Rechenoperationen und Elementare Algorithmen im Quantencomputing

Michael Goerz (goerz@physik.fu-berlin.de)
FU Berlin

Lehrseminar Quantencomputer SoSe 2007
3. März 2007

Inhalt

1 Einführung

Nach dem bekannten Mooreschen Gesetz verdoppeln Computerchips alle zwei Jahre ihre Leistung, d.h., dass auch die auf ihnen zu findenden Strukturen exponentiell kleiner werden. Diesem Wachstum sind jedoch Grenzen gesetzt: Werden die Strukturen genügend klein (unter 10-20 nm), ist es bald nicht mehr möglich, mit dem Bild der klassischen Elektronik abstrakt Informationen in Schaltungen zu verarbeiten. Zwischen den mikroskopischen Bauteilen kommt es zu Quanteneffekten, die als solche für die Elektronik Störungen darstellen. Es ist absehbar, dass in naher Zukunft die quantenphysikalische Grenze des Mooreschen Gesetz erreicht wird und dass sich das exponentielle Wachstum der Computerleistung nicht mehr fortsetzen kann.

Diesem Problem lässt sich auf mehrere Arten begegnen: Zum einen könnte man die ab der Unterschreitung einer gewissen Größe notwendig auftretenden Quanteneffekte nicht mehr als Störung behandeln, sondern aktiv in der Elektronik ausnutzen. Dabei würden die Möglichkeiten der Quantenmechanik als mehr oder weniger weitreichende Erweiterung der hergebrachten Methoden zum Bau von elektronischen Schaltkreisen angesehen.

Es gibt allerdings noch eine zusätzliche Herangehensweise, auf die nahende Krise der Informationsverarbeitung zu reagieren, nämlich ein Paradigmenwechsel von der klassischen Informationverarbeitung von Bits und der daran gebundenen Elektronik hin zu einer Quanteninformationsverarbeitung. Die Quantenmechanik legt nicht unwesentlich eine Interpretation in Informationsbegriffen nahe, wenn man sich etwa die Unschärferelation oder die abstrakte Darstellung in Dirac-Notation in Erinnerung ruft.

Dieser Text soll denn auch den Paradigmenwechsel von der klassischen zur Quanteninformationsverarbeitung im Detail umreißen. Dazu sollen zunächst die verschiedenen klassischen Computermodelle kurz beschrieben werden, dann das Quantencomputermodell und die Beziehung zwischen beiden.

2 Computermodelle

Algorithmus Ein Algorithmus ist eine endliche Folge formaler Anweisungen zur Lösung eines Problems.

Computer Ein Computer ist eine Maschine, die einen in passender Form kodierten Algorithmus ausführen kann.

Will man Algorithmen theoretisch betrachten, so ist der Aspekt der Kodierung und Formalisierung zentral. Es existieren mehrere Computermodelle, die jeweils eine Darstellung eines Algorithmus festlegen.

Drei wesentliche Modelle sollen im folgenden vorgestellt werden. Jeder Algorithmus kann in jedem dieser Computermodelle dargestellt werden, sie sind äquivalent.

2.1 Die Turingmaschine

Das Modell der Turingmaschine ist für die theoretische Informatik sehr wichtig. Es ist in Abb. 1 skizziert. Man stelle sich ein unendliches Speicherband vor, wobei in jeder Speicherzelle ein Buchstabe aus einem definierten Alphabet

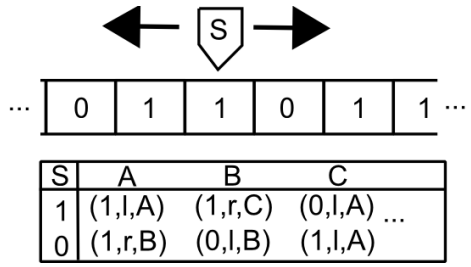


Abb. 1: Vereinfachte Darstellung einer Turingmaschine

(im Beispiel der Skizze 1 und 0) steht. Zusätzlich gibt es einen programmgesteuerten Kopf, der sich entlang des Speicherbands bewegen und die Symbole auf dem Band lesen und schreiben kann. Der Kopf kann sich zusätzlich in einem Zustand aus einer Menge definierter Zustände befinden. Das Programm hat die Form einer Tabelle. Abhängig davon, was der Kopf an seiner aktuellen Position für den Inhalt der Speicherzelle ausließt und seinem aktuellen Zustand (A,B,C,...) wird festgelegt, welcher Wert in die aktuelle Zelle geschrieben werden soll, in welche Richtung sich der Kopf auf dem Band weiterbewegen soll, und in welchen Zustand der Kopf übergehen soll. So würde z.B. in Abb. 1 der Kopf, wenn er im Zustand C den Wert 1 läse, diese 1 mit einer 0 überschreiben, auf dem Band eine Zelle nach links gehen, und in den Zustand A wechseln. Eine solche Turingmaschine ist nicht allzuweit von der in heutigen PCs umgesetzten „Von-Neumann-Architektur“ von Arbeitsspeicher und CPU entfernt, allerdings mit dem wesentlichen Unterschied, dass der Arbeitsspeicher endlich ist und das Programm nicht getrennt vorliegt, sondern ebenfalls auf das Band geschrieben wird.

2.2 Codedarstellungen und Komplexitäten

Eine zweite mögliche Darstellung von Algorithmen ist die weniger formale Codedarstellung, wie sie jedem Programmierer geläufig ist. Der Algorithmus wird dabei in einer Programmiersprache wie C oder Fortran (oder hunderte andere) beschrieben. Diese Darstellung zeichnet sich durch die Verwendung von Variablen und Kontrollstrukturen (Schleifen, Bedingungen) aus. Will man sich nicht auf eine konkrete Sprache festlegen, beschreibt man den Algorithmus oft im sog. Pseudocode, einer etwas freieren Formulierung, die ebenfalls auf die in allen Programmiersprachen vorkommenden Kontrollstrukturen zurückgreift[1], sich aber nicht um technische Details kümmern muss.

An der Codedarstellung lässt sich besonders gut die *Komplexität* eines Algorithmus analysieren. Dazu zählt man schlicht in Abhängigkeit von der Anzahl der Eingangsdaten (n) die Menge elementarer Anweisungen, die der Algorithmus durchlaufen muss, um zum Ziel zu kommen. Das Ergebnis wird dann meistens in der gebräuchlichen O-Notation geschrieben, bei der das asymptotische Verhalten für große n unter Vernachlässigung konstanter Faktoren angegeben

[1] Dabei soll hier nur von imperative Sprachen wie C, Fortran, Pascal die Rede sein, es gibt auch andere Programmiersprachen, die keine Kontrollstrukturen verwenden, etwa funktionale Programmiersprachen wie Lisp oder Haskell, diese orientieren sich an einem anderen Computermodell (siehe 2.3)

wird. So kann eine Algorithmus beispielsweise lineare ($O(n) = n$), quadratische ($O(n) = n^2$), linear-logarithmische ($O(n) = n \log n$) [2] oder auch exponentielle ($O(n) = a^n$) Laufzeit haben. Man unterscheidet hier zwei wesentliche Klassen von Algorithmen: solche, die noch in polynomieller Laufzeit ($O(n) = P_a(n)$) arbeiten (oder besser) werden mit P bezeichnet, solche, die nur noch in exponentieller Zeit bewältigbar sind, mit *NP*.[3]

Manchmal wird bei der Analyse eines Algorithmus nicht die Laufzeit, sondern der verwendete Speicherplatz nach derselben Methode beschrieben.

