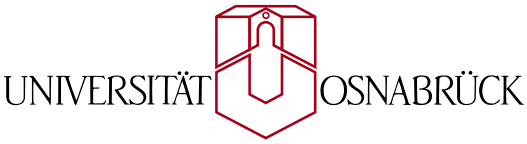
Untersuchungen zum Grover-Algorithmus

Bachelorarbeit

Uwe Sander
Fachbereich Physik



21. Juli 2004

Inhaltsverzeichnis

1 Zusammenfassung **5**

2 Abstract **6**

3 Einleitung **7**

4 Quantencomputer **9**
4.1 Was ist ein Quantencomputer? . . . 9
4.2 Quantenbits . . . 10
4.2.1 Formalismus . . . 10
4.2.2 Vom Bit zum Qubit . . . 12
4.3 Operationen auf Qubits . . . 12
4.3.1 Ein-Qubit-Gatter . . . 13
4.3.2 Mehr-Qubit-Gatter . . . 14
4.4 Gatter als unitäre Operationen . . . 16
4.5 Schaltkreise . . . 18
4.6 Algorithmen . . . 20
4.7 Physikalische Realisierung von Quantencomputern . . . 21
4.7.1 Quantenoptische Systeme . . . 22
4.7.2 Kernmagnetische Resonanz . . . 23
4.7.3 Festkörpersysteme . . . 23
4.7.4 Zusammenfassung . . . 23

5 Der Grover-Algorithmus **24**
5.1 Überblick . . . 24
5.2 Das Orakel . . . 25
5.3 Die Prozedur . . . 27
5.4 Geometrische Interpretation . . . 29
5.5 Laufzeit und Erfolgswahrscheinlichkeit . . . 31
5.6 Inversion um den Mittelwert . . . 32
5.7 Eine Beispielimplementation mit zwei Qubits . . . 34
5.8 Anwendungsmöglichkeiten . . . 35
5.8.1 Quantum counting . . . 35
5.8.2 NP-vollständige Probleme . . . 38
5.8.3 Die Suche in unstrukturierten klassischen Datenbanken . . . 39
5.9 Der Grover-Algorithmus ist optimal . . . 43

6 Simulation auf einem klassischen Computer **46**
6.1 Zielsetzung . . . 46
6.2 Die Programmiersprache Java . . . 47
6.3 Die Klassen in der Simulation . . . 47
6.4 Die Klassen im Detail . . . 48
6.4.1 `Complex.java` im Package `math` . . . 48

6.4.2 `Matrix.java` im Package `math` . . . 49
6.4.3 `Gate.java` im Package `gates` . . . 49
6.4.4 `CNot.java` im Package `gates` . . . 49
6.4.5 `Hadamard.java` im Package `gates` . . . 49
6.4.6 `Id.java` im Package `gates` . . . 50
6.4.7 `Not.java` im Package `gates` . . . 50
6.4.8 `Qubit.java` im Package `grover` . . . 50
6.4.9 `Register.java` im Package `grover` . . . 50
6.4.10 `PhaseShift.java` im Package `grover` . . . 51
6.4.11 `Oracle.java` im Package `grover` . . . 51
6.4.12 `GroverIteration.java` im Package `grover` . . . 52
6.4.13 `Measurement.java` im Package `grover` . . . 52
6.4.14 `Grover.java` im Package `grover` . . . 53
6.5 Analyse . . . 54
6.6 Vergleich mit *QuCalc* . . . 55

7 Ausblick **56**

A Eine sehr kurze Einführung in die Theoretische Informatik **59**
A.1 Die Turingmaschine . . . 59
A.2 Algorithmen . . . 59
A.3 Komplexitätsklassen . . . 59
A.4 Landau-Notation . . . 60

B Phasenabschätzung **60**

C Die Quelltexte **62**
C.1 Das Package `math` . . . 62
C.1.1 Die Klasse `Complex.java` . . . 62
C.1.2 Die Klasse `Matrix.java` . . . 65
C.2 Das Package `gates` . . . 69
C.2.1 Die Klasse `CNot.java` . . . 69
C.2.2 Die Klasse `Gate.java` . . . 70
C.2.3 Die Klasse `Hadamard.java` . . . 71
C.2.4 Die Klasse `Id.java` . . . 72
C.2.5 Die Klasse `Not.java` . . . 73
C.3 Das Package `grover` . . . 74
C.3.1 Die Klasse `Grover.java` . . . 74
C.3.2 Die Klasse `GroverIteration.java` . . . 80
C.3.3 Die Klasse `Measurement.java` . . . 81
C.3.4 Die Klasse `Oracle.java` . . . 84
C.3.5 Die Klasse `PhaseShift.java` . . . 85
C.3.6 Die Klasse `Qubit.java` . . . 87
C.3.7 Die Klasse `Register.java` . . . 90

Literatur **93**

Danksagung **96**

Eidesstattliche Erklärung **97**

1 Zusammenfassung

Der Grover-Algorithmus konnte mit der Hochsprache Java auf einem klassischen Computer simuliert werden. Meine Applikation konnte eine einfache Datenbank nach einem oder mehreren Elementen durchsuchen. Die unitären Transformationen des Algorithmus ließen sich ohne Probleme auf algorithmische Strukturen abbilden, wenn man mit Matrizen rechnet. Der hohe Speicherbedarf und die vielen rechenintensiven Matrixoperationen begrenzten jedoch die Anwendbarkeit der Simulation auf kleine Suchräume. Pro zusätzlichem Qubit verdoppelte sich der Speicherbedarf und verzehnfachte sich die Laufzeit des Programms. Im Allgemeinen wächst der Aufwand auf klassischen Rechnern linear mit der Zahl der Datenbankelemente; der Grover-Algorithmus bietet eine quadratische Beschleunigung.
Andere Software wie zum Beispiel *Mathematica* mit dem Paket *QuCalc* benötigte zwar weniger Speicher als mein Programm, der zeitliche Aufwand hingegen war in etwa derselbe. Hier zeigte sich, dass ein klassischer Computer mit solchen Aufgaben schnell überfordert werden kann.
Ein Schwachpunkt der Implementierung ist das Orakel, dem man vorab die Lösungen mitteilen muss. Wünschenswert wäre ein „echtes“ Orakel, das tatsächlich die Lösung(en) anhand eines bestimmten Kriteriums in der Datenbank erkennt. Die quantenmechanische Realisierung einer solchen Operation scheint allerdings nicht einfach zu sein. Es ist mir jedenfalls nicht gelungen, Literaturquellen zu diesem Thema zu finden.

2 Abstract

Grover's algorithm could be simulated on a classical computer using the high-level programming language Java. My application could search for one or more elements in in a simple database. I was able to map the unitary transformations required by the algorithm onto algorithmic structures. The simulation makes use of many matrix operations, which lead to a huge demand of memory and computing power. So my application can only simulate small search spaces. Each additional qubit doubled the demand of memory and increased tenfold the running time of my program. In general, the complexity depends linearly on the number of elements in the database; Grover's algorithm gives us a quadratic speedup.
Other software like *Mathematica* with the *QuCalc* package needed less memory than my application, but the running time was roughly the same. This means any classical computer can easily be overextended by this kind of search problems.
The oracle was a weak point in my implementation as one had to tell the solutions before running the algorithm. A „real“ oracle would recognise the solution(s) within the database using a certain criteria. But it seems to be hard to realise this operation quantum mechanically. At least I could not find any papers dealing with this aspect in depth.

3 Einleitung

Der Grover-Algorithmus ist einer von wenigen grundlegenden Algorithmen für Quantencomputer. Er sucht nach der „Nadel im Heuhaufen“, was bedeutet, dass sich mit seiner Hilfe unstrukturierte Datenbanken durchsuchen lassen. Diese Aufgabe erledigt er schneller, als jeder klassische Suchalgorithmus es könnte. Sein Anwendungsgebiet umfasst Probleme aus der Theoretischen Informatik, die Suche in klassischen Datenbanken (zum Beispiel in einem Telefonbuch) und das Auffinden des Schlüssels in Kryptographieverfahren. Dadurch wird der Algorithmus für die Praxis wichtiger als zum Beispiel der Deutsch-Josza-Algorithmus, der erstmals die Überlegenheit eines Quantenalgorithmus gegenüber einem klassischen demonstrierte, aber kaum praktische Anwendungen hat.
Lov Grovers ursprünglicher Vorschlag [Gro96, Gro97] aus dem Jahr 1996 für einen Quanten-Suchalgorithmus beschreibt das Vorgehen für genau *eine* „Nadel“. Im Jahr 1998 erweiterten Boyer, Brassard, Høyer und Trapp [BBHT98] diese Idee für Fälle, in denen es mehr als eine Lösung gibt. Seitdem ist der Algorithmus kaum noch verändert oder erweitert worden, was vor allem zwei Gründe hat: Zum einen ist bewiesen, dass es keinen schnelleren Orakel-basierten Suchalgorithmus für Quantencomputer geben kann [BBBV97]. Zum anderen ist die Entwicklung von Quantenalgorithmen sehr komplex und kaum intuitiv. Sie unterscheidet sich grundsätzlich von der klassischen Algorithmenentwicklung. In den letzten Jahren beschränkte man sich darauf, Anwendungsgebiete (zum Beispiel klassische Datenbanken [RG02]) und Interpretationen des Algorithmus (zum Beispiel die geometrische Interpretation in [Aha99]) zu untersuchen. Eine der letzten Detailverbesserungen stammt von Grover selbst [Gro02] und beschreibt, wie die Zahl der Operationen verringert werden kann. Er äußert die Hoffnung, dass noch weitere Verbesserungen möglich sind und gefunden werden.
Diese Bachelorarbeit hat einen theoretischen und einen praktischen Teil. Im theoretischen Teil beschreibe ich, was ein Quantencomputer ist und wie der Grover-Algorithmus funktioniert. Im praktischen Teil simuliere ich den Algorithmus auf einem klassischen Computer.
Kapitel 4 beschreibt die Grundlagen der Quanteninformatik. Dieses Modell geht auf Richard Feynman zurück, der 1982 erkannte, dass wir quantenmechanische Effekte ausnutzen können, um Quantencomputer zu bauen, die klassischen Rechnern überlegen sind. Statt Bits rechnet ein Quantencomputer mit Quantenbits oder kurz Qubits, die in einer Überlagerung von zwei Zuständen sein können. Um die durch Qubits repräsentierten Informationen verarbeiten zu können, brauchen wir Quantengatter (im Folgenden nur als Gatter bezeichnet). Wie wir sehen werden, lassen sich diese durch unitäre Transformationen darstellen. Es gibt universelle Sätze von Gattern, so dass wir jede Operation aus wenigen elementaren Gattern zusammensetzen können. Gatter lassen sich zu Schaltkreisen kombinieren, die einen gegebenen Anfangszustand so manipulieren, dass eine Messung des Endzustandes ein sinnvolles klassisches Ergebnis liefert. Desweiteren widmet sich das Kapitel den Quantenalgorithmen, deren Entwicklung alles andere als einfach ist. Neben dem Grover-Algorithmus gibt es als wichtige Quantenalgorithmen noch den bereits erwähnten Deutsch-Josza-Algorithmus und den

Shor-Algorithmus zur Faktorisierung großer Zahlen. Am Ende des Kapitels werden einige Ansätze zur physikalischen Realisierung von Quantencomputern vorgestellt. Ob man jemals die technischen Hürden bei der Implementation der „Hardware“ überwinden kann, ist noch ungewiss. Den letzten großen Erfolg feierte IBM im Jahr 2001, als es gelang, einen Quantencomputer mit 7 Qubits auf Basis von NMR[1] zu bauen.
In Kapitel 5 stelle ich den Grover-Algorithmus vor, der das Durchsuchen einer unstrukturierten (oder unsortierten) Datenbank beschleunigt. Klassisch würde eine lineare Suche $O(N)$ Schritte benötigen, um das gesuchte Element aus N Möglichkeiten zu finden. Der Grover-Algorithmus schafft es in $O(\sqrt{N})$ Schritten. Am Ende des Kapitels wird bewiesen, dass diese Beschleunigung nicht verbessert werden kann, der Grover-Algorithmus also optimal ist. Das Herz des Algorithmus bildet das Orakel, welches im Grunde eine binäre Funktion ist, die uns sagt, ob ein Element die Suchkriterien erfüllt. Die Wirkung des Orakels lässt sich, wie alle anderen Schritte des Algorithmus auch, sehr elegant geometrisch beschreiben, nämlich als Rotationen eines Vektors in einem zwei-dimensionalen Vektorraum. Eine weitere Veranschaulichung bietet sich, wenn wir einzelne Schritte des Orakels als *Inversion um den Mittelwert* beschreiben. Wie sich der Algorithmus konkret implementieren lässt, zeige ich anhand eines Beispiel mit $N = 4$. Dafür sind nur wenige elementare Gatter notwendig. Es schließt sich die Frage an, bei welchen praktischen Problemen uns der Grover-Algorithmus einen Vorteil gegenüber klassischen Suchverfahren verschafft. Wir werden sehen, dass dies vor allem Probleme der Theoretischen Informatik aus der Klasse NP-vollständig sind, zum Beispiel das bekannte Problem des Handlungsreisenden. Die Suche in klassischen Datenbanken ließe sich zwar auch beschleunigen, jedoch gibt es einige Einschränkungen, die den Einsatz des Grover-Algorithmus heutzutage noch unökonomisch machen.
In Kapitel 6 bespreche ich meine Simulation des Grover-Algorithmus auf einem klassischen Computer. Den entsprechenden Schaltkreis habe ich mit der objektorientierten Programmiersprache *Java* nachgebildet. Qubits und Gatter sind eigenständige Objekte, die miteinander interagieren. Das Programm basiert im wesentlichen auf Operationen mit Matrizen, die komplexe Zahlen enthalten. Da die Matrizengrößen sich mit jedem zusätzlichen Qubit verdoppeln, wird interessant zu untersuchen sein, wieviel Ressourcen (Speicher, Rechenzeit) die Simulation benötigt. Jedes Qubit im Register vervierfacht den Speicherbedarf und dementsprechend länger dauert die Berechnung. Die Laufzeit meines Programmes steigt exponentiell mit der Registerbreite. Das Kapitel schließt mit einem kurzen Vergleich meines Programmes mit dem *Mathematica*-Paket *QuCalc*, welches ebenfalls einen Quantencomputer simulieren kann. Auch dort ist der Ressourcenverbrauch stark von der Zahl der Qubits abhängig.

[1] Nuclear Magnetic Resonance

4 Quantencomputer

> *„Nobody understands quantum mechanics. No, you're not going to be able to understand it... You see, my physics students don't understand it either. That is because I don't understand it. Nobody does... The theory of quantum electrodynamics describes Nature as absurd from the point of view of common sense. And it agrees fully with experiment. So I hope you can accept Nature as She is — absurd.“*
> Richard Feynman

> *„For reasons which nobody fully understands, entangled states play a crucial role in quantum computation [...].“*
> Michael Nielsen and Isaac Chuang

4.1 Was ist ein Quantencomputer?

Ein Quantencomputer ist ein mikroskopisches System, dessen Verhalten von den Gesetzen der Quantenmechanik bestimmt wird. Dieses System enthält Informationen, die durch Zeitentwicklung verändert werden. Schließlich überführen wir die „Quanteninformation“ durch eine Messung in eine klassische Information. Durch das geschickte Ausnutzen quantenmechanischer Besonderheiten kann ein Quantencomputer bestimmte Berechnungen prinzipiell schneller ausführen als sein klassisches Pendant. Es können aber nur bestimmte Arten von Problemen schneller berechnet werden. Daher braucht man neue Algorithmen, Quantenalgorithmen (siehe Abschnitt 4.6), die die Besonderheiten des Quantencomputers ausnutzen. Die zwei wichtigsten sind der Shor-Algorithmus zur Faktorisierung von natürlichen Zahlen und der Grover-Algorithmus zur effizienten Suche in unstrukturierten Datenbanken. Die bei der Entwicklung dieser beiden Algorithmen benutzten Techniken bilden die Grundlage für viele andere Algorithmen.
Die beiden zentralen Besonderheiten der Quantenmechanik, die einen Quantencomputer schneller als jeden klassischen Computer machen, sind Superposition und Verschränkung. Diese Eigenschaften ermöglichen massiven Parallelismus und Effekte wie Quantenteleportation oder Quantenkryptografie.
Durch die Ausnutzung dieser Phänomene, die kein klassisches Analogon haben, sind Quantencomputer prädestiniert, quantenmechanische Systeme zu simulieren. Dies war auch die ursprüngliche Idee Richard Feynmans [Fey82], der sich in den 80er Jahren Gedanken darüber machte, wie Computern aussehen müssten, um diese Simulationen effizient durchführen zu können.
Ein Quantencomputer und ein klassischer Computer können sich gegenseitig simulieren, allerdings mit unterschiedlichem Aufwand. Da ein Quantencomputer das Repertoire eines klassischen Computers umfasst, kann er alle klassischen Algorithmen mit ähnlicher Effizienz berechnen. Umgekehrt lassen sich die erweiterten Fähigkeiten, die

sich durch Superposition und Verschränkung ergeben, auf einem klassischen Computer nur mit exponentiellem Aufwand abbilden.
Auch für einen Quantencomputer kennt man eine kleinste Informationseinheit: das Quantenbit, kurz Qubit, auf das in Abschnitt 4.2.2 näher eingegangen wird. Das Qubit kann jedoch exponentiell mehr Information tragen als das klassische Bit. Wie im klassischen Computer werden mehrere Qubits zu Registern zusammengefasst, auf denen man mit Quantengattern arbeitet. Diese Quantengatter entsprechen unitären Operatoren auf Hilberträumen, die wir in den Abschnitten 4.3 und 4.4 näher erklären. Man kann zeigen, dass bestimmte Sätze von Gattern universell sind [BBC$^+$95], das heißt, dass man jedes beliebige Gatter mit beliebiger Genauigkeit als Folge von Gattern aus dem universellen Satz darstellen kann. So ist es möglich, jeden Algorithmus durch einen universellen Satz zu implementieren, da er sich als Verkettung unitärer Operatoren darstellen lässt. Details zu Quantenschaltkreisen und deren Notation bietet Abschnitt 4.5.
Es gibt verschiedene Ansätze, Qubits und Gatter physikalisch zu realisieren: Ionenfallen (Qubits als Anregungszustände der Ionen, Gatter durch gezielte Manipulation mit Lasern) [CZ95], Nuclear Magnetic Resonance (Qubits als Kernspin, Gatter durch magnetische Felder) [Vin95] und Festkörpersysteme (Qubits durch Elektronenpaare, Gatter durch Elektrostatik) [NPT99, LD98]. Diese Verfahren stellen wir in Abschnitt 4.7 kurz dar.
Ein großes Problem bei dem Bau von Quantencomputern ist Dekohärenz: Die Interaktion des quantenmechanischen Systems mit seiner Umgebung stört die gespeicherte Quanteninformation und führt damit zu Informationsverlust. Die Wechselwirkung mit der Umgebung wirkt wie ein Messprozess, der Superposition und Verschränkung aufhebt. Die Güte eines Quantencomputers hängt entscheidend von der Zahl der möglichen Operationen ab, die durchgeführt werden können, bevor Dekohärenzeffekte auftreten.

