Quantum Many-Body Systems **Boulder School**
Ashwin Nayak (U. Waterloo & Perimeter) **July 21, 2010**

Quantum Complexity Theory I

In these two lectures, we review computational complexity theory with the goal of understanding the barriers to simulating quantum many-body systems.

Formulating computational problems

Computational problems are abound in many areas of the natural and mathematical sciences. One such widely studied class of problems from statistical mechanics pertains to the Ising model. The model is formalized as follows.

Let $G = (V, E)$ be a graph with vertex set $V = [n] = \{1, 2, \ldots, n\}$ and m edges. This represents a neighbourhood structure, and could be, for example, a 2d lattice with periodic boundary conditions. Let $c_{ij} \in \mathbb{Q}$ be the *interaction coefficient* associated with the edge $\{i, j\} \in E$. A *spin configuration* σ (also called a *state*) in G is an assignment of a *spin* $\sigma_i \in \{+1, -1\}$ to each vertex $i \in V$. The spin at i interacts with an external magnetic field with strength $b_i \in \mathbb{Q}$. The energy $H(\sigma)$ of a configuration σ is given by the *Hamiltonian*:

$$H(\sigma) \quad = \quad - \sum_{\{i,j\} \in E} c_{ij} \sigma_i \sigma_j + \sum_{i \in V} b_i \sigma_i \ . \tag{1}$$

Note that we restricted ourselves to rational numbers for the interaction coefficients and strengths in the description above. This is so that when given as input to a computational problem, they have finite representation.

We are often interested in studying the properties of a *ground state* of the Ising model. The ground state is a spin configuration with minimum energy. The corresponding computational problem may be stated as: Given a graph $G = (V, E)$, the matrix interaction coefficients $c = (c_{ij})$ for the edges, and the vector of strengths $b = (b_i)$ for the vertices as input, produce as output a ground state.

The problem above is that of optimization, one of *search* for a state that minimizes the energy. In another problem pertaining to the Ising model, we may ask for the *partition function*, a positive real number, as the output. In order to be able to relate different such computational problems, it helps to reformulate these problems as *decision* problems. For example, instead of asking for a ground state, we ask if there exists a spin configuration with energy at most k, where $k \in \mathbb{Q}$ is given as an additional input. The solution to a decision problem is either YES or NO.

The decision version of the problem is "easier" than the search/functional version, as a solution to the latter immediately gives us a solution to the former. In the problem of computing the ground state of the Ising model, we can also find a ground state given a procedure for the decision version. This is described in detail in Appendix 1. Such a relationship of the decision version of the problem to the functional version holds for the problems we typically encounter in computer science, and justifies the focus on decision problems.

The set of instances (or inputs) of a decision problem for which the answer is YES is called a *language*. We often refer to a problem by the name of the associated language. The *size* or *length* of an input to the problem is the number of bits it takes to represent it in a canonical manner.

We denote the language corresponding to the problem of deciding whether there is a state with energy at most k for an instance (G, c, b) of the Ising model as ISING GROUND STATE. If we specify the input to this problem by giving the integers n, m, a list of the m edges along with the interaction coefficients $c = (c_{ij})$, and the list of strengths $b = (b_i)$, the input length would be, up to rounding errors,

$$\log_2 n + \log_2 m + \sum_{\{i,j\} \in E} (1 + \log_2 c_{ij}^{(1)} + \log_2 c_{ij}^{(2)}) + \sum_{i=1}^{n} (1 + \log_2 b_i^{(1)} + \log_2 b_i^{(2)}) + (1 + \log_2 k^{(1)} + \log_2 k^{(2)}),$$

where the rationals $c_{ij} = c_{ij}^{(1)}/c_{ij}^{(2)}, b_i = b_i^{(1)}/b_i^{(2)}, k = k^{(1)}/k^{(2)}$ are represented as ratios of integers, and the extra bit indicates the sign of the rational number.