2.3 Funktionale Theorie und logische Gatter

Im dritten Computermodell wird der Algorithmus durch Funktionen dargestellt, die einer Eingabe (einem Tupel von Werten aus einem Eingangsalphabet) eine Ausgabe (ein ebensolches Tupel) zuordnen. Die Funktionen können beliebig verschachtelt werden und so auch komplexe Algorithmen darstellen. Funktionale Programmiersprachen wie Lisp und Haskell beruhen auf diesem Konzept.

Ein Spezialfall der funktionalen Theorie ist die Darstellung durch Boolesche Funktionen. Hier besteht das Alphabet nur aus den Werten 1 und 0. Die Definition einer Funktion geschieht über eine Tabelle. Man kann sich damit leicht vor Augen führen, dass die Anzahl der möglichen Booleschen Funktionen für eine feste Zahl von Ein- und Ausgangswerten endlich ist, für n Eingangsbits und ein Ausgangsbit genau 2^{2^n}.

Einige Beispiele für zweistellige Boolesche Funktionen sind in Abb. 2 dargestellt. Es handelt sich um das logische Und, Oder, Exklusives-Oder, Nicht-Und, Nicht-Oder, und die Implikation.

a	b	AND
0	0	0
0	1	0
1	0	0
1	1	1

a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1

a	b	XOR
0	0	0
0	1	1
1	0	1
1	1	0

a	b	NAND
0	0	1
0	1	1
1	0	1
1	1	0

a	b	NOR
0	0	1
0	1	0
1	0	0
1	1	0

a	b	IMP
0	0	1
0	1	1
1	0	0
1	1	1

Abb. 2: Auswahl sechs zweistelliger Boolescher Funktionen

Die Booleschen Funktionen lassen sich als *logische Gatter* direkt in die Elektronik übertragen und finden in dieser Form Verwendung in jedem Computerchip. Weiterhin lässt sich zeigen, dass sich jede boolesche Funktion als Verkettung von NANDs darstellen lässt, d.h. dass NAND ein sog. *universelles Gatter* ist. Mit der Schreibweise

$$\mathrm{NAND}(a, b) = a \mid b; \qquad \mathrm{AND}(a, b) = a \wedge b$$

[2] Vor allem bei Such- und Sortieralgorithmen kommt diese Laufzeit häufig vor.

[3] Strenggenommen steht *NP* für „nichtdeterministisch polynomiell“, d.h. ein nichtdeterministisch arbeitender Computer könnte den Algorithmus in polynomieller Laufzeit ausführen.

gilt z.B.:

$$a \wedge b = (a \mid b) \mid (a \mid b) \quad (1)$$

$$a \wedge b \wedge c = (((a \mid b) \mid (a \mid b)) \mid c) \mid (((a \mid b) \mid (a \mid b)) \mid c) \quad (2)$$

Die Gatter-Darstellung von Gl. (1) ist in Abb. 3 dargestellt. Man muss also nur

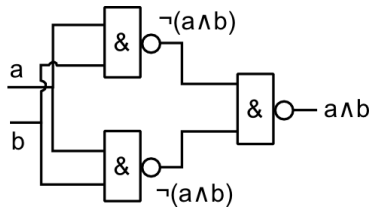


Abb. 3: AND(a, b) aus NAND-Gattern

das NAND-Gatter technisch umsetzen, um jede beliebige Boolesche Funktion und damit jeden beliebigen Algorithmus zu implementieren.

3 Quantencomputing

Um das Konzept des Quantencomputers zu entwickeln, verbleiben wir im Modell der logischen Funktionen und der mit ihnen verbundenen Gatter. Wir müssen allerdings sowohl das zugrundeliegende Alphabet als auch die auf es wirkenden Funktionen neu betrachten.

3.1 Qbits und Operatoren

Bei den Booleschen Funktionen haben wir als zugrundeliegendes Alphabet die klassischen Bits 1 und 0 verwendet. Für den Quantencomputer definieren wir sogenannte Qbits. Wir nehmen uns ein quantenphysikalisches System mit zwei Eigenzuständen und bezeichnen diese mit $|0\rangle$ und $|1\rangle$. Bekanntlicherweise kann sich ein System (solange es nicht ausgemessen wird) aber nicht nur in seinen Eigenzuständen, sondern auch in jeder Linearkombinationen derselben befinden. Wir definieren ein QBit $\Psi^{(1)}$ daher als

$$\Psi^{(1)} = \alpha_0 |0\rangle + \alpha_1 |1\rangle \quad (3)$$

Die Wahrscheinlichkeitsamplituden α_i geben quadriert die Wahrscheinlichkeit $P(i)$ dafür an, dass bei einer Messung das System im Eigenzustand $|i\rangle$ zu finden ist:

$$\mathrm{P}(i) = |\alpha_i|^2 \quad (4)$$

Betrachtet man Systeme von mehreren Qbits, muss man entsprechend mehr Eigenzustände verwenden:

$$\Psi^{(2)} = \alpha_{00} |00\rangle + \alpha_{01} |01\rangle + \alpha_{10} |10\rangle + \alpha_{11} |11\rangle \quad (5)$$

$$\Psi^{(3)} = \alpha_{000} |000\rangle + \alpha_{001} |001\rangle + \alpha_{010} |010\rangle + \alpha_{011} |011\rangle + \ldots \quad (6)$$

Formell geschieht diese Erweiterung über das Tensorprodukt:

$$\Psi^{(2)} \quad = \quad \Psi_1^{(1)} \otimes \Psi_2^{(1)}. \tag{7}$$

Es ist offensichtlich, dass die Anzahl der Eigenzustände und damit der Parameter (Wahrscheinlichkeitsamplituden) exponentiell mit der Anzahl der Bits wächst, genauer mit 2^n.

Während ein klassisches System von n Bits genau ein n-Tupel von $\{0,1\}$-Werten enthält, beschreibt ein QBit alle möglichen dieser n-Tupel gleichzeitig und beliebig überlagert.

Da bei einer festen Basis von Eigenzuständen das QBit durch die Wahrscheinlichkeitsamplituden vollständig beschrieben ist, kann man es als 2^n-komponentigen Vektor schreiben:

$$\vec{\Psi}^{(1)} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}; \qquad \vec{\Psi}^{(2)} = \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix}; \qquad \ldots \tag{8}$$

In dieser Darstellung wird dann auch das Tensorprodukt explizit ausgeführt.

Vektoren des $\mathbb{C}^n$ können bekanntlich durch Matrizen beliebig transformiert werden, daher liegt es nahe, auch die auf Qbits operierenden Funktionen („Quantengatter“) als $n \times n$-Matrizen zu schreiben. Da es für jedes n unendliche viele solcher Matrizen gibt, ist die Anzahl von Quantengattern im Gegensatz zu den klassischen logischen Gattern unendlich.

