Exercises

8.15 **NP-completeness and kSAT.**[1] [2] (Computer science, Computation, Mathematics) ④

In this exercise you will numerically investigate a phase transition in an ensemble of problems in mathematical logic, called **kSAT** [8, 93]. In particular, you will examine how the computational difficulty of the problems grows near the critical point. This exercise ties together a number of fundamental issues in critical phenomena, computer science, and mathematical logic.

The **kSAT** problem we study is one in a class of problems called **NP**–complete. In other exercises, we have explored how the speed of algorithms for solving computational problems depends on the size N of the system. (Sorting a list of N elements, for example, can be done using of order $N \log N$ size comparisons between elements.) Computer scientists categorize problems into *complexity classes*; for example, a problem is in **P** if it can guarantee a solution[3] in a time that grows no faster than a polynomial in the size N. Sorting lists is in **P** (the time grows more slowly than N^2, for example, since $N \log N < N^2$ for large N); telling whether an N digit number is prime has recently been shown also to be in **P**. A problem is in **NP**[4] if a proposed solution can be *verified* in polynomial time. For example, factoring an integer with N digits is not known to be in **P** (since there is no known algorithm for finding the factors[5] of an N-digit integer that runs in a time polynomial in N), but it is in **NP**.

(a) *Given two proposed factors of an N digit integer, argue that the number of computer operations needed to verify whether their product is correct is less than a constant times N^2.*

There are many problems in **NP** that have no known polynomial-time solution algorithm. A large family of them, the **NP**–complete problems, have been shown to be maximally difficult, in the sense that they can be used to efficiently solve any other problem in **NP**. Specifically, any problem in **NP** can be translated (using an algorithm that runs in time polynomial in the size of the problem) into any one of the **NP**–complete problems, with only a polynomial expansion in the size N. A polynomial-time algorithm for any one of the **NP**–complete problems would allow one to solve all **NP** problems in polynomial time.

- The *traveling salesman problem* is a classic example. Given N cities and a cost for traveling between each pair and a budget K, find a round-trip path (if it exists) that visits each city with cost $< K$. The best known algorithm for the traveling salesman problem tests a number of paths that grows exponentially with N—faster than any polynomial.
- In statistical mechanics, the problem of finding the lowest-energy configuration of a spin glass[6]

[1]From *Statistical Mechanics: Entropy, Order Parameters, and Complexity* by James P. Sethna, copyright Oxford University Press, 2007, page 186. 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 and the associated software were developed in collaboration with Christopher Myers, with help from Bart Selman and Carla Gomes.
[3]**P** and **NP**–complete are defined for deterministic, single-processor computers. There are polynomial-time algorithms for solving some problems (like prime factorization) on a quantum computer, if we can figure out how to build one.
[4]**NP** does not stand for 'not polynomial', but rather for *non-deterministic polynomial* time. **NP** problems can be solved in polynomial time on a hypothetical *non-deterministic* parallel computer—a machine with an indefinite number of CPUs that can be each run on a separate sub-case.
[5]The difficulty of factoring large numbers is the foundation of some of our *public-key cryptography* methods, used for ensuring that your credit card number on the web is available to the merchant without being available to anyone else listening to the traffic. Factoring large numbers is not known to be NP-complete.
[6]Technically, as in the traveling salesman problem, we should phrase this as a decision problem. Find a state (if it exists) with energy less than E.

is also **NP**–complete (Section 12.3.4).

- Another **NP**–complete problem is 3-colorability (Exercise 1.8). Can the N nodes of a graph be colored red, green, and blue so that no two nodes joined by an edge have the same color?

One of the key challenges in computer science is determining whether **P** is equal to **NP**—that is, whether all of these problems can be solved in polynomial time. It is generally believed that the answer is negative, that in the worst cases **NP**–complete problems require exponential time to solve.

