- Otherwise, if `AcceptTransition` == 0: keep the current x and E
- Repeat (go back to ▶) as needed.

§12. First Application: Sudoku Solver

In this application we will use the Metropolis algorithm to solve Sudoku puzzles. As we'll see in the next section, there are more direct ways to solve these puzzles, but applying Markov chain methods here is fun.

Sukoku puzzles work as follows. On a 9×9 grid, several sites are populated with the digits $\{1, 2, \ldots, 9\}$. Your task is to populate the remainder of the grid (the "solved-for" grid sites) with these nine digits in such a way that: each column has no digit repetitions; each row has no repetitions; and each of the highlighted 3×3 blocks has no repetitions. A puzzle and its unique solution are shown below.

Puzzle

	6							3
		2	9					
4		7		5	1			9
					6		9	5
		9		2		7		
8	3		5					
6			1	8		9		4
					4	1		
2							3	

Solution

9	6	5	2	4	7	8	1	3
3	1	2	9	6	8	5	4	7
4	8	7	3	5	1	2	6	9
7	2	4	8	1	6	3	9	5
1	5	9	4	2	3	7	8	6
8	3	6	5	7	9	4	2	1
6	7	3	1	8	2	9	5	4
5	9	8	6	3	4	1	7	2
2	4	1	7	9	5	6	3	8

Metropolis solves the puzzle by simulating a Markov chain via Monte Carlo. The state space S consists of all possible ways to assign the digits $\{1, 2, 3, \ldots, 9\}$ to the grid sites that must be solved for. (The clue sites remain fixed.) This puzzle has 27 clues and $81 - 27 = 54$ solved-for sites. Since each solved-for site can assume 9 possible values, there are $9^{54} \approx 3.38 \times 10^{51}$ states in S. That's a huge number. Imagine the PTM! Most states are very far from the solution, like the one below at the left. States like this will be assigned a high energy. A relatively small number of states are close to the solution, like the one below at the right where only the three starred sites differ from the solution. States like this will be assigned a low energy.

High Energy State

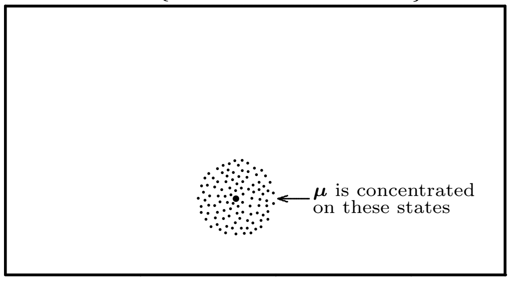
1	6	1	1	1	1	1	1	3
1	1	2	9	1	1	1	1	1
4	1	7	1	5	1	1	1	9
1	1	1	1	1	6	1	9	5
1	1	9	1	2	1	7	1	1
8	3	1	5	1	1	1	1	1
6	1	1	1	8	1	9	1	4
1	1	1	1	1	4	1	1	1
2	1	1	1	1	1	1	3	1

Low Energy State

6^*	6	5	2	4	7	8	1	3
3	1	2	9	6	3^*	5	4	7
4	8	7	3	5	1	2	6	9
7	2	4	8	1	6	3	9	5
1	5	9	7^*	2	3	7	8	6
8	3	6	5	7	9	4	2	1
6	7	3	1	8	2	9	5	4
5	9	8	6	3	4	1	7	2
2	4	1	7	9	5	6	3	8

The Markov chain is constructed so that its invariant distribution $\boldsymbol{\mu}$ is concentrated on states close to the solution — those with low energy. These states are represented below by the small dots; the big dot represents the solution. As the Markov chain is simulated, it converges to the invariant distribution. At this point, the Markov chain is sampling from a relatively limited number of states close to the solution. It eventually lands on the solution, at which point simulation stops and the result is reported.

$S \;=\; \{\text{all } 3.38 \times 10^{51} \text{ states}\}$



We have already specified the state space:

$$S \;=\; \{\text{all possible assignments of the digits } 1-9 \text{ to the solved-for sites}\}.$$

Regarding the energy function $E(\cdot)$, we will call a pair of two (distinct) grid sites **in conflict** if they are in the same row, column, or 3×3 block and they are allocated the same digit in the configuration. For any configuration x, $E(x)$ reflects the number of pairs of grid sites that are in conflict: any pair that involves only non-clue sites contributes 1 to $E(x)$; any pair where one of the sites is a clue site contributes 5 to $E(x)$. The penalty of 5 is arbitrary, but it reflects the fact that a conflict with a clue *must* be wrong (I learned this trick from Anik Roy, a former student).

For example, if x is the close-to configuration above, we have $E(x) = 20$: the 6 has two conflicts with clues (that's 10); the 3 has three conflicts with non-clues (that's 3 more); and the 7 has one conflict with a clue and two conflicts with non-clues (that's 7 more). (The far-from configuration above has an energy of 556.) There is only *one ground state* (the solution) and the energy of the ground state is 0.

Finally, call two configurations neighbors if they agree everywhere except for one grid site, where they disagree. (The site where they disagree must be a non-clue site, as all configurations agree at the clue sites.)