4.2 Quantenbits

4.2.1 Formalismus

Quantenmechanische Zustände (wie z. B. der Zustand eines Quantenbits oder eines Quantenregisters) lassen sich durch Vektoren in einem Hilbertraum beschreiben. In dieser Arbeit wird die Dirac-Notation benutzt. Dabei wird ein Vektor aus dem betrachteten Hilbertraum „Ket“ genannt und mit $|\phi\rangle$ (sprich: Ket Phi) bezeichnet. Die Klammer $|\rangle$ deutet dabei den Vektorcharakter an, während der Name (hier ϕ) zur Unterscheidung verschiedener Vektoren dient. In endlich-dimensionalen Hilberträumen lässt sich ein Ket als Spaltenvektor mit im Allgemeinen komplexen Einträgen schreiben.
Das Dualraumäquivalent eines Kets $|\phi\rangle$ bezeichnet man mit $\langle\phi|$ und nennt es „Bra“. In endlich-dimensionalen Hilberträumen lässt sich ein Bra als Zeilenvektor schreiben. Als Element des Dualraumes ist ein Bra eine lineare Abbildung vom betrachteten Hilbertraum in die zugrunde liegenden komplexen Zahlen. Bildet man mit dem Bra

ein Ket ab, so schreibt man $\langle\phi|\psi\rangle$. Das Ergebnis ist also ein Skalar.
Eine wichtige Operation mit Kets ist das Tensorprodukt. Das Tensorprodukt verknüpft zwei Vektorräume U und V zu einem größeren Vektorraum $W = U \otimes V$. Eine Basis des Produkt-Hilbertraumes W erhält man durch das Tensorprodukt von jedem Basisvektor $|i\rangle$ aus U mit jedem Basisvektor $|j\rangle$ aus V.
Das Tensorprodukt für zwei Vektoren ist über ihre Indizes definiert. Wenn i ein Index für die Elemente ϕ_i vom Vektor ϕ ist und j ein Index für die Elemente ψ_j von ψ, dann ist ij ein Index für das Tensorprodukt. Dabei werden i und j aber nicht multipliziert, sondern als Ziffern angesehen. Wenn i und j beispielsweise von 0 bis 1 laufen, dann durchläuft ij die Werte 00, 01, 10 und 11, also vier Werte.
Damit ergibt sich die Formel:

$$(\phi \otimes \psi)_{ij} \equiv \phi_i \, \psi_j \quad . \tag{1}$$

Man schreibt auch $|ij\rangle$ anstatt $|i\rangle \otimes |j\rangle$. Jeder Basiszustand von $U \otimes V$ ist damit ein Produktzustand. Viele Superpositionen von Basiszuständen lassen sich nicht als Tensorprodukt schreiben, zum Beispiel $\frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$. Solche Zustände nennt man *verschränkt*. Sie spielen eine wichtige Rolle (siehe Zitat am Anfang des Kapitels) in der Quanteninformatik, vor allem in Bereichen wie Quantenkryptografie oder Quantenteleportation.
Entsprechend lässt sich das Tensorprodukt für Operatoren $A : U \to U$ und $B : V \to V$ definieren. Es wirkt einfach jeder Operator auf seinen Vektorraum: $(A \otimes B) : U \otimes V \to U \otimes V$. Damit gilt für $|a\rangle \in A$ und $|b\rangle \in B$:

$$(A \otimes B)(|a\rangle \otimes |b\rangle) \equiv (A\,|a\rangle) \otimes (B\,|b\rangle) \tag{2}$$

oder

$$(A \otimes B)_{i\nu,j\mu} \equiv A_{ij} B_{\nu\mu} \quad . \tag{3}$$

Da das Tensorprodukt von A und B auch auf Vektoren wirken kann, die sich nicht als Tensorprodukt von zwei Vektoren aus U und V darstellen lassen, muss man es noch linear fortsetzen.
Ein Beispiel: Seien

$$A = \begin{pmatrix} 2 & 0 \\ -1 & 1 \end{pmatrix}, B = \begin{pmatrix} 1 & 1 \\ -2 & -1 \end{pmatrix} \quad . \tag{4}$$

Dann ist das Tensorprodukt

$$(A \otimes B) = \begin{pmatrix} 2 \cdot B & 0 \cdot B \\ -1 \cdot B & 1 \cdot B \end{pmatrix} = \begin{pmatrix} 2 & 2 & 0 & 0 \\ -4 & -2 & 0 & 0 \\ -1 & -1 & 1 & 1 \\ 2 & 1 & -2 & -1 \end{pmatrix} \quad . \tag{5}$$

4.2.2 Vom Bit zum Qubit

In der klassischen Informatik ist die kleinste Informationseinheit das Bit. Ein Bit hat entweder den Zustand 0 oder 1, so dass sich mit n Bits 2^n Werte darstellen lassen.
In der Quanteninformatik gibt es als analoge kleinste Informationseinheit das Quantenbit oder Qubit [Sch95]. Ein Qubit ist ein quantenmechanischer Zustand in einem zweidimensionalen Hilbertraum. Dieser wird von zwei Basiszuständen aufgespannt, die man üblicherweise mit $|0\rangle$ und $|1\rangle$ bezeichnet. Sie bilden die sogenannte Rechenbasis (bei Registern, die aus mehreren Qubits bestehen, heißt die Basis $\{|00...0\rangle, |00...1\rangle, \cdots, |11...1\rangle$ Rechenbasis). Ein Qubit mit dem Zustand $|0\rangle$ (bzw. $|1\rangle$) ist vergleichbar mit dem klassischen Zustand 0 (bzw. 1), allerdings kann ein Qubit auch in einem Superpositionszustand dieser beiden Basiskets sein. Dabei ist jede Linearkombination $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$ erlaubt mit $|\alpha|^2 + |\beta|^2 = 1$. α und β sind komplexe Zahlen. Die Quanteninformatik interpretiert eine Superposition von $|0\rangle$ und $|1\rangle$ als *gleichzeitige* Speicherung von 0 und 1. Ein Qubit ist damit nicht wie das klassische Bit entweder im Zustand 0 oder im Zustand 1, sondern in beliebiger Überlagerung.
Im Gegensatz zum klassischen Bit, bei dem es jederzeit durch eine Messung möglich ist, seinen Zustand (0 oder 1) genau zu bestimmen, lässt sich bei einem Qubit sein Quantenzustand, also die Werte von α und β, nicht feststellen. Man kann genau eine Messung durchführen, das Ergebnis ist zufällig. Dies bedeutet, dass man den Zustand $|0\rangle$ mit der Wahrscheinlichkeit $|\alpha|^2$ und den Zustand $|1\rangle$ mit der Wahrscheinlichkeit $|\beta|^2$ misst. Da die Messung den ursprünglichen Quantenzustand auf den gemessenen Zustand projiziert, ist die gespeicherte Quanteninformation also nicht komplett als klassische Information verfügbar, was zur Folge hat, dass wir mit versteckten Informationen arbeiten, die wir nie (durch Beobachtung oder Messung) zu Gesicht bekommen. Gute Quantenalgorithmen müssen also möglichst ohne Zwischenmessungen auskommen, da bei Messungen Quanteninformation verloren geht und damit der Vorteil des Quantencomputers gegenüber dem klassischen Computer.
Als Beispiel betrachten wir den Zustand $|\phi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. Die Wahrscheinlichkeit, den Zustand $|0\rangle$ zu messen beträgt also

$$\left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2} = 50\%.$$

Die Wahrscheinlichkeit für $|1\rangle$ ist genauso groß.

4.3 Operationen auf Qubits

Möchte man mit Qubits arbeiten, dann benötigt man genau wie im klassischen Fall Operationen, die Qubits manipulieren (Gatter [BBC+95], [BCDP96]). Das Ziel ist, den durch die Qubits repräsentierten Anfangszustand in einen Zustand zu überführen, aus dessen Messung wir ein sinnvolles Ergebnis bekommen.

4.3.1 Ein-Qubit-Gatter

Zunächst soll nur ein einziges Qubit betrachtet werden, sozusagen der einfachste Quantencomputer, den man sich denken kann. Wie können wir dieses Qubit manipulieren? Klassische Computer arbeiten mit Gattern, ein triviales Beispiel ist das NOT-Gatter X, welches jedes Bit invertiert. Aus 0 wird 1, aus 1 wird 0. Wie man sich leicht vorstellen kann, existiert eine analoge Operation für Qubits. Aus dem Zustand $|0\rangle$ wird der Zustand $|1\rangle$ und umgekehrt. Im Falle einer Superposition zweier Zustände, z. B. dem Zustand $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$, arbeitet das Quanten-NOT-Gatter linear, das heißt in diesem Fall, die Rollen von $|0\rangle$ und $|1\rangle$ werden vertauscht: $|\phi'\rangle = X|\phi\rangle = \alpha|1\rangle + \beta|0\rangle$. Diese Linearität ist eine wichtige Eigenschaft in der Quanteninformatik. Verallgemeinert können wir die NOT-Operation als $X|a\rangle = |a \oplus 1\rangle$ für einen beliebigen Basiszustand $|a\rangle$, $a \in \{0, 1\}$, schreiben, wobei $\oplus$ die Addition modulo 2 meint. Mathematisch lässt sich eine solche Operation, nennen wir sie U, durch eine 2×2-Matrix repräsentieren. Damit ist sicher gestellt, dass es sich um eine lineare Operation handelt. Weiterhin muss die Normierungsbedingung $|\alpha|^2 + |\beta|^2 = 1$ auch nach der Matrixoperation gelten, weswegen die Matrix unitär sein muss. Es muss also gelten $U^\dagger U = I$. Dies ist die einzige Bedingung, die eine Matrix erfüllen muss, um ein erlaubtes Quantengatter darzustellen. So gibt es theoretisch unendlich viele Gatter für einzelne Qubits.
Eine weitere Folge dieser einen Bedingung ist die Umkehrbarkeit der Operation, das heißt, im Gegensatz zu klassischen Gattern ist ein Quantengatter immer reversibel. Im klassischen Fall gibt es zum Beispiel die Gatter XOR oder NAND, die nicht invertierbar sind und somit zu einem Informationsverlust führen. Dies ist bei Quantengattern nicht möglich. Die Gesamtwahrscheinlichkeit des Systems muss vor und nach einer Operation gleich sein, was zusammen mit der Linearitätsbedingung nur umkehrbare Operatoren zulässt. Dieser Punkt bedarf besonderer Beachtung, wenn man versucht, klassische Algorithmen und Schaltlogik auf Quantencomputer zu übertragen.
Wenn man Hilfs-Qubits einführt, kann man allgemein auch nicht reversible Funktionen in einem Quantenschaltkreis berechnen. Dazu benötigt man zwei n-Qubit-Register. Die Funktion sei $f : \{0,1\}^n \to \{0,1\}^n$. Dann ist die Abbildung $U_f : |x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle$ unitär, also auf einem Quantencomputer ausführbar, und für $|y\rangle = |0 \ldots 0\rangle$ erhält man den Funktionswert im zweiten Register.
Bezogen auf unser Beispiel können wir die Matrix, die das NOT-Gatter darstellt, wie folgt definieren:

$$X \equiv \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{6}$$

oder in Dirac-Notation:

$$X = |0\rangle\langle 1| + |1\rangle\langle 0| \quad . \tag{7}$$

Zur Verdeutlichung schreiben wir den Zustand ϕ in Vektorschreibweise als

$$\phi = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}.$$

Wenden wir X auf diesen Vektor an, erhalten wir

$$X \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}.$$

Dass X unitär ist, lässt sich leicht erkennen.
X ist eine von vier Pauli-Matrizen, die zu den wichtigsten Ein-Qubit-Quantengattern gehören:

$$I \equiv \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}; \; X \equiv \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}; \; Y \equiv \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}; \; Z \equiv \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad . \tag{8}$$

Die verwendete grafische Darstellung orientiert sich an [NC00] und entspricht damit der gängigen Notation in der Fachliteratur.

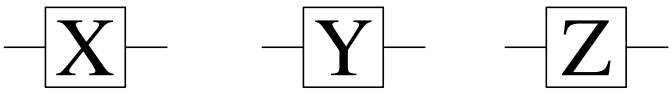


Abbildung 1: Die drei wichtigen Pauli-Matrizen in der für Quantenschaltkreise üblichen Notation

Daneben spielen noch drei weitere Matrizen eine entscheidende Rolle in der Quanteninformatik: das Hadamard-Gatter H, das Phasen-Gatter S und das $\pi/8$-Gatter T:

$$H \equiv \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}; \; S \equiv \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}; \; T \equiv \begin{pmatrix} 1 & 0 \\ 0 & \exp(i\pi/4) \end{pmatrix} \quad . \tag{9}$$

Dabei ist anzumerken, dass T historisch bedingt $\pi/8$-Gatter heißt, obwohl in der Definition $\pi/4$ auftaucht (Wenn man $e^{i\pi/8}$ aus der Matrix ausklammert, so taucht auch in der Matrix in den Exponenten $1/8$ auf).

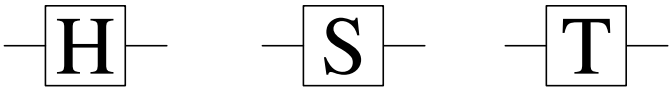


Abbildung 2: Von links nach rechts: Symbol für das Hadamard-Gatter, das Phasen-Gatter und das $\pi/8$-Gatter.

4.3.2 Mehr-Qubit-Gatter

Weil niemand einen Quantencomputer mit nur einem Qubit haben möchte, gibt es Quantengatter, die auf mehreren Qubits arbeiten. Auch diese werden durch unitäre $2^n \times 2^n$ Matrizen repräsentiert, wobei n die Anzahl der Qubits ist. Der Anfangszustand wird hierbei mathematisch durch das Tensorprodukt aller Qubits repräsentiert. Am Beispiel des CNOT-Gatters (Abbildung 3), welches dem klassischen XOR entspricht, soll das Manipulieren mehrerer Qubits verdeutlicht werden.
CNOT steht für *controlled*-NOT, was bedeutet, dass es bedingte Operationen nach dem Schema „Wenn x wahr ist, tue y“ ausführt. Das CNOT-Gatter bekommt zwei

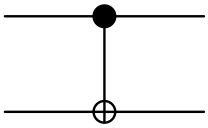


Abbildung 3: Das CNOT-Gatter. Die obere Linie repräsentiert das Kontroll-Qubit c, die untere Linie das Ziel-Qubit q. Ist c auf 1 gesetzt, wird q invertiert. Die Notation geht auf Feynman zurück [Fey85].

Qubits als Eingänge, ein *Kontroll*-Qubit und ein *Ziel*-Qubit. Das Kontroll-Qubit wird dabei nicht verändert. Das Ziel-Qubit wird geflippt (invertiert), falls das Kontroll-Qubit auf 1 gesetzt ist. Andernfalls (Kontroll-Qubit auf 0) wird nichts verändert.

$$|00\rangle \rightarrow |00\rangle$$
$$|01\rangle \rightarrow |01\rangle$$
$$|10\rangle \rightarrow |11\rangle$$
$$|11\rangle \rightarrow |10\rangle$$

Verkürzt kann man die Wirkung des Gatters auch durch $|x, y\rangle \rightarrow |x, y \oplus x\rangle$ beschreiben. Dabei meint das Symbol $\oplus$ eine Addition modulo 2. Das CNOT-Gatter stellt somit eine Verallgemeinerung des klassischen XOR-Gatters dar. In Matrixform schreibt sich CNOT wie folgt:

$$U_{CNOT} \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Man kann leicht zeigen, dass U_{CNOT} unitär ist, so dass die Normierungsbedingung auch hier eingehalten wird. Und wie alle Quantengatter ist U_{CNOT} natürlich reversibel, das heißt, von den Ausgangszuständen kann man immer auf die Eingangszustände rückschließen.
An diesem Beispiel zeigt sich auch das Phänomen der Verschränkung. Die zwei Qubits sind nach der CNOT-Operation verschränkt, sie lassen sich nicht mehr als Tensorprodukt der Zustände der Einzel-Qubits schreiben.
Natürlich kann nicht nur die NOT-Operation bedingt ausgeführt werden, sondern jede unitäre Operation U. Man spricht dann von einer *controlled-U* Operation, bei welcher ein Kontroll-Qubit bestimmt, ob U auf das Ziel-Qubit angewandt wird.
Ein-Qubit-Gatter U lassen sich zerlegen in

$$U = \exp(i\alpha) AXBXC \tag{10}$$

mit A, B und C als beliebige unitäre Operatoren, die $ABC = I$ erfüllen. Dadurch lässt sich die *controlled-U* Operation durch Ein-Qubit-Gatter und zwei CNOT-Operationen darstellen (Abbildung 4).
Dabei muss das Kontroll-Qubit nicht zwangsläufig auf 1 gesetzt sein, damit U auf dem Ziel ausgeführt wird. Genauso gut ist es möglich, dass die 0 die bedingte Operation

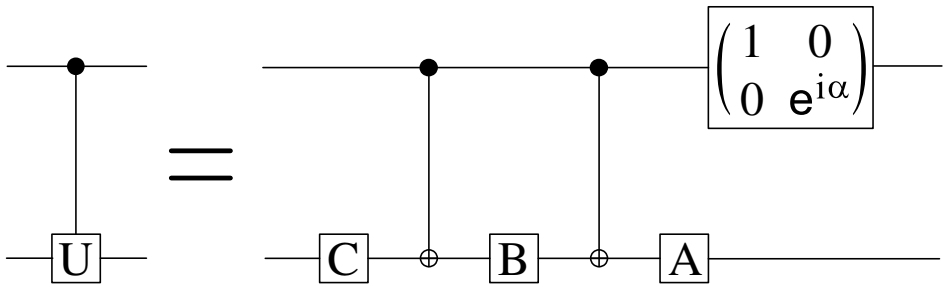


Abbildung 4: Zerlegung einer beliebigen bedingten Operation U in Ein-Qubit-Gatter A, B, C mit $ABC = I$ und CNOT-Gatter.

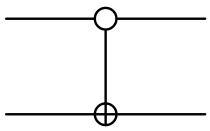
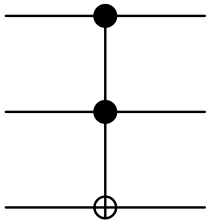


Abbildung 5: Symbol für das CNOT-Gatter, bei dem das Ziel-Qubit invertiert wird, wenn das Kontroll-Qubit auf 0 gesetzt ist.

auslöst. Für beide Möglichkeiten gibt es unterschiedliche Notationen (Abbildung 5, nach [NC00]).
Außerdem kann es sein, dass U von mehr als einem Kontroll-Qubit abhängt, so zum Beispiel beim Toffoli-Gatter (Abbildung 6), wo beide Kontroll-Qubits gesetzt sein müssen, damit U auf das Ziel-Qubit angewandt wird. Anders herum kann ein Kontroll-Qubit auch mehrere Ziel-Qubits steuern (Abbildung 7).

Abbildung 6: Symbol für das Toffoli-Gatter



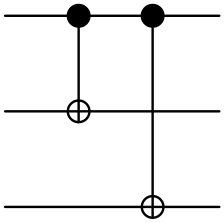


Abbildung 7: Ein Kontroll-Qubit steuert mehrere Ziel-Qubits.

4.4 Gatter als unitäre Operationen

In der klassischen Informatik betrachtet man n-Bit-Register und darauf arbeitende Abbildungen. Eine solche Abbildung ordnet jedem möglichen Zustand des Eingangsregisters einen Zustand des Ausgaberegisters zu. Da man die beiden möglichen Zustände des Bits als *wahr* und *falsch* deuten kann, nennt man die Abbildungen logische oder Boolsche Funktionen. Sowohl Eingabe- als auch Ausgaberegister können aber nur endlich viele Zustände annehmen. Es gibt damit auch nur endlich viele Boolsche Funktionen für ein gegebenes Register.
Ein wichtiger Satz ist nun, dass es Sätze von einfachen Boolschen Funktionen gibt (normalerweise Funktionen die von einem Ein- oder Zwei-Bit-Register auf ein Ein-Bit-Register abbilden), die ausreichen, um damit alle anderen Funktionen zusammenzusetzen.

Die Nicht-Und (NAND) Funktion ist durch die Wahrheitstabelle in 1 gegeben.

