Quantum Algorithms via Linear Algebra

— Basics of quantum computations

Dao-Yun Xu

College of Computer Science and Technology, Guizhou University

Outline

4. Boolean Functions, Quantum Bits, and Feasibility

A Boolean function f is a mapping from $\{0,1\}^n$ to $\{0,1\}^m$, for some numbers n and m:

$$f(x_1, ..., x_n) = (y_1, ..., y_m).$$

When $m = 1$, we can also think of f as a predicate: x satisfies the predicate if and only if $f(x) = 1$.

Basic functions:

- NOT: the unary function.
- AND: $f(x_1, ..., x_n) = 1$ if and only if every argument is 1.
- OR: $f(x_1, ..., x_n) = 1$ if and only if the number of 1's in $x_1, ..., x_n$ is non-zero.
- XOR: $f(x_1, ..., x_n) = 1$ if and only if the number of 1's in $x_1, ..., x_n$ is odd.

Boolean inner product $x \bullet y = XOR(x_1 \wedge y_1, ..., x_n \wedge y_n)$.

The following two Boolean functions should be not regarded as basic:

- PRIME: The function $f(x_1, ..., x_n)$ defined as 1 if the Boolean string $x = x_1, ..., x_n$ represents a number that is a prime number.

- FACTOR: The function $f(x_1, ..., x_n, w_1, ..., w_n)$ is regarded as with integers x and w as arguments. Note that we can pad w as well as x by leading 0's.

  It returns 1 if and only if x has no divisor greater than w, aside from x itself.

Therefore, $PRIME(x) = FACTOR(x, 1)$ for all x. This implies that a circuit for $FACTOR$ immediately gives one for solving the predicate $PRIME$.

4.1. Feasible Boolean Functions

The number of gates is identified with the amount of work expended by the circuit, and this in turn is regarded as the sequential time for the circuit to execute.

The critical counting gates is that only **basic operations** can be used, and that they can only apply to previously computed values.

The classical complexity for functions AND and XOR are computable in a linear time $O(n)$.

It is known circuits for $PRIME$ take more than linearly many steps, but the time is still polynomial $(n^{O(1)})$.

But there are also Boolean functions that require time exponential in n. Many people believe that $FACTOR$ is one of them, but nobody knows for sure.

Representing Boolean functions by their **truth tables** is always possible, but is not always **feasible**. The tables will be large when there are thirty inputs.

The technical concept we need is having not just a single Boolean function but rather a **family** $[f_n]$ of Boolean functions, each f_n taking n inputs, that are conceptually related. That is, the $[f_n]$ constitute a single function f on strings of all lengths, so, we write $f : \{0,1\}^* \to \{0,1\}^*$.

DEFINITION 4.1. A Boolean function $f = [f_n]$ is **feasible** provided the individual f_n are computed by circuits of size $n^{O(1)}$.

4.2. An Example

Consider the Boolean function $MAJ(x1, x2, x3, x4, x5)$, which takes the majority of five Boolean inputs.

The first idea is to compute it using applications of OR and AND as follows:

For every three-element subset $S = \{i, j, k\}$ of $\{1, 2, 3, 4, 5\}$, we compute $y_S = OR(x_i, x_j, x_k)$.

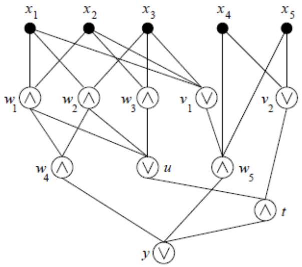
Define y to be the AND of each of the *ten* subsets S. That is: $y = AND(y_{S_1}, \cdots, y_{S_{10}})$.

Then $y = 1 \Leftrightarrow$ no more than 2 bits of $x_1, ..., x_5$ are $0 \Leftrightarrow$ $MAJ(x1, x2, x3, x4, x5)$ is true.

The complexity is counted as 11 operations and, importantly, 35 total arguments of those operations.

The second idea: we can find a program of slightly lower complexity. Consider the Boolean circuit diagram in the following. This program has 10 operations and only 24 applications to arguments.

To see that it is correct, note that w_4 is true if and only if $x_1 = x_2 = x_3 = 1$, and w_5 is true if and only if $x_4 = x_5 = 1$ and one of x_1, x_2, x_3 is 1. Finally, t is true if and only if two of x_1, x_2, x_3 and one of x_4, x_5 are true, which handles the remaining six true cases.



Problem: Is MAJ feasible?

For a string x of length n, the function $MAJ(x)$ returns 1 if more than $n/2$ of the bits of x are 1.

We ask the important question: **Is MAJ feasible?**

The first idea is failure. Because when generalized from "5" to "n", it needs to take the AND of every r-sized subset of [n], where $r = \lfloor n/2 \rfloor + 1$, and and feed that to an OR. However, there are $\binom{n}{r}$ such subsets, which is exponential when $r \sim n/2$.

The trouble shorter program with the second idea is its being rather *ad hoc* for $n = 5$. The question whether or not there are programs like it with only AND and OR gates that scale for all n is a famous historical problem in complexity theory.

The answer is known to be yes, but no convenient recipe for constructing the programs for each n is known, and their size $O(n^{5.3})$ is comparatively high.

4.3. Quantum Representation of Boolean Arguments

Let $N = 2^n$. Every coordinate in N-dimensional Hilbert space corresponds to a binary string of length n. The standard encoding scheme assigns to each index $j \in [0, ..., n-1]$ the n-bit binary string that denotes j in binary notation, with leading $0'$s if necessary.

This produces the **standard lexicographic ordering on strings**. For instance, with $n = 2$ and $N = 4$, we show the indexing applied to a permutation matrix:

$$\begin{array}{c|cccc|}
 & 00 & 01 & 10 & 11 \\
\hline
00 & 1 & 0 & 0 & 0 \\
01 & 0 & 1 & 0 & 0 \\
10 & 0 & 0 & 0 & 1 \\
11 & 0 & 0 & 1 & 0 \\
\hline
\end{array} = M$$

The matrix M defines a mapping f:

$$f(u_1u_2) = v_1v_2 \Leftrightarrow M[u_1u_2, v_1v_2] = 1.$$

$$f(00) = 00, f(01) = 01, f(10) = 11, f(11) = 10$$

and in general $f(x_1, x_2) = (x_1, x_1 \oplus x_2)$.

The operator writes the XOR into the second bit while leaving the first the same.

We can also say that **it negates the second bit if-and-only-if the first bit is** 1. This negation itself is represented on one bit by a matrix:

$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array}, \quad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The negation is controlled by the first bit, the name "Controlled-NOT" (CNOT) for the whole 4×4 operation.

To get a general Boolean function $y = f(x_1, ..., x_n)$, we need $n + 1$ Boolean coordinates, which entails $2N = 2^{n+1}$ matrix coordinates.

What we really compute is the function

$$F(x_1, ..., x_n, z) = (x_1, ..., x_n, z \oplus f(x_1, ..., x_n)).$$

Formally, F is a Boolean function with outputs in $\{0, 1\}^{n+1}$ rather than just $\{0, 1\}$.