3.2 Bedingungen für Quantengatter

Aus den bisherigen Definitionen für Qbits und Quantengatter folgen einige wichtige Einschränkungen.

3.2.1 Erhaltung von Qbits

Da wir ein QBit durch Matrizen transformieren wollen, folgt sofort, dass wir die Zahl der Qbits nicht verändern können. Eine n-QBit-System transformiert wieder in ein n-QBit-System. Dies steht im Gegensatz zu den klassischen Booleschen Funktionen, wo beispielsweise die NAND-Funktion zwei Eingangsbits ein Ausgangsbit zuordnet.

3.2.2 Unitarität der Quantentransformationen

Wir wissen aus der Quantenmechanik, dass bei einer Messung das System auf jeden Fall in irgendeinem Eigenzustand zu finden ist. Das heißt für die Wahrscheinlichkeitsamplituden, dass sie sich zu Eins aufaddieren müssen:

$$\sum_i |\alpha_i|^2 \quad = \quad 1, \tag{9}$$

oder für die Vektorschreibweise:

$$|\vec{\Psi}| = 1. \tag{10}$$

Wenn diese Gleichungen sowohl vor als auch nach der Transformation erhalten sein sollen, ergibt sich eine wichtige Einschränkung für die in Frage kommenden Transformationmatrizen: Sie müssen längenerhaltend sein. Es muss also gelten:

$$\begin{aligned} \langle \Psi | \Psi \rangle &\stackrel{!}{=} \langle U\Psi | U\Psi \rangle \\ &= \left\langle UU^\dagger \Psi | \Psi \right\rangle \\ &= \left\langle \Psi | U^\dagger U \Psi \right\rangle \end{aligned} \tag{11}$$

Es lässt sich sofort feststellen, dass $U^\dagger U = UU^\dagger = I$ sein muss. Dies ist genau die Bedingung, die die Transformation U als unitär definiert. Wir können festhalten: Die Matrizen, mit denen die QBit-Vektoren transformiert werden, müssen unitär sein.

3.2.3 Reversibilität

Wir wissen, dass jede unitäre Matrix U ein Inverses hat, nämlich genau $U^\dagger$. Zusammen mit der Tatsache, dass die Anzahl der Qbits erhalten bleibt, ergibt sich daraus, dass alle Operationen auf Qbits reversibel sind. Für klassische logische Schaltungen ist dies nicht der Fall.

3.2.4 Schleifen, FanIn, FanOut

Als Konsequenz aus der Reversibilität ergibt sich insbesondere, dass bei Quantenschaltungen im Gegensatz zu klassischen Schaltungen keine Schleifen, kein FanIn (Zusammenführen zweier Leitungen) und kein FanOut (Teilen einer Leitung), siehe Abb. 4, möglich ist.

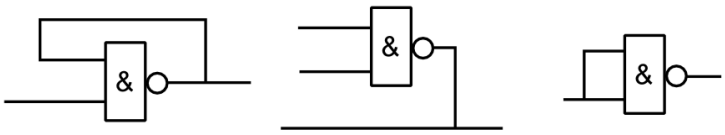


Abb. 4: Schleifen, FanIn und FanOut bei klassischen Gattern

Das FanOut wird durch das No-Cloning-Theorem verboten, welches besagt, dass es nicht möglich ist, eine Kopie eines beliebigen Quantenzustands anzufertigen.[4] Das FanIn erhält die Anzahl der QBits nicht, und eine Schleife würde, wenn man den Schaltkreis umkehren wollte, einen FanOut darstellen.

4 Das Quantum-Circuit-Gate-Model

Wir wollen nun die Quantengatter genauer betrachten. Dabei ist insbesondere auszuführen, wie komplexe Schaltungen aus einfachen Quantengattern zusammengesetzt werden. Ziel der Betrachtung ist es, einen Satz von universellen Gattern aufzustellen, mit denen alle möglichen Transformationen dargestellt werden können.

[4] siehe z.B. [3, S. 532]

4.1 Ein-Bit-Gatter

Zunächst sollte man die einfachsten Gatter betrachten, welche nur auf jeweils ein QBit wirken. Zwar gibt es davon unendlich viele, einige haben allerdings besondere Bedeutung. Eine Auswahl der wichtigsten Ein-Bit-Gatter ist in Abb. 5 zu sehen. Hervorzuheben sind insbesondere die ersten beiden Gat-

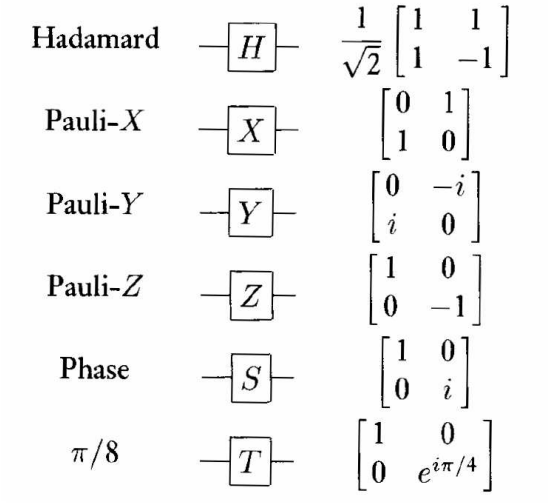


Abb. 5: Einige wichtige Ein-Bit-Gatter (aus [3])

ter: das Hadamard-Gatter erzeugt aus Eigenzuständen Überlagerungszustände und das Pauli-X-Gatter ist das quantenmechanische Äquivalent des klassischen NOT-Gatters (es vertauscht die beiden Wahrscheinlichkeitsamplituden, sodass $|1\rangle$ zu $|0\rangle$ wird und umgekehrt.

4.2 Das Dekompositionstheorem

Den ersten Schritt auf dem Weg der Darstellung komplexer durch einfache Gatter stellt das Dekompositionstheorem dar. Es besagt, dass sich jedes Ein-Bit-Gatter in eine Phasenverschiebung, eine Drehung um die z-Achse, eine Drehung um die y-Achse, und schließlich noch eine Drehung um die z-Achse zerlegen lässt. Wenn hier von Drehungen die Rede ist, bezieht sich dies auf die Bloch-Darstellung von Qbits als Zeiger auf die Oberfläche der Einheitskugel.[5]

$$U = e^{i\alpha} \underbrace{\begin{pmatrix} e^{-i\frac{\beta}{2}} & 0 \\ 0 & e^{i\frac{\beta}{2}} \end{pmatrix}}_{\mathrm{R}_z(\beta)} \underbrace{\begin{pmatrix} \cos\frac{\gamma}{2} & -\sin\frac{\gamma}{2} \\ \sin\frac{\gamma}{2} & \cos\frac{\gamma}{2} \end{pmatrix}}_{\mathrm{R}_y(\gamma)} \underbrace{\begin{pmatrix} e^{-i\frac{\delta}{2}} & 0 \\ 0 & e^{i\frac{\delta}{2}} \end{pmatrix}}_{\mathrm{R}_z(\delta)}. \quad (12)$$

Um das Theorem zu beweisen, muss gezeigt werden, dass sich jede unitäre Matrix darstellen lässt als

$$U = \begin{pmatrix} e^{i(\alpha-\beta/2-\delta/2)}\cos(\gamma/2) & -e^{i(\alpha-\beta/2+\delta/2)}\sin(\gamma/2) \\ e^{i(\alpha+\beta/2-\delta/2)}\sin(\gamma/2) & e^{i(\alpha+\beta/2+\delta/2)}\cos(\gamma/2) \end{pmatrix}. \quad (13)$$

[5] siehe [3, S. 15]

Ist dies geschehen, kann man sich leicht davon überzeugen, dass Gl. (13) schlicht das Matrizenprodukt der drei in Gl. (12) stehenden Matrizen ist.

