbf{R}$ is finite and $\mathbf{Q}$ is larger than $\mathbf{P}$, the test in line 7 succeeds but the test in line 9 fails. As a result, the sequence of statements in lines 8 and 13–17 is executed whereby balls 4 and 1 have scheduled a collision at time 25 and the index m_* of the third party is remembered. Since there was no partner in the *new* event previously scheduled by ball 1, m_* becomes Λ and lines 19 and 20 are skipped.

Time 25 becomes the smallest one in the event-list and the next two cycles, 5 and 6, are spent on processing two events, *event*[1, *new*[1]] and *event*[4, *new*[4]], both representing the collision of balls 1 and 4 at time 25 but from the "viewpoints" of two different balls. Processing the collision event by ball 1 generates a new boundary crossing scheduled for time 94. Processing the collision by ball 4 then generates another collision scheduled for time 87 with ball 2. The latter collision preempts the previously scheduled boundary crossing by ball 2 for time 124. The result of all this processing is shown in Fig. 5.2c. Two velocity vectors are indicated

[2] The order 1, 2, 4, 3 of ball selection for the first four cycles is compatible with the algorithm. However, a persistent reader might ask why this order is not 1, 2, 3, 4. Initially, the author suspected an error in the program when noticing the transposition of 3 and 4. However, no error was found. The explanation of the transposition is presented in Appendix thereby giving more details of the *heap* algorithm.

for each colliding ball in Fig. 5.2c, before and after the collision. As seen, ball 1 has collided, not with ball 4, but with its periodic image.

The sequence of snapshots shown in Fig. 5.3 corresponds to the initial condition of the balls in Fig. 5.2, but without sectoring. During cycles 1 to 4, two collisions are scheduled: balls 1 and 4 for time 25 and balls 2 and 3 for a distant time 388. However, after cycle 5, the more distant collision of balls 2 and 3 is preempted by a collision of balls 1 and 2 for earlier time 226. As a result, ball 3 is left without a collision; its tentative collision is turned into a no-partner event which will be convenient to call *advancement*. At cycle 6, the preempting collision of balls 1 and 2 for time 226, is itself preempted by a collision of balls 2 and 4 scheduled for even earlier time 87. As a result, ball 1 has now scheduled an advancement event, the one previously listed as a collision schedued for time 226.

It seems that events develop faster in the experiments without sectoring shown in Fig. 5.3 than in those with sectoring in Fig. 5.2. Without sectoring, the balls schedule their new events with larger horizons and are more "aggressive." However, each cycle here takes more computing time. We have continued both experiments for 10^5 collisions, with each pairwise collision being counted twice. Without sectoring it takes more computing time than with sectoring. (The ratio is 3:1.)

This is so because to schedule a collision with sectoring, a ball should check nine neighboring sectors including its own, where it finds at most three other balls. Without sectoring a ball should check the same three balls and their 3×8 periodic boundary images. Functions *interaction_time* are formally different in the two cases. In the case without sectors, the time of a next collision with a ball A is in fact given not as (2.2) but as the minimum of nine times. One of these represents a collision with A and is given by (2.2), and the other eight represent collisions with eight periodic images of A.

6. Comments on the Implementation of the Algorithm

Overlaps. The billiard simulation should be tolerant with respect to a small overlap of the balls. Figure 5.3 shows "a preemption of a preemptor" phenomenon when ball 1 has preempted a collision of balls 2 and 3 by scheduling an earlier collision with ball 2 (Fig. 5.3b), only to be later preempted by ball 4 which schedules an even earlier collision with ball 2 (Fig. 5.3c). In simulations with thousands of balls, more involved phenomena of this kind occur. While combined with the roundoff, they occasionally cause slight overlappings as shown in the following example. Suppose a scheduled collision of balls A and B for time t_{AB} is later preempted by scheduling a collision of B and C for time $t_{BC} < t_{AB}$. As a result, the collision event for A becomes an advancement for time t_{AB}. Suppose that later in the computations, a collision of C and D scheduled for time $t_{CD} < t_{BC}$ preempts the collision of B and C. As a result, the collision event for B becomes an advancement for time t_{BC}. Now the originally scheduled collision of A and B for time t_{AB} needs to be scheduled again. However, it will be done starting with different initial positions. If formula (2.2) is used in this scheduling, then $c=0$ and $t=0$, because max(*time* 1, *time* 2) $= t_{AB}$. Because of roundoff errors and different computational paths, c may be slightly negative as if balls A and B were slightly overlapping at time t_{AB} causing t to be negative. The existing program handles this problem as follows: whenever *interaction_time* computes a negative but small by absolute value t in (2.2), the value of t is replaced by zero.

Advancement events. A preempted two-component interaction is turned into an advancement for the third party. For example, the preempted collision for time 388 of balls 2 and 3 in Fig. 5.3a is turned for ball 3 into an advancement in Fig. 5.3b. A more aggressive strategy would perform a full-fledged new event scheduling for ball 3. Such strategy is less efficient partly because advancements are usually planned far into the future and have a great chance of being rescheduled. It is not worthwhile to waste precomputations on them. Only a small fraction of scheduled advancements "survive" rescheduling. In most simulated cases less than 15% of all processed events are advancements. More importantly, the fraction of the processed advancements does not grow with N. (No theoretical analysis of this statement is available.)