Its first virtue, which is necessary to the underlying quantum physics, is that it is **invertible**. In fact, F is its own inverse:

$$\begin{aligned} F(F(x_1, ..., x_n, z)) &= F(x_1, ..., x_n, z \oplus y) \\ &= (x_1, ..., x_n, (z \oplus y) \oplus y) \\ &= (x_1, ..., x_n, z). \end{aligned}$$

Its second virtue is having a $2N \times 2N$ permutation matrix P_f that is easy to describe: the unique one "1" in each row $x_1 x_2 \cdots x_n z$ is in column $x_1 x_2 \cdots x_n b$, where $b = z \oplus f(x_1, \dots, x_n)$.

If f is a Boolean function with m outputs $(y_1, \dots, y_m)$ **rather than a single bit**, then we have the same idea with

$$F(x_1, \dots, x_n, z_1, \dots, z_m) = (x_1, \dots, x_n, z_1 \oplus y_1, \dots, z_m \oplus y_m)$$

The matrix P_f is still a permutation matrix, although of even larger dimensions $2^{n+m} \times 2^{n+m}$.

Often left unsaid is what happens if we need h-many "helper bits" to compute the original f.

The simple answer is that we can treat them all as extra outputs of the function, allocating extra z_j variables as **dummy inputs** so that the $\oplus$ **trick preserves invertibility**. Because h is generally polynomial in n, this does not upset feasibility.

In the above scheme, we gave for classical computation, everything is laid out in $n' = n + m + h$ rows, with the input x laid out in the first column.

Each row is said to represent a qubit. In order to distinguish the row from the idea of a qubit as a **physically observable object, we often prefer to say qubit line for the row itself in the circuit**. The h-many helper rows even have their own fancy name as **ancilla qubits**, or **helper**.

A big $2^{n'} \times 2^{n'}$ matrix, just for a permutation, is of course not feasible. This is a chief reason we prefer to think of operators P_f as pieces of code.

The qubit lines are really coordinates of binary strings that represent indices to these programs. These strings have size n', and their own indices $1, ..., n'$ are what we call **quantum coordinates**.

4.4. Quantum Feasibility

In the above scheme, we confine ourselves to linear algebra operations that are efficiently expressible via these n' quantum indices, we can hope to keep things feasible.

A quantum algorithm applies a series of unitary matrices to its start vector.

Can we apply any unitary matrix we wish? The answer is no.

Of course, **if the quantum algorithms are to be efficient, then there must be a restriction on the matrices allowed.**

If we look at the matrices P_f in section 4.3, we see several issues.

(1) The design of P_f seems to take no heed of the complexity of the Boolean function f but merely creates a permutation out of its exponential-sized truth table. **Because infeasible (families of) Boolean functions exist, there is no way this alone could scale.**

(2) Even for simple functions like $AND(x_1, x_2, \cdots, x_n)$, the matrix still has to be huge-even larger than 2^n on the side.

How do we distinguish **"basic feasible operations"**?

(3) What do we use for variables? If we have a 2^n-sized vector, do we need exponentially many variables?

The answer is to note that if we keep the number k of arguments for any operation to a **(bounded) constant**, then 2^k stays constant. We can therefore use $2^k \times 2^k$ matrices that apply to just a few arguments.

But what are the arguments? They are not the same as the Hilbert space coordinates $0, ..., N-1$, which would involve us in exponentially many.

The quantum coordinates start off being labeled $x_1, x_2, \ldots, x_n$ as for Boolean input strings and extend to places for outputs and for **ancillae (helpers)**.

With these differences understood, **the notion of feasible for unitary matrices is the natural extension of the one for Boolean circuits**.

Any unitary matrix B of dimension 2^k where k is constant, we will have that $k \leq 3$ is feasible.

Such a matrix is allowed to operate on any subset of k quantum coordinates, provided it leaves the other $n' - k$ coordinates alone.

A tensor product of B with identity matrices on the other quantum coordinates is a **basic matrix**. We could require that the entires of B be simple in some way, but it will suffice to take B from a small fixed finite family of gates.

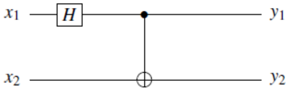
Let U be any unitary matrix of dimension N, we will say that it is feasible provided there is a way to construct it easily out of basic matrices.

To show concreteness, **one can express U via a quantum circuit of basic gate matrices**.

Quantum circuits

We will stay informal with quantum circuits as we did for Boolean circuits while formalizing quantum computations in terms of matrices.

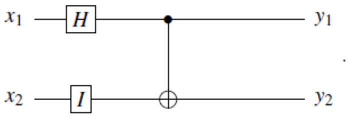
The first n lines correspond to the inputs $x_1, \ldots, x_n$, while all other qubit lines are conventionally initialized to 0. The only “crossing wires” are parts of multi-ary gates, either running invisibly inside boxes or shown explicitly for some gates like the *CNOT* operation.



In above figure, the circuit is composed of one Hadamard gate on qubit line 1, followed by a CNOT with its control on line 1 and its target on line 2:

Underneath the Hadamard gate is an invisible identity gate, expressing that in the first time step, the second qubit does not change.

We could draw this into the circuit if we wish:



Whenever two gates can be placed vertically this way, a tensor product is involved.

Thus, the matrix form of the computation is the composition $V_2 \cdot V_1$ of the 4×4 matrices.

$$V_2 = CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$V_1 = H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}.$$

$$U = V_2 \cdot V_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix}$$

$U = V_2 \cdot V_1$ acts on the input vector $x = [1,0,0,0]^T$, we obtain

$$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

The input vector denotes the input string 00, i.e. e_{00} or $x_1 = 0, x_2 = 0$.

The output vector is not a basis vector. It is an equal-weighted sum of the basis vector for 00 and 11.

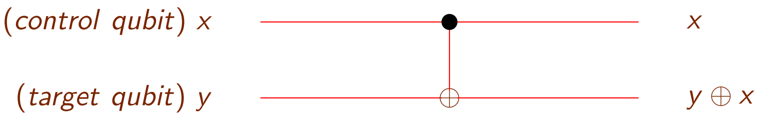
This means we do not have a unique output string y in quantum coordinates, although we have a simple output vector v in the $N = 2^2 = 4$ Hilbert-space coordinates.

Much of the power of quantum algorithms will come from such outputs v, from which we need further interaction in the form of measurements, even repeatedly, to arrive at a final Boolean output y.

DEFINITION 4.2. A quantum computation C on s qubits is feasible provided

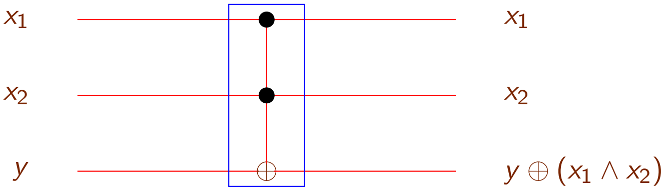
$$C = U_t U_{t-1} \cdots U_1$$

where each U_i is a feasible operation, and s and t are bounded by a polynomial in the designated number n of input qubits.