Another problem, k-SAT (for some positive integer k), that we encounter frequently in computer science arises from boolean logic. An input to this problem is a boolean formula ϕ over n variables $x = (x_i)$, in *k-conjuntive normal form* (k-CNF). A formula in conjunctive normal form is the boolean AND (denoted by '$\wedge$') of some number of clauses, each clause is the boolean OR (denoted by '$\vee$') of some number of *literals*, and each literal is some variable x_i or its negation $\neg x_i$. Additionally, if every clause involves at most k literals, the formula is said to be in k-CNF. For example, the formula

$$\phi(x) \quad = \quad (x_1 \vee \neg x_2 \vee x_5) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \tag{2}$$

over 5 variables is an instance of 3-SAT. The associated question is whether the formula is *satisfiable*, i.e., whether there is an assignment of truth values 0 or 1 to the variables such that the formula evaluates to 1. The formula $\phi(x)$ in Eq. (2) is satisfiable with the (partial) assignment $x_1 = 1, x_2 = 1, x_3 = 0$.

The language k-SAT consists of all satisfiable k-CNF formulae. The input size is of the order of $mk \log_2 n$, when the instance contains m clauses and is over n variables, as it suffices to specify the k indices for the variables in each clause, and whether a variable is negated or not.

As in the case of ISING GROUND STATE, we can quickly construct a satisfying assignment for k-CNF formulae, given a procedure to solve the decision problem k-SAT. We leave this as an exercise to the reader.

What is an algorithm?

Early in the 20th century, when the notion of an algorithm was being formalized, several models of computation were proposed: the Turing machine, the λ-calculus by Church, recursive functions by Church, Kleene, and Rosser, etc. In its own way, each captured the intuition that an algorithm consists of a sequence of elementary steps that may be followed mechanically to compute functions. However, it was soon discovered that these apparently different models all single out the *same* set of functions as being computable. Subsequently defined models, including the Random Access Machine (RAM), on which high-level programming languages are based, and the Boolean Circuit model, on which current day computer architecture is based, are also equivalent to the Turing machine. Moreover, it was found these models simulate each other efficiently (a term we will define rigorously below). This lead to an empirical observation, *the strong Church-Turing thesis*, that may be paraphrased as

> "A probabilistic Turing machine can efficiently simulate any realistic model of computation."

From a complexity-theoretic perspective it is most convenient to work with boolean circuits, as defined below. From the perspective of algorithm design it is more convenient to work with a high-level programming language, and as we do in presenting algorithms. We assume familiarity with writing "pseudo-code" and do not attempt to formally define a programming language here. The Church-Turing thesis along with the fact that (probabilitic) boolean circuits can efficiently simulate (probabilistic) Turing machines guarantees that there is no loss in generality in following our convention.

A boolean circuit C of size T over n variables $x = (x_i)$ consists of a sequence of T logical gates $g_1, g_2, \ldots, g_T$, where each $g_j \in \{\wedge, \neg\}$ and is associated either a single input wire or a pair of them, denoted by w_j. If the gate g_j is the gate $\wedge$, $w_j = (a_j, b_j)$ where a_j, b_j may be any of the variables x_i or (the output of) any preceding gate g_k ($k < j$). Similarly, when g_j is $\neg$, w_j is a single such input. The function computed by the circuit for an assignment of values to x is the value of the gate g_T. Figure 1 depicts such a circuit.

We may verify that any boolean function over n variables can be computed by a circuit of the above form, so the set $\{\wedge, \neg\}$ is a *complete* or *universal* gate set. As a consequence, any other finite set of gates can be simulated by finite size boolean circuits over $\{\wedge, \neg\}$. In effect, circuits over other finite gate sets can therefore be simulated by the circuits defined above with only a constant factor blow up in size.

An algorithm for a decision problem or language L is family of circuits (C_n), one for each input length n, such that for every n, and every input $x \in \{0,1\}^n$ of length n, we have $C_n(x) = 1$ if and only if $x \in L$.