Delayed update. There exists a subtle inefficiency in the algorithm in Fig. 4.1. When scheduling an interaction, the algorithm applies *advance* and *jump* operations. If the event is later preempted, these computations are wasted. For example, when scheduling a collision of balls 2 and 3 for time 388 (Fig. 5.3a), new velocities are computed, using *jump*. Later, however, this collision is preempted (Fig. 5.3b). To correct this inefficiency, the application of *advance* and *jump* should be delayed until the latest possible moment when the scheduled event is being processed. Figure 6.1 represents a variant of the algorithm which uses this idea.

The encoding of *partner* is different in the algorithm in Fig. 6.1 compared with that in Fig. 4.1. Assuming the interaction has not been processed yet, in the new version we have

$$\mathit{partner}[i, \mathit{new}[i]] = \begin{cases} A & \text{for an advancement} \\ \text{the index of the partner} & \text{for a two-component interaction} \\ N + \text{the index of the obstacle} & \text{for a one-component interaction.} \end{cases}$$

After the interaction has been processed by one participant i_* but not by the other $j_\#$, $\mathit{partner}[j_\#, \mathit{new}[j_\#]]$ becomes negative to indicate that no state update by the second participant $j_\#$ is required. This is so done because i_* has updated both states.

This code does not save a great deal in the billiard case because here *advance* and *jump* are much lighter computationally than *interaction_time*. The update pattern of array *time* $[1:N, 1:2]$ in the algorithm in Fig. 6.1 is the same as in the one in Fig. 4.1.

```
initially $current\_time \leftarrow 0$ and for $i = 1,2,...N$ :
$new[i] \leftarrow 1$, $old[i] \leftarrow 2$, $time[i,1] \leftarrow 0$, $partner[i,1] \leftarrow \Lambda$,
$state[i,1] \leftarrow$ initial state of component $i$, $event[i,2] \leftarrow event[i,1]$

1.  while $current\_time < end\_time$ do {
2.    $current\_time \leftarrow \min_{1 \le i \le N} time[i, new[i]]$ ;
      $i_* \leftarrow$ an index which supplies this minimum (i.e., $current\_time$) ;
3.    $state1 \leftarrow advance(state[i_*, old[i_*]], time[i_*, old[i_*]], time[i_*, new[i_*]])$ ;
4.    $j_\# \leftarrow partner[i_*, new[i_*]]$ ;
5.    if $j_\# = \Lambda$ then $state[i_*, new[i_*]] \leftarrow state1$
6.    else /* case $j_\# \ne \Lambda$ */
7.      if $j_\# > 0$ then /* state update required */
8.        if $j_\# > N$ then /* one-component interaction */
            $state[i_*, new[i_*]] \leftarrow jump(state1, j_\# - N)$
9.        else { /* $1 \le j_\# \le N$, two-component interaction */
10.         $state2 \leftarrow advance(state[j_\#, old[j_\#]], time[j_\#, old[j_\#]], time[j_\#, new[j_\#]])$ ;
11.         $(state[i_*, new[i_*]], state[j_\#, new[j_\#]]) \leftarrow jump(state1, state2)$ ;
12.         $partner[j_\#, new[j_\#]] \leftarrow -i_*$ ; /* negative partner flags no state update for $j_\#$ */
          } ; /* end two-component interaction close,
              end state update required close, end $j_\# \ne \Lambda$ close */
13.   $new[i_*] \leftarrow old[i_*]$ ; $old[i_*] \leftarrow 3 - new[i_*]$ ;
14.   $\mathbf{P} \leftarrow \min_{j \in A(i_*)} P_{i,j}$ , where $A(i_*) = \{j \mid 1 \le j \le N,\ j \ne i_*,\ time[j, new[j]] \ge P_{i,j}\}$ ;
      if $\mathbf{P} < +\infty$ then $j_* \leftarrow$ an index which supplies this minimum (i.e., $\mathbf{P}$) ;
15.   $\mathbf{Q} \leftarrow \min_{j \in B} Q_{i,k}$ , where $B = \{k \mid 1 \le k \le K\}$ ;
      if $\mathbf{Q} < +\infty$ then $k_* \leftarrow$ an index which supplies this minimum (i.e., $\mathbf{Q}$) ;
16.   $\mathbf{R} \leftarrow \min\{\mathbf{P}, \mathbf{Q}\}$ ; $time[i_*, new[i_*]] \leftarrow \mathbf{R}$ ;
17.   if $\mathbf{R} < +\infty$ then
18.     if $\mathbf{Q} < \mathbf{P}$ then $partner[i_*, new[i_*]] \leftarrow N + k_*$
19.     else { /* case $\mathbf{Q} \ge \mathbf{P}$ */
20.       $time[j_*, new[j_*]] \leftarrow \mathbf{R}$ ;
21.       $m_* \leftarrow partner[j_*, new[j_*]]$ ;
22.       $partner[i_*, new[i_*]] \leftarrow j_*$ ; $partner[j_*, new[j_*]] \leftarrow i_*$ ;
23.       if $m_* \ne \Lambda$ and $m_* \ne i_*$ then $partner[m_*, new[m_*]] \leftarrow \Lambda$ ;
        } /* end $\mathbf{Q} \ge \mathbf{P}$ close */
    } /* end while loop */
```

FIG. 6.1. A version of the simulation algorithm with delayed state update.

Can the third party be identical to the first party? In both versions of the algorithm, the third party update is conditioned to $m_* \ne i_*$ (lines 18, 19, and 20 in Fig. 4.1 and line 23 in Fig. 6.1), which requires the third party to be distinct from the first party, the initiator of the update. The existing program for billiard balls is supposed to report an occurrence of $m_* = i_*$. This condition has never been reported. Is identity $m_* = i_*$ at all possible?