Tabelle 1: Die klassische NAND-Verknüpfung

Bit 1	Bit 2		Ergebnis
0	0	→	1
0	1	→	1
1	0	→	1
1	1	→	0

Es lässt sich nun zeigen, dass allein dieses NAND-Gatter ausreicht, um alle anderen Boolschen Funktionen zu erzeugen. In dieser Hinsicht ist das NAND-Gatter also ein Erzeugendensystem der Menge aller Boolscher Funktionen.
Eine gegebene Boolsche Funktion nur durch NAND-Gatter darzustellen kann aber ungünstig sein. Welche Gatter man nutzt, um daraus größere Gatter zu bauen, spielt auch für Quantencomputer eine Rolle und wird später diskutiert.
Wie oben schon erwähnt sind die möglichen Operationen auf Qubit-Gattern genau die Operationen, die sich durch unitäre Abbildungen auf Hilberträumen darstellen lassen. Aber selbst für den Fall von Ein-Qubit-Gattern gibt es unendlich viele solcher Operationen, zumal die Menge der möglichen Zustände eines einzelnen Qubits schon überabzählbar ist.
Umso erstaunlicher ist es, dass es auch hier universelle Sätze gibt [BMP$^+$99], aus denen sich jeweils alle möglichen Gatter erzeugen lassen. Dabei muss man allerdings gewisse Einschränkungen in Kauf nehmen.
Die wichtigste Überlegung ist zunächst, dass sich jede unitäre Operation als Verkettung von einfachen unitären Operationen schreiben lässt. Dabei meint *einfach*, dass die Operation maximal zwei Qubits verändert. Um also Operationen auf beliebig vielen Qubits durchführen zu können, braucht man nach dieser Überlegung nur alle Ein- und Zwei-Qubit-Operationen. Dies sind aber immer noch unendlich viele.
Die zweite Überlegung ist, dass man nur ein Zwei-Qubit-Gatter braucht, quasi um Informationen zwischen Qubits auszutauschen, und ansonsten nur alle weiteren Ein-Qubit-Gatter. Dazu kann man aber nicht jedes Zwei-Qubit-Gatter verwenden. Es muss ein Zwei-Qubit-Gatter sein, welches sich nicht als Tensorprodukt von Ein-Qubit-Gattern darstellen lässt. Leider hat man damit immer noch unendlich viele Gatter.
Nun gibt es zwei Möglichkeiten der Vereinfachung. Die erste stützt sich darauf, dass sich jede unitäre 2×2-Matrix (bis auf eine Phase) exakt durch die Verkettung von Rotationsmatrizen darstellen lässt [NC00]. Der Winkel, um den bei jeder Rotation gedreht wird, ist dabei ein Parameter. Wenn man also Gatter für die Rotationen sowie die Phase hat, die jeweils durch einen kontinuierlichen Parameter anpassbar sind, so hätte man zumindest im Prinzip endlich viele Gatter. Aber natürlich ist ein kontinuierlicher Parameter in der Realität nur näherungsweise möglich. Je nach Genauigkeit wird eine beliebige unitäre Operation also nur approximiert.

Die zweite Möglichkeit ist, einen geschickten Satz von festen, also parameterunabhängigen Gattern zu nutzen. Dieser lässt sich so wählen, dass sich jede beliebige Ein-Qubit-Operation mit beliebiger Genauigkeit approximieren lässt. Ob sich aber für jede Operation die gewünschte Genauigkeit effizient, also mit polynomiell anwachsendem Ressourcenaufwand, erreichen lässt, ist dadurch nicht gesagt.
Ein solcher Satz ist zum Beispiel $\{H, T, CNOT\}$ mit CNOT als Zwei-Qubit-Gatter. Die Vollständigkeit dieses Gatter-Satzes wurde in [BMP+99] bewiesen.

4.5 Schaltkreise

So wie es für klassische Computer grafische Notationen zum Beschreiben von Schaltkreisen oder Gattern gibt, so benutzt man eine äquivalente Notation für Quantencomputer, die auf Feynman [Fey85] zurückgeht. Auch hier gibt es „Drähte“ und logische Gatter, welche die durch den Schaltkreis transportierten Informationen verändern. Ein Beispiel für eine solche grafische Repräsentation eines Quantenschaltkreises ist in Abbildung 8 zu sehen. Es handelt sich dabei um einen Schaltkreis, der mit zwei Qubits arbeitet und deren Zustände vertauscht,

$$|ab\rangle \rightarrow |ba\rangle \,.$$

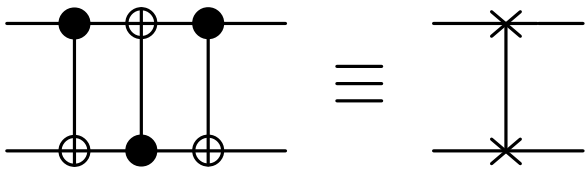


Abbildung 8: Quantenschaltkreis, der zwei Qubits vertauscht, mit einer alternativen Notation

Entsprechend gibt es zwei horizontale Linien, die die „Drähte“ darstellen sollen. Wir benutzen die Anführungszeichen um zu verdeutlichen, dass dies keine Drähte im herkömmlichen physikalischen (oder elektrotechnischen) Sinn sein müssen. Durch die Linien soll vielmehr veranschaulicht werden, dass sich dort eine Information in dem Schaltkreis entwickelt bzw. verändert. Es könnte zum Beispiel ein zeitlicher Verlauf dargestellt sein oder die Bewegung eines Teilchens (z. B. eines Photons), das ein Qubit repräsentiert.
Die Abbildung, oder besser gesagt der Schaltkreis, muss von links nach rechts gelesen werden. Links ist also der Anfangszustand (Input) und rechts der Endzustand (Output). Der Anfangszustand kann links angegeben sein, muss es aber nicht. Bei vielen Quantenschaltkreisen ist der Anfangszustand $|00\cdots0\rangle$, da es zu den Forderungen an einen Quantencomputer gehört [Div95], dass er in diesem Zustand initialisierbar ist.
Unser Beispiel zeigt drei CNOT-Operationen auf den zwei Qubits. Das Kontroll-Qubit ist dabei durch den dicken schwarzen Punkt angedeutet, das Ziel-Qubit dementsprechend durch den Kreis, der das Plus-Zeichen umschließt. Wie im Abschnitt 4.3.2 bereits erwähnt, lassen sich beliebige unitäre Operationen bedingt ausführen. Die Notation ändert sich dabei nicht, man ersetzt lediglich das eingekreiste Plus-Zeichen durch einen

quadratischen Kasten mit dem Buchstaben 'U' darin (Abbildung 9). Ein CNOT ist also nur ein Spezialfall mit $U = X$.

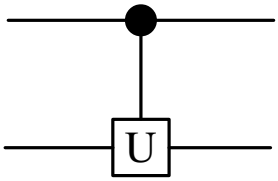


Abbildung 9: Symbol für eine beliebige bedingte unitäre Operation U

Darüberhinaus gibt es ein letztes wichtiges Symbol in Quantenschaltkreisen, nämlich das für die Messung. Diese macht aus einem Qubit ein klassisches Bit. War das Qubit vor der Messung im Zustand $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$, dann ist das klassische Bit nach der Messung mit der Wahrscheinlichkeit $|\alpha|^2$ im Zustand 0 und mit der Wahrscheinlichkeit $|\beta|^2$ im Zustand 1. Zur Verdeutlichung, dass es sich um ein klassisches Bit handelt, werden zwei horizontale Linien gezeichnet, die aus dem Messoperator hinausführen (Abbildung 10). Da die Messung die quantenmechanischen Informationen zerstört und somit nicht reversibel ist, steht sie im Allgemeinen am Ende eines Quantenschaltkreises.
Dazu sind zwei Prinzipien wichtig: Das *Prinzip der aufgeschobenen Messung* sagt, dass man Messungen immer erst am Ende durchführen kann, anstatt sie innerhalb des Quantenschaltkreises durchzuführen. Sollen Messinformationen den weiteren Verlauf des Schaltkreises bestimmen, so kann man diesen (klassischen) Einfluss durch kontrollierte Quantengatter ersetzen.
Das *Prinzip der impliziten Messung* besagt, dass jedes Qubit am Ende eines Schaltkreises, welches nicht explizit gemessen wird oder werden muss, dennoch als gemessen angenommen werden kann.

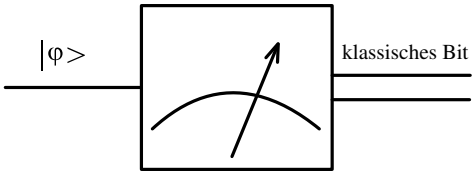


Abbildung 10: Symbol für einen Messvorgang am Quantenschaltkreis

Einige aus klassischen Schaltkreisen bekannten Eigenschaften können in Quantenschaltkreisen nicht benutzt werden, weil wir immer beachten müssen, dass die Operationen reversibel sein müssen. Eine Anweisung wie „$a = 5$“ ist nicht möglich, da wir danach nicht mehr auf den ursprünglichen Wert der Variable a schließen können. Im Gegensatz dazu ist die Anweisung „$a = a + 5$“ erlaubt, weil wir hier aus dem Wert nach der Addition den Anfangswert rekonstruieren können.
Desweiteren verbietet das *no-cloning Theorem* das Kopieren von Qubits [Die82], während das in klassischen Schaltkreisen an der Tagesordnung ist.

4.6 Algorithmen

Die ersten Entwicklungen im Bereich der Algorithmik liegen weit in der Vergangenheit: Keilschriftfunde belegen, dass schon etwa 1750 v. Chr. bei den Babyloniern fortgeschrittene Algorithmen bekannt waren. Doch erst 1936 wurde dieses Gebiet auf ein festes Fundament gestellt, als Alan Turing die abstrakte Turingmaschine vorschlug.
In seiner Arbeit zeigte er, dass es eine universelle Turingmaschine gibt, die jede andere Turingmaschine simulieren kann. Zusätzlich behauptete er, dass die universelle Turingmaschine exakt das berechnen kann, was mit algorithmischen Prinzipien berechenbar ist (Church-Turing-These). Das bedeutet, dass wenn ein Algorithmus auf irgendeinem Gerät berechenbar ist, dann sollte es einen äquivalenten Algorithmus für die universelle Turingmaschine geben.
Beobachtungen an analogen Computern und an probabilistischen Systemen führten zu einigen Modifikationen der Church-Turing-These, aber erst David Deutsch [Deu85, Deu89] schaffte es, eine Formulierung zu finden, die nicht nur auf Beobachtungen basierte, sondern sich auf physikalische Theorien stützt.
Dazu ersann er einen Computer, der beliebige physikalische Systeme effizient simulieren kann, indem er quantenmechanische Prinzipien nutzt. Dieses Konzept eines Quantencomputers, eine Art Quantenanalogon zur Turingmaschine, könnte prinzipiell Algorithmen ermöglichen, die weder auf einer normalen noch auf einer probabilistischen Turingmaschine effizient berechnet werden können, aber auf einem Quantencomputer schon.
Er schlug auch einen Algorithmus vor, der schneller funktioniert als jeder bekannte klassische Algorithmus. Doch erst die Entdeckung des Shor-Algorithmus [Sho94], der anders als der Deutsch-Josza-Algorithmus praktische Relevanz hat, führte zu allgemeinem Interesse an dem Konzept des Quantencomputers.
Es ist jedoch problematisch, definitive Aussagen über Algorithmen zu treffen. Denn obwohl man keinen klassischen Algorithmus kennt, der so effizient ist wie der Deutsch-Josza-Algorithmus, ist nicht für alle Zeiten ausgeschlossen, dass irgendwann einer gefunden wird. Bisher ist weder die Existenz noch die Nicht-Existenz bewiesen.
Man kann auch nicht sagen, dass es Quantenalgorithmen in großer Anzahl gibt oder regelmäßig neue gefunden werden. Denn ein sinnvoller Quantenalgorithmus muss sich der sehr anti-intuitiven Quantenmechanik bedienen und zudem noch besser sein als jeder bekannte klassische. Niemand weiß, für welche Art von Problemen sich eine solche Suche lohnt, zumal auch niemand weiß, ob das Konzept des Quantencomputers dem Konzept der Turingmaschine wirklich überlegen ist.
Fest steht nur, dass zum Zeitpunkt der Erstellung dieser Arbeit im Wesentlichen drei Bereiche bekannt sind, in denen Quantencomputer Vorteile gegenüber aktuellen klassischen Verfahren bieten: das Problem der versteckten Untergruppe [Sim94, Sim97], das Problem der Suche in unstrukturierten Daten [Gro97] und die Simulation von quantenmechanischen Systemen.
Das Problem der versteckten Untergruppe besteht darin, ein Erzeugendensystem einer Untergruppe (die versteckte Untergruppe) zu finden, auf der eine bestimmte Funktion konstant ist. Diese sehr abstrakte Formulierung umfasst sehr viele Probleme, unter

Tabelle 2: Übersicht über verschiedene Ansätze zur Realisierung von Quantencomputern. Aus: [NC00], Seite 278. Legende: Dekohärenzzeit τ_Q in Sekunden, Operationszeit τ_{Op} in Sekunden, maximale Anzahl von Operationen n_{Op}, geschätzte Werte

System	τ_Q	τ_{Op}	$n_{Op} = \frac{\tau_Q}{\tau_{Op}}$
Kernspin	$10^{-2} - 10^{8}$	$10^{-3} - 10^{-6}$	$10^{5} - 10^{14}$
Elektronenspin	10^{-3}	10^{-7}	10^{4}
Ionenfalle	10^{-1}	10^{-14}	10^{13}
Elektronen-Au	10^{-8}	10^{-14}	10^{6}
Elektronen-GaAs	10^{-10}	10^{-13}	10^{3}
Quanten-Punkt	10^{-6}	10^{-9}	10^{3}
Optische Kavität	10^{-5}	10^{-14}	10^{9}
Mikrowellenkavität	10^{0}	10^{-4}	10^{4}

anderem die Faktorisierung von natürlichen Zahlen (Shor-Algorithmus). Bei diesen Problemen kann eine Verbesserung von exponentieller zu polynomieller Komplexität erreicht werden. Komplexität beschreibt den Aufwand an Zeit und Hardware, den ein Algorithmus für seine Ausführung benötigt. Die Komplexitätsklasse beschreibt, wie dieser Aufwand von der Bitbreite der Eingabe abhängt.
Das Problem der Suche in unstrukturierten Daten wird durch den Grover-Algorithmus gelöst, der sich eines *Orakels*[2] bedient. Dabei wird die klassische lineare Komplexität zu Quadratwurzel-Komplexität verbessert. Es wurde gezeigt, dass bei einem Orakelbasiertem Algorithmus keine Verbesserung über diese Komplexität hinaus möglich ist [BBBV97].
Bei der Simulation von quantenmechanischen Systemen besteht das Problem, dass noch nicht bewiesen wurde, ob sich wirklich jedes System simulieren lässt. Für die Systeme, für die es aber schon gezeigt wurde, ergibt sich eine exponentielle Verbesserung des Ressourcenaufwandes, da die Dimension der Qubit-Hilberträume genauso schnell wächst wie die Dimension von zu simulierenden Systemen.

4.7 Physikalische Realisierung von Quantencomputern

Die Quanteninformatik ist als theoretisches Themengebiet schon enorm spannend und lehrreich, da sie eine tiefgreifende Verknüpfung von Physik und klassischer Informatik aufzeigt. Dennoch stellt sich natürlich die Frage, ob sich ein Quantencomputer tatsächlich realisieren lässt, und wenn ja, wie.
Da es die verschiedensten quantenmechanischen Systeme gibt, gibt es entsprechend viele Ansätze für die Realisierung von Quantencomputern. Einen Überblick liefert Tabelle 2 aus [NC00].
Um als Quantencomputer geeignet zu sein verlangt man im Allgemeinen fünf Eigenschaften von einem System, die auf [Div95] zurückgehen:

[2]Begriff aus der Informationstheorie. Ein Orakel ist eine Black Box, die eine Funktion in einem Schritt berechnet.

- **Qubits**: Das System muss in der Lage sein, Quanteninformation zu speichern, das bedeutet Information mit der Möglichkeit zur Superposition und Verschränkung. Dazu ist vor allem wichtig, dass der Hilbertraum des Systems endlich ist, damit man die einzelnen Zustände unterscheiden kann. Auch muss die Dekohärenzzeit lang sein, damit möglichst viele Operationen durchführbar sind, bevor die Quanteninformation verloren geht.

- **Gatter**: Es muss möglich sein, die Zeitentwicklung des Systems zumindest so genau steuern zu können, dass die elementaren Gatteroperationen (H, T, CNOT) ausgeführt werden können. Damit lässt sich jede Zeitentwicklung approximieren, also jede Quantenrechnung durchführen.

- **Anfangszustand**: Das System muss sich in einen definierten Anfangszustand bringen lassen.

- **Messung**: Die Messung des Endzustands muss die möglichen Ergebnisse mit den quantenmechanischen Wahrscheinlichkeiten liefern. Denn nur so lassen sich Algorithmen, die probabilistisch arbeiten, sinnvoll ausführen.

- **Skalierbarkeit**: Die Schwierigkeit, Gatter und Messungen durchzuführen, darf nicht zu stark von der Anzahl der Qubits abhängen, da ansonsten die Vorteile der Quantenalgorithmen verloren gehen.

Im Laufe der letzten Jahre haben sich im Wesentlichen drei Ansätze als erfolgsversprechend herausgestellt, die im Folgenden näher erläutert werden: Quantenoptische Systeme, Kernmagnetische Resonanz (NMR) und Festkörpersysteme.

4.7.1 Quantenoptische Systeme

Bei diesen Systemen werden die Qubits durch Atome oder Ionen repräsentiert. Die beiden Basiszustände der Qubits sind zwei Energieniveaus oder Spinzustände. Durch die präzise Nutzung von Lasern werden diese Zuständen manipuliert und damit Gatteroperationen implementiert. Messungen werden durch Laser und anschließende Fluorenszensmessung realisiert.
Diese Systeme erreichen eine enorme Präzision. Quantenmechanische Effekte lassen sich sehr gut beobachten, weil die Wechselwirkung mit der Umgebung auf ein Minimum reduziert ist. In vielen Experimenten (zum Beispiel mit Ionenfallen [CZ95]) wurden erfolgreich Gatteroperationen mit mehreren Qubits demonstriert.
Das große Problem bei diesen Systemen ist, dass sie nur sehr schlecht skalierbar sind, da die verschiedenen Energieniveaus immer dichter zusammenrücken und nur noch sehr schwer unterschieden werden können. Rechnungen mit mehr als nur ein paar Qubits sind noch nicht durchgeführt worden.
Einen Ausweg bieten so genannte Quantennetzwerke [KMW02], die verschiedene kleine Einheiten mit wenigen Qubits so verbinden, dass das ganze wie ein großes System mit vielen Qubits wirkt.

4.7.2 Kernmagnetische Resonanz

In diesem Verfahren werden Qubits durch den Kernspin von Atomen in großen Molekülen repräsentiert. Man verwendet Kernspins mit $s = \frac{1}{2}$, so dass der Hilbertraum zweidimensional ist. Durch Radiowellen bestimmter Frequenz und Dauer kann man die Kernspins manipulieren und auf diese Weise den Qubitzustand verändern [Vin95]. Die dafür notwendigen Apparate sind (z. B. in der Medizin und Chemie) bereits vorhanden und ausgereift. Normalerweise betrachtet man ein ganzes System von Molekülen (Größenordnung 10^{25}) und nicht nur ein einzelnes.
Durch starke Magnetfeldpulse werden die Spins manipuliert (NMR) und damit Gatteroperationen implementiert. Durch entsprechende Vorgänge können die Spins wieder ausgelesen, also gemessen werden.
Der Anfangszustand muss in einem NMR-System nicht unbedingt ein reiner Zustand sein. Statt dessen kann es auch ein „pseudoreiner“ Zustand sein. Das bedeutet gemischt, aber mit starken Einschränkungen, auf die in [CFH97] eingegangen wird.
Dadurch ist der Endzustand im Allgemeinen kein reiner Zustand. Messungen an einem gemischten Zustand liefern aber nur Mittelwerte, was bei bestimmten Anwendungen (zum Beispiel bei der Kettenbruchentwicklung beim Shor-Algorithmus) zu Problemen führt. Prinzipiell ist das Auslesen, auch praktisch, möglich.
Die Messung wird zudem immer schwieriger, je mehr Qubits man hat. Daher ist eine Skalierung nur sehr begrenzt möglich.