CNOT

$$(x, y) \to (x, x \oplus y);$$

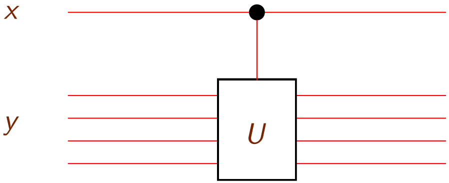


CCNOT

$$(x_1, x_2, y) \to (x_1, x_2, (x_1 \wedge x_2) \oplus y);$$

Controlled unitary matrices

CU :

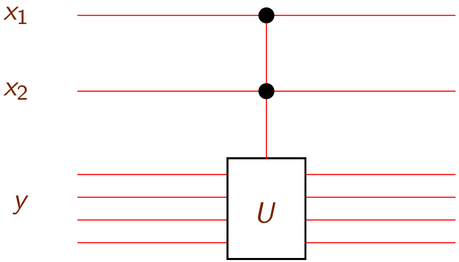


If x then U else I; $\quad U' = \begin{bmatrix} I_{2^k} & \\ & U_{2^k} \end{bmatrix}$

If $x = 1$ then Uy else Iy; $\quad CU|x\rangle|y\rangle = |x\rangle|Uy\rangle$;

Controlled unitary matrices

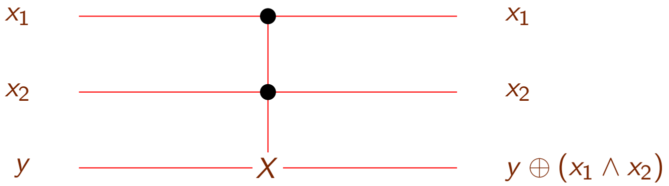
$CCU = C^2U:$



If $(x = 1) \wedge (x_2 = 1)$ then Uy else Iy;

$CU\big|x_1x_2\big\rangle\big|y\big\rangle = \big|x_1x_2\big\rangle\big|Uy\big\rangle$;

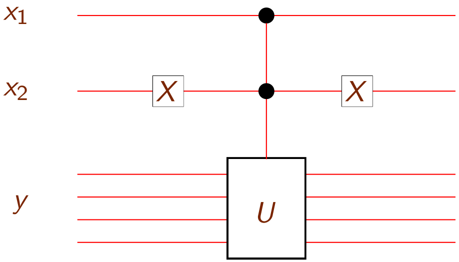
$CCX = CCNOT = TOF$: (Toffoli gate)



$$CCNOT \; e_{x_1 x_2 y} = \begin{cases} e_{x_1 x_2 y} & if \;\; x_1 = x_2 = 1 \\ e_{x_1 x_2 (1 \oplus y)} & o.w. \end{cases}$$

Controlled unitary matrices

$C\overline{C}U = C^{(1,0)}U$:



If $(x = 1) \wedge (x_2 = 0)$ then Uy else Iy;

$$C^{(1,0)}U\big|x_1x_2\big\rangle\big|y\big\rangle = \big|x_1x_2\big\rangle\big|U^{(1,0)}y\big\rangle$$

$\Leftrightarrow$ if $(x_1 = 1) \wedge (x_2 = 0)$ then $\big|x_1x_2\big\rangle\big|Uy\big\rangle$ else $\big|x_1x_2\big\rangle\big|y\big\rangle$.

Outline

5. Special Matrices

Given our view that quantum algorithms are simply the result of applying a unitary transformation to a unit vector.

There are just a few families of such matrices that are used in most quantum algorithms.

Two of the families correspond to transforms that are well studied through mathematics and computer science theory and have many applications in many areas besides quantum algorithms.

(1) Hadamard Matrices; **(2) Fourier Matrices**

When is a transformation a transform? The latter term connotes that the output is a new way of interpreting the input.

All quantum transformations are invertible :

$$y = Ux, x = U^* y$$

5.1. Hadamard Matrices

The first family of unitary transforms are the famous Hadamard matrices. Note that because we mainly stay with the standard basis of e_k vectors, we will identify transforms with their matrices, and this should cause no confusion. Here we lock in our convention that N is always 2^n for some n.

DEFINITION 5.1. The Hadamard matrix H_N of order N is recursively defined by $H_2 = H$ and for $N \geq 4$:

$$H_N = H \otimes H_{N/2} = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{N/2} & H_{N/2} \\ H_{N/2} & -H_{N/2} \end{bmatrix}$$

We could also use $H_1 = [1]$ as the basis, and write $H^{\otimes n} = H_{2^n}$.

LEMMA 5.2. For any row r and column c,

$$H_N[r, c] = (-1)^{r \cdot c},$$

recalling that $r \cdot c$ is the inner product of r and c treated as Boolean strings. An example:

	00	01	10	11
00	1	1	1	1
01	1	−1	1	−1
10	1	1	−1	−1
11	1	−1	−1	1

For any vector a, the vector $b = H_N a$ is defined by

$$b(x) = \sum_{t=0}^{N-1} (-1)^{x \cdot t} a(t).$$

where x, t are treated as Boolean strings.

This is the way that we will view the transform in the analysis of most algorithms. In a quantum circuit with n qubit lines, H_N is shown as a column of n-many single-qubit Hadamard gates.

Hadamard matrix and Mean, Parallel,...

$$\frac{1}{\sqrt{8}}\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{8}}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

5.2. Fourier Matrices

The second family of unitary transforms are Fourier Matrices.

DEFINITION 5.3. Let $\omega = e^{2\pi i/N}$ be"N-th root of unity". The Fourier matrix F_N of order N is:

$$\frac{1}{\sqrt{N}}\begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{(N-2)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{N-3)} \\ \vdots & \vdots & & & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{N-2} & \omega^{N-3} & \cdots & \omega \end{bmatrix}$$

Clearly, $F_N[i,j] = \omega^{ij (mod\ N)}$.

It is well known that F_N is a unitary matrix over the complex Hilbert space.

For any vector a, the vector $b = F_N a$ is defined by

$$b(x) = \frac{1}{\sqrt{N}} \sum_{t=0}^{N-1} \omega^{xt (mod\ N)} a(t).$$

Note that: $\sum_{t=0}^{N-1} \omega^{xt} = 0$ for $x = 1, 2, \cdots, N-1$. (by $\omega^N = 1$)

Fourier Matrices and Period, Parallel,...

$$\begin{array}{cc}
 & \frac{1}{\sqrt{N}}\begin{bmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\
1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{(N-2)} \\
1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{N-3)} \\
\vdots & \vdots & & & \ddots & \vdots \\
1 & \omega^{N-1} & \omega^{N-2} & \omega^{N-3} & \cdots & \omega
\end{bmatrix} \\
\text{Period: } r = & \begin{matrix} 0 & 1 & 2 & 3 & \cdots & N-1 \end{matrix}
\end{array}$$

Shor's Algorithm: (Finding a period of the function F_N)

$$F_N(a) = y^a \ mod \ N \ (0 < y < N, gcd(y, N) = 1)$$

5.3. Reversible Computation and Permutation Matrices

Every $N \times N$ permutation matrix is unitary. However, in terms of n with $N = 2^n$, **not all of them can be feasible.**

The problem is that which permutation matrices are feasible?