We can imagine a scenario when equality $m_* = i_*$ is caused by two components interacting twice with the second interaction occurring after the first one without other components or obstacles intervening in between. In the billiard case with periodic boundary conditions, subsequent collisions of the same pair of balls is highly improbable for large N. In a different system, such occurrences may be probable even for large N. That is why the execution is safeguarded with the test $m_* \neq i_*$.

7. Consistency of Basic Operations

In an application, the three basic functions of Section 2 are derived from a consistent model: by integrating differential equations of motion of a system, using conservation laws, etc. However, the formulation of the algorithm in Section 4 employs no additional model. Obviously, arbitrarily "bad" basic functions can cause arbitrarily bizarre behavior even in a "good" algorithm. If we wish to analyze the algorithm correctness, we should request certain consistency properties in the basic functions to start. Thus, we introduce the following conditions:

(I) Function *interaction_time* is commutative with respect to the components; i.e., it depends on the unordered pair of components, although in (2.1) the two participants in the interaction are represented in a particular order.

(II) Similarly, function *jump* depends only on the unordered pair of arguments. This means that assignment (*new_state*2, *new_state*1) $\leftarrow$ *jump* (*state* 2, *state*1) produces the same *new_state*1, and *new_state*2 as assignment (2.5).

(III) Function *advance* (*state*0, *time*1, *time*2) satisfies a two-parametrical semigroup property with respect to its second and third argument, i.e., for any $t_1 \leqslant t_2 \leqslant t_3$ we have *advance* (*advance* $(s, t_1, t_2), t_2, t_3) =$ *advance* (s, t_1, t_3) for any state s.

(IV) Moreover, there is a proper associativity between *advance* and *interaction_time*. Namely, if $t =$ *interaction_time* $(s_1, t_1, .)$, and $t_1 \leqslant t_2 \leqslant t$, then $t =$ *interaction_time* (*advance* $(s_1, t_1, t_2), t_2, .)$. Here dot $(\cdot)$ replaces either an appropriate pair (*state*, *time*), if we have a two-component interaction, or an *obstacle*, if we have a one-component interaction. For a two-component *interaction_time*, this property, coupled with (I), implies a similar associativity with respect to the second set of arguments or with respect to both sets.

(V) Components are never stuck with each other. Namely, if two components 1 and 2 with *state*1 and *state*2 are interacting, i.e., (2.3) holds, *jump* is applied and *new_state*1 and *new_state*2 are computed according to (2.5), then *interaction_time* (*new_state*1, *time*, *new_state*2, *time*) $>$ *time*. Similarly, if a component with *state* is interacting with *obstacle*, i.e. (2.8) holds, *jump* is applied and *new_state* is computed according to (2.7), then *interaction_time* (*new_state*, *time*, *obstacle*) $>$ *time*.

Computationally conditions (I)–(IV) might be "slightly" violated because of the roundoff. This can cause the simulated history to be dependent upon the processing order. In the billiard simulation, if processing is organized in two different ways, usually after a few dozen collisions by each ball, an accumulation of small quantitative roundoff errors causes qualitative divergence of the history, which determines, for example, which ball collides with which. Computational physicists are aware of such divergence [3] and consider it a variant of physical irreproducibility. It is worth stating, however, that the second run of exactly the same serial program starting with the same input data produces exactly the same results.

Now we are going to introduce a condition of a different kind. Consider the set of components and obstacles $I(t)$ interacting at a particular time t. If $I(t)$ is nonempty, we may introduce a binary relation Δ among the elements in $I(t)$, assuming $i \Delta j$ if i is interacting with j at t. Let $\tilde{}$ be a reflexive, symmetric, and transitive closure of Δ, so that $\tilde{}$ is an equivalence. With this definition, the condition is:

(VI) No equivalence class for the relation $\tilde{}$ contains more than two elements.

For example, in the billiard case (VI) prohibits participation of more than two balls in the same collision, but several disjointed pairwise collisions may take place at the same time. Figure 7.1 shows such a prohibited triple collision where (2.4) holds for the pair $(i=1, j=2)$ and, separately, for the pair $(i=2, j=3)$, but not for

and their velocities v_1, v_2, and v_3, is mirror symmetrical with respect to the middle vertical line M. There are two possible orders for processing this collision by the

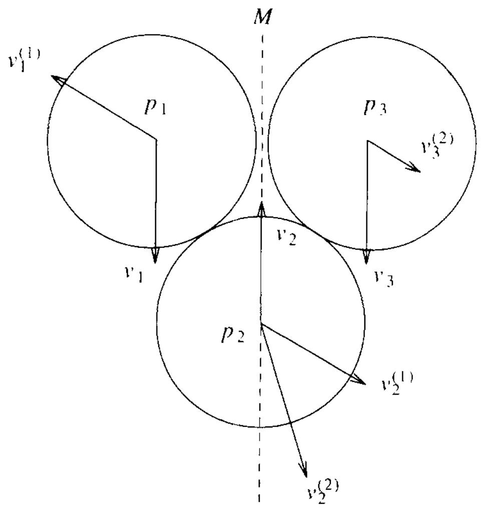


FIG. 7.1. A triple collision.

algorithm. In one order, balls 1 and 2 first collide and obtain new velocities $v_1^{(1)}$ and $v_2^{(1)}$. Then balls 2 and 3 collide and obtain new velocities $v_2^{(2)}$ and $v_3^{(2)}$. The initial velocity of ball 2 for the second pairwise collision is $v_2^{(1)}$ as if the second collision occurred later than the first one. The net result of the triple collision is the three balls moving away from the collision site with velocities $v_1^{(1)}$, $v_2^{(2)}$, and $v_3^{(2)}$, which are not mirror symetrical with respect to M. Hence the coutcome of the triple collision depends on the order of processing as does the history of the entire simulation.