4.7.3 Festkörpersysteme

Bei diesem System betrachtet man Elektronenpaare (Cooperpaare [NPT99] oder Quantenpunkte [LD98]) in Halbleitern. Ein Qubit ist ein solches Paar. Durch kontrollierte Coulomb-Wechselwirkungen (und andere Effekte) können Operationen auf Qubits durchgeführt werden und Messungen vorgenommen werden (heutige Feldeffekttransistoren können die Bewegung einzelner Ladungen detektieren).
Dieser Ansatz ist prinzipiell vielversprechend, da ein solches System sehr gut skalierbar wäre. Hinzu kommt, dass die Mikroelektronik schon durch die Entwicklung bei klassischen Computern sehr ausgereift ist. Nachteilig ist jedoch, dass eine Abkopplung von der Umgebung prinzipiell nicht möglich ist, da sich das Geschehen in einem Festkörper abspielt. Damit ist die Dekohärenzzeit sehr kurz.

4.7.4 Zusammenfassung

Die betrachteten Ansätze haben alle vielversprechende Eigenschaften, aber auch teils gravierende praktische Nachteile, die bis heute die Entwicklung eines brauchbaren Quantencomputers verhindert haben. Brauchbar meint, dass ausreichend viele Qubits vorhanden sind, um komplexere Aufgaben zu verrichten, und dass das System skalierbar ist.
Im Jahr 2001 hat IBM einen 7-Qubit-Quantencomputer auf der Basis von NMR vorgestellt [VSB+01]. Dieses System konnte den Shor-Algorithmus für die Zahl 15 ausführen.

Bis heute ist es nicht gelungen, ein System mit wesentlich mehr Qubits zu konstruieren, auf dem noch sinnvoll gerechnet werden kann.
Auf Grund der offensichtlichen Schwierigkeiten vertreten viele Wissenschaftler die Meinung, dass ein nutzbarer Quantencomputer nicht möglich ist und damit niemals gebaut wird. Andere Wissenschaftler lassen sich eher von den bisherigen Erfolgen beflügeln und meinen, dass in absehbarer Zukunft ein Quantencomputer gebaut werden kann, zumal es keine prinzipiellen Hindernisse gibt, nur praktische. Ob es allerdings so weit kommt, dass jeder, der heute einen klassischen Computer auf dem Schreibtisch stehen hat, zukünftig einen Quantencomputer besitzt, ist mehr als fraglich. Die Stärken von Quantencomputern liegen eindeutig im wissenschaftlichen Bereich, wo die enorme Leistungsfähigkeit zum Tragen kommt. Klassische Computer werden ihre heutigen Aufgaben weitestgehend behalten, nur in Spezialbereichen wie der Faktorisierung oder der Datenbanksuche, wo Quantencomputer grundlegend schneller sind, werden diese die klassischen Rechner ersetzen. Ein „Quantencomputer von Aldi“ scheint aus heutiger Sicht unrealistisch. Allerdings dachte man so auch über klassische Computer:

> *„I think there is a world market for maybe five computers.“*
> Thomas Watson, Vorsitzender von IBM, 1943

5 Der Grover-Algorithmus

5.1 Überblick

> *„I think it's a very nice piece of work.“*
> Peter Shor

Mit Hilfe der Quantenmechanik ist es möglich, klassische Suchvorgänge zu beschleunigen. Angenommen, man hat eine unsortierte Liste mit 1000 Einträgen, welche Titel von Zeitschriften sind. Möchte man diese Liste nach einer bestimmten Zeitschrift durchsuchen, so wird man im klassischen Fall nacheinander jeden Eintrag darauf prüfen, ob er dem Suchkriterium entspricht. Man spricht auch von *brute force* Methode. Dieser einfache Algorithmus (*lineare Suche*) benötigt im besten Fall genau einen Schritt, im schlechtesten Fall 1000 Schritte, und im Schnitt 500 Schritte, bis das gesuchte Exemplar gefunden wird.
Das Beispiel lässt sich leicht verallgemeinern. Man spricht dann nicht von einer Liste, sondern von einer Datenbank, die eine Sammlung von Informationen enthält. Gibt es in dieser Datenbank N Elemente, dann braucht man mit der oben beschriebenen Methode auf einem klassischen Computer durchschnittlich $N/2$ Operationen, um das gewünschte Element zu finden, das heißt, die Laufzeit dieses Algorithmus beträgt

$O(N)$.
Der Grover-Algorithmus, entwickelt 1996 von Lov K. Grover [Gro96, Gro97], ist ein Quantenalgorithmus für die Suche in solchen Datenbanken, die keinerlei Struktur oder Sortierung aufweisen, wie im obigen Beispiel beschrieben. In diesen Fall kann man das Resultat bereits nach $O(\sqrt{N})$ Schritten erhalten, was eine quadratische Verbesserung im Vergleich zum klassischen Algorithmus darstellt. In der Laufzeit unterscheidet sich der Grover-Algorithmus ein wenig von anderen Quantenalgorithmen, die oft eine exponentielle Beschleunigung gegenüber herkömmliche Algorithmen bieten. Wenn N groß ist, dann bedeutet auch eine „nur“ quadratische Verbesserung jedoch einen enormen Geschwindigkeitsvorteil.
Man kann zeigen, dass es keinen Orakel-basierten Quantenalgorithmus gibt, der das Suchproblem schneller als in $O(\sqrt{N})$ lösen kann, das heißt, der Grover-Algorithmus ist optimal. Diesen Beweis lieferte [BBBV97].
Der Algorithmus kann die Datenbank nach einem oder mehreren Elementen durchsuchen. Prinzipiell unterscheidet sich das Vorgehen dabei nicht. Die Erweiterung auf mehrere Suchergebnisse geht auf [BBHT98] zurück.
Dieses Kapitel ist wie folgt unterteilt: Abschnitt 5.2 beschreibt die Funktion des so genannten *Orakels*, dem eine zentrale Funktion zukommt. Danach (Abschnitt 5.3) stelle ich die einzelnen Schritte des Algorithmus vor, die sich durch eine geometrische Beschreibung veranschaulichen lassen (5.4). Der schon erwähnte Geschwindigkeitsvorteil wird in Abschnitt 5.5 hergeleitet. Ein anderes Bild, was die Wirkung des Grover-Algorithmus auf einen gegebenen Anfangszustand verdeutlicht, beschreibe ich in 5.6. Daran schließt sich ein Beispiel an für einen Suchraum mit 4 Elementen. Abschnitt 5.7 zeigt, welche elementaren Gatteroperationen in diesem Fall nötig sind, um den Grover-Algorithmus zu implementieren. Drei Anwendungsmöglichkeiten des Algorithmus zeigt Abschnitt 5.8. Zum Schluss beweise ich, dass es keinen schnelleren Orakel-basierten Quantenalgorithmus geben kann (5.9).
Die Darstellung der Sachverhalte in den folgenden Kapiteln orientiert sich an [NC00]. Die dortigen Ausführungen bieten einen gelungenen Einstieg in die Thematik.

5.2 Das Orakel

Das System, das durchsucht werden soll, habe N Elemente, wobei wir zur Vereinfachung annehmen, dass N eine Zweierpotenz ist, $N = 2^n$. Jedes Element im System kann über einen Index adressiert werden, welcher eine Zahl zwischen 0 und $N - 1$ ist. Das bedeutet, dass wir den Index in n klassischen Bits gespeichert können. Unter diesen N Elementen soll es M geben, die das Suchkriterium erfüllen, also die Lösungen zum Suchproblem darstellen: $1 \leq M \leq N$.
Die Annahme ist jetzt, dass es eine binäre Funktion f gibt, mit der wir herausfinden können, ob wir eine Lösung gefunden haben. Das Argument von f ist eine ganze Zahl x, $x \in [0; N-1]$. Das Ergebnis von f zeigt uns an, ob wir eine Lösung gefunden haben. Erfüllt x unser Suchkriterium, ist $f(x) = 1$, andernfalls ist $f(x) = 0$. Dabei soll f das Ergebnis in *einem* Schritt bestimmen können. Das Problem besteht also darin, ein x zu finden, so dass $f(x) = 1$.

Aus der Theoretischen Informatik kennt man für eine solche Funktion f den Begriff des *Orakels*. Ein Orakel ist eine Black Box, das heißt, die interne Implementation bleibt uns verborgen und ist nicht vom Grover-Algorithmus vorgegeben. Wir wissen nur, dass das Orakel die Lösung in einem Schritt erkennen kann. Als unitärer Operator O beschrieben, lässt sich die Wirkung des Orakels wie folgt darstellen:

$$|x\rangle\,|q\rangle \xrightarrow{O} |x\rangle\,|q \oplus f(x)\rangle\,. \tag{11}$$

x ist das Indexregister, $\oplus$ bedeutet Addition modulo 2 und q ist ein Qubit für das Orakel. Falls x eine Lösung ist, folgt $f(x) = 1$ und $|q\rangle$ wird invertiert. Ist x keine Lösung, bleibt $|q\rangle$ unverändert. Das Orakelqubit zeigt also an, ob wir ein Lösung gefunden haben.
Es ist hilfreich, wenn wir das Orakelqubit anfangs im Zustand $(|0\rangle - |1\rangle)/\sqrt{2}$ präparieren. Dieser Anfangszustand entsteht, wenn wir das Hadamard-Gatter auf den Basiszustand $|1\rangle$ anwenden. Der Vorteil bei diesem Vorgehen ist, dass sich nur das Vorzeichen des Zustandes ändert, wenn $x = x_S$ Lösung ist:

$$|x_S\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) \xrightarrow{O} -|x_S\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right). \tag{12}$$

Falls x keine Lösung ist, bleibt der Zustand $(|0\rangle - |1\rangle)/\sqrt{2}$ unverändert. Zusammenfassend kann mal also schreiben

$$|x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) \xrightarrow{O} (-1)^{f(x)}\,|x\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right). \tag{13}$$

Somit bekommen wir nur einen Vorzeichenwechsel bei passendem x. Das Orakelqubit wird nicht verändert, so dass wir es für die folgenden Überlegungen weglassen können. Die verkürzte Schreibweise für die Wirkung des Orakels ist dann

$$|x\rangle \xrightarrow{O} (-1)^{f(x)}\,|x\rangle \tag{14}$$

oder

$$O\,|x_S\rangle = -|x_S\rangle \tag{15}$$

$$O\,|x\rangle = |x\rangle \qquad \forall\, x \neq x_S \tag{16}$$

An dieser Stelle wird der Zweck des Orakels im Grover-Algorithmus deutlich. Es erkennt den Lösungszustand, ändert dessen Phase und markiert dadurch die Lösung. Die Betonung liegt auf *erkennen*, im Unterschied zu *kennen*. Letzteres hieße, wir könnten das Orakel direkt nach der Lösung fragen und bekämen die richtige Antwort. Das ist aber nicht möglich, da das Orakel nur binäre Antworten geben kann im Sinn von 'ja/nein' oder 'richtig/falsch', so dass wir für jeden Zustand aus dem Register fragen müssen, ob es sich um eine Lösung handelt. Es ist dann unsere Aufgabe, die richtige Lösung mit möglichst wenig Orakelaufrufen zu finden.

5.3 Die Prozedur

Der Grover-Algorithmus setzt sich aus wenigen unitären Operationen zusammen, die alle aus elementaren Gattern (siehe Abschnitt 4.4) aufgebaut werden können. Das Ziel ist es, aus dem Suchraum ein bestimmtes Element zu finden, das die Lösung des Suchproblems darstellt. Dabei sollen möglichst wenige Orakelaufrufe erfolgen.
Abbildung 11 zeigt den schematischen Aufbau des Algorithmus.

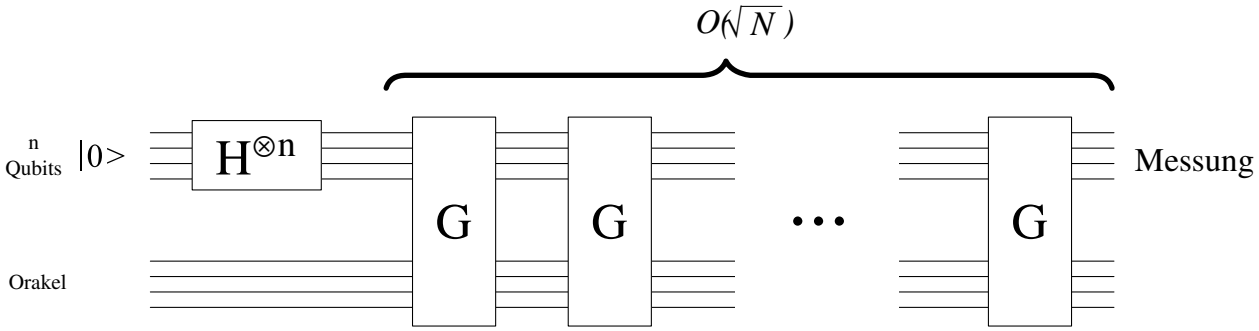


Abbildung 11: Schematischer Aufbau des Grover-Algorithmus. Der Arbeitsbereich des Orakels kann mehrere Qubits umfassen, spielt aber für die Analyse keine Rolle.

Es gibt zwei Register: eines mit n Qubits für die Indizes der zu durchsuchenden Elemente, und ein zweites Register für das Orakel. Dessen Implementation spielt aber für die weiteren Betrachtungen keine Rolle, weswegen ich es nicht näher erläutern werde. Die folgenden Ausführungen betreffen ausschließlich das Indexregister.
Das Register wird im Anfangszustand $|0\rangle^{\otimes n}$ präpariert. Durch die Hadamard-Operation $H^{\otimes n}$ konstruieren wir eine gleichmäßige Superposition,

$$|\psi\rangle \;=\; \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle \,, \tag{17}$$

so dass jeder Zustand die gleiche Amplitude $1/\sqrt{N}$ hat. Nun wenden wir die so genannte *Grover-Iteration* G an, die aus folgen vier Schritten besteht:

1. Wir wenden das Orakel O an, das heißt, der Lösungszustand erhält ein negatives Vorzeichen: $|x_S\rangle \longrightarrow -\,|x_S\rangle$.

2. Anwendung der Hadamard-Operation $H^{\otimes n}$.

3. Jeder Zustand, mit Ausnahme von $|0\rangle$, wird um π phasenverschoben, wechselt also das Vorzeichen.

4. Anwendung der Hadamard-Operation $H^{\otimes n}$.

Die letzten drei Schritte der Grover-Iteration dienen dazu, die Amplitude des Lösungszustandes zu erhöhen, so dass sich die Erfolgswahrscheinlichkeit bei einer Messung

erhöht. Der dritte Schritt, die bedingte Phasenverschiebung, lässt sich als unitärer Operator $2\left|0\right\rangle\left\langle 0\right| - I$ schreiben. Man sieht sofort, dass dieser Operator den Zustand $\left|0\right\rangle$ nicht verändert, alle anderen Zustände aber das Vorzeichen wechseln. Die Schritte 2), 3) und 4) lassen sich zum Operator

$$H^{\otimes n}(2\left|0\right\rangle\left\langle 0\right| - I)H^{\otimes n} = 2\left|\psi\right\rangle\left\langle\psi\right| - I \tag{18}$$

zusammenfassen. Somit ist die ganze Grover-Iteration

$$G = (2\left|\psi\right\rangle\left\langle\psi\right| - I)O. \tag{19}$$

In Abbildung 12 ist der Schaltkreis für die Grover-Iteration schematisch dargestellt. Die tatsächliche Implementation des Orakels ist durch den Algorithmus nicht festgelegt. Es ist aber möglich, das Orakel durch elementare Gatteroperationen zu realisieren, was ich in Abschnitt 5.7 anhand eines Beispiels besprechen werde. Abbildung 13 zeigt, wie die bedingte Phasenverschiebung aus elementaren Gattern aufgebaut werden kann.

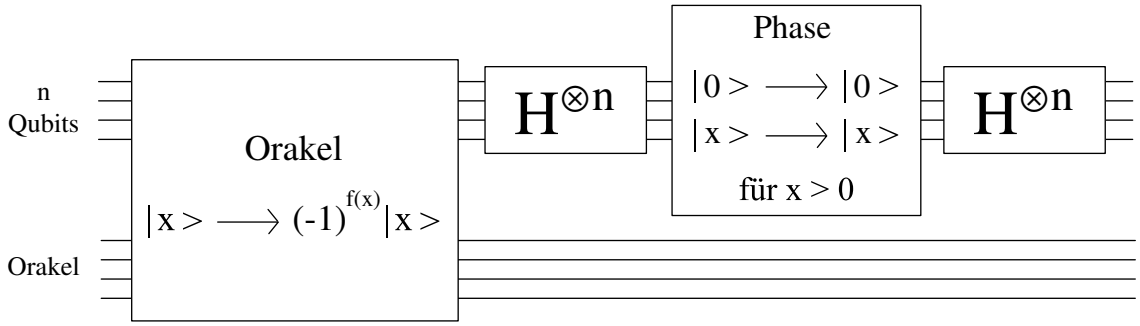


Abbildung 12: Schaltkreis für die Grover-Iteration

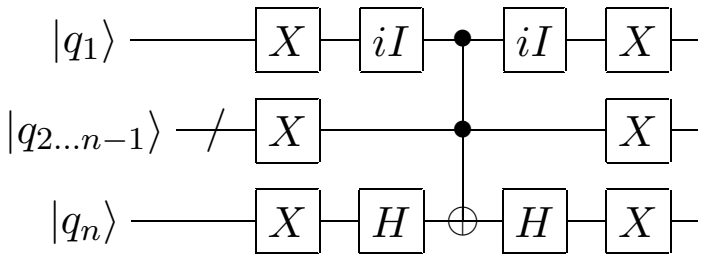


Abbildung 13: So kann man die bedingte Phasenverschiebung $2\left|0\right\rangle\left\langle 0\right| - I$ aus elementaren Gattern zusammensetzen.

Der letzte Schritt des Algorithmus ist die Messung in der Rechenbasis. Wir werden den gesuchten Zustand $\left|x\right\rangle$ mit hoher Wahrscheinlichkeit messen, wenn wir die Grover-Iteration oft genug ausführen (Abschnitt 5.5) und mit jedem Schritt die Amplitude von $\left|x\right\rangle$ erhöhen. Dieses Vorgehen lässt sich auf den Fall, dass es mehrere Lösungen gibt, übertragen.