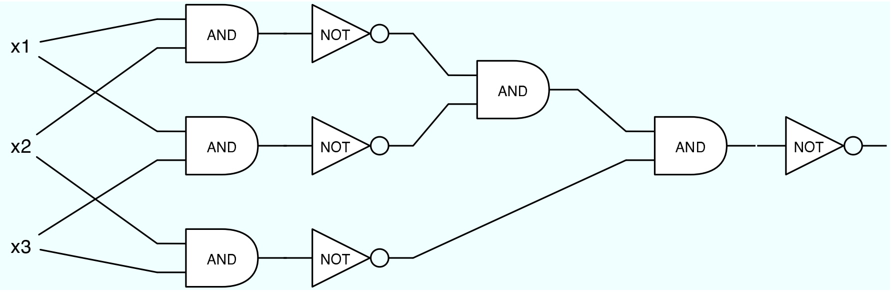

Figure 1: A boolean circuit computing the majority of three bits.

We would like to emphasize that an algorithm is guaranteed to produce the correct answer for every input, and is different in that respect from heuristics for which no such guarantees are available. A subtlety in the definition of an algorithm is that the above description of the circuit C_n is required to be computable by a deterministic Turing machine in $p(T(n))$ steps, where p is a polynomial function of its argument, and $T(n)$ is the size of the circuit C_n. Otherwise, we can design circuit families for any function, even for a function that is deemed uncomputable with respect to other reasonable models of computation. Families of circuits which are constructible in the manner described above are said to be *(polynomial-time) uniform*.

The existence of uncomputable functions may seem to be at odds with the fact that every function of n variables can be realised with a circuit. However, the generic such construction of the circuit presupposes knowledge of the function values for input length n, and therefore corresponds to a different procedure for computing f for each input length. This is clearly at odds with our intuitive understanding of an algorithm as we would like an algorithm to follow the same kind of rules for computing a function for any input length. Uniform constructibility ensures this condition.

Efficient algorithms and P

Computational complexity theory is the study of the physical resources such as space and time that are required to solve problems. Our focus is the time-complexity of problems. The insistence on elementary operations that consume only unit resources, be it time, space, or energy, ensures that most resource requirements are captured by the notion of time and space complexity. In the case of boolean circuits, the size of the circuit, i.e., the number of gates in it, is the relevant measure of complexity and corresponds to time complexity. Indeed, if we evaluated a function specified by a circuit, we would take time proportional to the number of gates in it (with some additional overhead required to store and retrieve the intermediate results).

Another parameter, the *depth* of a circuit, captures the amount of time a parallel processor would take to evaluate the function. The depth is the longest path in the circuit from an input variable to the output gate. This complexity measure is outside the scope of these lectures.

How large are the circuits for ISING GROUND STATE, or for k-SAT?

If the interactions in the Ising model are all ferromagnetic, i.e., $c_{ij} \geq 0$, and the $b_i \geq 0$, there is a simple solution: a ground state is one in which all spins are -1, regardless of the underlying graph. In next week's lectures by Barbara Terhal, you will see an algorithm that makes $\mathrm{O}(n)$ arithmetic operations, when the underlying graph is a 1d chain of length n (i.e., the path graph with n vertices) and computes a ground state. Addition of ℓ-bit numbers takes $\mathrm{O}(\ell)$ operations using the method we learnt in elementary school.

As mentioned in the previous section, we only bound the number of steps taken by high-level descriptions of the algorithms we present. The corresponding circuit size is larger by up to a fixed polynomial in the bounds we present.

Another case is when the underlying graph is planar, and there is no external magnetic field. An elegant construction due to Kasteleyn, Fisher, and others, reduces the computation of the ground state to computing a minimum weight perfect matching, a problem that may be solved using $\mathrm{O}(n^{2.5})$ arithmetic operations, for graphs with n vertices. Using a further connection to determinants, even the partition function may be computed exactly with $\mathrm{O}(n^3)$ addition and multiplication operations. Multiplication of ℓ-bit numbers takes $\mathrm{O}(\ell^2)$ operations using the method we learnt in elementary school. (More efficient methods for arithmetic exist, which take nearly linear in ℓ operations. Similarly, more efficient methods to compute the determinant also exist.)

The above algorithms break down when the additional constraints we imposed above are lifted. When the interactions may be anti-ferromagnetic, or when the underlying graph is not planar, or when there is an external magnetic field, the best known algorithms do not perform qualitatively better than a brute-force search through all 2^n spin configurations. The latter is an exponential in the size of the input, in stark contrast to the polynomial number we had before.