With infinite precision computations, in the case of chaotically colliding billiard balls, the probability of violating (VI) is zero. However, in our finite precision experiments multiple collisions could practically occur and hence (VI) could be violated. The proof in Section 8 of the correctness of the simulated trajectory should be understood as an assurance that if the machine precision is infinite, the correctness holds for as long as (VI) holds.

In order to show that the algorithm in Fig. 4.1 reconstructs the trajectory of each component "correctly" we must know what a "correct" trajectory is. With assumptions (I)–(VI), starting with a global state at time 0, we can uniquely define the system state at any positive time using the naive algorithm discussed in the Introduction. We *call* the obtained trajectory the correct one.

The algorithm in Fig. 4.1 ignores many events on correct trajectory; our task is to prove that despite this fact, the trajectory does not change.

8. Invariants and Correctness Proof

The actions of both simulation algorithms in Fig. 4.1 and in Fig. 6.1 can be summarized as follows: a repeated update of arrays *new*[1:N], *old*[1:N], and *event*[1:N, 1:2] in such a way that the conditions

$$\max_{1\leqslant i\leqslant N} \mathit{time}[i, \mathit{old}[i]] \leqslant \min_{1\leqslant i\leqslant N} \mathit{time}[i, \mathit{new}[i]] \tag{8.1}$$

remain invariant. For $i = 1, 2, ..., N$ we have

$$\mathit{time}[i, \mathit{new}[i]] \leqslant \min\{\min_{1\leqslant j\leqslant N, j\neq i, \mathit{time}[j, \mathit{new}[j]]\geqslant P_{ij}} P_{ij}, \min_{1\leqslant k\leqslant K} Q_{ik}\} \tag{8.2}$$

and

either *partner*[i, *new*[i]] $= \Lambda$,
or $j =$ *partner*[i, *new*[i]] is an integer in the interval $N+1 < j \leqslant N+K$, (8.3)
or j is an integer in the interval $1 \leqslant j \leqslant N$ and *partner*[j, *new*[j]] $= i$.

Conditions (8.1), (8.2), and (8.3) are trivially satisfied in the beginning of the simulation. Invariance of condition (8.1) is obvious. As to (8.2) and (8.3), their invariance can be violated temporarily after a cycle during which one component participating in a two-component interaction has been processed but the other has not been yet. After both components have been processed, and no other two-component interaction processing has been started, (8.2) and (8.3) hold. For (8.2), it follows from lines 4, 5, and 6 in Fig. 4.1 and for (8.3), it follows from symmetricity of matrix P_{ij}. This symmetry is an obvious implication of (I) and (II). Observe, that the invariance of (8.1) and (8.2) requires no consistency conditions (I)–(VI).

Invariant (8.2) is the key to understanding the "wasteful" strategy of the data update in this algorithm. Consider an example. Let $N=3$, $K=0$. Figure 8.1 shows trajectories of three billiard balls A, B, and C. We assume that at time $t=0$ the balls are positioned on the same horizontal line and we suppose that these are their *old* positions, i.e., those stored in array *event*[·, *old*[·]].

On the basis of the *old* events only, C can see two immediate collisions, one with B when the balls occupy positions $B2$ and $C2$ (call it collision $B2, C2$), and the other with A, namely collision $A2, C1$. C also notes that both A and B have a scheduled event at time earlier than times of either $A2$, $C1$ and $B2$, $C2$. Thus, the set of balls X over which the minimum of P_{CX} is to be taken according to (8.2) is empty, and this minimum together with the time of the immediate next interaction for C is $+\infty$.

With the given *old* events, the following assignment of *new* times would satisfy (8.2): both *time*[A, *new*[A]] and *time*[B, *new*[B]] are equal to the time of collision $A1, B1$, end *time*[C, *new*[C]] $= +\infty$. With such an assignment, three inequalities (8.2) turn into equalities.

The assignment *time*[i, [*new*[i]] $= +\infty$ simply means that C sees to future

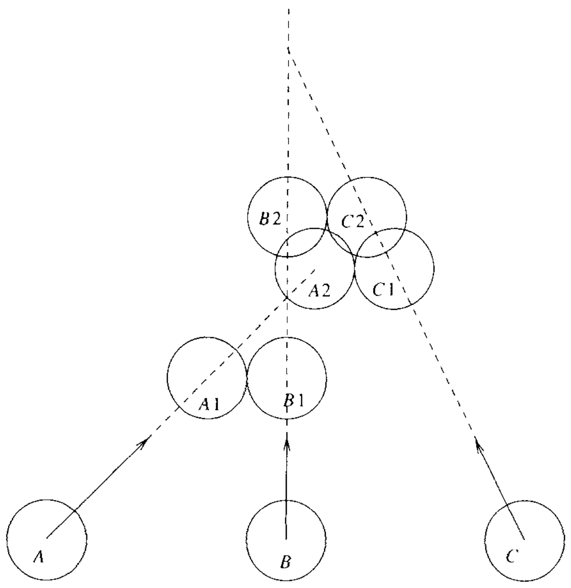


FIG. 8.1. Asynchronous collisions of three billiard balls.

interaction at this stage of simulation. A more aggressive strategy of precomputation, in which C would look one more step ahead and would examine possible collisions with A and B based on their velocities after an as yet unprocessed collision $A1, B1$, is possible. However, the proposed algorithm does not use such strategy. The aggressive strategy might work well for a small number of balls. For many balls, the aggressive strategy would require a complicated data structure to support an arbitrary many-step lookahead. In our algorithm, C does not look for more than one step ahead, thus allowing us to keep the data structure simple.