Proving a new type of problem to be **NP**–complete usually involves translating an existing **NP**–complete problem into the new type (expanding N at most by a polynomial). In Exercise 1.8, we introduced the problem of determining *satisfiability* (**SAT**) of Boolean logical expressions. Briefly, the **SAT** problem is to find an assignment of N logical variables (true or false) that makes a given logical expression true, or to determine that no such assignment is possible. A logical expression is made from the variables using the operations OR ($\vee$), AND ($\wedge$), and NOT ($\neg$). We introduced in Exercise 1.8 a particular subclass of logical expressions called **3SAT** which demand simultaneous satisfaction of M clauses in N variables each an OR of three literals (where a literal is a variable or its negation). For example, a **3SAT** expression might start out

$$[(\neg X_{27}) \vee X_{13} \vee X_3] \wedge [(\neg X_2) \vee X_{43} \vee (\neg X_21)] \dots \quad (1)$$

We showed in that exercise that **3SAT** is **NP**–complete by translating a general 3-colorability problem with N nodes into a **3SAT** problem with $3N$ variables. As it happens, **SAT** was the first problem to be proven to be **NP**–complete; *any* **NP** problem can be mapped onto **SAT** in roughly this way. **3SAT** is also known to be **NP**–complete, but **2SAT** (with clauses of only two literals) is known to be **P**, solvable in polynomial time.

Numerics

Just because a problem is **NP**–complete does not make a typical instance of the problem numerically challenging. The classification is determined by worst-case scenarios, not by the ensemble of typical problems. If the difficult problems are rare, the average time for solution might be acceptable even though some problems in the ensemble will take exponentially long times to run. (Most coloring problems with a few hundred nodes can be either quickly 3-colored or quickly shown to need four; there exist particular maps, though, which are fiendishly complicated.) Statistical mechanics methods are used to study the average time and distribution of times for solving these hard problems.

In the remainder of this exercise we will implement algorithms to solve examples of **kSAT** problems, and apply them to the ensemble of random **2SAT** and **3SAT** problems with M clauses. We will see that, in the limit of large numbers of variables N, the fraction of satisfiable **kSAT** problems undergoes a *phase transition* as the number M/N of clauses per variable grows. Each new clause reduces the scope for possible solutions. The random **kSAT** problems with few clauses per variable are almost always satisfiable, and it is easy to find a solution; the random **kSAT** problems with many clauses per variable are almost always not satisfiable, and it is easy to find a contradiction. Only near the critical point where the mean number of solutions vanishes as $N \to \infty$ is determining satisfiability typically a challenge.

A logical expression in conjunctive normal form with N variables X_m can conveniently be represented on the computer as a list of sublists of non-zero integers in the range $[-N, N]$, with each integer representing a literal ($-m$ representing $\neg X_m$) each sublist representing a disjunction (OR) of its literals, and the list as a whole representing the conjunction (AND) of its sublists. Thus $[[-3, 1, 2], [-2, 3, -1]]$ would be the expression $((\neg X_3) \vee X_1 \vee X_2) \wedge ((\neg X_2) \vee X_3 \vee (\neg X_1))$.

Download the hints and animation software from the computer exercises portion of the text web site [129].

(b) *Do exercise 1.8, part (b). Generate on the computer the conjunctive normal form for the 3-colorability of the two graphs in Fig. 1.8.* (Hint: There should be $N = 12$ variables, three for each node.)

The DP (Davis–Putnam) algorithm for determining satisfiability is recursive. Tentatively set a variable to true, reduce the clauses involving the variable, and apply DP to the remainder. If the remainder is satisfiable, return satisfiable. Otherwise set the variable to false, again reduce the clauses involving the variable, and return DP applied to the remainder.

To implementing DP, you will want to introduce (i) a data structure that connects a variable to the clauses that contain it, and to the clauses that contain its negation, and (ii) a record of which clauses are already known to be true (because one of its literals has been tentatively set true). You will want a *reduction routine* which tentatively sets one variable, and returns the variables and clauses changed. (If we reach a dead end—a contradiction forcing us to unset the variable—we'll need these changes in order to back up.) The *recursive solver* which calls the reduction routine should return not only whether the network is satisfiable, and the solution if it exists, but also the number of dead ends that it reached.

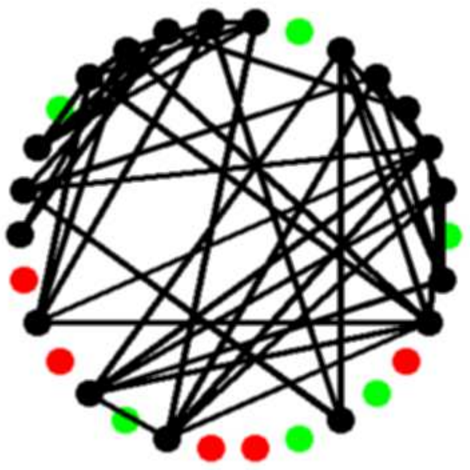

Fig. 8.20 D–P algorithm. A visualization of the Davis–Putnam algorithm during execution. Black circles are unset variables, the other shades are true and false, and bonds denote clauses whose truth is not established.