Als Beweis von Gl. (13) lassen sich explizit die Parameter $\alpha, \beta, \delta, \gamma$ angeben. Angenommen, es soll die Matrix

$$U = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

mit der Determinanten $D = \pm 1$ (Unitarität) dargestellt werden, dann gilt:

$$\alpha = -i \ln\left(-\sqrt{D}\right), \tag{14}$$

$$\beta = -2i \ln\left(\left(\frac{-cd}{ab}\right)^{1/4}\right), \tag{15}$$

$$\delta = -2i \ln\left(\left(\frac{-bd}{ac}\right)^{1/4}\right), \tag{16}$$

$$\gamma = 2 \arccos\left(-\sqrt{\frac{ad}{D}}\right). \tag{17}$$

Mit diesen Angaben kann man Gl. (13) durch Einsetzen verifizieren.[6]

Zur späteren Verwendung schreiben wir Gl. (12) erneut um, und zwar in die Form

$$U = e^{i\alpha} AXBXC \tag{18}$$

mit

$$A = R_z(\beta) R_y(\gamma/2) \tag{19}$$

$$B = R_y(\gamma/2) R_z(-(\delta+\beta)/2) \tag{20}$$

$$C = R_z((\delta-\beta)/2) \tag{21}$$

X bezeichnet die Pauli-X-Matrix. Es gilt zusätzlich

$$ABC = I \tag{22}$$

4.3 Kontrollierte Operationen

Der erste Schritt beim Übergang von Ein-Bit- zu Mehr-Bit-Gattern ist die Implementierung von kontrollierten Operationen. Ein entsprechendes Gatter hat zwei Eingänge, das sog. Control-Bit und das Target-QBit. Auf das Target-QBit wird eine beliebige Transformation ausgeführt, aber nur, wenn sich das Control-QBit im Eigenzustand $|1\rangle$ befindet.[7] Wir wollen zunächst das wichtigste kontrollierte Gatter betrachten: das CNOT. Seine symbolische Darstellung, Transformationsmatrix, und Wirkung auf die Eigenzustände ist in Abb. 6 gezeigt.

Wie man sieht, macht das CNOT-Gatter nichts anderes als die Wahrscheinlichkeitsamplituden das Target-Qbits zu vertauschen, falls das Control-QBit gesetzt ist, und sie unverändert zu lassen, falls es nicht gesetzt ist.

[6] Bei der Rechnung sollte man sich in Erinnerung rufen, dass aus der vierten Wurzel beliebig viele i's herausgezogen werden können.

[7] Das Verhalten ist nur für $|1\rangle$ und $|0\rangle$ definiert.

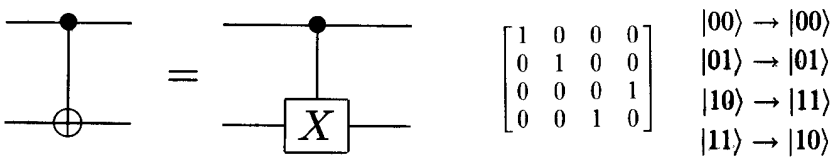


Abb. 6: das CNOT-Gatter

Wie kann man nun allgemeine kontrollierte Gatter implementieren? Wir verwenden dazu die Zerlegung aus Gl. (18) mit der Bedingung aus Gl. (22). Nun muss diese Zerlegung nur noch eins-zu-eins in Gatter implementiert werden, die NOTs (X) werden dabei aber durch CNOTs ersetzt. Die Schaltung ist in Abb. 7 dargestellt.

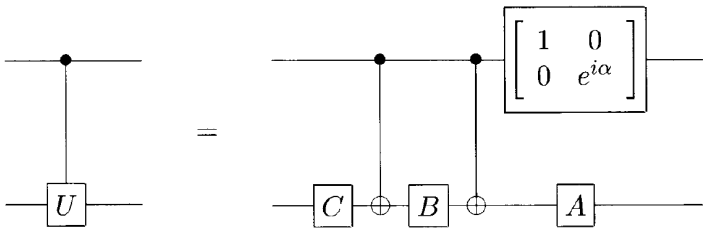


Abb. 7: Implementierung eines allgemeinen kontrollierten Gatters (aus [3])

Ist nun das Control-Bit gesetzt, wird die volle Reihe von Gattern durchlaufen und insgesamt die Transformation U ausgeführt. Ist das Control-Bit allerdings nicht gesetzt, fallen die NOT-Gatter weg, und übrig bleibt noch ABC. Dies ist jedoch nach Gl. (22) die Einheitsoperation, sodass das Target-Bit nach dem gesamten Durchlauf unverändert ist. Wir haben insgesamt also genau die gewünschte kontrollierte Operation implementiert.

Allgemein gibt es noch einige Erweiterungen der bisher besprochenen kontrollierten Operationen. So kann z.B. das Control-Bit invertiert werden, die Operation wird also ausgeführt, wenn es im Zustand $|0\rangle$ statt $|1\rangle$ ist. Außerdem lässt sich eine größere Zahl sowohl von Control- als auch von Target-Bits verwenden. Zum Beispiel ist die Implementierung von zwei Control-Bits in Abb. 8 angegeben. Sie lässt sich realisieren, indem man eine Transformation V findet, sodass $V^2 = U$. Diese Konzepte lassen sich beliebig kombinieren und erweitern.

4.4 Mehr-QBit-Gatter

An dieser Stelle ist ein allgemeiner Kommentar zum Verständnis von Mehr-Bit-Gattern angebracht. Wie bereits in Abschnitt 3.1 gesagt, muss für jede Anzahl von Qbits eine entsprechende eigene Basis von 2^n Eigenzuständen gewählt werden, die ihrerseits als Vektoren geschrieben werden. Jeder Quantenschaltkreis kann prinzipiell in seiner Gesamtheit als eine entsprechend große unitäre Transformationsmatrix geschrieben werden. Allerdings wollen wir in der Regel

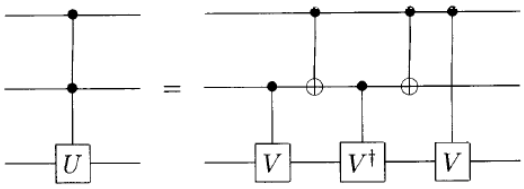


Abb. 8: Eine kontrolliertes Gatter mit zwei Control-Bits (aus [3])

Schaltkreise wie in Abb. 7 oder 8 darstellen, zusammengesetzt aus Ein- und Zwei-QBit-Gattern, welche für sich nur durch entsprechende 2×2- bzw. 4×4-Block-Matrizen dargestellt werden.