Invariants (8.1) and (8.2) imply the following useful invariant

$$\min\{\min_{1 \leqslant i,j \leqslant N} P_{ij}, \min_{1 \leqslant i \leqslant N, 1 \leqslant k \leqslant K} Q_{ik}\} \geqslant \min_{1 \leqslant i \leqslant N} \mathit{time}[i, \mathit{new}[i]]. \tag{8.4}$$

To prove the correctness of the algorithm we will show that if

(*) the simulated trajectory is identical to the "correct" one defined in Section 7, for all t in the interval $0 \leqslant t \leqslant \max_{1 \leqslant i \leqslant N} \mathit{time}[i, \mathit{old}[i]]$, then

(**) after all events with times equal to $\min_{1 \leqslant i \leqslant N} \mathit{time}[i, \mathit{new}[i]]$ will be processed, the extended simulated trajectory will be identical to the "correct" trajectory for all t in the interval $0 \leqslant t \leqslant \min_{1 \leqslant i \leqslant N} \mathit{time}[i, \mathit{new}[i]]$.

This would constitute the inductive step. The basis for the induction is obviously satisfied since (*) is correct for the program state initialized for $t = 0$ as described in Fig. 4.1.

The "correct" trajectory has no interaction on the open interval $\max_{1 \leqslant i \leqslant N} \mathit{time}[i, \mathit{old}[i]] < t < \min_{1 \leqslant i \leqslant N} \mathit{time}[i, \mathit{new}[i]]$, because if it did, (8.4) would be violated. Hence the simulated trajectory is identical to the "correct" one for all t in the interval $0 \leqslant t < \min_{1 \leqslant i \leqslant N} \mathit{time}[i, \mathit{new}[i]]$. By (I)–(VI), this property extends to the point $t = \min_{1 \leqslant i \leqslant N} \mathit{time}[i, \mathit{new}[i]]$ and this completes the proof.

9. An Application Example: A Disk Packing Problem

The following model is simulated in [7]: N points are placed randomly within an $L \times L$ square. Periodic boundary condition apply in both directions. The N points are assigned random initial velocities and in the absence of subsequent collisions would move with these velocities along straight lines threading through an infinite sequence of periodic images of the basic square. However, the points also begin to grow at a common rate into elastic rigid disks, with diameters that are given by linear function of time $D(t) = at$, $t > 0$. As a result, particule collisions become possible, and increase in frequency as $D(t)$ increases. We permit $D(t)$ to grow until the system "jams up," thus obtaining the final packing.

This is a variant of the billiard simulations. Two differences are:

(1) instead of equation $|p + vt|^2 = D^2$ as in Fig. 2.1, equation $|p + vt|^2 = (at)^2$ has to be solved; the latter is still a quadratic equation with respect to t;

(2) the normal components of v_1^{new} and v_2^{new} (velocities of balls after a collision, see Fig. 2.2), have to be increased to guarantee that balls do not overlap or stick to each other. Any additive velocity larger than $a/2$ would be appropriate.

Energy or momentum conservation are lost with such an additive; as the simulation progresses the system "heats up," and computational precision may be lost as ball speeds increase. The existing program once in a while interrupts the simulation projecting all the ball positions into a particular time value, then scales down and balances the velocities. (The velocities v_i, $i = 1, ..., N$, of N balls of equal masses are balanced if $\sum_{1 \leqslant i \leqslant N} v_i = 0$.)

Figures 9.1 and 9.2 show some results of these experiments, in particular the so-called "rattler" balls which remains unjammed within the walls of jammed neighbors [7]. In the experiment presented in Fig. 9.2, the large square is sub-