The story of k-SAT is similar; there seems to be an abrupt jump in the complexity of the best known algorithms from polynomial in n to exponential, as we consider its variants. When $k = 1$, the formula is a conjunction of some number of literals. It is satisfiable if and only if the conjunction does not include both a variable and its negation. This may be determined in by sorting the literals by the indices of the variables in them, and scanning the sorted list for contradictory literals. This takes $\mathrm{O}(n \log n)$ operations.

The problem 2-SAT also turns out to be computationally easy, and a solution that takes time linear in the size of the formula is sketched in Appendix 2. No similar algorithm is known for 3-SAT. In fact the best known algorithm takes time of the order of $n^\alpha 2^{\beta n}$, for some constants $\alpha > 0$, and $0 < \beta < 1$. Again we see a polynomial versus exponential gap in the complexities of the two closely related problems.

Polynomial versus exponential complexity of course has dramatic implications for practical computations. For example, even with the fastest of computers available today, solving 3-SAT over a few thousand variables would take more than the estimated lifetime of the universe! A 2-SAT instance with the same numner of variables can be solved within a few minutes. It should therefore not come as a surprise that we call polynomial-time algorithms *efficient*.

Formally, we say a language L is solvable in polynomial time (or efficiently solvable), if it has an algorithm, i.e., a uniform family of circuits $\{C_n\}$ with size $T(n)$, such that T is a polynomial in n. The complexity class **P** is the collection of all languages that are solvable in polynomial time.

Just as the set of functions that are deemed computable does not change under different reasonable models of computation, the class **P** is robust under change of the model of computation. This forms the basis of the complexity-theoretic version of the Church-Turing hypothesis, and lends further credence to it being an appropriate definition of efficiently solvable problems.

We currently do not have a precise characterization of which computational problems belong to the class **P**, and which do not. Indeed, it has taken several decades of efforts by computer scientists to develop polynomial time alorithms for fundamental problems such as linear programming and testing primality of integers. Next, we see how in some cases overwhelming theoretical evidence suggests that polynomial time algorithms may not exist.

Appendix 1: From decision to search

Here we describe a method to efficiently compute a ground state of the Ising model given a procedure to solve the decision version.

We first search for the ground state energy k_0 by performing a binary search in the range between $C = -\sum_{ij} |c_{ij}| - \sum_i |b_i|$ and 0 (the minimum energy is always in this range). In other words, we set $k = C/2$, and run the decision procedure. If the answer is YES, the minimum energy is at most $C/2$. So we set k to be the midpoint of the interval on the left, i.e., $k = 3C/4$, and recurse. If the answer is NO, we set $k = C/4$ and recurse on the interval on the right. The recursion ends when we have an interval smaller than the precision with which we have specified the rationals in the problem. This search procedure identifies the minimum energy k_0. We are still left with the task of identifying a ground state with energy k_0. We may do so by a different reduction to the decision procedure, in which we successively set the spin at $i = 1, 2, \ldots, n$

to $+1$, and ask if there is a state with energy at most k_0. If YES, we set $\sigma_i = +1$, otherwise we set it to -1, and move to the next spin. This gives us a ground state with energy k_0.

A point which we swept under the rug: the original formulation of the problem did not include partial assignments of spin. We may easily accommodate partial assignments by modifying the parameters of the model. Fixing the value of the spin at a vertex i to $+1$ is equivalent to removing the vertex i from the graph G, so also all incident edges $\{i, j\}$, and modifying the strength b_j to $b_j - c_{ij}$ for every vertex j adjacent to i. We also modify the minimum energy from k_0 to $k_0 - b_i$. Call the new instance (G', c', b'). There is a ground state with energy $k' = k_0 - b_i$ in the new instance (G', c', b') if and only if there is a ground state with energy k_0 in the original instance (G, b, c) with ith spin equal to $+1$.

The kind of reduction we saw above is called a *Turing* or *Cook reduction*, as it involves the solution to multiple instances of the decision problem. In contrast, the Karp reductions we saw in proofs of **NP**-hardness involve only one instance.

