C/CS/Phys C191 Complexity and Reversible Computing 9/27/07
Fall 2007 Lecture 10

1 Readings

Benenti, Casati, and Strini:

Reversible computation Ch.1.5 - 1.6

Turing Machines Ch. 1.1

Complexity Theory Ch. 1.3 - 1.4

2 Reversible Computation

The classical NAND gate is irreversible. Landauer first pointed out that there is a minimal amount of energy dissipated from the computer into the environment with every irreversible computation. This minimal energy loss is equal to $kTln2$ and derives from the entropy decrease of the information on erasure (initialization) of one bit. Entropy decrease of the information is accompanied by an entropy increase of the environment and dissipation of energy into this. Note that information content is equivalent to our ignorance of the message, i.e., of the actual state of the bit.

Energy dissipation in electronic circuits has decreased substantially over the last 40 years, by a factor of about 10 every 4 years, going from about 1 x 10^{-2}J per logical operation in 1940 to about 1 x 10^{-7}J per logical operation today. Extrapolation of this trend (is this valid?) would imply that the energy dissipated per logical operation will reach the thermal limit kT at $T = 300$ K within 10-15 years from now. In that situation spontaneous fluctuations could cause the circuits to switch and computations to cease being reliable. The anticipation of this situation led to the development of reversible computation schemes.

We shall illustrate this here by supposing that we are given a classical circuit, for example for primality testing of an input number M (note that the length of the input M is $\lceil \log_2 M \rceil$), and showing how to construct a corresponding reversible circuit.

Any classical circuit can be built from three basic pieces: the AND gate $\wedge(a,b) = a \cdot b$, the NOT gate $NOT(a) = 1-a$, and fan-out (copying). The NOT gate is a reversible, unitary one-qubit operation. But the AND gate clearly cannot be reversible; since it takes two bits to just one, some information must be lost. (In particular, after applying a NOT gate, we cannot distinguish the cases where the inputs were 00 versus 01 or 10.) The AND gate erases information, which is not reversible. However, copying of classical states is reversible, $(a,0) \rightarrow (a,a)$. There are a number of ways to construct a reversible circuit corresponding to a nonreversible circuit. For example, one can build the necessary pieces out of a controlled swap gate (Fredkin gate), a CNOT gate, and a NOT gate. Here, we will construct a corresponding reversible circuit using the Toffoli gate and the NOT gate. We encountered the Toffoli gate a couple of lectures ago: it is a doubly-controlled NOT gate with action (a,b,c) to $(a,b,c+ab \bmod 2)$. It is its own inverse, so it is by definition reversible.

Now we consider the standard, possibly irreversible, circuit C taking input x to $y = C(x)$, shown in Figure 1. The Toffoli and NOT gates can be used to replace each of the circuit components to build a reversible circuit $\hat{C}$ taking $(x,0^k)$ to (x,y). Actually, some number of ancilla bits, initialized to 0 will also be useful, so $\hat{C}$ takes

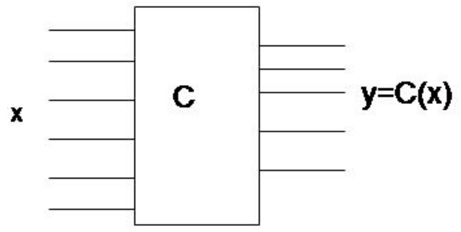

Figure 1: The classical, possibly nonreversible circuit C

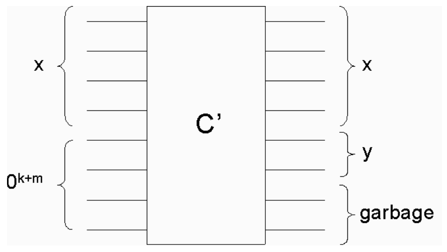

Figure 2: The reversible circuit C' that produces extra garbage

$(x, 0^k, 0^m)$ to $(x, y, 0^m)$. (Notice that the m ancilla bits are unchanged. . . which will require putting them back to their initial states.) The replacement procedure is summarized below and the resulting reversible circuit is shown in Figure 2.

- To achieve an AND gate on a and b, we use the Toffoli gate on input $(a, b, 0)$. The output is $(a, b, a \wedge b)$, so the third output wire has the result of the AND.
- To copy a bit a, use the Toffoli gate on input $(a, 1, 0)$. The output is $(a, 1, a)$, so we have copied a. Note that this uses both 0 and 1 ancillas; to get a 1 ancilla, we can simply apply a NOT gate to a constant 0 wire.

This method allows us to construct a circuit C' corresponding to C which reversibly takes $(x, 0^k, 0^m)$ to $(x, C(x), \text{garbage}_x)$. The garbage is left over in the original ancilla wires because our operations had extra inputs and outputs. For example to achieve a NOT we had three output wires, whereas the nonreversible NOT gate only has one output wire. The extra two wires are just garbage which we can reset to their initial states for a clean circuit.

Now let us consider how things look with quantum circuits. As a historical note, quantum computation actually was originally (in the late 70s and early 80s) studied to understand whether unitary constraint on quantum evolution provided *limits* beyond those explored in classical computation. A unitary transformation taking basis states to basis states must be a permutation. (Indeed, if $U|x\rangle = |u\rangle$ and $U|y\rangle = |u\rangle$, then $|x\rangle = U^{-1}|u\rangle = |y\rangle$.) Therefore quantum mechanics imposes the constraint that its classical analog must be reversible computation.