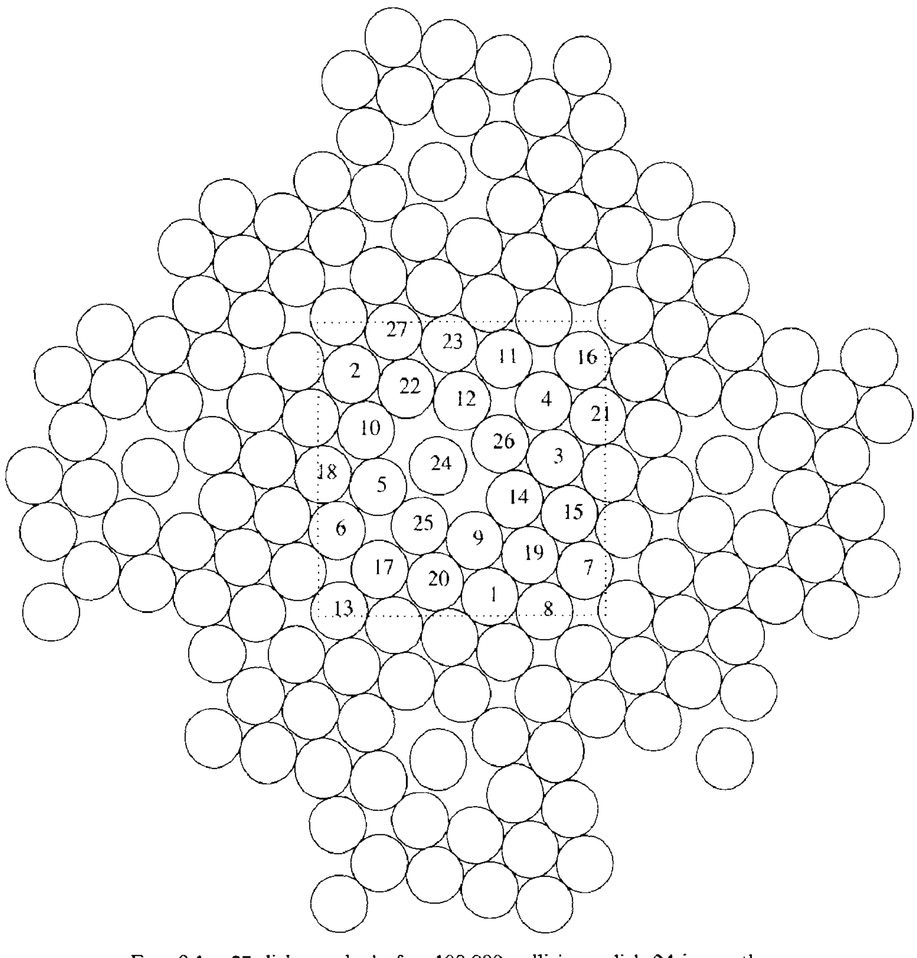
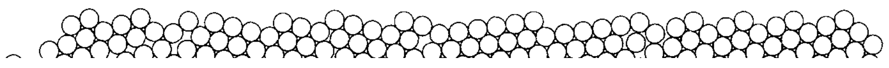


FIG. 9.1. 27 disks packed after 100,000 collisions; disk 24 is a rather.



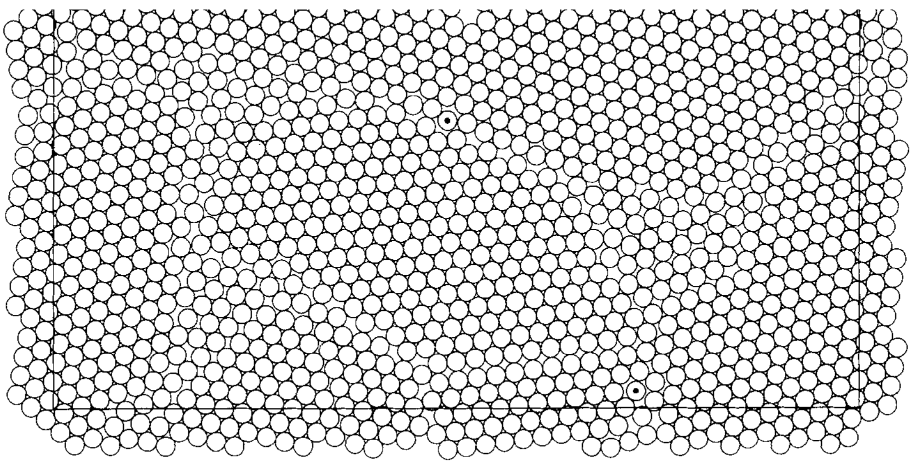


FIG. 9.2. 2000 disks packed after 42×10^6 collisions; dots mark significant rattlers.

divided into 40×40 small square sectors (not shown in Fig. 9.2). Rather than checking a possible next collision with 8×1999 candidates, only about 10 disk candidates for the next collision are checked.

10. Comparing the Performance of This Algorithm with That of Other Algorithms

Physicists often study hard-sphere and hard-disk models using computing experiments. However, with the exception of [3, 1], nobody discusses the details of the algorithms used, and with the exception of [1], nobody gives performance data. We read in [1]: "The IBM704 calculator handles about 2000 collisions per hour for 100 molecules and about 500 collisions per hour for 500 molecules."

Assuming that IBM704 was not slower than 0.02 MFLOP [2], this scales to no more than 30 collisions per second for a 1 MFLOP machine. The speed of our calculations is in the range 150–450 pairwise collisions per second (independently of the number of balls) on VAX8550 which has speed 1 MFLOP. Thus, even the most pessimistic comparison with [1] gives about an order of magnitude speed-up of our algorithm.

Simulation of 50,000 to 55,000 committed events in a random configuration of 160 disks is reported in [4]. Let us count one pairwise collision as two committed events, and one sector boundary crossing or cushion reflection as one committed event.

The model [4] is different from the one we simulated in that instead of periodic boundary conditions, rigid elastic "cushions" are employed to guard the cell boundaries. To compensate for the difference, let us equare an external boundary crossing in our program with one cushion reflection in [4]. Note that when scheduling a collision close to the cell boundary, our program considers not only internal disks as the candidates for collision, as program [4] does, but also their periodic images. This additional complexity in our program more than compensates for a possible loss of complexity due to substituting a cushion reflection with a boundary crossing.

In our measuring run, sector boundary crossings were counted only for 16 sectors specified in series I in [4]. The run was continued until the number reached 52,000 as in [4]. It took 90 s CPU to complete this run.