First we demonstrate that this notion of neighbor satisfies the good neighbor rules of Chapter 3:

(i) $x \not\leftrightarrow x$;
(ii) each state has the same number of neighbors, call it N;
(iii) $x \leftrightarrow y \implies y \leftrightarrow x$; and
(iv) for any two states x and y, there is a neighbor-to-neighbor walk from x to y.

The first is clear: $x \leftrightarrow y$ means they differ at one site, so $x \neq y$. As for (ii), in the above puzzle, for example, each configuration has $N = 54 \times 8 = 432$ neighbors. (Choose one of the 54 solved-for sites and change its digit to one of the 8 other digits.) As for (iii), if y differs from x at one site then x differs from y at one site. Regarding (iv), one can go from any configuration to any other configuration by changing one site at a time.

The algorithm starts with a randomly selected configuration. This is arbitrary. By Fact 2 of Chapter 1, the Markov chain will converge to its invariant distribution regardless of how it's started. Suppose after n steps of the Markov chain the configuration is x whose energy is E. One of x's N neighbors, call it

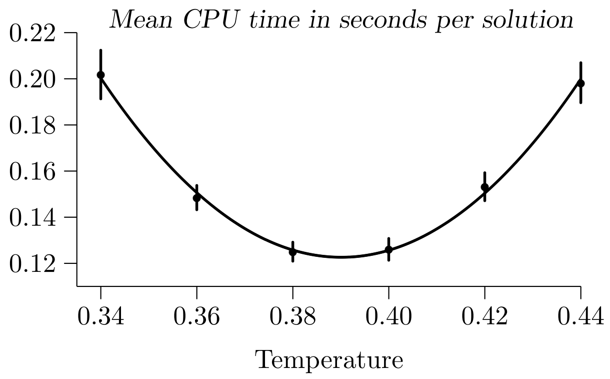
y, is selected randomly as discussed above. If $\Delta E(x,y) \leq 0$ we accept the transition. Otherwise, if $T > 0$, we calculate $p = \exp\left(-\frac{\Delta E(x,y)}{T}\right) < 1$ and generate a **fresh uniform** $U \sim \text{Uniform}\,(0,1)$, i.e. one that is independent of previous uniforms. (For all eight applications here we use the Mersenne Twister as our random number generator — see [MN].) If $U \leq p$ (which happens with probability p) we accept the transition. If we accept the transition, take new $x = y$ and new $E = E + \Delta E(x,y)$; otherwise we keep x and E, so new x = current x, and new E = current E. When we eventually have $E(x) = 0$ we stop the Markov chain — we have found the solution.

One way to compute $\Delta E(x,y)$ is to compute $E(y)$ and put $\Delta E(x,y) = E(y) - E(x)$. In this application, computing the energy of a state is computationally expensive (time intensive). These Markov chains run for hundreds of thousands (sometimes millions) of steps and if this is done at every step things add up! Sometimes (as in this example) there is a shortcut to computing $\Delta E(x,y)$. The code `SudokuSolver.cpp` includes a function `int Conflicts (int r, int c, int d)` which computes, in the current x configuration, the conflicts with the row `r`, column `c`, site if the digit at that site is `d`. If, for example, going from x to y involves changing the digit at row 6, column 3, from a 1 to a 5, then

$$\Delta E(x,y) \;=\; \texttt{Conflicts(6,3,5)-Conflicts(6,3,1)}.$$

These calculations involve sites only in row 6 and column 3 and the block containing site $(6,3)$. Computing $E(y)$, by contrast, involves all 81 grid sites. With this approach, only the energy E of the *original* state must be computed. At each step, we compute $\Delta E(x,y)$ directly and, if the transition is accepted, update the energy by simply adding $\Delta E(x,y)$ to the current energy.

The selection of the temperature T can be a tricky business. The figure below shows the average time in seconds (the •s) it takes for the algorithm to repeatedly solve 50 randomly selected puzzles for various temperatures T with $0.34 \leq T \leq 0.44$ (outside of this range run-time increases dramatically). Since these average times were computed via Monte Carlo simulation, 95% confidence intervals for the results are also shown (the vertical bars). A quadratic OLS regression is superimposed which minimizes at $T \approx 0.39$. Those working with artificial intelligence would call this "training" the model's one parameter.



This algorithm is implemented in `SudokuSolver.cpp`, where you are invited to tinker with the temperature parameter. The program reads in a `.txt` data file that describes the puzzle to be solved. For the puzzle above, that file looks like

```
0 6 0 0 0 0 0 0 3
0 0 2 9 0 0 0 0 0
4 0 7 0 5 1 0 0 9
0 0 0 0 0 6 0 9 5
0 0 9 0 2 0 7 0 0
8 3 0 5 0 0 0 0 0
6 0 0 1 8 0 9 0 4
0 0 0 0 0 4 1 0 0
2 0 0 0 0 0 0 3 0,
```

where the non-zero digits are the clues and the zeros are at the solved-for sites. In the next section you will learn how to generate additional puzzles of varying degrees of difficulty to test this program.

§13. Making Sudoku Puzzles

In §12 we saw how to use Metropolis to solve Sudoku puzzles. Solving these puzzles may be hard (for a human), but making up Sudoku puzzles is harder! The set of clues must obviously have a solution, but the solution must also be *unique*. Here we describe a procedure for generating puzzles that uses Metropolis in two stages. Web sites like `websudoku.com` that generate puzzles