Now from the perspective of a quantum algorithm, the extra garbage left over is quite significant because it can prevent desirable destructive interference when we run the circuit on states which are not just computational basis states. Thus in a quantum circuit we definitely want a clean output. Fortunately, there is a simple trick for removing all the garbage; we just run the circuit in reverse to erase it! Of course, we don't

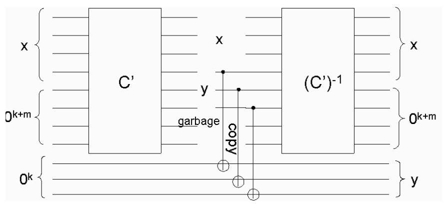

Figure 3: The clean reversible circuit $\hat{C}$ built out of C' and $(C')^{-1}$.

want to forget our final answer, so before running C' in reverse we need to copy $y = C(x)$ to some additional ancilla wires. This copying just uses one additional 1 ancilla, and does not create any further garbage. The sequence of steps is then

$$(x, 0^k, 0^m, 0^k, 1) \xrightarrow{C'} (x, y, \text{garbage}_x, 0^k, 1) \xrightarrow{\text{copy } y} (x, y, \text{garbage}_x, y, 1) \xrightarrow{(C')^{-1}} (x, 0^k, 0^m, y, 1) \ .$$

Overall, this gives us a clean reversible circuit $\hat{C}$ corresponding to C, shown in Figure 3.

3 Complexity Theory

Complexity theory deals with the scaling of a computation with the resources, in particular, with the number of bits (qubits) specifying the input (the size of the input). An algorithm is defined as a set of instructions for solving a given problem, e.g., $a + b =?$.

Read the section on Turing machines in Benenti et al., Ch. 1.1. The Church-Turing thesis states that any function computable by an algorithm can be computed on class of a Turing machine, the paradigm of a computational device. Furthermore, there exists a universal Turing machine that given a descriptor of a given computation from a specific Turing machine, can perform that computation with at most polynomial slowdown. This is related to the strong version of the Church-Turing thesis, which states that any model of computation can be simulated by a probabilistic Turing machine with at most polynomial increase in the number of gates. Since gates are equivalent to time, the converse of this implies that if a computational problem cannot be solved in polynomial time on a probabilistic Turing machine, then it is not solvable. Shor's algorithm challenges this since there is a polynomial quantum algorithm, but no known polynomial time classical algorithm. Whether one may exist or not is still an open problem.

Now consider how computations scale with number of bits. E.g., finding the square of an integer, x. We need $L = log_2 x$ bits to represent the number and require $s \propto L^2$ steps to compute x^2. Thus finding the square of a numbers is in P, i.e., $s \propto L^k$ where k is an integer. We say that problems with polynomial complexity are 'easy', while problems with superpolynomial complexity are 'hard'. These include scalings $s \propto exp(L), s \propto 2^n, s \propto n!$. Why this distinction? Several reasons:

- usually if one has an algorithm scaling algorithmically one has taken advantage of some mathematical insight into the problem - polynomial algorithms are generally low order ($k = 1, 2, 3$) and one rarely finds $k > 10$
- suppose that today you solve a problem of size n and are asked tomorrow to solve the same problem for a number of size $n+1$. If your algorithm scales as $O(2^n)$ you will need twice as much computation time tomorrow, but if your algorithm scales as $O(n^2)$ then you will need only a small fraction more than today (consider $2n+1$ versus n^2).

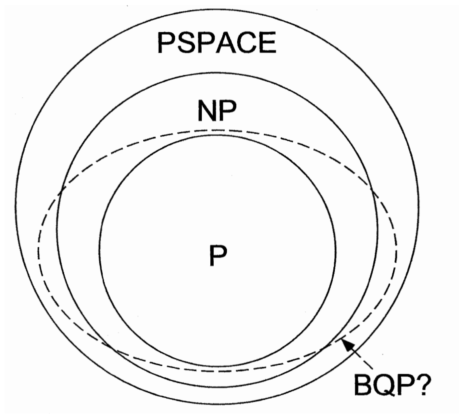

Figure 4: Schematic of possible hierarchy of complexity classes.

Examples of complexity classifications:

- matrix mutliply $O(n^3)$ (now lower order but greater than 2 is possible)
- sorting n items $O(nlogn)$
- factorization of an integer N by number field sieve $expO(n^{1/3}(logn)^{2/3})$ (here $n = logN$ is the input size). For a number N with 250 digits, this requires about 10^6 years on a 200 MIPS machine.

Complexity classes

P = solve algorithm in time (gates) polynomial in the number of bits

NP = can verify a solution in polynomial time, e.g., whether numbers a and b factor N (just multiply them together)

$P \subset NP$ but it is an open questions as to whether NP is actually larger than P...

NPC = any NP problem can be reduced to it ($NP - complete$), e.g., the traveling salesman problem.

$PSPACE$ = polynomial in space resources but no limit on time

$P \subset PSPACE$ but it is also an open question whether $PSPACE$ is actually bigger than P...

BPP = bounded probabilistic polynomial algorithm

BQP = quantum probabilistic algorithm with bounded error and running in polynomial time

What is known is that

$P \subseteq BPP \subseteq BQP \subseteq PSPACE$. The extent of BQP is not well understood.

For an introductory overview to complexity theory, see S. Mertens, cond-mat/0012185.