Recall the definition of the permutation matrix P_f from the invertible extension F of a Boolean function f, where

$$F(x_1, \cdots, x_n, z) = (x_1, \cdots, x_n, z \oplus f(x_1, \cdots, x_n)).$$

THEOREM 5.4. All classically feasible Boolean functions f have feasible quantum computations in the form of P_f.

The proof of this theorem stays entirely classical–that is, the quantum circuits are the same as Boolean circuits that are reversible, which in turn efficiently embed any given Boolean circuit computing f.

We need only one new gate—**Toffoli gate, TOF**, which is the 8×8 matrix of P_f where f is the binary AND function.

Toffoli gate

The Toffoli gate is the ternary Boolean function

$$TOF(x_1, x_2, x_3) = (x_1, x_2, x_3 \oplus (x_1 \wedge x_2)).$$

The Toffoli gate induces the permutation in 8-dimensional Hilbert space that **swaps the last two entries**, which correspond to the strings 110 and 111, and leaves the rest the same.

	000	001	010	011	100	101	110	111
000	1	0	0	0	0	0	0	0
001	0	1	0	0	0	0	0	0
010	0	0	1	0	0	0	0	0
011	0	0	0	1	0	0	0	0
100	0	0	0	0	1	0	0	0
101	0	0	0	0	0	1	0	0
110	0	0	0	0	0	0	0	1
111	0	0	0	0	0	0	1	0

x_1	x_2	x_3	x_3'
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

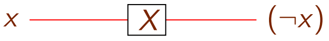
This extends the idea of CNOT with x_1, x_2 as “controls” and x_3 as the “target”. Clearly,

$$TOF = CCNOT = CCX.$$

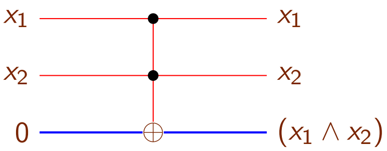
That this simple swap is universal for Boolean computation is conveyed by the following two facts for Boolean **bit arguments** a, b: $NOT(a) = TOF(1, 1, a)$; $AND(a, b) = TOF(a, b, 0)$.

Transform a Boolean circuit into a quantum circuit:

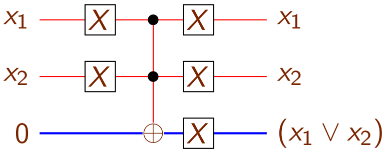
NOT :



AND :



OR :



Proof of Theorem 5.4.

Because *AND* and *NOT* is a universal set of logic gates, we may start with a Boolean circuit C computing $f(x_1, ..., x_n)$ using r-many *NOT* and s-many binary *AND* gates.

The *NOT* gates we can leave alone because we already have the corresponding 2×2 matrix X as a basic quantum operation.

Hence, we need only handle the s-many *AND* gates.

A simple idea is that we can simulate them by s-many Toffoli gates each with **an common ancilla line** set to 0 for input, but this is superseded by the issue of possibly needing multiple copies of the result c of an *AND* gate on lines a, b—that is, one for each wire out of the gate.

This is where the Toffoli gate shines. For each output wire w, we allocate a fresh ancilla z and put a Toffoli gate with target on line z and controls on a and b. This automatically computes $z \oplus (a \wedge b)$, which with z initialized to 0 is what we want. Multiple Toffoli gates with the same controls do not affect each other.

Hence, the overhead is bounded by the number of wires in C, which is polynomial, and the only ancilla lines we need already obey the convention of being initialized to 0. $\square$

The "polynomial" property is good enough for our present discussion of feasibility. Thus, a permutation matrix — which is a deterministic quantum operation –is feasible if it is induced by a classical feasible function on the quantum coordinates.

5.4. Feasible Diagonal Matrices

Any diagonal matrix whose entries have absolute value 1 is unitary. Hence, it can be a quantum operation.

Similarly, which of these operations are feasible?

Of course, if the size of the matrix is a small fixed number, e.g. 2, we can call it basic and hence feasible.

But, even if we limit to entries 1 and -1, we have one such matrix U_S for every subset S of $[N]$, that is, $S \subset \{0,1\}^n$:

$$U_S[x,x] = \begin{cases} 1 & \textit{if } x \in S; \\ -1 & \textit{otherwise.} \end{cases}$$

Because there are doubly exponentially many S, there are doubly exponentially many U_S, so most of them are not feasible.

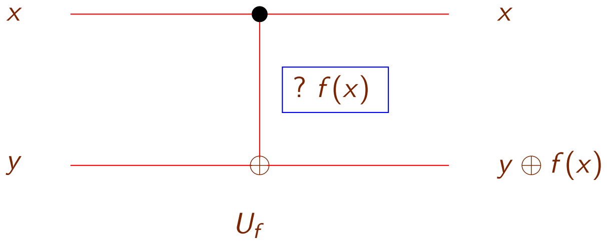
Can we tell which are feasible?

We give a partial answer. When S is the set of arguments that make a Boolean function f true, i.e.,

$$f(x_1, \cdots, x_n) = 1 \Leftrightarrow (x_1, \cdots, x_n) \in S$$

We write U_f in place of U_S.

The matrix U_f is called the Grover oracle for f.



$$U_f(|x\rangle|y\rangle) = |x\rangle|y \oplus f(x)\rangle$$

THEOREM 5.6. If f is a feasible Boolean function, then its Grover oracle U_f is feasible.

We defer the proof until section 6.5 in the next chapter.

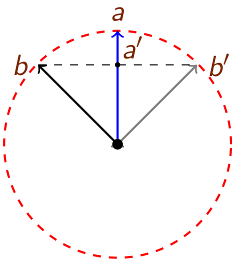
The question of whether any other families of functions f make U_f meet our quantum definition of feasible is a deep one whose answer is long unknown and is related to issues in chapter 16.

We can be satisfied for now that we have a rich vocabulary of feasible operations, and the next chapter will give some tricks for combining them. Here we give one more family of operations.

5.5. Reflections

Given any unit vector a, we can create the unitary operator Ref_a, which **reflects** any other unit vector b around a.

Geometrically, this is done by dropping a line from the tip of b that hits the body of a in a right angle and continuing the line the same distance further to a point b'. Then b' likewise lies on the unit sphere of the Hilbert space.



The operation mapping b to b' preserves the unit sphere and is its own inverse, so it is unitary.

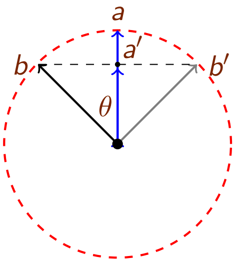
In geometrical terms, the point on the body of a is the projection of b onto a and is given by $a' = a\langle a, b\rangle = a \cdot cos(\theta)$.