Um einen solchen Schaltkreis zu analysieren und die Gesamttransformation zu finden, muss man die n-QBit-Basis auflösen, und als eine Reihe von Ein-QBit-Vektoren schreiben (das Tensorprodukt also rückgängig machen). Z.B. wird aus dem Zwei-QBit-Vektor $(0, 0, 1, 0)$ die beiden einzelnen Qbits $(0, 1)$ und $(1, 0)$. Nun kann man die Matrizen der Schaltung auf die einzelnen Qbits anwenden und sie anschließend wieder per Tensorprodukt zusammenführen.

Mit dieser Methode kann man sich auch leicht Klarheit über eine scheinbare Unstimmigkeit in Abb. 7 verschaffen: Es scheint zunächst so, als ob der Phasenshift $e^{i\alpha}$ nicht wie gewünscht auf das Target-Bit, sondern auf das Control-Bit angewendet wird. Eine genauere Betrachtung zeigt: der Zustand $|10\rangle$ muss zerlegt werden in $|1\rangle = (0, 1)$ und $|0\rangle = (1, 0)$ und transformiert zu $e^{i\alpha} |1\rangle \otimes |0\rangle$, was sich wieder zu $e^{i\alpha} |10\rangle$ zusammensetzt. Entsprechend transformiert $|11\rangle \rightarrow e^{i\alpha} |11\rangle$, $|00\rangle \rightarrow |00\rangle$, und $|01\rangle \rightarrow |01\rangle$. Das Ergebnis ist also dasselbe, als wenn wir die Transformation

$$U = \begin{pmatrix} e^{i\alpha} & 0 \\ 0 & e^{i\alpha} \end{pmatrix}$$

auf das zweite QBit angewendet hätten, was der eigentlichen Intuition entspräche.

4.5 Universelle Gatter

Wir können nun unsere Betrachtung der Mehr-Bit-Gatter ausweiten auf beliebige unitäre $n \times n$-Matrizen. Um solche allgemeinen Transformationen zu handhaben, wird man auf universelle Systeme angewiesen sein, also eine begrenzte Anzahl von Gattern, mit denen man sämtliche Transformationen implementieren kann (analog zum klassischen NAND-Gatter). Schließlich ist es (wahrscheinlich) praktisch nicht möglich, einen Quantencomputer zu bauen, der direkt jede beliebig Transformation durchführen kann. Wir wünschen uns stattdessen einen endlichen Satz von Operationen, die wir einfach implementieren und dann beliebig zusammensetzen können. Wie sich zeigen wird, haben wir alle benötigten Elemente schon kennen gelernt.

Es sollen im folgenden zwei solcher universellen Systeme betrachtet werden.

4.5.1 Das System der Ein-Bit- und CNOT-Gatter

Es lässt sich zeigen, dass sich aus der Gesamtheit der Ein-QBit- und der CNOT-Gatter (mit Varianten) jede beliebige Transformation, also jede unitäre $n \times n$-

Matrix, implementieren lässt. Dazu muss man in zwei Schritten vorgehen.

Im ersten Schritt lässt sich zeigen: Jede unitäre $n \times n$-Matrix lässt sich in eine Reihe sog. Zwei-Level-Matrizen zerlegen. Eine Zwei-Level-Matrix ist eine Einheitsmatrix beliebiger Größe, bis auf vier nichttriviale Elemente, und zwar so, dass sich die Matrix auf eine eben aus diesen vier Elementen bestehende unitäre 2×2-Matrix reduziert, wenn man alle Zeilen und Spalten streicht, in denen kein nichttriviales Element steht (zwei der nichttrivialen Elemente müssen auf der Hauptdiagonalen liegen). Z.B. reduziert sich die Zwei-Level-Matrix

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha & 0 & \beta \\ 0 & 0 & 1 & 0 \\ 0 & \gamma & 0 & \delta \end{pmatrix}$$

zu der 2×2-Matrix

$$\tilde{U} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}.$$

Es ist offensichtliche Eigenschaft von Zwei-Level-Matrizen, dass sie nur auf zwei der Eigenvektoren eine verändernde Wirkung haben.

Die Zerlegung einer allgemeinen Matrix U soll nun die Form

$$U = U_1^\dagger U_2^\dagger \dots U_m^\dagger \tag{23}$$

haben mit den m Zwei-Level-Matrizen $U_1 \dots U_m$, $m \leq \frac{n(n-1)}{2}$ und der Eigenschaft

$$U_m U_{m-1} \dots U_1 U = I. \tag{24}$$

Die Zerlegungsmatrizen zu finden ist nach einem festen System möglich. Im Wesentlichen versucht man, in dem Produkt $U_\mu U_{\mu-1} \dots U_1 U$ mit wachsendem μ immer eine Null mehr in der ersten Spalte (und später weiteren Spalten) zu bekommen. Ein Beispiel:

Es soll die 4×4-Matrix

$$U = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}$$

zerlegt werden. Man geht wie folgt vor:

$$M := U;$$

$$U_1 = \begin{pmatrix} \frac{M_{11}^*}{\sqrt{|M_{11}|^2+M_{21}|}} & \frac{M_{21}^*}{\sqrt{|M_{11}|^2+M_{21}|}} & 0 & 0 \\ \frac{M_{21}}{\sqrt{|M_{11}|^2+M_{21}|}} & \frac{-M_{11}}{\sqrt{|M_{11}|^2+M_{21}|}} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}; \quad \begin{array}{l} M := U_1 M, \\ M_{21} \text{ wird 0;} \end{array} \tag{25}$$

$$U_2 = (\dots); \qquad M := U_2 M, \quad M_{31} \text{ wird 0;} \tag{26}$$

$$U_3 = \begin{pmatrix} \frac{M_{11}^*}{\sqrt{|M_{11}|^2+M_{41}|}} & 0 & 0 & \frac{M_{41}^*}{\sqrt{|M_{11}|^2+M_{41}|}} \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{M_{41}}{\sqrt{|M_{11}|^2+M_{41}|}} & 0 & 0 & \frac{-M_{11}}{\sqrt{|M_{11}|^2+M_{41}|}} \end{pmatrix}; \quad \begin{array}{l} M := U_3 M, \\ M_{41} \text{ wird 0.} \end{array} \tag{27}$$

An dieser Stelle hat $M = U_3U_2U_1U$ die Form

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -\frac{1}{2}(-1)^{3/4} & \frac{1}{\sqrt{2}} & \frac{\sqrt[4]{-1}}{2} \\ 0 & \frac{1}{2}\sqrt{\frac{4}{3}+i} & -\frac{1}{\sqrt{6}} & \frac{1}{2}\sqrt{\frac{4}{3}-i} \\ 0 & \frac{i}{\sqrt{3}} & \frac{1}{\sqrt{3}} & -\frac{i}{\sqrt{3}} \end{pmatrix},$$

wie erwünscht haben wir also in der ersten Zeile und ersten Spalte nur Nullen abseits des Hauptdiagonalelements. Es verbleibt aber noch eine unitäre 3×3-Untermatrix. Auf diese ist nun dasselbe Verfahren anzuwenden:

$$U_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{M_{22}^*}{\sqrt{|M_{22}|^2+M_{32}|}} & \frac{M_{32}^*}{\sqrt{|M_{22}|^2+M_{32}|}} & 0 \\ 0 & \frac{M_{32}}{\sqrt{|M_{22}|^2+M_{32}|}} & \frac{-M_{22}}{\sqrt{|M_{22}|^2+M_{32}|}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}; \quad \begin{array}{l} M := U_4\, M, \\ M_{32} \text{ wird } 0; \end{array} \tag{28}$$

$$U_5 = (\dots) \qquad M := U_5\, M; \quad M_{42} \text{ wird } 0. \tag{29}$$

Wir haben nun

$$\begin{aligned} M &= U_5U_4U_3U_2U_1U \\ &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -i & -i \end{pmatrix} \end{aligned}$$