A similar run in [4] (Series I), took 440 s on one PE and 62 s on 32 PEs, nodes of a hypercube MARK III. (For 32 and 64 sectors, it took, respectively, 44 and 42 s in [4].) One node of MARK III is about 60% faster than our VAX 8550. Besides, our algorithm is a Fortran code while program [4] is written in C-language, both compiled under UNIX. This yields an additional 10% in favor of our algorithm, since Fortran is slower than C under a UNIX compilation. Thus our serial algorithm runs about as fast as the parallel Time Warp [4] on a 32-node hypercube.[3]

11. Conclusion: Other Billiard-like Simulations and an Unsolved Problem

A collision of two billiard balls of radii $D/2$ can be considered as an interaction of two zero-size particles with potential $V(r) = 0$ for distances $r > D$ and

[3] After the measuring run was completed, A. P. Wieland informed the author that one sector boundary crossing is actually counted as two events in [4] rather than as one, as is assumed in this paper. Also, one pairwise disk collision is counted not as two events as assumed, but as $4 + m$ events, where variable m is the number of disks located in the involved sectors at the time of the collision. Suppose only one sector is involved in each collision, and there are totally 16 sectors (as in Series I in [4]). Then m is about 10. This makes the total count of events generated in [4] during a comparable simulation time interval several times higher than was assumed in the experiments. This, in turn, makes our program faster than program [4].

$V(r) = +\infty$ for $r \leqslant D$. More general piece-wise constant potentials can be dealt with using the same algorithm, e.g., the square-well potential [1],

$$\begin{aligned} V &= +\infty, && \text{if} \quad r < \sigma_1 \\ V &= V_0, && \text{if} \quad \sigma_1 < r < \sigma_2 \\ V &= 0, && \text{if} \quad \sigma_2 < r, \end{aligned}$$

where V_0, σ_1, and σ_2 are finite constants. We can imagine two concentric balls: the "hard core" ball of diameter σ_1 and a larger "soft shell" ball of diameter σ_2. We could then have two types of "collisions": internal, of the hard-cores, and external, of the soft-shells. Each type has its own *jump* function.

By the mean of a Monte-Carlo simulation, [9] shows that larger particles move against the gravitational force if they are placed together with smaller particles in a vibrating container. The balls of different diameters can be easily handled in our scheme, if the ball diameter becomes a part of its state. Model [9] can be easily simulated using direct represetations of particle dynamics, instead of Monte Carlo.

Components lacking homogeneity can be treated in the same way, i.e., by making the type or the class identification of a component an unchangeable part of its state. Perhaps, certain granular flow models can be treated in this way. Combat simulations [11] present such inhomogeneity to a large extent, since here components represent military units of opposing armies, and types of units vary.

Collisions may be generalized to any state changes, including changes that do not immediately lead to a trajectory change. A typical simulation rule in [11] is: "if within radius σ, a unit detects m units of the same army and n units of the opposing army, then it takes course of action $c(n, m)$, from the time of detecting this situation until the time when another rule becomes applicable," We can represent these rules by surrounding a zero-sized unit by several circles, each representing a rule. A counter "inside" the unit state gets an instantaneous increment, when a particular circle "collides" with another unit. The counter change may or may not trigger a change in the course of the action. Such mechanisms can be represented within the discussed framework and simulated using the algorithm in Fig. 4.1.

According to [10], a variant of the dense packing algorithm can be used in finding optimal spherical codes. Here the task is to find N points p_i, $i = 1, ..., N$, on the sphere in the k-dimensional Euclidean space in such a way that

$$\min_{i \neq j} \text{distance}(p_i, p_j) \rightarrow \max.$$

We would start with a random configuration of N "seed" points and then grow "caps" of equal size, each cap having a seed in the center. Caps are prevented from the overlap by collisions.

Although the algorithm is practically efficient, no theoretical model which explains this is available. The model should explain, for example, why the number of overhead advancement events remains bounded from the above independent of N.

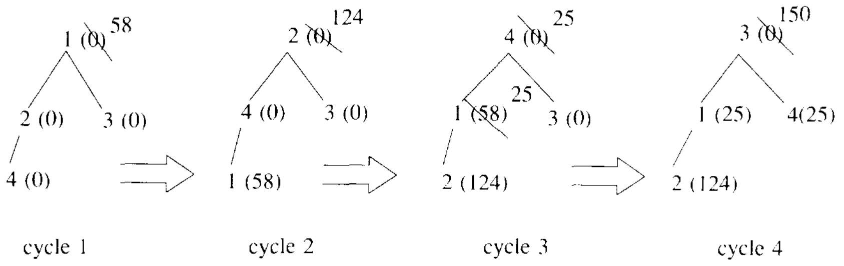


FIG. A.1. Evolution of the heap tree for the example in Fig. 5.2.