Thus,

$$b' = b - 2(b - a') = b - 2(b - a\langle a, b\rangle) = (2P_a - I)b,$$

where $P_a (= |a\rangle\langle a|)$ is the operator doing the projection: for all b,

$$P_a b = a' = a\langle a, b\rangle = a \cdot cos(\theta).$$



For example, let a be the unit vector with entries $\frac{1}{\sqrt{N}}$, which we call j. Then the projector is the matrix whose entries are all $\frac{1}{N}$, which we call J in our matrix font. Finally, the reflection operator is

$$V = 2J - I = \begin{bmatrix} \frac{2}{N} - 1 & \frac{2}{N} & \frac{2}{N} & \cdots & \frac{2}{N} \\ \frac{2}{N} & \frac{2}{N} - 1 & \frac{2}{N} & \cdots & \frac{2}{N} \\ \vdots & \vdots & & \ddots & \vdots \\ \frac{2}{N} & \frac{2}{N} & \cdots & \frac{2}{N} & \frac{2}{N} - 1 \end{bmatrix}$$

We claim that this matrix is feasible.

Of course this leads to the question: which reflection operations are feasible?

$$a = ?$$

An important case of reflection is when a is the characteristic vector of a nonempty set S, that is:

$$a(x) = \begin{cases} \frac{1}{\sqrt{|S|}} & \textit{if } x \in S; \\ 0 & \textit{otherwise}. \end{cases}$$

Suppose we apply Ref_a to vectors b with the foreknowledge that all entries $c = b(x)$ for $x \in S$ are equal. Let $k = |S|$. Then we have $\langle a, b \rangle = kc/\sqrt{k} = c\sqrt{k}$, and taking the projection $a' = P_a b$, we have

$$a'(x) = \begin{cases} c & \textit{if } x \in S; \\ 0 & \textit{otherwise}. \end{cases}$$

The reflection $b' = 2a' - b$ thus satisfies

$$b'(x) = \begin{cases} b(x) & \textit{if } x \in S; \\ -b(x) & \textit{otherwise}. \end{cases}$$

In the case $x \in S, b'(x) = 2c - b(x) = 2c - c = c = b(x)$.

Thus the action is the same as multiplying by the diagonal matrix that has -1 for the coordinates that are not in S, that is, by the Grover oracle for the complement of S.

Because the negation of a feasible Boolean function is feasible, this together with the case of V implies:

THEOREM 5.7. For all feasible Boolean functions f, provided we restrict to the linear subspace of argument vectors whose entries indexed by the "true set" S_f of f are equal, reflection about the characteristic vector of S_f is a feasible quantum operation.

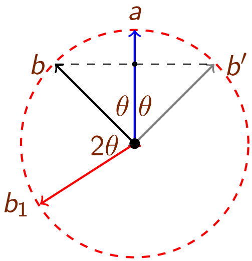
Happily, the set of such argument vectors forms a linear subspace and always contains the vector j, which we will use as a "start" vector.

Moreover, reflections by a and b, when applied to vectors already in the linear subspace $Span(a, b)$ spanned by a and b, stay within that subspace.

We will use this when presenting Grover's algorithm and search by quantum random walks.

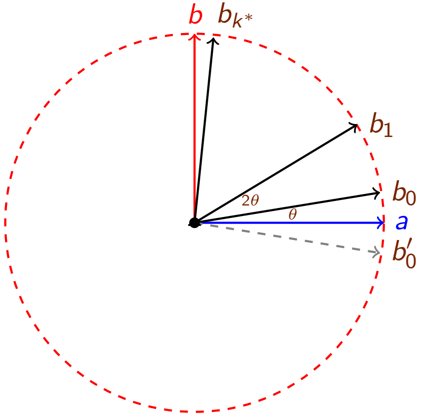
5.6. A application of reflections: Grover's search algorithm

$$U = 2P_a - I = 2|a\rangle\langle a| - I$$
$$b' = (2P_a - I)b,$$



$$b_1 = (2P_b - I)b' = (2P_b - I)(2P_a - I)b.$$

The basic idea of Grover's search algorithm



$b_1 = (2P_{b_0} - I)b_0' = (2P_{b_0} - I)(2P_a - I)b_0 = Gb_0$.

$b_{k+1} = (2P_{b_0} - I)b_k = (2P_{b_0} - I)(2P_a - I)b_k = G^k b_0$.

$Gb_0 = cos(3\theta)a + sin(3\theta)b$, $\theta_0 = \theta, \theta_1 = 3\theta, \theta_k = (2k+1)\theta$.

Take $k^* = max\{k : \ (2k \pm 1)\theta \leq \frac{\pi}{2}\}$.

Outline

6. Tricks

6.1. Start Vectors

A quantum algorithm needs to start on a simple vector.

We usually restrict algorithms to start in a simple state. It may assume that all memory locations are set to zero. The simplest start state possible is $e_0 = [1, 0, 0, ..., 0]$. But there are exceptions, e.g. $e_1 = [0, 1, 0, 0]$. This is also an elementary vector, it is reasonable to allow this as the initial state.

To show how to move from e_0 to e_1 , the idea is that with respect to the indexing scheme, 0 corresponds to the string 0^n and 1 to $0^{n-1}1$ **(local changing)**. We can invert the last bit, which entails swapping e_0 and e_1, is accomplished by $I^{\otimes(n-1)} \otimes X$:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

It is just the linear algebraic way of applying a NOT gate to the last string index.

We can do this on the r-th bit from the right, inducing permutations of $[N]$ that move indices up or down by 2^r.

Note that we have not transposed **only** e_0 **and** e_1; we must be aware of other effects on the Hilbert space.

Interchanging e_1 and e_2 involves a different operation. In string indices, we need to swap $...01$ with $...10$. This is **not totally local as it involves two indices**. We need to tensor the 4×4 swap matrix,

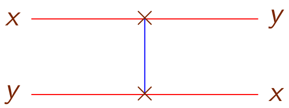
$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

after $I^{\otimes(n-2)}$, i.e. $I^{\otimes(n-2)} \otimes SWAP$.

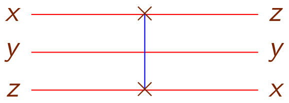
This can be regarded as a invertible function $f(a,b) = (b,a)$.

$SWAP$:

	00	01	10	11
00	1	0	0	0
01	0	0	1	0
10	0	1	0	0
11	0	0	0	1



$SWAP_{1,3}$:



$$SWAP_{1,3} = \begin{array}{c|cccccccc} & 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\ \hline 000 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 001 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 010 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 011 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 100 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 101 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 110 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 111 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

$$SWAP_{1,3} = (SWAP \otimes I_2)(I_2 \otimes SWAP)(SWAP \otimes I_2)$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$$

Another interesting start vector j is the sum of all the e_k, which must be divided by $\sqrt{N}$ to keep it a unit vector.

$$j_N = \tfrac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e_k.$$

We can obtain this from e_0 and the Hadamard matrix $H_N = H^{\otimes n}$

$$j_N = H_N e_0.$$

Again by the tensor product feature, this operation is local to each individual string index.

In terms of strings, it creates a weighted sum over all of $\{0,1\}^n$.

$$H_4 e_0 = H_4 e_{00} = \tfrac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \tfrac{1}{2} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

If we apply this to any other e_k, then we get a vector with some -1 entries in place of $+1$.

$$H_4 e_2 = H_4 e_{10} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 1 \\ 1 \\ -1 \\ -1 \end{pmatrix}$$

Finally, we may wish to extend our start vectors to initialize helper bits.