Im letzten Schritt setzen wir

$$U_6 = M^{-1} \tag{30}$$

und können verifizieren, dass gilt

$$U_1^\dagger U_2^\dagger U_3^\dagger U_4^\dagger U_5^\dagger U_6^\dagger = U, \tag{31}$$

$$U_6U_5U_4U_3U_2U_1 = I. \tag{32}$$

Nach demselben Schema lassen sich beliebige unitäre $n \times n$-Matrizen in Zwei-Level-Matrizen zerlegen.

Nachdem wir nun also beliebige unitäre Matrizen in Zwei-Level-Matrizen zerlegt haben, ist der nächste Schritt, diese aus Ein-QBit-Gattern zusammenzusetzen. Wie schon erwähnt verändern die Zwei-Level-Matrizen nur genau zwei der Eigenvektoren. Zur Zerlegung hält man sich nun an die folgenden Instruktionen:

- Suche die beiden Eigenvektoren, die das Gatter verändert.
- Schreibe den sog. „Gray-Code“ zwischen beiden Eigenvektoren aus.
- Setze den Gray-Code mit erweiterten CNOT-Gattern um.
- Setze im letzten Schritt das $\tilde{U}$-Gatter an die Stelle des sich ändernden QBits.

- Implementiere den Gray-Code rückwärts.

Unter dem Gray-Code versteht man die Verbindung von zwei Bit-Zeichenketten durch Zwischenschritte, bei denen jeweils nur genau ein Bit verändert werden darf. Der Gray-Code zwischen (000) und (111) ist beispielsweise

$$(000) \rightarrow (001) \rightarrow (011) \rightarrow (111). \tag{33}$$

Es ist nicht schwer einzusehen, dass sich jeder Schritt in einem solchen Code direkt als CNOT-Gatter schreiben lässt: das sich ändernde QBit ist das Target-QBit, die übrigen sind die Control-QBits, negiert falls sie Null sind.

Nach dieser Vorgehensweise lässt sich die 8×8-Matrix

$$U = \begin{pmatrix} \alpha & 0 & 0 & 0 & 0 & 0 & 0 & \beta \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \gamma & 0 & 0 & 0 & 0 & 0 & 0 & \delta \end{pmatrix}$$

mit der Untermatrix

$$\tilde{U} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}$$

darstellen. Die Implementierung ist in Abb. 9 zu sehen. Man kann leicht sehen, dass der Gray-Code aus Gl. (33) umgesetzt wird.

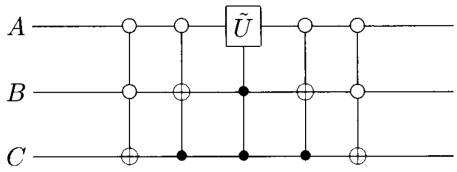


Abb. 9: Zerlegung und Implementierung einer Zwei-Level-Matrix (aus [3])

4.5.2 Hadamard-, Phase-, CNOT-, und $\frac{\pi}{8}$-Gatter

Das bisher kennengelernte System aus Ein-QBit- und CNOT-Gattern stellt zwar schon eine erhebliche Vereinfachung dar, allerdings müssen wir immer noch unendlich viele Gatter implementieren können, sodass dieses universelle System für die praktische Umsetzung weniger interessant ist. Zum Glück gibt es aber auch noch ein endliches universelles System, welches aus Hadamard-, Phase-, CNOT- und $\frac{\pi}{8}$-Gatter besteht. Eine kleine Einschränkung zeigt sich allerdings: beliebige Transformationen können durch diese Gatter nur genähert (aber mit beliebiger Genauigkeit) dargestellt werden; je genauer die Näherung, desto mehr Gatter werden benötigt. Allerdings lässt sich zeigen, dass eine kleine Abweichung in der

Transformation auch nur eine kleine Abweichung im Ergebnis nach sich zieht, sodass die Universalität nicht gefährdet ist.

Das System im Detail auszuführen würde an dieser Stelle den Rahmen sprengen, auch wenn es für die experimentelle Umsetzung von Quantencomputern von großer Wichtigkeit ist (es wird auch als Standardsystem bezeichnet). Kurz gesagt lässt sich zeigen, dass aus dem Hadamard- und dem $\frac{\pi}{8}$-Gatter jedes beliebige Ein-Bit-Gatter genähert werden kann, und zwar indirekt über Näherung der Rotationsmatrizen aus dem Dekompositionstheorem in Gl. (12).

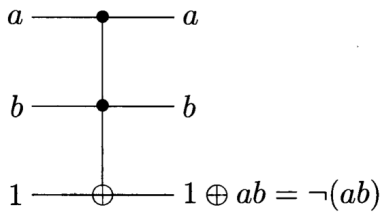
5 Klassische Algorithmen vs. Quantenalgorithmen

Nach der Ausführung des Apparats der Quantengatter sind wir nun in der Lage, auf n-QBit-System beliebige Transformationen anzuwenden. Im nächsten Schritt soll nun die Beziehung zwischen klassischen Schaltkreisen und Quantengattern untersucht werden.

5.1 Von logischen Gattern zu Quantengattern

Da die Implementierung von Algorithmen mit klassischen logischen Gattern sehr ausgereift ist, stellt sich die Frage, ob es möglicherweise eine einfache Methode gibt, solche klassischen Gatter direkt in eine Implementierung durch Quantengatter zu überführen. Wir wollen also nur auf Eigenzuständen operieren, und diese so transformieren, dass der resultierende Eigenzustand genau dem klassischen Ergebnis entspricht. Dass dies auf einfache Art und Weise gelingt, ist nicht selbstverständlich, schließlich haben die Quantengatter zunächst einige Einschränkungen gegenüber den klassischen (siehe 3.2). Es wird sich aber zeigen lassen, dass sich diese Beschränkungen für den Grenzfall der Eigenzustände umgehen lassen.

Wie bereits erwähnt, stellt das klassische NAND-Gatter ein universelles System für logische Schaltkreise dar: jeder klassische Algorithmus kann nur mit NAND-Gattern implementiert werden. Nun kann man sich davon überzeugen, dass bei den Quantengattern das sog. Toffoli-Gatter (Abb. 10) ein vollständiges Analogon zum NAND-Gatter ist.