5.4 Geometrische Interpretation

Man kann sich die Grover-Iteration anschaulich als Rotation eines Vektors in einem zweidimensionalen Vektorraum vorstellen, wie [Aha99] zeigte. Dieser Raum wird durch zwei Basisvektoren $|\alpha\rangle$ und $|\beta\rangle$ aufgespannt. $|\alpha\rangle$ ist wie folgt definiert:

$$|\alpha\rangle \equiv \frac{1}{\sqrt{N-M}} \sum_x{}'' |x\rangle \tag{20}$$

Dabei bezeichnet $\sum_x{}''$ die Summe über alle Zustände, die nicht das Suchkriterium erfüllen, also nicht Lösungen sind. Man kann sagen, dass $|\alpha\rangle$ die gleichmäßige Superposition aller „Nichtlösungen“ ist. Für $|\beta\rangle$ gilt:

$$|\beta\rangle \equiv \frac{1}{\sqrt{M}} \sum_x{}' |x\rangle \tag{21}$$

$\sum_x{}'$ ist die Summe über alle Zustände im Suchraum, die das Suchkriterium erfüllen. $|\beta\rangle$ ist demnach eine Superposition aller Lösungen. Geometrisch beschrieben stehen die beiden Vektoren α und β senkrecht aufeinander und spannen eine Ebene auf.
Der Anfangszustand $|\psi\rangle$ lässt sich als Summe von $|\alpha\rangle$ und $|\beta\rangle$ schreiben:

$$|\psi\rangle = \sqrt{\frac{N-M}{N}} |\alpha\rangle + \sqrt{\frac{M}{N}} |\beta\rangle = \frac{1}{N} \sum_{x \epsilon \{0,1\}^n} |x\rangle \tag{22}$$

Wie lässt sich in diesem Bild die Grover-Iteration verstehen? Der erste Schritt der Iteration, das Orakel O, stellt eine Spiegelung um den Basisvektor $|\alpha\rangle$ dar:

$$O(a|\alpha\rangle + b|\beta\rangle) = a|\alpha\rangle - b|\beta\rangle \tag{23}$$

Der zweite Schritt ist die bedingte Phasenverschiebung, die durch den Operator $2|\psi\rangle\langle\psi| - I$ beschrieben wird. Diese Operation entspricht wieder einer Spiegelung, diesmal um den Vektor $|\psi\rangle$. Zusammen mit der ersten Spiegelung ergibt sich insgesamt eine Rotation. Also bleibt $|\psi\rangle$ durch die mehrfache Anwendung von G in der Ebene aus $|\alpha\rangle$ und $|\beta\rangle$.
Wir können den Winkel der Rotation berechnen. Sei

$$\cos\frac{\theta}{2} = \sqrt{\frac{N-M}{N}} \tag{24}$$

und

$$\sin\frac{\theta}{2} = \sqrt{\frac{M}{N}}, \tag{25}$$

das heißt

$$|\psi\rangle = \cos\frac{\theta}{2}|\alpha\rangle + \sin\frac{\theta}{2}|\beta\rangle. \tag{26}$$

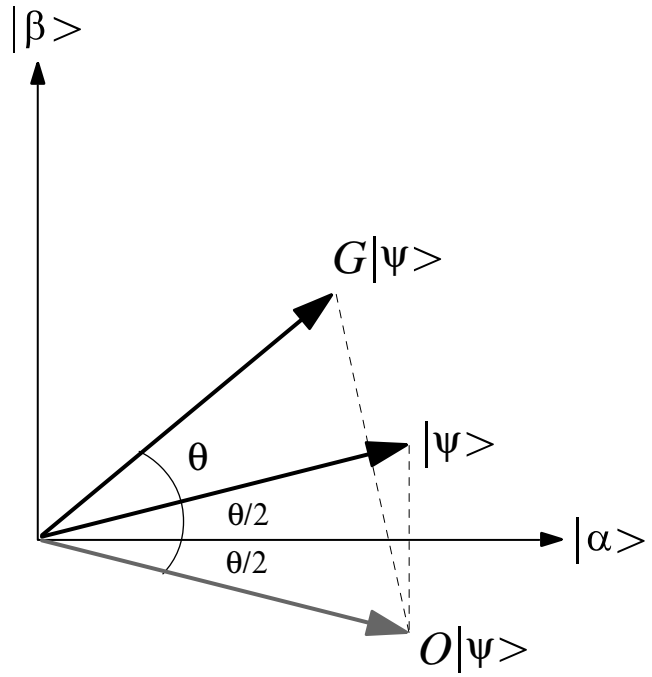


Abbildung 14: Die Wirkung der Grover-Iteration G: Zuerst bewirkt der Orakelaufruf, dass $|\psi\rangle$ an $|\alpha\rangle$ gespiegelt wird. Anschließend wird der neue Zustand an $|\psi\rangle$ gespiegelt. Insgesamt wird also um θ rotiert. Das Ziel ist es, $|\psi\rangle$ durch mehrere Grover-Iterationen möglichst nah an $|\beta\rangle$ zu drehen.

Abbildung 14 zeigt, dass die Grover-Iteration G den Startzustand $|\psi\rangle$ nach $|\psi\rangle'$ rotiert, mit

$$G\,|\psi\rangle \;=\; \cos\frac{3\theta}{2}\,|\alpha\rangle \;+\; \sin\frac{3\theta}{2}\,|\beta\rangle \tag{27}$$

In der $|\alpha\rangle$, $|\beta\rangle$ Basis geschrieben, ist die Grover-Iteration

$$G \;=\; \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}, \tag{28}$$

was einer gewöhnlichen Drehmatrix in der Ebene entspricht. Wenn man mehrfach iteriert, dann führt das zu

$$G^k\,|\psi\rangle \;=\; \cos\Big(\frac{2k+1}{2}\theta\Big)\,|\alpha\rangle \;+\; \sin\Big(\frac{2k+1}{2}\theta\Big)\,|\beta\rangle\,. \tag{29}$$

Der Operator G dreht $|\psi\rangle$ also in der durch $|\alpha\rangle$ und $|\beta\rangle$ aufgespannten Ebene um den Winkel θ in Richtung des „Lösungsvektors“ $|\beta\rangle$. Genügend Iterationen vorausgesetzt, wird man am Ende mit hoher Wahrscheinlichkeit einen Zustand messen, der das Suchkriterium erfüllt. Rotiert man allerdings zu weit, dann entfernt sich der Vektor $|\psi\rangle'$ wieder von $|\beta\rangle$ und die Wahrscheinlichkeit, einen Lösungszustand zu messen, sinkt. Dies ist ein wichtiger Punkt: Anders als man erwarten würde, steigt die Wahrscheinlichkeit nicht mit der Anzahl der Iterationen, sondern sie oszilliert zwischen 0

und 1 mit einer gewissen Periodizität. Das bedeutet, man muss die günstigste Zahl an Iterationen bestimmen, bevor man in der Rechenbasis messen kann.

5.5 Laufzeit und Erfolgswahrscheinlichkeit

Wieviele Grover-Iterationen nötig sind, sagt uns die Gleichung

$$k\theta + \frac{\theta}{2} = \frac{\pi}{2}. \tag{30}$$

Der Wert k gibt dabei die Anzahl der benötigten Grover-Iterationen an. $\theta/2$ ist der ursprüngliche Winkel zwischen $|\alpha\rangle$ und $|\psi\rangle$, $\pi/2 = 90°$ ist der gewünschte Winkel, bei dem $|\psi\rangle$ genau auf $|\beta\rangle$ rotiert wird. Weil θ mit der Größe des Suchraums N und der Zahl der Lösungen M zusammenhängt (siehe Gleichung 25), bestimmen diese beiden Parameter, wie oft wir die Grover-Iteration durchführen müssen.
Da k ganzzahlig sein muss, runden wir:

$$k = \text{round}\Big(\frac{\pi - \theta}{2\theta}\Big). \tag{31}$$

Für k lässt sich eine obere Grenze angeben, wenn wir für θ wieder Gleichung 25 berücksichtigen und die Näherung $\theta/2 \geq \sin(\theta/2)$ machen:

$$k = \text{round}\Big(\frac{\pi - \theta}{2\theta}\Big) = \text{round}\Big(\frac{\pi}{2\theta} - \frac{1}{2}\Big) \leq \text{round}\Big(\frac{\pi}{4}\sqrt{\frac{N}{M}}\Big) \tag{32}$$

Somit haben wir eine obere Grenze für die Zahl k der Grover-Iterationen gefunden:

$$k = \left\lceil \frac{\pi}{4}\sqrt{\frac{N}{M}} \right\rceil \tag{33}$$

Die Laufzeit des Grover-Algorithmus ist demnach $O(\sqrt{N/M})$, eine quadratische Beschleunigung im Vergleich zum klassischen Algorithmus, der $O(N/M)$ Schritte braucht. Die Wahrscheinlichkeit, das gewünschte Element am Ende durch eine Messung in der Rechenbasis zu finden, ist

$$p = \sin^2\Big(\frac{2k+1}{2}\theta\Big), \tag{34}$$

wie aus 29 folgt. Abbildung 15 zeigt p für n von 2 bis 30 bei $M = 1$. Für $n = 2$ ist die Wahrscheinlichkeit genau 1, weil der Winkel $\theta/2$ zwischen $|\psi\rangle$ und $|\alpha\rangle$ in diesem Fall genau $\pi/6$ ist, wie aus Gleichung 24 hervorgeht. Wenn man dann G anwendet, rotiert man $|\psi\rangle$ genau auf $|\beta\rangle$.
Die Näherung $\theta/2 \geq \sin(\theta/2)$ gilt nur, wenn $M \leq N/2$, das heißt, maximal die Hälfte aller Zustände sind Lösungen des Suchproblems. Ist das nicht der Fall, gibt es zwei Möglichkeiten. Falls M bekannt ist, denn können wir einfach ein zufälliges Element wählen und per Orakelaufruf prüfen, ob es eine Lösung ist. Da mehr als die Hälfte

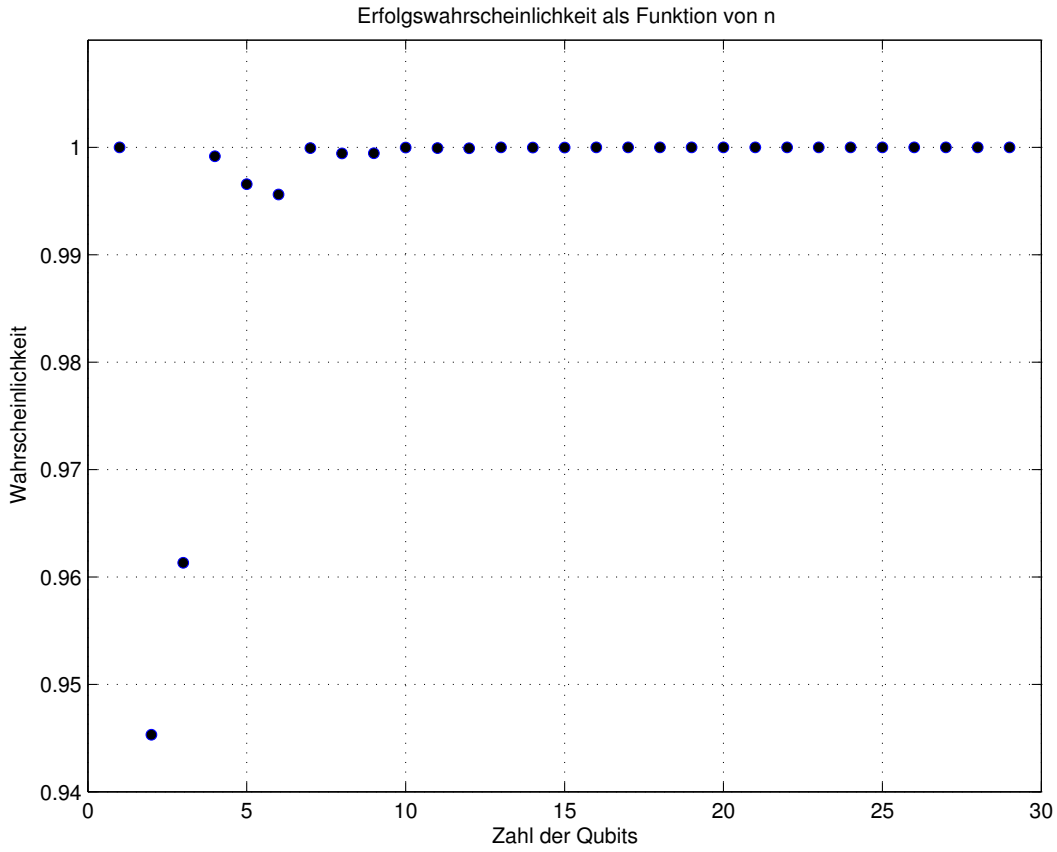


Abbildung 15: Die Wahrscheinlichkeit, die richtige Lösung zu messen, als Funktion von $n = \log N$

aller Zustände diese Bedingung erfüllen, ist die Erfolgswahrscheinlichkeit größer 1/2 bei nur einem Orakelaufruf.
Da die Zahl M aber nicht immer vorher bekannt sein dürfte, ist der zweite Ansatz, die Zahl der Elemente im Suchraum zu verdoppeln durch N Elemente, die keine Lösungen darstellen. Dazu addiert man ein Qubit zum bestehenden Register und konstruiert ein neues, erweitertes Orakel für diese Zahl von Qubits. Die Laufzeit ist in diesem Fall $k = \pi/4\sqrt{2N/M}$, was immer noch $O(\sqrt{N/M})$ ist.

5.6 Inversion um den Mittelwert

Die Wirkung des Operators $D^3 = 2\,|\psi\rangle\,\langle\psi| - I$ lässt sich veranschaulichen, wenn man die Amplituden der Zustände im Suchraum betrachtet. $|\psi\rangle$ ist dabei die gleichmäßige Superposition aller Zustände (Gleichung 17).
Sei $|\alpha\rangle = \sum_x \alpha_x\,|x\rangle$ ein allgemeiner Zustand. Das Skalarprodukt von $|\alpha\rangle$ und $|\psi\rangle$ ist

$$\langle\psi|\alpha\rangle = \frac{1}{\sqrt{N}}\sum_x \alpha_x = \sqrt{N}\,A, \tag{35}$$

[3] D wird auch Diffusionsoperator genannt.

mit $A = \frac{1}{N}\sum_x \alpha_x$. A ist also der Mittelwert der Amplituden. D angewandt auf $|\alpha\rangle$ gibt

$$\begin{aligned} D\,|\alpha\rangle &= 2\,|\psi\rangle\,\langle\psi|\alpha\rangle - |\alpha\rangle && (36)\\ &= 2\sum_x |x\rangle\,A - \sum_x \alpha_x\,|x\rangle && (37)\\ &= \sum_x (2A - \alpha_x)\,|x\rangle && (38) \end{aligned}$$

Die Wirkung von D lässt sich folgendermaßen zusammenfassen:

$$D:\ \alpha_x \longrightarrow 2A - \alpha_x \qquad (39)$$

Das heißt, die Koeffizienten von $|x\rangle$ werden am Mittelwert gespiegelt. Man spricht deshalb von *Inversion um den Mittelwert* (engl. *inversion about average*). Ziel der Grover-Iteration ist es, die Amplitude des Lösungszustandes mit jedem Iterationsschritt zu erhöhen. Wie das geschieht, kann man sehr gut graphisch verstehen, wie Grover in der Erweiterung [Gro97] seines Originalartikels beschreibt. Für mehrere Lösungen gilt diese Beschreibung natürlich auch.

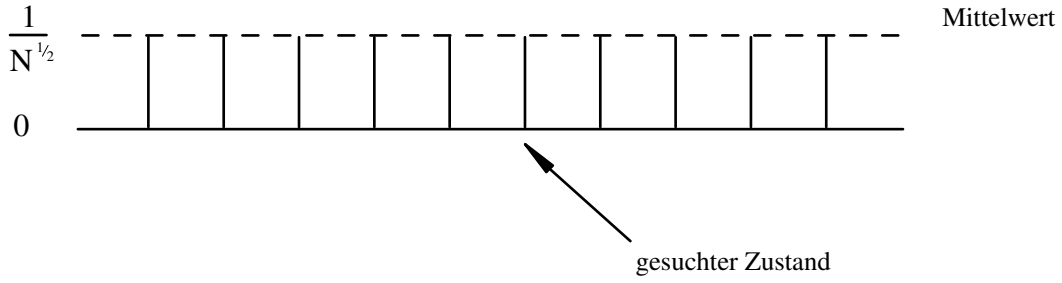


Abbildung 16: Die Anfangssituation: Alle Zustände haben die gleiche Amplitude $1/\sqrt{N}$.

Abbildung 16 zeigt die Amplituden der (Basis-)Zustände nach der $H^{\otimes n}$-Operation. Sie sind alle gleich groß.

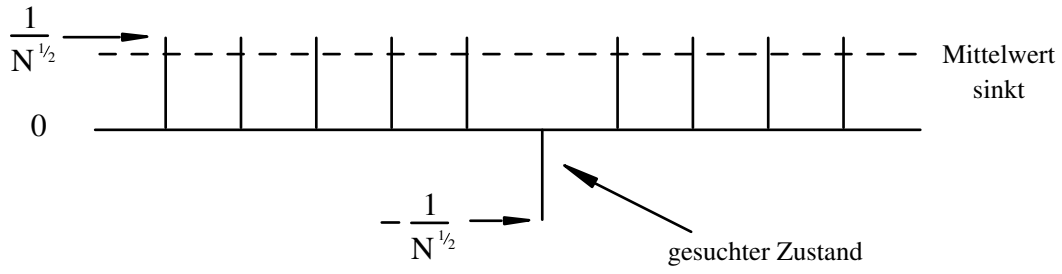


Abbildung 17: Nach dem Orakelaufruf hat die Amplitude des Lösungszustandes ein anderes Vorzeichen. Der Mittelwert sinkt leicht.

Nach dem Aufruf des Orakels hat der Lösungszustand das Vorzeichen gewechselt (Abbildung 17). Dadurch sinkt der Amplituden-Mittelwert ein wenig. Je größer der Suchraum ist, desto kleiner ist diese Änderung.

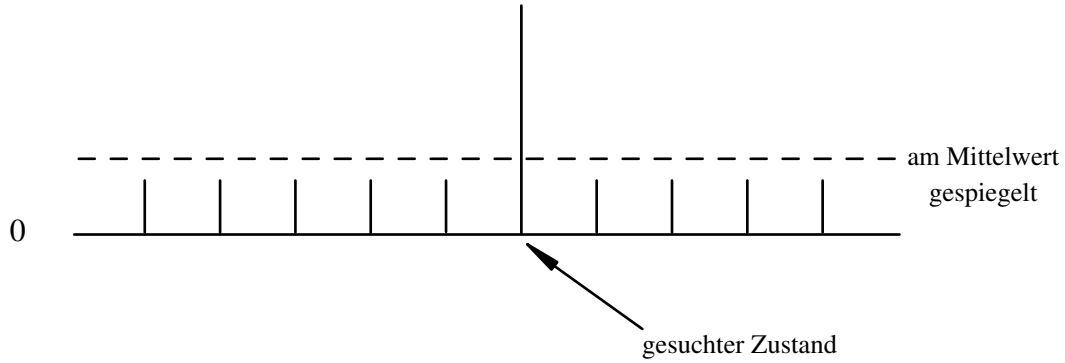


Abbildung 18: Inversion um den Mittelwert: Die Wahrscheinlichkeit, den gesuchten Zustand zu messen, hat sich deutlich erhöht

Im dritten Schritt werden alle Amplituden am Mittelwert gespiegelt. So erhöht sich die Amplitude der Lösung stark, während alle anderen Amplituden kleiner werden, siehe Abbildung 18. Auf diese Weise steigt die Amplitude des gesuchten Zustandes mit jeder Grover-Iteration, so dass am Ende eine Messung in der Rechenbasis mit hoher Wahrscheinlichkeit das gewünschte Ergebnis liefert.

5.7 Eine Beispielimplementation mit zwei Qubits

In Abbildung 19 ist eine konkrete Implementation für den Grover-Algorithmus zu sehen, die auf [NC00] und [LMP03] basiert. Der Suchraum umfasst 4 Elemente, die wir durch zwei Qubits repräsentieren können, welche anfangs im Zustand $|0\rangle$ präpariert werden. Das dritte Qubit in dem gezeigten Schaltkreis ist das Hilfsqubit für das Orakel. Es ist am Anfang im Zustand $|1\rangle$.