Appendix 2: A linear time algorithm for 2-SAT

Here we describe an efficient solution to 2-SAT. To simplify the exposition of the solution, we modify a given input formula ϕ so that all its clauses contain two literals. Any clause of size 1 is replaced by an OR of the associated literal with itself. Now note that an OR of two literals is equivalent to an implication: $(l_1 \vee l_2)$ is true iff $(\neg l_1 \implies l_2)$ is true under any assignment of truth values. We construct a directed graph G_ϕ with $2n$ vertices, where n is the number of variables in the input formula. Each vertex corresponds to a variable or its negation. For every clause $(l_1 \vee l_2)$ in ϕ we place directed edges $(\neg l_1, l_2)$ and $(\neg l_2, l_1)$ in the graph G_ϕ. If ϕ has m clauses, the graph G_ϕ has at most $2m$ edges.

The graph corresponding to the formula

$$\psi(x) \quad = \quad (x_1 \vee \neg x_4) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_2) \wedge (x_2 \vee \neg x_4) \tag{3}$$

is shown in Figure 2.

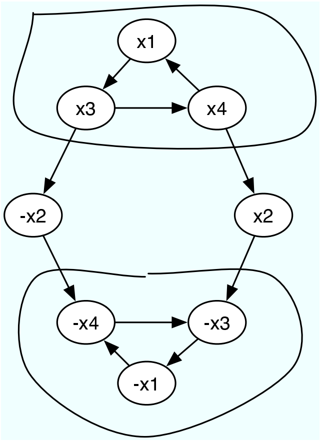

Figure 2: The directed graph G_ψ corresponding to the formula in Eq. (3). Two of its SCCs are circled.

Every directed graph may be decomposed into its *strongly connected components* (SCCs). These are maximal subgraphs in which there is a directed path from every vertex to every other vertex. There are four SCCs in the graph G_ψ in Figure 2. Two have been circled, and the remaining two consist of single vertices x_2 and $\neg x_2$. The SCCs in any directed graph can be identified by exploring the graph (twice) in a systematic fashion called *depth first search*. This takes time linear in the number of vertices and edges of the graph. If we shrink all the vertices in an SCC into a "super vertex", we see that the resulting directed graph has no directed cycles, i.e., is acyclic. Any directed acyclic graph (DAG) has a *source*, a vertex which has no incoming edges.

We use the decomposition of the graph G_ϕ into its SCCs to determine whether the formula ϕ is satisfiable or not. If there is an SCC in G_ϕ which contains both a variable x_i and its negation $\neg x_i$, the formula contains two chains of implications, one equivalent to $x_i \implies \neg x_i$, and the other to $\neg x_i \implies x_i$. These together assert that both x_i and $\neg x_i$ be true, which is impossible.

If there is no SCC in G_ϕ that contains both a variable and its negation (call this the consistency condition), we can find a satisfying assignment as follows. We find a source SCC in the graph. We assign all the literals in it the value 0. This fixes to 1 the value of their negations. In turn, this fixes to 1 the value of all the literals in the SCC to which the negations belong also. In fact, this fixes to 1 the value of all literals reachable from the negations. We delete all the vertices whose values have been fixed, and the incident edges. This leaves us with a subgraph of the DAG corresponding to G_ϕ. We repeat the same procedure with this new DAG until all variables have been assigned truth values. We leave it to the reader to verify that we can assign truth values to all variables because of the consistency condition, and that the truth values satisfy the formula ϕ. Note that the step of assignment of values also takes time linear in the size of the graph.

We may run this algorithm on the graph G_ψ in Figure 2. We see that no SCC contains both a variable and its negation, so we proceed to assign truth values to variables. The SCC at the top is a source, so we first assign value 0 to the variables x_1, x_3, x_4. This fixes the values in the SCC at the bottom to 1. We delete both these SCCs. The remaining SCCs are both sources, and we may pick either of them, say, $\neg x_2$, and assign it the value 0. This concludes the process, and we see that the resulting assignment satisfies ψ.

Note that adding the clause x_1 to ψ causes the corresponding graph to have an additional edge from $\neg x_1$ to x_1, so that all vertices now belong to the same SCC and a consistent assignment is impossible. We may verify that the formula $\psi(x) \wedge x_1$ is not satisfiable.