Generally, this means extending the underlying binary string with some number m of 0s. In that case, because we already regard e_0 as our generic start vector, we need do nothing.

Algebraically what we are doing is working in the product Hilbert space $H_N \otimes H_M$, with $M = 2^m$, because e_0 in the product space is just the tensor product of the first basis vectors of the two spaces.

If we want to change any state a to $a \otimes e_0$, then we may suppose the extra helper bits were there all along.

6.2. Controlling and Copying Base States

Can we change any state a to $a \otimes a$? (Copying a)

Algebraically, the latter means the state b such that indexing by strings $x, y \in \{0,1\}^n$, we have

$$b(xy) = a(x)a(y).$$

No-cloning theorem: There is no $2^{2n} \times 2^{2n}$ unitary operation U such that **for all** a,

$$U(a \otimes e_0) = a \otimes a.$$

However, a limited kind of copying is possible that can replicate computations and help to amplify the success probability of algorithms after taking measurements.

Proof of no-cloning theorem: Suppose that there is a unitary matrix U such that for any vector a, $U(a \otimes e_0) = a \otimes a$.

By taking $a = e_0$ and $a = e_1$ respectively, we have

$$\begin{aligned} U(e_0 \otimes e_0) &= e_0 \otimes e_0, \\ U(e_1 \otimes e_0) &= e_1 \otimes e_1 \end{aligned}$$

Let $a = \frac{1}{\sqrt{2}}(e_0 + e_1)$ again. Then, by linearity of U

$$\begin{aligned} U(a \otimes e_0) &= \frac{1}{\sqrt{2}} U(e_0 + e_1) \otimes e_0 \\ &= \frac{1}{\sqrt{2}} (U(e_0 \otimes e_0) + U(e_1 \otimes e_0)) \ . \\ &= \frac{1}{\sqrt{2}} (e_0 \otimes e_0 + e_1 \otimes e_1) \end{aligned}$$

But, by the copying property of U, we have

$$\begin{aligned} U(a \otimes e_0) &= \frac{1}{2} (e_0 + e_1) \otimes (e_0 + e_1) \\ &= \frac{1}{2} (e_0 \otimes e_0 + e_0 \otimes e_1 + e_1 \otimes e_0 + e_1 \otimes e_1) \ . \\ &\neq \frac{1}{\sqrt{2}} (e_0 \otimes e_0 + e_1 \otimes e_1) \end{aligned}$$

It is a contradiction. $\Box$

THEOREM 6.1. For any $n \geq 1$, we can efficiently build a $2^{2n} \otimes 2^{2n}$ unitary operation C_n that converts any vector a' into b such that for all $x, y \in \{0,1\}^n$,

$$b(xy) = a'(x(x \oplus y)).$$

In particular, if $a' = a \otimes e_{0^n}$, then we get for all x,

$$a(x) = a'(x0^n) = b(xx),$$

so that measuring b yields xx with the same probability that measuring a yields x.

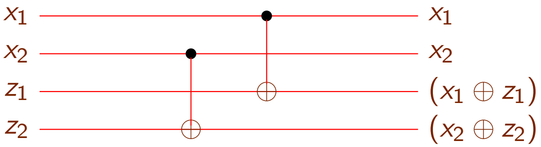
Proof. Consider $n = 1$. The operator must make $b(00) = a'(00)$, $b(01) = a'(01), b(10) = a'(11)$, and $b(11) = a'(10)$.

This is done by the 4×4 permutation matrix

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

For $n = 2$, the indices are length-4 strings $y_1 y_2 z_1 z_2$, which are permuted into $y_1 y_2 (y_1 \oplus z_1)(y_2 \oplus z_2)$.

This is a composition of two $CNOT$ operations, one on the first and third indices (preserving the others), which we denote by $C_{1,3}$ and the other on the second and fourth, written as $C_{2,4}$.



For general n, the final operator is the composition

$$C_n = C_{1,n+1} C_{2,n+2} \cdots C_{n,2n}. \square$$

$C_{1,3} = C'_{1,3} \otimes I_2$:

$$C'_{1,3} = \begin{array}{c|cccccccc} & 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \\ \hline 000 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 001 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 010 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 011 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 100 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 101 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 110 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 111 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Similarly, we can define a 8×8 matrix $C'_{2,4}$, and then get $C_{2,4} = I_2 \otimes C'_{2,4}$.

Magically, what this does is clone every basis state at once. If $a = e_x$, then b is the same as $a \otimes a$ after all.

An example of why this doesn't violate the no-cloning theorem is that when a is a non-basis state, such as $\frac{1}{\sqrt{2}}(e_x + e_y)$, $a \otimes a$ is generally not the same as $\frac{1}{\sqrt{2}}(e_{xx} + e_{yy})$.

We can now do various things.

We can run two operations U_f computing a function $f(x)$ on x side by side. Or we can do just one, applying $I^{\otimes n} \otimes U_f$ to e_{xx} to get $e_{xf(x)}$.

Essentially, we are using two Hilbert spaces that we put together by a product. We can also arrive at this kind of state in the manner shown next.

6.3. The Copy-Uncompute Trick

Suppose that we wish to compute $f : \{0,1\}^n \to \{0,1\}^m$, where $m < n$. Such function f is not invertible, so we cannot expect to map an input state e_x to a quantum state that uniquely corresponds to y.

We have already seen the idea of replacing f by the function

$$F(x, v) = (x, v \oplus f(x)).$$

Then $F : \{0,1\}^{n+m} \to \{0,1\}^{n+m}$ is a bijection, and the original function f is recoverable via $F(x, 0^m) = (x, f(x))$.

Now suppose we have any quantum operation U on the "x" part, where $f(x)$ might be embedded as a substring in m indexed places **in the first n indices**.

We can automatically obtain the corresponding $F(x, 0^m)$ via the computation:

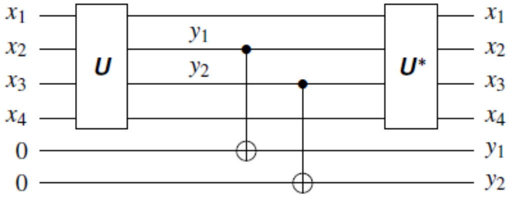
$$(U^* \otimes I_m) C_m (U \otimes I_m)(e_x \otimes e_{0^m}),$$

where the C_m is applied to **those** m **index places in the first** n **indices** and to m **ancilla** places.

This effectively lifts out and copies $f(x)$ into **the last** m **fresh places**.

The final U^* then inverts what U did in the first n places, "cleaning up" and leaving x again.

Here is a diagram for $n = 4$ and $m = 2$, where the values $f(x) = y_1 y_2 \in \{0,1\}^2$ are computed on the second and third wires and then copied to the ancillae:

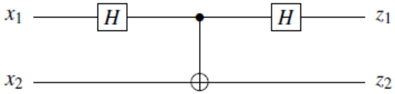


This trick is called **copy-uncompute or compute-uncompute**.

