Exercises

8.14 **Hysteresis algorithms.**[1 2] (Complexity, Computation) ④

As computers increase in speed and memory, the benefits of writing efficient code become greater and greater. Consider a problem on a system of size N; a complex algorithm will typically run more slowly than a simple one for small N, but if its time used scales proportional to N and the simple algorithm scales as N^2, the added complexity wins as we can tackle larger, more ambitious questions.

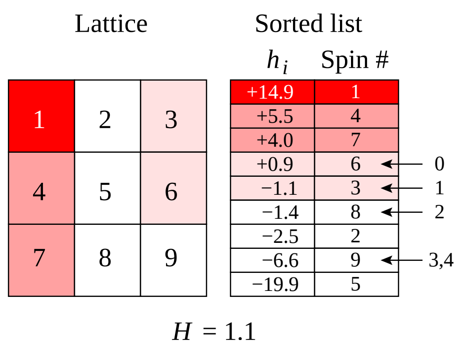

Fig. 8.19 Using a sorted list to find the next spin in an avalanche. The shaded cells have already flipped. In the sorted list, the arrows on the right indicate the `nextPossible[nUp]` pointers—the first spin that would not flip with `nUp` neighbors at the current external field. Some pointers point to spins that have already flipped, meaning that these spins already have more neighbors up than the corresponding `nUp`. (In a larger system the unflipped spins will not all be contiguous in the list.)

In the hysteresis model (Exercise 8.13), the brute-force algorithm for finding the next avalanche for a system with N spins takes a time of order N per avalanche. Since there are roughly N avalanches (a large fraction of all avalanches are of size one, especially in three dimensions) the time for the brute-force algorithm scales as N^2. Can we find a method which does not look through the whole lattice every time an avalanche needs to start?

We can do so using the *sorted list* algorithm: we make[3] a list of the spins in order of their random fields (Fig. 8.19). Given a field range $(H, H+\Delta)$ in a lattice with z neighbors per site, only those spins with random fields in the range $JS + H < -h_i < JS + (H+\delta)$ need to be checked, for the $z+1$ possible fields $JS = (-Jz, -J(z-2), \dots, Jz)$ from the neighbors. We can keep track of the locations in the sorted list of the $z+1$ possible next spins to flip. The spins can be sorted in time $N \log N$, which is practically indistinguishable from linear in N, and a big improvement over the brute-force algorithm.

Sorted list algorithm.

(1) Define an array `nextPossible[nUp]`, which points to the location in the sorted list of the next spin that would flip if it had `nUp` neighbors. Initially, all the elements of `nextPossible[nUp]` point to the spin with the largest random field h_i.

(2) From the $z+1$ spins pointed to by `nextPossible`, choose the one `nUpNext` with the largest internal field in `nUp - nDown` $+ h_i =$ `2 nUp -` $z + h_i$. Do not check values of `nUp` for which the pointer has fallen off the end of the list; use a variable `stopNUP`.

(3) Move the pointer `nextPossible[nUpNext]` to the next spin on the sorted list. If you have fallen off the end of the list, decrement `stopNUP`.[4]

[1]From *Statistical Mechanics: Entropy, Order Parameters, and Complexity* by James P. Sethna, copyright Oxford University Press, 2007, page 185. A pdf of the text is available at pages.physics.cornell.edu/sethna/StatMech/ (select the picture of the text). Hyperlinks from this exercise into the text will work if the latter PDF is downloaded into the same directory/folder as this PDF.

[2]This exercise is also largely drawn from [69], and was developed with the associated software in collaboration with Christopher Myers.

[3]Make sure you use a packaged routine to sort the list; it is the slowest part of the code. It is straightforward to write your own routine to sort lists of numbers, but not to do it efficiently for large lists.

[4]Either this spin is flipped (move to the next), or it will start the next avalanche (flip

(4) If the spin `nUpNext` has exactly the right number of up-neighbors, flip it, increment the external field $H(t)$, and start the next avalanche. Otherwise go back to step (2).

Implement the sorted list algorithm for finding the next avalanche. Notice the pause at the beginning of the simulation; most of the computer time ought to be spent sorting the list. Compare the timing with your brute-force algorithm for a moderate system size, where the brute-force algorithm is slightly painful to run. Run some fairly large systems[5] *(*2000^2 *at* $R = (0.7, 0.8, 0.9)$ *or* 200^3 *at* $R = (2.0, 2.16, 3.0)$*), and explore the avalanche shapes and size distribution.*

To do really large simulations of billions of spins without needing gigabytes of memory, there is yet another algorithm we call *bits*, which stores the spins as bits and never generates or stores the random fields (see [69] for implementation details).

and move to the next), or it has too few spins to flip (move to the next, flip it when it has more neighbors up).

[5]Warning: You are likely to run out of RAM before you run out of patience. If you hear your disk start swapping (lots of clicking noise), run a smaller system size.