A. Gupta
Vazirani Class #4
September 4, 1997

IV. Reversible Computation

1 Reversible Computation

It is interesting to note that all the Quantum gates we have studied are reversible. Indeed, the Hadamard gate and the controlled NOT gates are their own inverses; the rotation gate R_θ has the inverse $R_{-\theta}$, while the phase shift gate $P(\phi_1, \phi_2)$ can be inverted by the gate $P(-\phi_1, -\phi_2)$. This is not by chance, but because the laws of quantum mechanics require the evolution of the state vector to be unitary and therefore time reversible.

Now, suppose we restrict our quantum circuits so that the state of each wire is either $|0\rangle$ or $|1\rangle$, but never a non-trivial superposition, then the circuit reduces to a classical circuit — with the additional constraint that it is time reversible. So, a priori, it is not even clear that quantum circuits are as powerful as classical circuits. Fortunately, the power of reversible computation had already been studied in the context of inherent bounds on the amount of heat dissipated in a circuit. In an early paper, von Neumann had made the comment that it was clear from thermodynamic considerations that each step of computation requires the dissipation of kT units of energy. Bennett analyzed this assertion more rigorously and concluded that energy is dissipated only while erasing information, not while computing with it. Furthermore, he showed how any classical circuit or Turing Machine could be simulated efficiently by a reversible circuit or Turing Machine. These constructions are the topic of this lecture.

2 Reversible Classical Logic Circuits

Reversible logic circuits can be built using Fredkin gates (see figure 1). This gate has 3 inputs: the third input c is the control and $c' = c$. When c is set to 1, then the first two bits x and y are swapped, else they are left untouched. Clearly this is a reversible gate. To see that this gate is universal, note that if the x is set to 1, then $x' = y \wedge z$, thus simulating the AND gate. If $x = 0$ and $y = 1$, then y' gives us $\neg c$, while x' gives us another copy of c, hence giving us both the NOT and FANOUT gates.

Clearly, any classical circuit C can be simulated by a reversible circuit C_r of Fredkin gates, albeit with the added presence of some clean 0 and 1 inputs. Further, if the size of a circuit is measured by the number of wires in it, then C_r is larger by no more than a constant factor. However, along with the required function output, we get some extra outputs (which we shall call "the junk").

But is the junk necessary ? Clearly, We will have some number of extra output bits if the number of (real) output bits is less than the number of input bits. To circumvent this problem, we instead try to implement the circuit for the function $x \mapsto \langle x, f(x)\rangle$, and re-ask the question.

And it turns out that we can indeed avoid the junk in the output, and this is done as in figure 3.

We compute C_r as before, getting $\bar{y} = f(\bar{x})$ and the junk. Then we make a safe copy of the output $\bar{y}$, and then feed one copy of this along with the junk into the inverse circuit C_r^{-1}. This gives us back some clean bits along with $\bar{x}$. Thus we have managed to perform the transformation

$$\langle \text{clean bits}, \bar{x} \rangle \longmapsto \langle \text{clean bits}, \bar{x}, f(x) \rangle.$$

Henceforth, we shall ignore the presence of the input clean bits which are again output as clean bits (even if they are transmuted within the circuit).

It turns out that we can do better in certain cases. If f is a bijection, we can compute f^{-1}. This allows us to do away with the artificial device of going to the auxiliary function $x \mapsto \langle x, f(x) \rangle$. Instead, we just consider the two circuits: first C_f, which performs the map f, i.e.

$$\langle \text{clean bits}, x \rangle \longmapsto \langle x, f(x) \rangle,$$

and the circuit $C_{f^{-1}}$ for f^{-1}:

$$\langle \text{clean bits}, f(x) \rangle \longmapsto \langle f(x), x \rangle.$$

It is a simple task to piece together C_f and $C_{f^{-1}}^{-1}$ (the inverse of $C_{f^{-1}}$) to get the map

$$\langle \text{clean bits}, x \rangle \longmapsto \langle \text{clean bits }, f(x) \rangle,$$

or with our assumption of ignoring clean bits, $x \mapsto f(x)$.

3 Reversible Turing Machines

A k-tape deterministic Turing machine is given by an alphabet Σ, a set of states Q, and a transition function $\delta : Q \times \Sigma^k \to Q \times \Sigma^k \times \{L, R\}^k$, which is a map from the curent state and the symbols being read on the tapes to the new states, the new symbols to be written on the tapes and the direction of motion for each of the k heads.

The fact that δ is a function implies that each configuration has a unique successor and that the machine is deterministic. For a reversible machine, we require that each state should have a unique predecessor, or in other words, any previous state of the computation can always be reconstructed given a description of the current state.