It is important to note that it works only when the quantum state after applying U and before C_m is a superposition of **only those basis states** that have $f(x)$ in the set of quantum coordinates to which the controls are applied. **If there is any disagreement there in the superposition, then the results can be different.**

This again is why the trick does not violate the no-cloning theorem.

For a simple concrete example, consider the following quantum circuit, noting that the Hadamard matrix is its own inverse, i.e., is self-adjoint:



On input e_{00}, that is, $x_1 = x_2 = 0$, the first Hadamard gate gives the control qubit a value that is a superposition. Hence, the second Hadamard gate does not "uncompute" the first Hadamard to restore $z_1 = 0$.

The action can be worked out by the following matrix multiplication (with an initial factor of $\frac{1}{2}$):

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 & -1 \\ 1 & 1 & -1 & 1 \\ 1 & -1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix}$$

This maps e_{00} to $\frac{1}{2}[1, 1, 1, -1]^T$, thus giving equal probability to getting 0 or 1 on the first qubit line.

However, if U includes a preamble transforming e_0 to e_x and then leaves a definite value y on the controlled lines before the rest of the circuit does U^*, then the computation does end with the first n places again zeroed out, i.e., in some state $f' = e_{0^n} \otimes e_y$.

This finally justifies why we can regard e_0 as the only input we need to consider.

It emphasizes the goal of efficiently preparing a state from which a desired value $f(x)$ can be recovered by measurement.

As long as we are careful to represent the linear algebra correctly, we will not be confused between these two eventualities. Then we can do more tricks with superpositions and controls.

6.4. Superposition Tricks

Recall the vector j_N and unitary matrix C_n, $C_n(a \otimes e_0) = a \otimes a$.

For the case $n = 2, N = 4$ is $\frac{1}{2}[1,1,1,1]^T$, we feed it on the first n of 2^n quantum coordinates and **following it with controls** y gives the following state:

$$(C_n(j_N \otimes e_0))(xy) = \begin{cases} \frac{1}{\sqrt{N}} & if\ y = x \\ 0 & o.w. \end{cases}$$

Furthermore, these ideas show that we can construct a vector b such that

$$b(xy) = \begin{cases} \frac{1}{\sqrt{N}} & if\ y = f(x) \\ 0 & o.w. \end{cases}$$

Here xy is just the concatenation of the strings x and y.

Moreover, by the last section, we can obtain a version of b even when y is just a single bit. In either case, we can also write

$$b = \tfrac{1}{\sqrt{N}} \textstyle\sum_{x\in\{0,1\}^n} (e_x \otimes e_{f(x)}).$$

The Dirac notation for this state is $b = \frac{1}{\sqrt{N}} \sum_{x\in\{0,1\}^n} \big|x\big\rangle \big|f(x)\big\rangle$.

DEFINITION 6.2. Given $f : \{0,1\}^n \to \{0,1\}^m$, the state

$$s_f = \tfrac{1}{\sqrt{2^n}} \textstyle\sum_{x\in\{0,1\}^n} \big|x\big\rangle \big|f(x)\big\rangle \ (= H_{2^n} \big|0^n\big\rangle \big|f(x)\big\rangle)$$

is called the **functional superposition of f**.

Note: $s_f \in \{0,1\}^{n+m}$.

We can also extend the conditional idea of C_n directly to any given quantum operation U. Define CU by

$$((CU)a)(0x) = a(x); ((CU)a)(1x) = (Ua)(x).$$

We have used extra parentheses to make clear that CU is a name, not the composition of matrices called C and U, and it is read **"Control-U"**.

Our $CNOT$ operation did this to our matrix X of the unitary NOT operation, which explains the name. We can also iterate this, for instance, to do $CCNOT$. This yields our friend the Toffoli gate again.

6.5. Flipping a Switch

A classical problem is that how many X-es does it take to change a light bulb?

In quantum computation, everything is reversible, and that applies to such jokes as well: **If you change a light bulb, how many X-es can you affect?**

The answer is: as many as you like.

Our light bulb can be the $(n+1)$st qubit, call it y. Suppose we multiply it by a unit complex number a, such as -1. It may seem that we are only flipping the sign of the last qubit, and we might even wrongly picture the $(n+1) \times (n+1)$ matrix that is the identity except for a in the bottom right corner.

The unitary matrices that are really involved, however, are $2^{n+1} \times 2^{n+1}$ acting on the Hilbert space, and by linearity, the **scalar multiplication** applies to all coordinates.

Put another way, if we start with a product state $z \otimes e_y$ and change the latter part e_y to ae_y, then the resulting tensor product is mathematically the same as $(az) \otimes e_y$, here a is a scalar.

With $a = -1$, we can interpret this as z being flipped instead. This feels strange, but both come out the same in the index-based calculations.

This becomes a great trick if we can arrange for a itself to depend on the basis elements e_x.

Given a Boolean function f with one output bit, let us return to the computation of the reversible function

$$F(x, y) = (x, (y \otimes f(x))).$$

Our quantum circuits for f have thus far initialized y to 0.

Let us instead arrange $y = 1$ and then apply a single-qubit Hadamard gate. Thus, instead of starting up with $e_x \otimes e_0$, we have $e_x \otimes d$, where d is the **“difference state”**:

$$d = (\tfrac{1}{\sqrt{2}}, \tfrac{-1}{\sqrt{2}}) = \tfrac{1}{\sqrt{2}}(e_0 - e_1)$$

Now apply the circuit computing F. By linearity we get:

$$\begin{aligned} F(x,d) &= \tfrac{1}{\sqrt{2}}(F(x0) - F(x1)) \\ &= \tfrac{1}{\sqrt{2}}(e_x \otimes e_{0\oplus f(x)} - e_x \otimes e_{1\oplus f(x)}) \\ &= \tfrac{1}{\sqrt{2}}(e_x \otimes (e_{0\oplus f(x)} - e_{1\oplus f(x)})) \\ &= e_x \otimes d' \end{aligned}$$

where

$$d' = \begin{cases} \tfrac{1}{\sqrt{2}}(e_0 - e_1) & \textit{if } f(x) = 0 \\ \tfrac{1}{\sqrt{2}}(e_1 - e_0) & \textit{if } f(x) = 1 \end{cases}$$

$$= (-1)^{f(x)} d.$$

Thus, we have flipped the last quantum coordinate by the value $a_x = (-1)^{f(x)}$.

Well actually no–by the above reasoning, what **we have equally well done is that when presented with a basis vector e_x as input, we have multiplied it by the x-dependent value a_x.**

We have involved the $(n+1)$st coordinate, but because we have obtained $a_x e_x \otimes d$, we can regard it as unchanged.

In fact, we can finish with another Hadamard and NOT gate on the last coordinate to restore it to 0. On the first n qubits, over their basis vectors e_x, what we have obtained is the action:

$$e_x \to (-1)^{f(x)} e_x.$$

This is the action of the Grover oracle.

We have thus proved theorem 5.6 in chapter 5. We can summarize this and the conclusion of section 6.4 in one theorem statement:

THEOREM 6.3. For all classically feasible (families of) functions

$$f : \{0,1\}^n \to \{0,1\}^m$$

both the mapping from e_{x0^m} to the functional superposition s_f, $e_{x0^m} \to s_f$, and the Grover oracle of f, $e_x \to (-1)^{f(x)} e_x$, are feasible quantum operations.