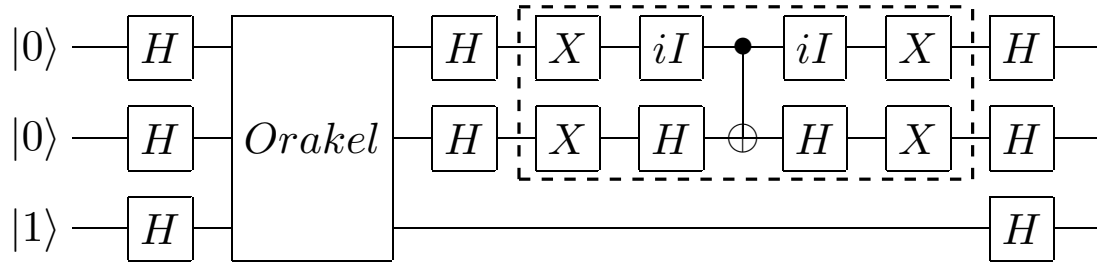


Abbildung 19: Der Schaltkreis für den Grover-Algorithmus mit $N = 4$. Das untere Qubit ist ein Hilfsqubit für das Orakel. In diesem speziellen Fall ist nur eine Grover-Iteration nötig. Die Gatter in dem gestrichelten Kasten implementieren die bedingte Phasenverschiebung.

Das Orakel selbst kann durch einen der vier Schaltkreise in Abbildung 20 realisiert

werden. Wie man leicht nachrechnen kann, wird dadurch die Transformation

$$|x\rangle \xrightarrow{O} (-1)^{f(x)} |x\rangle \tag{40}$$

implementiert, wobei $f(x) = 0$ für alle x außer $x = x_S$, für das $f(x_S) = 1$ gilt. x_S sei die Lösung des Suchproblems.

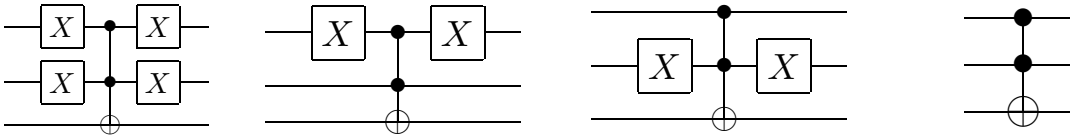


Abbildung 20: Implementationsmöglichkeiten für das Orakel. Ganz links ist der Schaltkreis für $x_S = 0$ (x_S sei die Lösung). Dann folgen die entsprechenden Schaltkreise für $x_S = 1, 2, 3$. Die beiden oberen Qubits bilden den Suchraum, das dritte Qubit ist der Arbeitsbereich des Orakels.

Der Schaltkreis in Abbildung 19 bringt die Qubits mit Hilfe der $H^{\otimes n}$-Operation in eine gleichmäßige Superposition. Danach wird die Grover-Iteration einmal durchgeführt. In diesem Fall reicht das, um durch eine Messung der ersten beiden Qubits x_S mit Wahrscheinlichkeit 1 zu finden. Denn aus Gleichung 25 folgt mit $N = 4$ und $M = 1$, dass $\theta/2 = \pi/6$. In dem geometrischen Bild aus Kapitel 5.4 schließt der Anfangszustand $|\psi\rangle = (|00\rangle + |01\rangle + |10\rangle + |11\rangle)/2$ mit der Achse $|\alpha\rangle$ den Winkel $\theta/2 = 30°$ ein. Die Grover-Iteration rotiert dann um $\theta = 60°$, so dass der Zustand genau auf der Achse $|\beta\rangle$ landet.
Der hier gezeigte Schaltkreis kann leicht verifiziert werden, zum Beispiel durch eine Simulation mit dem *Mathematica*-Paket *QuCalc* (siehe Kapitel 6.6).

5.8 Anwendungsmöglichkeiten

5.8.1 Quantum counting

Oft steht man vor der Frage: Wie viele Lösungen gibt es zu einem Suchproblem? Und wie schnell kann man ihre Zahl bestimmen? Klassisch würde man mit einem Orakel $\Theta(N)$ Schritte benötigen, um bei einem Suchraum der Größe N diese Fragen zu beantworten. Zur Notation $\Theta(N)$ siehe Anhang A.4. Mit einem Quantencomputer und dem Grover-Algorithmus kann das Problem schneller gelöst werden. Dies wurde detailliert in [BHT98] beschrieben. Die Verbesserung erreicht man, indem man auf ein Verfahren zurückgreift, das *Phasenabschätzung* heißt [Mos98]. Im Anhang B gehe ich näher auf dieses Verfahren ein.
Es gibt zwei Anwendungen von Quantum counting.

1. Die Suche nach Lösungen wird beschleunigt, sollte die Anzahl M der Lösungen nicht von vornherein bekannt sein. In diesem Fall kann man zuerst M bestimmen und danach den bekannten Suchalgorithmus anwenden. Bei der Grover-Iteration ist wichtig, wie groß M ist, um die Zahl der Wiederholungen bestimmen zu können.

2. Man kann prüfen, ob es überhaupt eine Lösung gibt. Dieser Punkt spielt unter anderem bei der Lösung von NP-vollständigen Problemen eine Rolle.

Die Grover-Iteration G aus Abschnitt 5.3 ist ein Operator mit bestimmten Eigenwerten, aus denen man M berechnen kann. Um die Eigenwerte zu finden, setzt man die Phasenabschätzung ein.
$|a\rangle$ und $|b\rangle$ seien die beiden Eigenvektoren von G, θ der Drehwinkel in der $|\alpha\rangle$-$|\beta\rangle$-Ebene. Gleichung 28 gibt uns die Eigenwerte: $e^{i\theta}$ und $e^{i(2\pi-\theta)}$. Unsere Aufgabe ist nun, θ zu suchen. Um die Beschreibung der folgenden Schritte zu vereinfachen, nehmen wir an, dass wir mit dem am Ende von Abschnitt 5.5 besprochenen erweiterten Orakel arbeiten, das auf einem Suchraum von $2N$ Elementen arbeitet. Dann gilt die Beziehung

$$\sin^2(\theta/2) \;=\; M/2N. \tag{41}$$

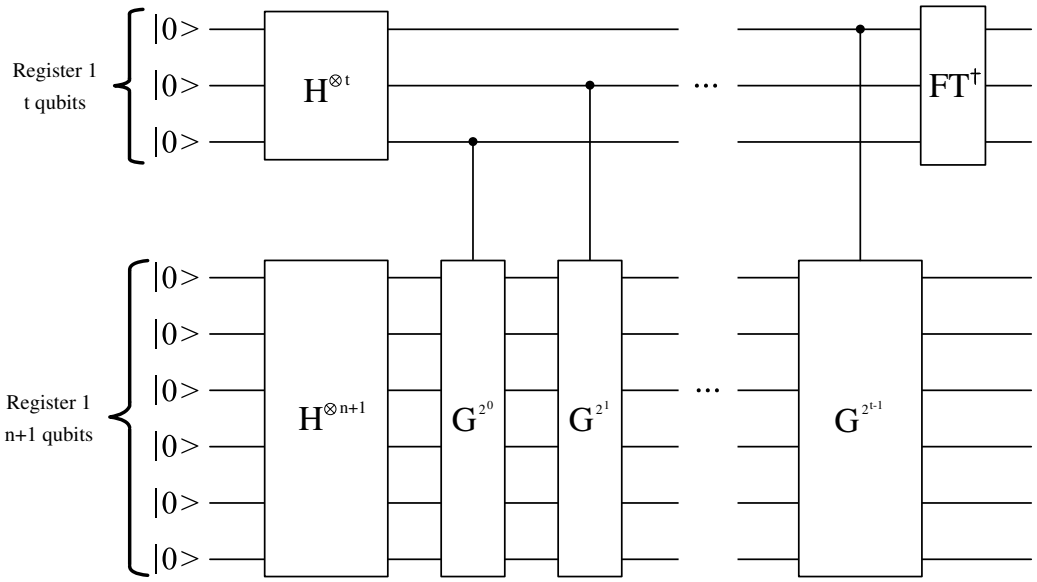


Abbildung 21: Der Schaltkreis für die Phasenabschätzung

Den Schaltkreis für die Phasenabschätzung zeigt Abbildung 21. Der Schaltkreis berechnet θ mit m-bittiger Genauigkeit und einer Wahrscheinlichkeit von mindestens $1-\epsilon$.
Im ersten Register sind t Qubits, wobei $t = m + \lceil \log(2 + 1/2\epsilon) \rceil$. t hängt ab von der Genauigkeit, mit der man das Ergebnis berechnen möchte, und von der Wahrscheinlichkeit, mit der die Prozedur Erfolg haben soll.
Im zweiten Register sind $n+1$ Qubits, also genau so viele, wie wir für die Grover-Iteration auf dem erweiterten Suchraum benötigen. Es wird t-mal iteriert. Der Startzustand dieses Registers ist wie üblich eine gleichmäßige Überlagerung aller Elemente, was durch Hadamard-Gatter erreicht wird. In Abschnitt 5.3 haben wir gesehen, dass dies eine Superposition aller Eigenvektoren $|a\rangle$ und $|b\rangle$ ist. Die Phasenabschätzung

liefert uns für diese Überlagerung eine Abschätzung für θ oder $2\pi - \theta$, und zwar mit einer Wahrscheinlichkeit von mindestens $1 - \epsilon$ und und der Genauigkeit $|\Delta\theta| \leq 2^{-m}$. Da eine Abschätzung von $2\pi - \theta$ äquivalent zur Abschätzung von θ ist, haben wir durch diesen Schaltkreis θ approximiert.
Mit diesem Wissen ist es leicht, über Gleichung 41 die Zahl der Lösungen M zu berechnen. Aber auch M wird mit einem Fehler ΔM behaftet sein, genau wie θ. ΔM können wir abschätzen:

$$\frac{|\Delta M|}{2N} = \left|\sin^2\left(\frac{\theta+\Delta\theta}{2}\right) - \sin^2\left(\frac{\theta}{2}\right)\right| \tag{42}$$

$$= \left(\sin\left(\frac{\theta+\Delta\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)\right)\left|\sin\left(\frac{\theta+\Delta\theta}{2}\right) - \sin\left(\frac{\theta}{2}\right)\right|. \tag{43}$$

Aus der Analysis kennen wir die Beziehung

$$\left|\sin\left(\frac{\theta-\Delta\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)\right| \leq \left|\frac{\Delta\theta}{2}\right|. \tag{44}$$

Einfache Trigonometrie gibt uns

$$\left|\sin\left(\frac{\theta+\Delta\theta}{2}\right)\right| < \sin\frac{\theta}{2} + \frac{\Delta\theta}{2}. \tag{45}$$

Daraus können wir folgern, dass

$$\frac{|\Delta M|}{2N} = \left(2\sin\frac{\theta}{2} + \frac{|\Delta\theta|}{2}\right)\frac{|\Delta\theta|}{2}. \tag{46}$$

Jetzt ersetzen wir $\sin^2(\theta/2) = M/2N$ und $|\Delta\theta| \leq 2^{-m}$, so dass wir schließlich die Abschätzung für den Fehler ΔM erhalten:

$$|\Delta M| < \left(\sqrt{2MN} + \frac{N}{2^{m+1}}\right)2^{-m}. \tag{47}$$

Betrachten wir ein Beispiel. Wir wählen $m = \lceil n/2 \rceil + 1$ und $\epsilon = 1/6$. Daraus ergibt sich $t = \lceil n/2 \rceil + 3$. Aus Abbildung 5.8.1 geht hervor, dass die letzte Grover-Iteration $G^{2^{t-1}}$ ist. Das heißt, wir brauchen $\Theta(\sqrt{N})$ Schritte, um M zu bestimmen, was eine quadratische Beschleunigung gegenüber dem klasischen Fall darstellt. Der Fehler ist $\Delta M < \sqrt{M/2} + 1/4 = O(\sqrt{M})$.
Dieses Beispiel verdeutlicht auch, dass wir mit Hilfe dieses Algorithmus feststellen können, ob es überhaupt ein Lösung gibt, also ob $M = 0$ oder $M \neq 0$. Im ersten Fall ist nämlich $|\Delta M| < 1/4$, so dass der Algorithmus mit der Wahrscheinlichkeit 5/6 das Ergebnis 0 liefert. Wenn es mindestens eine Lösung gibt, ist die Abschätzung dementsprechend mit 5/6 Wahrscheinlichkeit ungleich 0.
In der Praxis kann man also den Algorithmus erst zählen und anschließen suchen lassen, das heißt, man bestimmt zuerst die Anzahl der Lösungen und startet mit diesem Wissen dann die eigentliche Suche.

5.8.2 NP-vollständige Probleme

Der Begriff NP-Vollständigkeit[4] beschreibt eine Menge von Problemen, die mit Hilfe klassischer Computer als nicht lösbar gelten. Die Unlösbarkeit ist jedoch bis heute nicht bewiesen. Eines dieser Probleme ist der *Hamiltonkreis*. Sei G ein Graph ohne Mehrfachkanten, dann ist der Kreis H ein Hamiltonkreis, wenn er alle Knoten von G enthält. Das Problem, zu entscheiden, ob ein gegebener Graph einen Hamiltonkreis besitzt (also *hamiltonisch* ist), nennt man Hamiltonkreis-Problem. Dieses Problem ist NP-vollständig; man glaubt also, dass es für einen klassischen Computer im Allgemeinen nicht lösbar ist.
Ein Algorithmus, der das Hamiltonkreis-Problem lösen kann, muss alle möglichen Pfade, also alle Anordnungen der Knoten, durchsuchen. Dazu muss zuerst eine Liste mit allen möglichen Knotenfolgen $(v_1, \ldots, v_n)$ des Graphen erzeugt werden. Wiederholungen sind erlaubt, weil sie das Ergebnis im Kern nicht beeinflussen, aber das Aufstellen der Liste vereinfachen. Danach muss für jede Anordnung geprüft werden, ob es ein Hamiltonkreis ist. Falls nicht, wird der nächste Listeneintrag geprüft. Es gibt somit $n^n = 2^{n \log n}$ möglicher Knotenanordnungen, das heißt, man benötigt klassisch $2^{n \log n}$ Suchvorgänge im schlechtesten Fall.
Der Quantensuchalgorithmus aus dem vorherigen Abschnitt kann diese Suche beschleunigen. Wenn wir die Version benutzen, die abwechselnd zählt und sucht, ist es möglich die Suchgeschwindigkeit zu erhöhen. Wie wir feststellen werden, benötigt die Quantensuche nur die Quadratwurzel der Zahl an Operationen, die ein klassischer Algorithmus bräuchte.
Wie gehen wir vor? Sei $m = \lceil \log n \rceil$. Wir brauchen $m \cdot n$ Qubits, um alle Knoten abzubilden. Jeder Block von m Qubits repräsentiert den Index eines einzelnen Knotens. So können wir als Rechenbasis $|v_1, \ldots, v_n\rangle$ schreiben, wobei jedes $|v_i\rangle$ durch den passenden String von Qubits dargestellt wird.
Das Orakel soll erkennen, wenn ein Pfad ein Hamiltonkreis ist, und die Lösung durch die Änderung des Vorzeichens markieren:

$$0\,|v_1 \ldots, v_n\rangle = \begin{cases} -\,|v_1, \ldots, v_n\rangle\,, & \text{falls Hamiltonkreis;} \\ \\ |v_1, \ldots, v_n\rangle\,, & \text{falls kein Hamiltonkreis.} \end{cases} \tag{48}$$

Diese Transformation kann implementiert werden, indem man einen klassischen (nichtreversiblen) Schaltkreis nimmt, der mit polynomialen Aufwand einen Hamiltonkreis erkennt. Diesen Schaltkreis muss man in einen reversiblen konvertieren, der ebenfalls von polynomialen Aufwand ist. So kann man die Transformation

$$(|v_1, \ldots, v_n, q\rangle) \longrightarrow (v_1, \ldots, v_n, q \oplus f(v_1, \ldots, v_n)) \tag{49}$$

durchführen. Hierbei ist q das Qubit für das Orakel und $f(v_1, \ldots, v_n)$ wie üblich gleich 1, falls der Pfad ein Hamiltonkreis ist, und gleich 0 andernfalls. Ein solches Orakel

[4] siehe Anhang A.3

ist leicht zu entwerfen, wenn man eine Beschreibung des Graphen besitzt. Wichtig ist, dass die Zahl der den Schaltkreis bildenden Gatter polynomial mit n wächst.
Nimmt man die Variante des Quanten-Suchalgorithmus aus dem letzten Kapitel, dann hat dieser Algorithmus eine Laufzeit von $O(2^{m \cdot n/2}) = O(2^{n\lceil \log n \rceil/2})$. Er ist im Vergleich mit einem klassischen Algorithmus quadratisch besser, was die Zahl der Operationen angeht.
Der kombinierte Zähl-/Suchalgorithmus bestimmt also zuerst, ob es einen Hamiltonkreis gibt, und kann uns anschließend sagen, welche Knotenfolge diesen bildet. Diese Prozedur hat ein probabilistisches Ergebnis, während ein klassischer Algorithmus deterministisch[5] arbeitet. Der Fehler kann jedoch minimiert werden, wenn man die Prozedur mehrmals wiederholt.

5.8.3 Die Suche in unstrukturierten klassischen Datenbanken

Gegeben sei eine klassische Datenbank mit einer Vielzahl von unsortierten Einträgen. Das könnte zum Beispiel ein Telefonbuch sein, das wir nicht alphabetisch sondern nach einer bestimmten Telefonnummer durchsuchen wollen. Zur Vereinfachung nehmen wir an, dass diese Datenbank genau $N = 2^n$ Einträge enthält. Jeder Eintrag hat die Länge l Bits und kann über einen Index $d_1, \ldots, d_N$ adressiert werden. Unsere Aufgabe ist es nun, einen bestimmten Eintrag s (ebenfalls der Länge l Bit) aus der Datenbank zu finden.
Betrachten wir zunächst den klassischen Fall. Ein klassischer Computer ist im Allgemeinen in zwei Bereiche unterteilt. Der erste Teil ist die CPU[6], die Berechnungen mit den Daten durchführen kann. Der zweite Teil ist der Speicher, der die Daten hält, aber sie nicht manipulieren kann. Die CPU braucht selbst nur wenig Speicher, um temporär alle Daten zu speichern, mit denen sie gerade rechnet. Das sind die so genannten CPU-Register. Der Speicher ist in 2^n Blöcke unterteilt, die jeweils eine l-Bit lange Datenfolge (String) enthalten.
Es gibt drei Basisoperationen:

- Das Laden der Daten aus dem Speicher in die CPU (`LOAD`)
- Das Zurückschreiben der Daten von der CPU in den Speicher (`STORE`)
- Das Manipulieren der Daten in der CPU.

Das ist eine sehr vereinfachte Sicht der Dinge, heutige Rechner sind wesentlicher komplizierter aufgebaut. Aber für die folgenden Betrachtungen reicht diese Beschreibung aus.
Der effizienteste klassische Algorithmus, der ein gegebenes s in der Datenbank findet, hat folgenden Aufbau.

1. In der CPU wird ein Index der Länge l Bit angelegt.

[5] siehe dazu Anhang A.2
[6] Central Processing Unit

2. Der Index startet bei 0 und wird in jedem Schritt um 1 erhöht.

3. In den jeweils aktuellen Indexwert wird der entsprechende Datenstring aus dem Speicher geladen.

4. Dieser Eintrag wird mit s verglichen

5. Ist der Eintrag gleich s, hält das Programm an und gibt den aktuellen Index aus. Andernfalls wird der Index wieder um 1 erhöht.