(c) *Implement the DP algorithm. Apply it to your 3-colorability expressions from part (b).*

Let us now explore how computationally challenging a typical, random **3SAT** problem is, as the number M/N of clauses per variable grows.

(d) *Write a routine, given k, N and M, that generates M random* **kSAT** *clauses using N variables. Make sure that no variable shows up twice in the same clause (positive or negative). For N =5, 10, and 20 measure the fraction of* **2SAT** *and* **3SAT** *problems that are satisfiable, as a function of M/N. Does the fraction of unsatisfiable clusters change with M/N? Around where is the transition from mostly satisfiable to mostly unsatisfiable? Make plots of the time (measured as number of dead ends) you found for each run, versus M/N, plotting both mean and standard deviation, and a scatter plot of the individual times. Is the algorithm slowest near the transition?*

The DP algorithm can be sped up significantly with a few refinements. The most important is to remove singletons ('length one' clauses with all but one variable set to unfavorable values, hence determining the value of the remaining variable).

(e) *When reducing the clauses involving a tentatively set variable, notice at each stage whether any singletons remain; if so, set them and reduce again. Try your improved algorithm on larger problems. Is it faster?*

Heavy tails and random restarts. The DP algorithm will eventually return either a solution or a judgment of unsatisfiability, but the time it takes to return an answer fluctuates wildly from one run to another. You probably noticed this in your scatter plots of the times—a few were huge, and the others small. You might think that this is mainly because of the rare, difficult cases. Not so. The time fluctuates wildly even with repeated DP runs on the same satisfiability problem [49].

(f) *Run the DP algorithm on a* **2SAT** *problem many times on a single network with $N = 40$ variables and $M = 40$ clauses, randomly shuffling the order in which you select variables to flip. Estimate the power law $\rho(t) \sim t^x$ giving the probability of the algorithm finishing after time t. Sort your variables so that the next one chosen (to be tentatively set) is the one most commonly arising (positive or negative) in the clauses. Does that speed up the algorithm? Try also reversing the order, choosing always the least used variable. Does that dramatically slow down your algorithm?*

Given that shuffling the order of which spins you start with can make such a dramatic difference in the run time, why persist if you are having trouble? The discovery of the heavy tails motivates adding appropriate *random restarts* to the algorithm [49]; by throwing away the effort spent exploring the neighborhood of one spin choice, one can both improve the average behavior and avoid the heavy tails.

It is known that **2SAT** has a continuous phase transition at $M/N = 1$, and that **3SAT** has an abrupt phase transition (albeit with critical fluctuations) near $M/N = 4.25$. **3SAT** is thought to have severe critical slowing-down near the phase transition, whatever algorithm used to solve it. Away from the phase transition, however, the fiendishly difficult cases that take exponentially long for DP to solve are exponentially rare; DP typically will converge quickly.

(g) *Using your best algorithm, plot the fraction of* **2SAT** *problems that are* **SAT** *for values of* $N = 25$, 50, *and* 100. *Does the phase transition appear to extrapolate to* $M/N = 1$, *as the literature suggests? For* **3SAT**, *try* $N = 10$, 20, *and* 30, *and larger systems if your computer is fast. Is your phase transition near* $M/N \approx 4.25$? *Sitting at the phase transition, plot the mean time (dead ends) versus* N *in this range. Does it appear that* **2SAT** *is in* **P**? *Does* **3SAT** *seem to take a time which grows exponentially?*

Other algorithms. In the past decade, the methods for finding satisfaction have improved dramatically. **WalkSAT** [116] starts not by trying to set one variable at a time, but starts with a random initial state, and does a zero-temperature Monte Carlo, flipping only those variables which are in unsatisfied clauses. The best known algorithm, **SP**, was developed by physicists [92,48] using techniques developed to study the statistical mechanics of spin-glasses.