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

Abb. 10: Das Toffoli-Gatter

Es handelt sich beim Toffoli-Gatter um ein spezielles CNOT mit zwei Control-Bits. Man kann sich nun leicht davon überzeugen, dass, wenn das Target-QBit auf 1 (d.h. $|1\rangle$) vorinitialisiert wird, man bei beliebigen Control-QBits a und b als Ergebnis im Target-QBit das Ergebnis NAND(a, b) erhält.

Das Toffoli-Gatter hat noch eine weitere angenehme Eigenschaft: Man kann damit ein FanOut (Abzweigung) simulieren. Erhält das Toffoligatter nämlich $|1a0\rangle$ als Eingang, liefert es $|1aa\rangle$. Damit wurde effektiv das Bit a kopiert. Dass dies keine Verletzung des No-Cloning-Theorems ist, sieht man daran, dass dieser Kopiervorgang nur bei reinen Eigenzuständen funktioniert: für ein allgemeines QBit $|\Psi\rangle = (\alpha_0, \alpha_1)$ kann man leicht berechnen, dass durch das Toffoli-Gatter die Transformation

$$|1\rangle \otimes |\Psi\rangle \otimes |0\rangle \quad \longrightarrow \quad \alpha_0 |100\rangle + \alpha_1 |111\rangle \not\equiv |1\rangle \otimes |\Psi\rangle \otimes |\Psi\rangle$$

ausgeführt wird, man also nicht davon sprechen kann, dass $|\Psi\rangle$ kopiert würde.

Es ist allerdings nach wie vor zu beachten, dass das Quantengatter reversibel sein muss. Dies bedeutet insbesondere, dass an keiner Stelle neue QBits erzeugt oder alte „weggeschmissen“ werden dürfen; alle QBits, die an irgendeinem Punkt benötigt werden, müssen von Anfang an eingegeben werden: es ist im Allgemeinen eine größere Anzahl von vorinitialisierten Hilfsbits nötig. So zeigt Abb. 11 die Implementierung der in Abb. 3 gezeigten AND-Schaltung durch Toffoligatter. Neben den „echten“ Eingängen für a und b gibt es noch fünf weitere Hilfsbits.

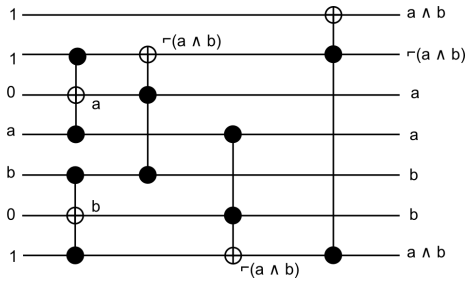


Abb. 11: Implementierung von AND mit Toffoli-Gattern

5.2 Der Deutsch-Algorithmus

Selbstverständlich soll es nicht nur darum gehen, klassische Algorithmen mit Quantengattern zu implementieren. Die Tatsache, dass ein QBit prinzipiell unendlich viel Informationen fassen kann, soll vielmehr genutzt werden, um Algorithmen zu entwickeln, die schneller arbeiten als äquivalente klassische.

Ein einfaches Beispiel ist der Deutsch-Algorithmus, der zwar kaum eine praktische Anwendung hat, aber dennoch gut demonstriert, wie „Quantenparallelismus“ benutzt werden kann, um schneller zu rechnen als dies mit einem klassischen Computer möglich wäre. Der Algorithmus berechnet, ob eine einstellige Boolesche Funktion konstant ist oder nicht. Die Implementierung ist in Abb. 12 zu sehen.

Man kann die Transformation schrittweise nachvollziehen und findet für den Zustand des Systems an den markierten Stellen:

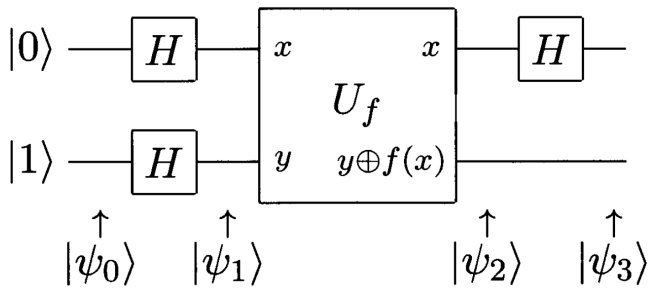


Abb. 12: Implementierung des Deutsch-Algorithmus (aus [3])

$$
\begin{aligned}
|\Psi_0\rangle &= |0\rangle \otimes |1\rangle \\
|\Psi_1\rangle &= \left[\frac{1}{\sqrt{2}}|0\rangle + |1\rangle\right] \otimes \left[\frac{1}{\sqrt{2}}|0\rangle - |1\rangle\right] \\
|\Psi_2\rangle &= \begin{cases} \pm\left[\frac{1}{\sqrt{2}}|0\rangle + |1\rangle\right] \otimes \left[\frac{1}{\sqrt{2}}|0\rangle - |1\rangle\right] & \text{falls } f(0) = f(1) \\ \pm\left[\frac{1}{\sqrt{2}}|0\rangle - |1\rangle\right] \otimes \left[\frac{1}{\sqrt{2}}|0\rangle - |1\rangle\right] & \text{falls } f(0) \neq f(1) \end{cases} \\
|\Psi_3\rangle &= \begin{cases} \pm|0\rangle\left[\frac{1}{\sqrt{2}}|0\rangle - |1\rangle\right] & \text{falls } f(0) = f(1) \\ \pm|1\rangle\left[\frac{1}{\sqrt{2}}|0\rangle - |1\rangle\right] & \text{falls } f(0) \neq f(1) \end{cases}
\end{aligned}
$$

Das 1. QBit enthält offensichtlich das gewünschte Ergebnis. Obwohl die Funktion nur einmal berechnet wurde, konnte eine globale Eigenschaft, d.h. eine Eigenschaft, die die Kenntnis der Funktion an *allen* Stellen voraussetzt, ermittelt werden! Ein klassischer Computer hätte beide Funktionswerte getrennt berechnen müssen. Es zeigt sich, dass Quantencomputer bei bestimmten Algorithmen fundamental schneller sein können. Im Fall des Deutsch-Algorithmus ergibt sich sogar ein exponentiell besseres Laufzeitverhalten: die Anzahl der möglichen Belegungen für eine Boolesche Funktion, die ein klassischer Computer alle ausrechnen müsste, steigt gemäß 2^n mit der Anzahl der Eingangsvariablen. Es wird aber nur *ein* Quantengatter benötigt, um die Funktion an allen Stellen gleichzeitig zu berechnen.

5.3 Die Mächtigkeit des Quantencomputers

Im Vergleich zwischen klassischem und Quantencomputer haben wir durch die vorangehenden Betrachtungen, etwas vereinfachend, folgende Feststellungen gemacht:

- Alles, was ein Computer effizient lösen kann, kann auch ein Quantencomputer (zumindest theoretisch) effizient lösen.
- Einiges von dem wir denken, dass es ein Computer nicht effizient lösen kann, kann der Quantencomputer effizient lösen.

Wenn wir uns an die in Abschnitt 2.2 erwähnten Komplexitätsklassen erinnern, bedeutet das, dass die Klasse der Probleme, die ein Quantencomputer effizient

lösen kann (BPQ) die Klasse der P Probleme ganz und die Klasse der NP-Probleme zum Teil erfasst (Abb. 13).