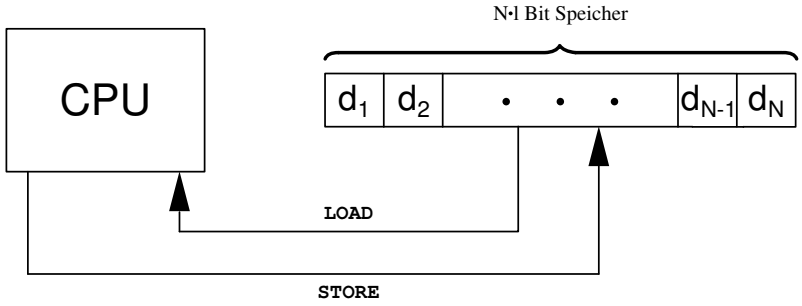


Abbildung 22: Die klassische Suche in einer Datenbank.

Dieser Algorithmus muss im schlechtesten Fall (das letzte Element ist die Lösung) alle Elemente überprüfen, benötigt dafür also $N = 2^n$ Schritte. Im besten Fall (das erste Element ist die Lösung) ist nur ein Schritt notwendig. Im Durchschnitt sind es folglich $N/2$ Schritte, bis das gesuchte Ergebnis gefunden wird. Da die Daten keinerlei Sortierung oder sonstige Struktur besitzen, kann es keinen schnelleren Algorithmus geben.
Wenn wir die Aufgabe mit dem Grover-Algorithmus lösen wollen, um die Suche zu beschleunigen, dann müssen bestimmte Voraussetzungen gegeben sein. An der Aufteilung des Computers in CPU und Speicher ändert sich nichts. Allerdings muss die CPU nun vier Register besitzen:

$|x\rangle$ Ein n-Qubit Register für den Index, das mit $|0\rangle^{\otimes n}$ initialisiert wird.

$|s\rangle$ Ein Register, das den gesuchten Zustand enthält. Es bleibt über die Dauer des Algorithmus unverändert. Mit diesem Register werden die Datenbankeinträge verglichen.

$|d\rangle$ Ein Datenregister aus l Qubits, das mit $|0\rangle^{\otimes n}$ initialisiert wird. In dieses Register werden die Daten aus dem Speicher geladen.

$|f\rangle$ Ein 1-Qubit Register, das mit $(|0\rangle - |1\rangle)/\sqrt{2}$ initialisiert wird.

Beim Aufbau des Speicher gibt es zwei Möglichkeiten. Zum einen könnten wir einen Quantenspeicher benutzen, der mit 2^n Speicherzellen arbeitet. Diese Zellen enthalten die l-Qubit langen Einträge, also $|d_1\rangle \ldots |d_N\rangle$, allgemein $|d_x\rangle$. Zum anderen lässt sich

auch ein klassischer Speicher benutzen, der mit einem quantenmechanischen Adressierungsschema ausgestattet ist. Wir arbeiten dann in diesem Fall mit einem Index x, der in einer Superposition mehrerer Werte sein kann. Wie das funktioniert, erkläre ich später. Die Speicherzellen selbst sind klassisch. Es gibt insgesamt N Stück der Länge l Bit. Sie enthalten die Einträge d_x. Wegen des „Quantenindex“ ist es möglich, eine Superposition von Speicherwerten in die CPU zu laden.
Für den Vergleich eines Datenbankeintrages mit den gesuchten Wert s lädt die CPU im ersten Schritt den Eintrag in das Datenregister $|d\rangle$. Welcher Eintrag geladen wird, bestimmt das Indexregister. Ist es im Zustand $|x\rangle$, wird der Inhalt des x-ten Speicherblocks in $|s\rangle$ geladen.

$$|d\rangle \longrightarrow |d \oplus d_x\rangle \tag{50}$$

Auch hier stellt sich wieder die Frage, wie das Orakel beschaffen sein muss. Es soll das Vorzeichen des Index vertauschen, welcher den Eintrag s im Speicher angibt. Angenommen, die vier Register in der CPU sehen so aus:

$$|x\rangle\,|s\rangle\,|0\rangle^{\otimes n}\,\frac{|0\rangle - |1\rangle}{\sqrt{2}}. \tag{51}$$

Nachdem wir einen Block von Daten aus dem Speicher in das dritte Register geladen haben (`LOAD`),

$$|x\rangle\,|s\rangle\,|d_x\rangle\,\frac{|0\rangle - |1\rangle}{\sqrt{2}}, \tag{52}$$

wird $|s\rangle$ mit $|d_x\rangle$ verglichen. Stimmen die Werte überein, soll sich das Vorzeichen des vierten Registers ändern. Andernfalls passiert nichts.

$$|x\rangle\,|s\rangle\,|d_x\rangle\,\frac{|0\rangle - |1\rangle}{\sqrt{2}} \longrightarrow \begin{cases} -\,|x\rangle\,|s\rangle\,|d_x\rangle\,(|0\rangle - |1\rangle)/\sqrt{2}, & \text{falls } d_x = s; \\ \\ |x\rangle\,|s\rangle\,|d_x\rangle\,(|0\rangle - |1\rangle)/\sqrt{2}, & \text{falls } d_x \neq s. \end{cases} \tag{53}$$

Danach führen wir die `LOAD`-Operation ein zweites Mal durch, um das dritte Register wieder in den Zustand $|0\rangle^{\otimes n}$ zu versetzen. Auf diese Weise ändern wir nur das Vorzeichen des ersten Registers, alle anderen bleiben unverändert.
Damit sind auch schon alle alle Voraussetzungen für das Orakel beschrieben. Jetzt können wir dieses Orakel für unseren Quanten-Suchalgorithmus verwenden und so den gesuchten Datenbankeintrag nach $O(\sqrt{N})$ Schritten bestimmen. Dies entspricht einer quadratischen Beschleunigung im Vergleich zum klassischen Algorithmus.
Abschließend stellt sich die Frage, ob wir den Speicher quantenmechanisch oder klassisch realisieren sollen. Für die zweite Alternative spricht die lange Erfahrung, die man mit klassischer Hardware hat. Außerdem reagiert klassischer Speicher unempfindlicher auf äußere Störungen, die die Informationen in einem quantenmechanischen System sofort zerstören würden. Wie aber können wir klassischen Speicher für quantenmechanische Informationen nutzen? Die Lösung ist ein quantenmechanisches Adressierungsschema, wie es in [NC00] vorgeschlagen wird. Abbildung 23 zeigt dieses Schema.

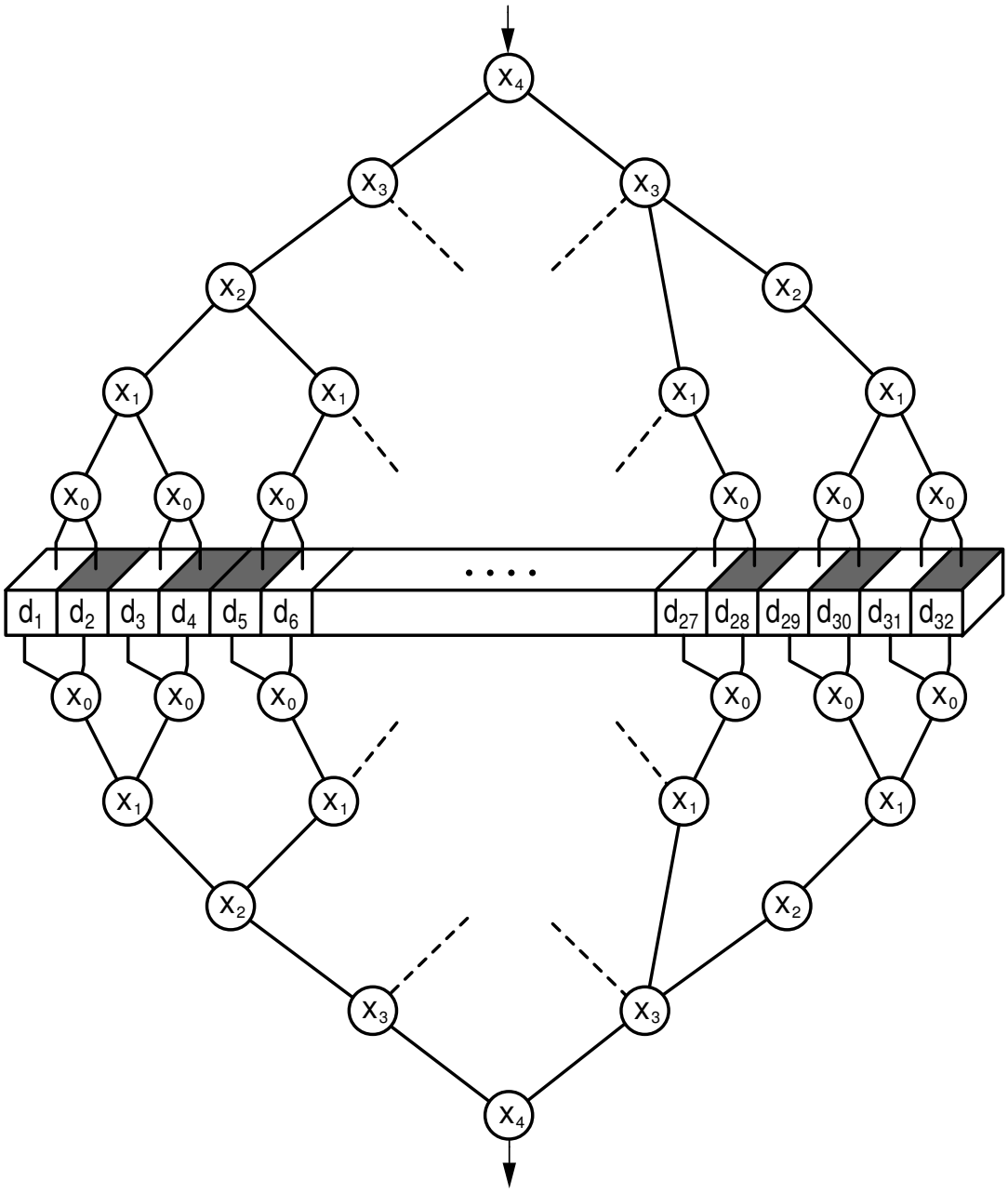


Abbildung 23: Ein Quanten-Adressierungsschema mit fünf Qubits spricht 32 klassische Speicherzellen an. Die Kreise sind Schalter, der Block in der Mitte stellt die klassische Datenbank dar. Dabei repräsentieren weiße Kästen eine 0, graue Kästen eine 1. Diese Darstellung stammt aus [NC00].

Der „Quantenindex“, der binär kodiert ist, wird in einen unären Index übersetzt, mit dem die klassischen Datenbankeinträge adressiert werden können. Nachdem der gewünschte Eintrag ausgelesen wurde, wird die Transformierung rückgängig gemacht. Jeder Kreis in Abbildung 23 entspricht einem Schalter. Die Beschriftung gibt an, welches Qubit den Schalter steuert. Ist das Qubit auf $|0\rangle$ gesetzt, wird das Eingangssignal nach links geleitet, ist es auf $|1\rangle$ gesetzt, wird das Signal nach rechts geleitet. Sollte das

Qubit im Zustand $(|0\rangle + |1\rangle)/\sqrt{2}$ sein, so nimmt das Eingangssignal eine gleichförmige Superposition beider Wege. Die Qubits aus dem Datenregister durchlaufen diese Struktur von oben nach unten, abhängig davon, wie die Schalter gesetzt sind. Am Ende hat man die gewünschte Adresse ausgelesen. Physikalisch kann dieses Schema durch einzelne Photonen implementiert werden, die die Qubits im Datenregister repräsentieren. Die klassische Datenbank könnte eine einfache Plastikscheibe sein, in der eine Null das Licht unverändert transmittiert und eine Eins die Polarisation um 90° ändert.
Damit ist gezeigt, dass wir den Grover-Algorithmus auch zur Suche in unstrukturierten klassischen Datenbanken einsetzen können. Allerdings gibt es zwei Einschränkungen, die den praktischen Nutzen in Frage stellen. Erstens sind die meisten Datenbanken nicht unstrukturiert, sondern besitzen eine Sortierung, meist auch nach mehr als einem Sortierkriterium. In diesen Fällen kann ein klassischer Suchalgorithmus das richtige Ergebnis in $O(\log N)$ Schritten finden. Trotzdem kann ein Quanten-Suchalgorithmus hier Vorteile bieten, da es Suchkriterien geben kann, die vorher nicht bekannt waren, und besonders bei komplexeren Datenbanken Suchanfragen mit einer komplizierten und unvorhergesehenen Struktur auftreten können.
Zweitens müssen wir den Aufwand für das Quanten-Adressierungsschema berücksichtigen. Dieses Schema benötigt $O(N \log N)$ Quantenschalter. Das ist genau so viel technischer Aufwand, wie wir für die Datenbank allein brauchen. Also verdoppelt sich der Hardware-Aufwand durch dieses Konzept. Ob es sich ökonomisch lohnt, einen Quantencomputer für solche Zwecke zu bauen, ist fraglich. Erst wenn diese Schalter eines Tages billig und in großer Stückzahl verfügbar sein sollten, dürfte das Konzept realisierbar sein. Bis dahin allerdings muss erst einmal der Quantencomputer selbst praxistauglich sein.
Es liegt also näher, den Grover-Algorithmus für bisher unlösbare Probleme der Theoretischen Informatik einzusetzen als für die Suche in unstrukturierten Datenbanken, für die es schon eine Vielzahl klassischer Lösungen gibt. Bei Aufgaben wie dem Hamiltonkreis-Problem und dem Problem des Handlungsreisenden scheint der praktische Nutzen, den ein Quantenalgorithmus bietet, größer zu sein als bei Suchproblemen in herkömmlichen Datenbanken.

5.9 Der Grover-Algorithmus ist optimal

Der Grover-Algorithmus braucht $O(\sqrt{N})$ Schritte, um die Lösung zu finden. Das heißt, das Orakel wird mindestens $\Omega(\sqrt{N})$-mal befragt. Im Folgenden möchte ich zeigen, dass diese Laufzeit für einen Orakel-basierten Quanten-Suchalgorithmus optimal ist. Es kann also keinen Quantenalgorithmus geben, der das Suchproblem schneller löst, also weniger Schritte benötigt. Der hier gezeigte Beweis geht auf [BBHT98] zurück.
Zur Vereinfachung nehmen wir an, dass das Suchproblem nur eine Lösung x hat. Wenden wir das Orakel O_x auf einen beliebigen Startzustand $|\psi_0\rangle$ an, bleiben alle Zustände bis auf $|x\rangle$ unverändert. $|x\rangle$ erfährt eine Phasenverschiebung um 90°, wechselt

also das Vorzeichen. Als Operator geschrieben hat O_x die Form

$$O_x = I - 2|x\rangle\langle x|. \tag{54}$$

Ausgehend von $|\psi_0\rangle$ wenden wir O_x k-mal an, wobei k aus Gleichung 33 folgt. Zwischen den Orakelaufrufen finden andere unitäre Operationen $U_1, U2, \ldots, U_k$ statt. Sei

$$|\psi_k^x\rangle \equiv U_k O_x U_{k-1} O_x \ldots U_1 O_x |\psi_0\rangle \tag{55}$$

und

$$|\psi_k\rangle \equiv U_k U_{k-1} \ldots U1 |\psi_0\rangle. \tag{56}$$

$|\psi_k\rangle$ ist also der Zustand, der aus Anfangszustand $|\psi_0\rangle$ hervorgeht, wenn man nur die unitären Operationen $U_1, \ldots, U_k$ anwendet, jedoch nicht die Orakeloperationen.
Jetzt führen wir eine neue Größe D_k ein:

$$D_k \equiv \sum_x ||\psi_k^x - \psi_k||^2. \tag{57}$$

In dieser Gleichung schreiben wir abkürzend ψ statt $|\psi\rangle$, damit die Terme übersichtlicher werden. Es soll gezeigt werden, dass D_k beschränkt ist. Die Größe D_k ist ein Maß für die vom Orakel verursachte „Abweichung nach k Schritten“, das heißt, die Abweichung von einer Entwicklung des Zustandes, die dieser ohne die Orakelaufrufe erfahren hätte. Wenn D_k klein ist, dann sind alle Zustände $|\psi_k^x\rangle$ ungefähr gleich groß und uns wird es kaum möglich sein, den Zustand $|x\rangle$ mit hoher Wahrscheinlichkeit zu messen.
Um zu zeigen, dass D_k beschränkt ist, sind zwei Schritte notwendig:

1. Es ist zu zeigen, dass D_x nicht stärker als $O(k^2)$ wächst.
2. Wir müssen beweisen, dass D_k $\Omega(N)$ ist, wenn wir N verschiedene Zustände unterscheiden wollen. Zur Notation $\Omega(N)$ siehe Anhang A.4.

Den ersten Beweis können mit Hilfe des Verfahrens der vollständigen Induktion führen.
Behauptung: $D_k \leq 4k^2$
Beweis:
(1) Die Behauptung ist wahr für $k = 0$:

$$D_0 = 0. \tag{58}$$

(2) Die Behauptung sei für ein beliebiges k wahr. Für $k + 1$ erhalten wir

$$D_{k+1} = \sum_x ||O_x \psi_k^x - \psi_k||^2 \tag{59}$$

$$= \sum_x ||O_x(\psi_k^x - \psi_k) + (O_x - I)\psi_k||^2. \tag{60}$$

Es gilt allgemein: $||a+b||^2 \leq ||a||^2+2||a||\,||b||+||b||^2$. In unserem Fall ist $a \equiv O_x(\psi_k^x-\psi_k)$ und $b \equiv (O_x - I)\psi_k = -2\,\langle x|\psi_k\rangle\,|x\rangle$. Daraus folgt

$$D_{k+1} \leq \sum_x \left(||\psi_k^x - \psi_k||^2 + 4||\psi_k^x - \psi_k||\,|\,\langle x|\psi_k\rangle\,| + 4|\,\langle\psi_k|x\rangle\,|^2\right). \tag{61}$$

Mit $\sum_x |\,\langle x|\psi_k\rangle\,|^2 = 1$ und der Cauchy-Schwarzschen Ungleichung angewandt auf den zweiten Term auf der rechten Seite von Gleichung 61 erhalten wir

$$\begin{aligned} D_{k+1} &\leq D_k + 4\left(\sum_x ||\psi_k^x - \psi_k||^2\right)^{1/2}\left(\sum_{x'} |\,\langle\psi_k|x'\rangle\,|^2\right)^{1/2} + 4 && (62)\\ &\leq D_k + 4\sqrt{D_k} + 4. && (63)\end{aligned}$$

Da die Behauptung für k gültig ist, folgt

$$D_{k+1} \leq 4k^2 + 8k + 4 = 4(k+1)^2. \tag{64}$$

Somit ist die Behauptung bewiesen.
Im zweiten Beweis müssen wir zeigen, dass $D_k\ \Omega(N)$ ist. Gilt diese Annahme, dann ist die Erfolgswahrscheinlichkeit des Algorithmus ausreichend hoch. Hoch soll in diesem Fall bedeuten, dass $|\,\langle x|\psi_k^x\rangle\,|^2 \geq \frac{1}{2}$, das heißt, wir können die Lösung x bei einer Messung mit der Mindestwahrscheinlichkeit $1/2$ messen.
Statt $|x\rangle$ können wir auch $e^{\imath\theta}\,|x\rangle$ schreiben, da die Phase nicht die Wahrscheinlichkeit beeinflusst, dass der Zustand gemessen wird. Deshalb können wir ohne Beschränkung der Allgemeinheit annehmen, dass

$$\langle x|\psi_k^x\rangle = |\,\langle x|\psi_k^x\rangle\,|. \tag{65}$$

Daraus folgt

$$||\psi_k^x - x||^2 = 2 - 2|\,\langle x|\psi_k^x\rangle\,| \leq 2 - \sqrt{2}. \tag{66}$$