6.6. Measurement Tricks

There are also several tricks involving measurement.

Suppose that the final state of some quantum algorithm is a. We now plan to take a measurement that will return y with probability $|a(y)|^2$.

In some cases, we can compute this in closed form, whereas in other cases, we can approximate it well.

In other algorithms, we use the following idea: **"Everybody has to be somewhere".**

Let S be a subset of the possible indices y, and suppose that we can prove

$$\sum_{y \in S} |a(y)|^2 \geq c > 0$$

for some constant c. Then we can assert that with probability at least c a measurement will yield a good y from the set S.

The power of this trick is that **we do not have to understand the values of $a^2(z)$ for z not in the set S.**

We need only understand those in the set S. This is used in chapter 11 when we study Shor's factoring algorithm.

When the set S is the set of all indices having a "1" in a particular place, this is called **measuring one qubit**.

Note that S includes exactly half of the indices, as does its complement, which equally well defines a one-qubit measurement.

The idea can be continued to define r-qubit measurements, each of which "targets" a particular outcome string $w \in \{0,1\}^r$ and involves the particular set S_r of $N/2^r$ indices that have w in the respective places.

Principle of deferred measurement:

Theoretically, after a one-place measurement, the quantum computation can continue on the smaller Hilbert space of the remaining places. But, all algorithms we cover do their measurements at the end of quantum routines.

We cover **the principle of deferred measurement**, which often removes the need to worry about this possibility because it illustrates the above controlled-U trick.

THEOREM 6.4. If the result b of a one-place measurement is used only as the test in one or more operations of the form **"if b then U"**, then exactly the same outputs are obtained upon replacing U by the quantum controlled operation CU with control index the same as the index place being measured and measuring that place later without using the output for control.

Proof. As before, we visualize the control index being the first index, **but it can be anywhere.**

Suppose in the new circuit the result of the measurement is 0. Then the CU acted as the identity, so on the first index, the same measurement in the old circuit would yield 0, thus failing the test to apply U and so yielding the identity action on the remainder as well.

If the new circuit measures 1, then because CU does not affect the index, the old circuit measured 1 as well, and in both cases the action of U is applied on the remainder. $\square$

6.7. Partial Transforms

Another trick is applying an operator to **“part” of the space**.

Suppose that U is a unitary transform defined on vectors a in some Hilbert space. Then by definition there is a function $u(x, k)$ so that

$$(Ua)(x) = \sum_k u(x, k)a(k).$$

Suppose that we want to extend U to apply to the vector $b(xy)$.
The natural idea is to imagine that b is really many different vectors, each of the form $b(xy_0)$ for a different fixed value of y_0.
If we want the result of applying U to one of these, it should be

$$\sum_k u(x, k)b(xy_0).$$

Therefore, the result of applying U to the vector b is:

$$c(xy) = \textstyle\sum_k u(x,k)b(k,y).$$

As an example, let $a(xy)$ be a vector. We can apply the Hadamard transform just to the **first** "x" **part** as follows.

The result is

$$b(xy) = \tfrac{1}{\sqrt{N}} \textstyle\sum_t (-1)^{x \cdot t} a(ty).$$

This is what is meant by applying the Hadamard only to the "x" coordinates.

Some important operations

(1) Hadamard operations: $b = H_{2^n} \otimes I_{2^m} a$

$$b(xy) = \frac{1}{\sqrt{N}} \sum_{t \in \{0,1\}^n} (-1)^{x \cdot t} a(ty)$$

where $y \in \{0,1\}^m$.

(2) Controlled operations: $C^x U$

(where $x \in \{0,1\}^n$, $order(U) = 2^m$)

$$C^x U \; e_{zy} = \begin{cases} e_z \otimes U e_y & if \;\; z = x \\ e_{zy} = e_z \otimes e_y & o.w. \end{cases}$$

where $y \in \{0,1\}^m$.

(3) Spiltting sensor product to product of matrices:

$$U \otimes V = (U \otimes I_{2^m})(I_{2^n} \otimes V)$$

$$(U \otimes V)(a \otimes b) = (Ua) \otimes (Vb)$$

(where $order(U) = order(a) = 2^n, order(V) = order(b) = 2^m$)

Some basic operations:

(1) Hadamard operations H:

$$He_0 = \frac{1}{\sqrt{2}}(e_0 + e_1),\ \ He_1 = \frac{1}{\sqrt{2}}(e_0 - e_1)$$

$$H \otimes H = (H \otimes I)(I \otimes H)$$

$$\begin{aligned}(H \otimes H)e_{10} &= He_1 \otimes He_0 \\ &= \frac{1}{\sqrt{2}}(e_0 - e_1) \otimes \frac{1}{\sqrt{2}}(e_0 + e_1) \\ &= \frac{1}{2}(e_{00} + e_{01} - e_{10} - e_{11})\end{aligned}$$

$$\begin{aligned}(H \otimes I)e_{10} &= (He_1) \otimes Ie_0 = (He_1) \otimes e_0 \\ &= \frac{1}{\sqrt{2}}(e_0 - e_1) \otimes e_0 = \frac{1}{\sqrt{2}}(e_{00} - e_{10})\end{aligned}$$

$$\begin{aligned}(I \otimes H)\frac{1}{\sqrt{2}}(e_{00} - e_{10}) &= \frac{1}{\sqrt{2}}((I \otimes H)e_{00} - (I \otimes H)e_{10}) \\ &= \frac{1}{2}((e_{00} + e_{01}) - (e_{10} + e_{11}))\end{aligned}$$

Some basic operations:

(2) Not operations X: $Xe_0 = e_1,\ Xe_1 = e_0)$

$CXe_{00} = e_{00},\ CXe_{01} = e_{01}, CXe_{10} = e_{11},\ CXe_{11} = e_{10}.$

$CCNOTe_{000} = e_{000}, \cdots, CCNOTe_{101} = e_{101},$

$CCNOTe_{110} = e_{111},\ CCNOTe_{111} = e_{110}$

$C^{101}Xe_{0000} = e_{0000},\ C^{101}Xe_{1010} = e_{1011}, C^{101}Xe_{1011} = e_{1010}.$

(3) Z-operation: $Ze_0 = e_0,\ Ze_1 = -e_1$

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$CZe_{00} = e_{00},\ CZe_{01} = e_{01}, CZe_{10} = e_{10},\ CZe_{11} = -e_{11}.$

(3) $SWAP$-operation:

$$SWAPe_{xx} = e_{xx}, SWAPe_{10} = e_{01}, SWAPe_{01} = e_{10}$$

$$CSWAPe_{0xy} = e_{0xy}, CSWAPe_{1xx} = e_{1xx}$$

$$CSWAPe_{110} = e_{101}, CSWAPe_{101} = e_{110}$$

Similarly, we can give operations for other basic matrices:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \; T = \begin{pmatrix} 1 & 0 \\ 0 & e^{\pi i/4} \end{pmatrix} \; Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$