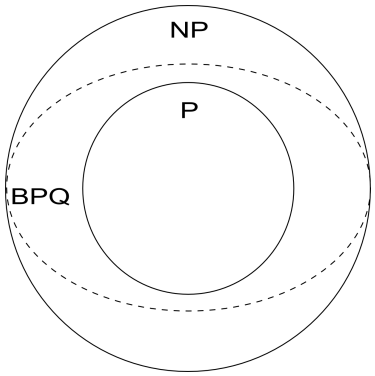


Abb. 13: Von einem Quantencomputer effizient lösbaren Probleme

Dabei ist allerdings generell unbewiesen, ob die Unterscheidung zwischen P- und NP-Problemen überhaupt existiert: Möglicherweise ist es doch möglich, die „schweren“ NP-Probleme mit einem klassischen Computer effizient zu lösen (auch wenn niemand ernsthaft daran glaubt). Falls dem so wäre, hätte der Quantencomputer keinen wesentlichen Vorteil. Zudem sollte man sich klarmachen, dass auch für den Quantencomputer einige Probleme bisher nicht effizient lösbar sind. Dies betrifft insbesondere alle NP-vollständigen Probleme wie etwa das Travelling-Salesman-Problem.

6 Ausblicke

6.1 Quantensimulation und Fourier- und Such-Algorithmen

Da der vorgestellte Deutsch-Algorithmus keine praktische Anwendung hat, soll an dieser Stelle noch ein kurzer Ausblick auf tatsächliche Anwendungen des Quantencomputers gegeben werden. Eine wichtige potentielle Anwendung findet sich in der Simulation von experimentell schwer zugänglichen Quantensystemen. Wie wir bereits gesehen haben, steigt die Anzahl der zur Beschreibung eines n-QBit-Systems nötigen Parameter (die Wahrscheinlichkeitsamplituden) exponentiell (mit 2^n). Ein klassischer Computer würde also exponentiell viel Speicher benötigen, was die Simulation größerer Systeme verhindert. Wenn es allerdings gelingt, ein System mit einem äquivalenten Hamiltonoperator zu finden, dass als Quantencomputer implementiert werden kann, reichen linear viele Qbits aus.[8]

Was die Algorithmik betrifft, so gibt es zwei fundamentale Klassen von Problemen, für die bis jetzt Quantenalgorithmen gefunden worden sind, die fundamental schneller als die bisher bekannten klassischen Algorithmen sind. Dies ist zum einen die Quantensuche, die gegenüber herkömmlichen Suchalgorithmen um n^2 schneller ist, und zum anderen die Klasse der Fouriertransformationsalgorithmen, für die sogar eine exponentielle Verbesserung erreicht werden kann.

[8] Die Quantensimulation geht wesentlich auf Feynman zurück [2].

Der wohl wichtigste Algorithmus letzterer Gruppe ist der Shor-Algorithmus zur Primzahlzerlegung (siehe [5]).

6.2 Die Quantenturingmaschine und Quantencode

Das Modell, in dem wir Quantencomputing betrachtet haben, war bisher ausschließlich das Modell der Schaltkreise. Es lässt sich aber noch die Brücke schlagen zu den beiden anderen in Abschnitt 2 erwähnten Computermodellen.

Die Quantenturingmaschine wurde wesentlich von Deutsch entwickelt [1]. Sie entspricht in weiten Teilen der klassischen Turingmaschine, allerdings bestehen sowohl Kopf als auch Band aus Qbits. Durch Quantenparallelismus ist es möglich, dass der Kopf in mehrere Richtungen gleichzeitig geht. Allein aus diesem Bild wird schon die potentielle Stärke des Quantencomputers ersichtlich.

Es existieren auch bereits Programmiersprachen für Quantencomputer, z.B. das an der TU Wien entwickelte QCL [4]. Die Sprache hat die üblichen Variablen und Strukturen klassischer Sprachen, und führt darüber hinaus spezielle Datentypen und Befehle für Qbits ein. Hier ein kurzes Beispiel einer Implementierung der diskreten Fourier-Transformation, wie sie für den Shor-Algorithmus verwendet wird:[9]

```
operator dft(qureg q) { // main operator
  const n=#q;           // set n to length of input
  int i; int j;         // declare loop counters
  for i=0 to n-1 {
    for j=0 to i-1 {    // apply conditional phase gates
      CPhase(2*pi/2^(i-j+1),q[n-i-1] & q[n-j-1]);
    }
    Mix(q[n-i-1]);      // qubit rotation
  }
  flip(q);              // swap bit order of the output
}
```

7 Zusammenfassung

Wir haben uns zunächst mit drei wichtigen klassischen Computermodellen vertraut gemacht, um dann unmittelbar zum Quantencomputing überzugehen. Dabei haben wir Qbits als Vektoren dargestellt, die mit unitären Matrizen manipuliert werden können. Für jedes n-QBit-System wird eine Basis von 2^n Eigenzuständen eingeführt. Die Transformation der Vektoren durch Matrizen lässt sich eins-zu-eins durch Quantengatter darstellen.

Im nächsten Schritt haben wir das System der Quantengatter systematisch aufgebaut. Ausgehend von Ein-QBit-Gattern wurde die Zerlegung in Rotationen (Dekompositionstheorem) gezeigt, und daraus zunächst kontrollierte Operationen gewonnen. Für Mehr-QBit-Gatter haben wir gesehen, wie sie in Zwei-Level-Gatter zerlegt werden können, welche dann durch CNOT und Ein-QBit-Gatter implementiert werden können. Schließlich sind wir kurz darauf eingegangen, dass mit Hadamard, Phase, CNOT, und $\frac{\pi}{8}$-Gattern auf Grundlage des Dekompositionstheorems beliebige Ein-QBit-Gatter genähert werden können. Wichtigste

[9] aus [4, S. 105])

Konsequenz ist, dass alle Gatter aus einem endlichen Satz von Universalgattern hergestellt werden können.

Im letzten Teil wurde dann erläutert, wie das Quantenanalogon klassischer Schaltkreise zu konstruieren ist, und im Anschluss der Deutsch-Algorithmus als Beispiel für einen „echten“ Quantenalgorithmus betrachtet. Es zeigt sich, dass Quantencomputer mit speziellen Algorithmen für bestimmte, aber nicht alle Probleme fundamental effizienter sind als klassische Computer.

Literatur

[1] Deutsch, David: Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer. In: *Proc. Roy. Soc. Lond. A* 400 (1985), S. 97

[2] Feynman, Richard P.: Simulating Physics with Computers. In: *Int. J. Theor. Phys.* 21 (1982), S. 467

[3] Nielsen, Michael ; Chuang, Isaac: *Quantum Computation and Quantum Information*. Cambridge University Press, 2000

[4] Ömer, Bernhard: *Quantum Programming in QCL*, TU Wien, Diplomarbeit, 2000

[5] Shor, Peter W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. In: *SIAM J. on Computing* 26 (1997), Nr. 5, S. 1484–1509

[6] Williams, Colin P. ; Clearwater, Scott H.: *Explorations in Quantum Computing*. Springer, 1997