To get a reversible computation of $x \mapsto \langle x, f(x) \rangle$, we add another tape to our TM which we refer to as the *history tape*. Now whenever we want to make a move on state q and tape symbols $\langle a_1, a_2, \ldots, a_k \rangle$, we do two steps: we first write down this information onto the history tape (we shall assume that our alphabet is expanded to allow this to be written down as a single symbol). Then we read this symbol from the history tape and perform the action according to it. Thus the history tape is a record of all the actions performed by the TM and can be used to undo the changes.

It is easy to see that the above process is reversible, and that we can safely talk about the reverse computation. Thus the actual computation (without any junk being produced) involves performing the steps described above, making a copy of the output $f(x)$ and then performing the reverse computation to get rid of the junk.

Note that the entire computation can be performed in $O(T)$ time, if the original irreversible computation was done in T time. However, we are losing a lot in terms of space, with the requirement going up from S to $O(S+T)$ space.

3.1 Reducing the Space Overhead

We can actually do better using a very elegant simulation technique, which requires $O(T^{1+\epsilon})$ time but only $O(S \log T)$ space. Noting that any non-looping computation takes time $T \leq 2^S$, we can always simulate any TM by a reversible one in $O(S^2)$ space.

The idea is this: suppose we simulate only the first half of the computation, save the configuration C (which consists of all the information on the tapes, the positions of the heads and the state of the machine) and clean up. Then starting from this saved half-way configuration C, we complete the run, save the result and undo everything back to C.

At this point, however, we do not have the information to roll back to the start. Thus we start the computation from the beginning again, reach C, erase the previous version of C (which is same as making a copy), and roll back to the beginning. Note that the space requirement has been almost halved (since we could re-use space) at the cost of doing 1.5 times as much work.

We can now recurse on each of the halves, which gives us even better results. In fact, the space requirement satisfies $\mathbf{Space}(n) = \mathbf{Space}(n/2) + S$, giving us $\mathbf{Space}(T) = O(S \log T)$. However, the recurrence for the time required is $\mathbf{Time}(n) \leq 6\mathbf{Time}(n/2) + O(n)$, which gives $\mathbf{Time}(T) = T^{\lg 6}$.

To get running times which are arbitrarily close to T, we have to modify our procedure slightly, breaking up into k parts intsead of 2. This would give us $\mathbf{Time}(n) \leq 4k\mathbf{Time}(n/k) + O(n)$ and thus $\mathbf{Time}(T) \leq T^{\log_k 4k}$ with a space requirement of $kS \log_k T$. We can now pick suitable values of k depending on the space-time tradeoff desired.

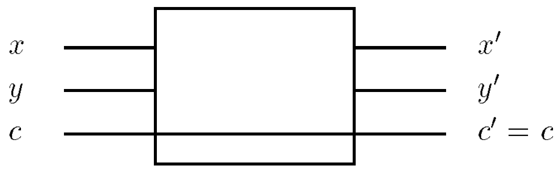

Figure 1: The Fredkin gate

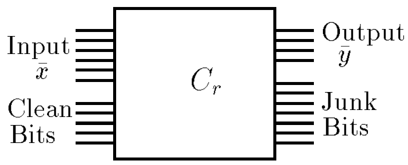

Figure 2: A reversible circuit

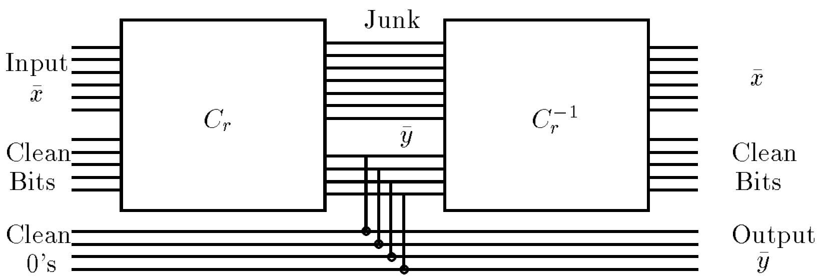

Figure 3: Computing $x \mapsto \langle x, f(x) \rangle$

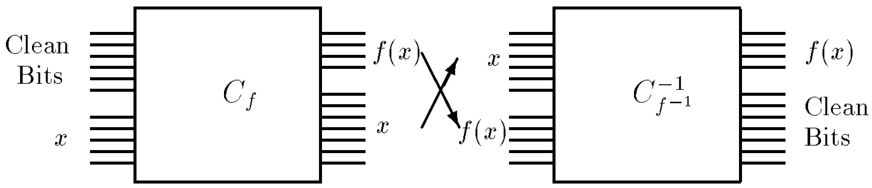

Figure 4: Computing a bijective function f