APPENDIX

The program forms the original heap tree in a natural order: ball 1 at the top, then balls 2 and 3 at the second level, and then ball 4 attached to ball 2 at the third level (cycle 1 in Fig. A.1). The value of the key *time* $[i, new[i]] = 0$ for all $i = 1, 2, 3, 4$. In Fig. 1, a key is indicated in parentheses after the ball number. At each cycle the ball at the tree root gets processed. Thus, at cycle 1 ball 1 is processed. As a result, key *time*$[1, new[1]]$ becomes 58. This "heavy" key together with the ball should move down to keep the heap discipline: any parent must be

to the left or to the right branch. The program examines the branches from left to right. Hence ball 1 moves down to the place of ball 4. "Light" balls 2 and 4 move up to the vacant places. Cycle 2 begins with ball 2 at the root. During cycle 2 ball a is processed, and key *time*$[2, new[2]]$ becomes 124. Now ball 2 must move down, while balls 1 and 4 move up to the vacant places. At the beginning of cycle 3, ball 4 (not ball 3!) happens to reach the root and therefore gets processed. During this processing, both *time*$[4, new[4]]$ and *time*$[1, new[1]]$ become 25. Finally, during cycle 4 ball 3 gets processed. Thus, the balls are processed in the order 1, 2, 4, 3, not in the order 1, 2, 3, 4.

Figures A.2 and A.3 show the fragments of the FORTRAN code dealing with heap-sort mechanism for $N = 2000$ balls. The heap structure is initialized at the beginning of the execution (Fig. A.2). Each time value *time*$[k, new\ [k]]$ is changed, subroutine *pull*$(k, 2000)$ (Fig. A.3) is invoked to adjust the heap.

```
c initialize heap
      do 10 k=1,2000
      pah(k)=k
      pha(k)=k
10  continue
```

FIG. A.2. Heap initialization fragment.

```
      subroutine pull(k,iend)
c pulling up or down in the heap-sort algorithm
c minimum at the root, heavy items go down
      common /theap/ time(2000,2),new(2000),pah(2000),pha(2000)
      integer pah,pha
      integer new
      double precision time
      double precision aa, aaj, aaj1

        if((k.ge.1).or.(k.le.iend))goto 2
          print *,'in pull: request for item at k=',k
          print *,'endlist=',iend
         stop
 2      continue
c PULL-UP INITIALIZE
        n1=pha(k)
        aa=time(n1,new(n1))
        j1=k
c SET SON i1 AND FATHER j1
 5      i1=j1
        j1=j1/2
        if(j1.lt.1) goto 8
        aaj=time(pha(j1),new(pha(j1)))
c IF FATHER SMALLER THAN SON THEN PULL-UP IS COMPLETE
        if(aaj.le.aa) goto 8
c PULL THE FATHER j1 DOWN TO THE VACANT SON PLACE i
        m1=pha(j1)
        pah(m1)=i1
        pha(i1)=m1
        goto 5
c PLACE THE GRANDSON UP
 8      pha(i1)=n1
        pah(n1)=i1
***********************************
cPULL-DOWN INITIALIZE
 9      j=k
        n=pha(k)
        aa=time(n,new(n))
c SET FATHER i AND SON j
 10     i=j
        j=2*j
c IF NO SONS THEN EXIT
        if(j.gt.iend)goto 30
c LOCATE THE LEFT SON j
        aaj=time(pha(j),new(pha(j)))
c IF ONLY ONE SON THEN BYPASS COMPARISON
        if(j.eq.iend)goto 20
c COMPARISON:  MAKE j TO BE THE SMALLEST SON
        aaj1=time(pha(j+1),new(pha(j+1)))
        if (aaj.le.aaj1) goto 20
         j=j+1
         aaj=aaj1
c IF GRANDFATHER SMALLER THAN THESE SONS THEN EXIT

 20     if(aa.le.aaj) goto 30
c PULL THE SMALL SON j UP TO THE VACANT FATHER PLACE i
        m=pha(j)
        pah(m)=i
        pha(i)=m
        goto 10
c PLACE THE OLD GRANDFATHER DOWN
 30     pha(i)=n
        pah(n)=i
        return
        end
```

FIG. A.3. Heap update subroutine.

ACKNOWLEDGMENT

Frank H. Stillinger has greatly stimulated this work by proposing the disk packing model described in Section 9. His contribution is presented in companion paper [7].

REFERENCES

1. B. J. ALDER AND T. E. WAINWRIGHT, *J. Chem. Phys.* **31**, No. 2, 459 (1959).
2. C. J. BASHE *et al.*, *IBM's Early Computers* (MIT Press, Cambridge, MA, 1986).
3. J. J. ERPENBECK AND W. W. WOOD, in *Statistical Mechanics. Part B: Time-Dependent Processes*, edited by J. B. Berne (Plenum, New York, 1977), p. 1.
4. P. HONTALES, B. BECKMAN, *et al.*, in *Proceedings, 1989 SCS Multiconference, Simulation Series* (Society for Comput. Simulation, San Diego, CA, 1989), Vol. 21, No. 2, p. 3.
5. J. KATZENELSON, *SIAM J. Sci. Statist. Comput.* **10**, No. 4, 787 (1989).
6. D. E. KNUTH, *Art of Computer Programming, Vol. 3 Sorting and Searching* (Addition, New York, 1973).
7. B. D. LUBACHEVSKY AND F. H. STILLINGER, *J. Statist. Phys.* **60**, No. 5/6, 561 (1990).
8. B. D. LUBACHEVSKY, in *Proceedings, 1990 SCS Multiconference, Simulation Series* (Society for Comput. Simulation, San Diego, CA, Vol. 22, No. 2, p. 194.
9. A. ROSATO *et al.*, *Phys. Rev. Lett.* **58**, No. 10, 1038 (1987).
10. W. SMITH, private communication.
11. F. WIELAND AND D. JEFFERSON, in *Proceedings, 1989 Int. Conf. Parallel Processing*, Vol. III, edited by F. Ris, and M. Kogge (Pennsylvania State University Press, University Park/London, 1989), p. 255.