Sei $E \equiv \sum_x ||\psi_k^x - x||^2$, dann folgt $E_k \leq (2-\sqrt{2})N$. Sei außerdem $F_k \equiv \sum_x ||x-\psi_k||^2$, dann ergibt sich

$$\begin{aligned} D_k &= \sum_x ||(\psi_k^x - x) + (x - \psi_k)||^2 && (67)\\ &\geq \sum_x ||\psi_k^x - x||^2 - 2\sum_x ||\psi_k^x - x||\,||x - \psi_k|| + \sum_x ||x - \psi_k||^2 && (68)\\ &= E_k + F_k - 2\sum_x ||\psi_k^x - x||\,||x - \psi_k||. && (69)\end{aligned}$$

Jetzt können wir wieder die Cauchy-Schwarzsche Ungleichung zu Hilfe nehmen und erhalten

$$\sum_x = ||\psi_k^x - x||\,||x - \psi_k|| \leq \sqrt{E_k F_k}, \tag{70}$$

woraus folgt

$$D_k \geq E_k + F_k - 2\sqrt{E_k F_k} = \left(\sqrt{F_k} - \sqrt{E_k}\right)^2. \tag{71}$$

Da $F_k \geq 2N - 2\sqrt{N}$ und $E_k \leq (2-\sqrt{2})N$, ergibt sich

$$D_k \geq cN \tag{72}$$

für genügend großes N. Dabei ist c eine beliebige Konstante kleiner $\left(\sqrt{2} - \sqrt{2-\sqrt{2}}\right)^2 \approx 0.42$. Da wir wissen, dass $D_k \leq 4k^2$, folgt schließlich

$$k \geq \sqrt{cN/4}. \tag{73}$$

Somit ist gezeigt, dass wir $\Omega(\sqrt{N})$ Orakelaufrufe brauchen, bis wir die Lösung des Suchproblems mit der Mindestwahrscheinlichkeit 1/2 bestimmen können. Kein Quantenalgorithmus kann schneller sein. Der Grover-Algorithmus ist mit seinen $O(\sqrt{N})$ Schritten also der bestmögliche Quanten-Suchalgorithmus.
Was können wir dieses Ergebnis interpretieren? Auf der einen Seite ist es ein gutes Ergebnis, bedeutet es doch, dass Lov Grover das Optimale herausgeholt hat. Ein besserer Algorithmus lässt sich nicht finden. Den linearen Aufwand eines klassischen Algorithmus hat Grover quadratisch reduzieren können und dabei die Stärken der Quantenmechanik ausgenutzt. Auf der anderen Seite ist das Ergebnis ein wenig enttäuschend, denn man hätte vielleicht erwarten können, dass sich eine noch größere Verbesserung ergibt. Der Shor-Algorithmus zum Beispiel bietet eine Verbesserung von exponentieller zu polynomieller Komplexität. Übertragen auf eine Suchalgorithmus hätte man also bei einem Suchraum mit N Elementen $O(\log N)$ Schritte erwarten können. Dass der Grover-Algorithmus tatsächlich „nur“ eine quadratische Verbesserung bietet, liegt in erster Linie an der fehlenden Struktur des Suchraums. Peter Shor fand bei der Entwicklung seines Algorithmus eine „versteckte“ Struktur, die er quantenmechanisch nutzen konnte. Das ist im Falle des Grover-Algorithmus nicht möglich.

6 Simulation auf einem klassischen Computer

6.1 Zielsetzung

Jeder klassische Computer kann einen Quantencomputer simulieren. Im folgenden untersuche ich, wie gut sich der Grover-Algorithmus mit Hilfe einer klassischen Programmiersprache, in diesem Fall Java, nachbilden lässt. Dazu werde ich mich an dem Schaltkreis orientieren, der in [NC00] beispielhaft für den Grover-Algorithmus mit zwei Qubits und einer Lösung angegeben ist, und diesen für den allgemeinen Fall mit mehr Qubits und mehreren Lösungen erweitern. Das Programm wird also elementare Gatter-Operationen verknüpfen und dabei die quantenmechanischen Regeln beachten. Am Ende, nach erfolgreicher Simulation, werde ich den Ressourcenaufwand auswerten, das heißt, wie viel Zeit und Speicher das Programm benötigt hat.

6.2 Die Programmiersprache Java

Die Simulation geschieht mit Java[7], einer objektorientierten und plattformunabhängigen Programmiersprache.
Objektorientiert bedeutet, dass abstrakte Objekte untereinander kommunizieren. Ein Objekt ist eine Kapselung von Daten und Funktionen. Über den so genannten *Konstruktor* werden Objekte erzeugt; der *Destruktor* zerstört Objekte. Im Laufe ihres Lebenszyklus können Objekte über Nachrichten miteinander interagieren. Auf diese Weise entsteht das Programm.
Plattformunabhängig meint, dass man beim Programmieren unabhängig vom Betriebssystem oder der eingesetzten Hardware ist. Der Java-Compiler erzeugt *Bytecode*, der von der Laufzeitumgebung (*Java Virtual Machine*) auf beliebigen Architekturen ausgeführt werden kann.
Der Begriff *Klasse* steht in Java für eine Einheit von zusammengehörigem Code. Eine Klasse bildet den Bauplan für ein Objekt. So gibt es in meinem Programm zum Beispiel die Klassen `Qubit.java` oder `Matrix.java`. Mehrere Klassen lassen sich zu einem *Package* zusammenfassen, wodurch man größere Projekte sinnvoll strukturieren kann.

6.3 Die Klassen in der Simulation

Die Simulation besteht aus den drei Packages `math`, `gates` und `grover`. Im Package `math` gibt es die beiden Klassen `Complex.java` und `Matrix.java`, die die Grundlage für alle mathematische Operationen in dieser Applikation bilden. Das Package `gates` enthält die Klassen für alle Quantengatter, die für den Grover-Algorithmus benötigt werden. Der eigentliche Algorithmus wird im Package `grover` zusammengebaut. Dort gibt es Klassen für Qubits und Quantenregister sowie für die bedingte Phasenverschiebung, das Orakel und die Iteration. Der Messvorgang, der das quantenmechanische System in ein klassisches überführt, ist ebenfalls eine eigene Klasse in diesem Package. Es folgt eine kurze Übersicht über alle Klassen:

Das Package `math`:

- `Complex.java`
- `Math.java`

Das Package `gates`:

- `CNot.java`
- `Gate.java`
- `Hadamard.java`
- `Id.java`

[7] http://java.sun.com

- `Not.java`

Das Package `grover`:

- `Grover.java`
- `GroverIteration.java`
- `Measurement.java`
- `PhaseShift.java`
- `Qubit.java`
- `Register.java`

6.4 Die Klassen im Detail

6.4.1 `Complex.java` im Package `math`

Da es in Java keine komplexe Zahlen gibt, war eine eigene Implementierung dieses Zahlentyps nötig. Eine komplexe Zahl besteht aus Real- und Imaginärteil, welche in dieser Klasse als `double`-Werte, also 64 Bit Gleitkommazahlen, hinterlegt sind. Neben den üblichen Konstruktoren, die die Werte initialisieren, gibt es Methoden, um den Real- und Imaginärteil neu setzen oder auszulesen, so genannte *Getter- und Setter-Methoden*. Die Variablen `re` und `im` selbst sind von außen nicht sichtbar, sondern nur über diese Methoden ansprechbar.
Weiterhin sind die Addition und die Multiplikation von komplexen Zahlen als eigene Methoden implementiert, so dass man alle benötigten Berechnungen ausführen kann. Die Methode `abs()` liefert den Betrag der komplexen Zahl.
Um eine String-Repräsentation dieses Datentyps auf der Konsole auszugeben, besitzt die Klasse eine eigene `toString()`-Methode, die die Zahl nach dem Muster $re + im \cdot i$ darstellt. Dabei berücksichtige ich eventuelle numerische Ungenauigkeiten, die beim Rechnen mit `double`-Werten auftreten können. Liegt zum Beispiel der Realteil nahe bei 0 (in einem Abstand kleiner ϵ, wobei $\epsilon = 10^{-15}$), so wird der Wert wie eine 0 behandelt und nur der Imaginärteil dargestellt. Für den Imaginärteil gilt das Entsprechende. Sind beide Werte 0, so wird die Zahl als 0.0 ausgegeben. Diese Maßnahmen sollen die Übersichtlichkeit bei der Darstellung von komplexen Zahlen erhöhen. Beim Rechnen mit komplexen Zahlen finden diese Überprüfungen nicht statt, weil sie zu viel Rechenzeit kosten würden.
Die Klasse `Complex` enthält das statische Feld `eins`, das eine komplexe Zahl mit Realteil 1 und Imaginärteil 0 liefert. Das dient als Abkürzung, weil diese Zahl häufig benutzt wird und auf diese Weise nicht jedes Mal ein neues Objekt erzeugt werden muss.

6.4.2 `Matrix.java` im Package `math`

Die Klasse repräsentiert eine $n \times m$-Matrix mit n Zeilen und m Spalten. Die Einträge der Matrix sind komplexe Zahlen, die als zweidimensionales Array abgelegt werden. Dieses Array ist `public`, also von außerhalb der Klasse veränderbar, so dass man einzelne Matrixelemente leicht verändern kann.
Es gibt den Standardkonstruktor, der eine leere 1×1-Matrix erzeugt, einen weiteren Konstruktor für eine 1×1-Matrix, deren Wert übergeben werden kann, und einen Konstruktor mit den Vorgaben für n und m, der eine leere Matrix dieser Größe erzeugt.
Darüber hinaus sind als Matrixoperationen das Matrixprodukt und das Tensorprodukt implementiert.
Die Zeilen- und Spaltenzahl einer Matrix kann über die Methoden `getRows()` und `getColumns()` abgefragt werden. Unterklassen können auch direkt auf die Felder `n` und `m` zugreifen.
Die Methode `toString()` stellt die Matrixelemente als Tabelle dar, ähnlich der üblichen Schreibweise auf Papier.

6.4.3 `Gate.java` im Package `gates`

Die Klasse `Gate` ist ein Marker-Interface, das heißt, sie besitzt keine eigene Funktionalität und definiert auch keine Felder oder Methoden. Sie soll nur Klassen „markieren“, die ein Quantengatter repräsentieren. Alle Klassen, die dieses Interface implementieren, kennzeichnen sich somit selbst als Gatter.

6.4.4 `CNot.java` im Package `gates`

Hier wird das `CNOT`-Gatter implementiert, das auf zwei Qubits wirkt. Die Klasse ist eine Unterklasse von `Matrix`, genau wie alle folgenden Gatter-Klassen. Der Standardkonstruktor erstellt eine 4×4-Matrix mit den entsprechenden Einträgen. Es gibt einen zweiten Konstruktor, bei dem die Anzahl der Kontroll-Qubits übergeben werden kann, so dass man eine Art generalisiertes Toffoli-Gatter mit beliebig vielen Kontroll-Qubits und einem Ziel-Qubit erstellen kann. Um die Applikation einfacher und übersichtlicher zu halten, sind diese Gatter auch Objekte des Typs `CNOT`, auch wenn ein `CNOT`-Gatter rein formal nur auf zwei Qubits wirken kann.
Die Klasse `CNOT` implementiert das `Gate`-Interface.

6.4.5 `Hadamard.java` im Package `gates`

Das Hadamard-Gatter ist als Ein-Qubit-Gatter durch eine 2×2-Matrix repräsentiert. Der Vorfaktor $1/\sqrt{2}$ ist ein statisches Feld. Es gibt eine weitere statische Methode `hadamard()`, die das Tensorprodukt beliebig vieler Hadamard-Gatter berechnet und die entsprechende Matrix zurückgibt. So wird der Code einfacher und lesbarer, wenn zum Beispiel auf jedes Qubit eines Quantenrechners ein Hadamard-Gatter wirken soll.
Auch die Klasse `Hadamard` implementiert das `Gate`-Interface.

6.4.6 `Id.java` im Package `gates`

Die Klasse `Id` stellt ein Identitätsgatter dar und implementiert somit auch das `Gate`-Interface. Dieses Quantengatter lässt ein Qubit unverändert und wird demnach durch eine 2×2-Einheitsmatrix beschrieben. Dieses Gatter ist notwendig, um die Gesamtmatrix berechnen zu können, wenn in einem Schritt des Algorithmus' bestimmte Qubits unverändert bleiben. Auch hier gibt es eine statische Methode, um das Tensorprodukt aus mehreren `Id`-Gattern zu berechnen (`id()`).

6.4.7 `Not.java` im Package `gates`

Hier wird das `NOT`-Gatter algorithmisch abgebildet. Der Konstruktor erstellt die Pauli Matrix σ_x. Es handelt sich also um ein Quantengatter, das ein Qubit invertiert. Die statische Methode `not()` berechnet das Tensorprodukt aus beliebig vielen `NOT`-Gattern.
Auch diese Klasse implementiert das `Gate`-Interface.

6.4.8 `Qubit.java` im Package `grover`

Diese Klasse ist eine Unterklasse von Matrix und repräsentiert ein Qubit, also einen Zustand der Form $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. Ein Qubit wird durch eine Matrix mit zwei Zeilen und einer Spalte dargestellt. In der ersten Zeile wird die komplexe Zahl α gespeichert, in der zweiten Zeile die komplexe Zahl β. Auf diese Weise können wir die üblichen Matrixoperationen und damit alle oben beschriebenen Gatter auf ein Qubit anwenden. Der Standardkonstruktor erstellt ein Qubit mit $\alpha = 1$ und $\beta = 0$. Desweiteren gibt es zwei Konstruktoren, um beide Werte direkt zu setzen, einmal für die übergebenen komplexen Zahlen und einmal für die übergebenen `double`-Werte, die in komplexe Zahlen konvertiert werden. Hinzu kommen die üblichen Getter- und Setter-Methoden für α und β, wobei eine Fehlermeldung erscheint, wenn die Gesamtwahrscheinlichkeit $|\alpha|^2 + |\beta|^2$ größer als 1 wird.
Auch in dieser Klasse gibt es eine eigene `toString()`-Methode, die ein Qubit mit seinen α- und β-Werten darstellt.

6.4.9 `Register.java` im Package `grover`

Ein Register speichert mehrere Qubits. Die Klasse `Register` ist eine Unterklasse von Matrix, da ein Quantenregister durch eine n×1-Matrix repräsentiert werden kann. Das kleinstmögliche Register besteht aus zwei Qubits und kann durch den entsprechenden Konstruktor instanziiert werden. Der zweite Konstruktor bekommt ein anderes Register als Argument und erzeugt davon eine Kopie. Das Feld `size` hält die Registerbreite und kann über die Methode `getSize()` abgefragt werden. Über die Methode `add()` kann man dem Register ein weiteres Qubit hinzufügen.
Auch diese Klasse hat ihre eigene `toString()`-Methode.

6.4.10 `PhaseShift.java` im Package `grover`

Diese Klasse implementiert die bedingte Phasenverschiebung auf beliebig vielen Qubits, deren Anzahl dem Konstruktor übergeben werden kann. Diese Operation ändert das Vorzeichen auf allen Zuständen bis auf den Zustand $|00\ldots0\rangle$, der unverändert gelassen wird. Dafür werden verschiedene elementare Gatter in fünf Schritten miteinander verknüpft (Abbildung 24).

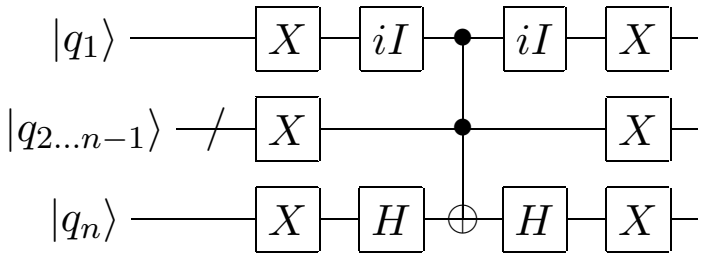


Abbildung 24: Der Schaltkreis um eine bedingte Phasenverschiebung $2\,|0\ldots0\rangle\langle0\ldots0 - I|$ für n Qubits durchzuführen. Diese Implementation basiert auf [LMP03]

Der Standardkonstruktor erzeugt ein `PhaseShift`-Objekt für zwei Qubits, also den einfachsten Fall. Der andere Konstruktor erwartet als Argument die Registerbreite und ruft die Methode `calcMatrix()` auf, die die entsprechenden Operationen durchführt. Für die fünf Schritte werden die Matrizen erstellt und anschließend multipliziert. Die Gesamtmatrix repräsentiert dann die die bedingte Phasenverschiebung.

6.4.11 `Oracle.java` im Package `grover`

Auch die Klasse `Oracle` ist eine Unterklasse von `Matrix`. Das Orakel markiert die „Nadel“ im Heuhaufen, indem es das Vorzeichen des entsprechenden Zustandes ändert. Dies lässt sich durch eine Einheitsmatrix repräsentieren, die an der richtigen Stelle statt einer 1 eine -1 enthält. Ursprünglich wollte ich das Orakel auch aus elementaren Gattern zusammenbauen, wie in [NC00] auf Seite 256 beschrieben, doch leider stellte sich heraus, dass sich das dortige Beispiel für zwei Qubits nicht so einfach auf mehr Qubits verallgemeinern ließ. Der Algorithmus funktionierte zwar, das heißt, der Lösungszustand hatte eine höhere Wahrscheinlichkeit als alle anderen Zustände, aber diese Wahrscheinlichkeit war kleiner als der theoretische Wert, der über Gleichung 34 berechnet werden kann. Je mehr Qubits der Suchraum umfasste, desto geringer war die Wahrscheinlichkeit des Lösungszustandes. Sie sank sogar deutlich unter 0.5. Deshalb habe ich mich entschlossen, das Orakel „fest“ zu implementieren. Weil es nicht aus elementaren Gatteroperationen zusammengesetzt ist, benötigt das Programm keine Hilfsqubits für das Orakel.
Der Konstruktor erwartet die Registerbreite und ein Array mit den Lösungszuständen als Parameter. Man muss also die Lösungen des Suchproblems vorher kennen, was aber den Sinn der Simulation nicht in Frage stellt. Wünschenswert wäre ein „echtes“ Quantenorakel, was in einer Datenbank eigenständig die Lösungszustände erkennt.

Leider habe ich in der Literatur nichts zu diesem Thema finden können. Eine eigene Implementation ließ sich angesichts der knappen Zeit nicht mehr verwirklichen.

6.4.12 `GroverIteration.java` im Package `grover`

Die Klasse `Grover-Iteration` liefert uns beim Aufruf eine Matrix, die eine vollständige Grover-Iteration repräsentiert. Diese besteht aus vier Schritten, die jeweils auf alle Qubits des Registers wirken:

- Orakelaufruf
- Hadamard-Gatter
- bedingte Phasenverschiebung
- Hadamard-Gatter

Dem Konstruktor der Klasse übergibt man die Registerbreite und ein Array mit den Lösungen. Mit diesen Angaben werden drei Objekte erzeugt: eines für das Orakel, eines für die Hadamard-Operationen und eines für die bedingte Phasenverschiebung. Anschließend berechnet die Methode `calcMatrix()` die Gesamtmatrix nach dem obigen Schema.

6.4.13 `Measurement.java` im Package `grover`

Die Klasse repräsentiert eine Messung, die das quantenmechanische System in ein klassisches überführt. Die Messung des Endzustandes beim Grover-Algorithmus soll den gesuchten Zustand ergeben, der am Ende des Algorithmus' in der Überlagerung aus allen Zuständen mit hoher Wahrscheinlichkeit vertreten ist.
Der Konstruktor dieser Klasse erhält die Matrix, die den zu messenden Zustand repräsentiert, und die Zahl der Lösungen des Suchproblems als Parameter. Die Klasse besitzt zwei Methoden: `measure()` und `examine()`. Nur `examine()` ist von außen zugänglich, wobei die eigentliche Messung jedoch in `measure()` vorgenommen wird. Dort wird eine klassische Zufallszahl zwischen 0 und 1 erzeugt und geprüft, ob die Wahrscheinlichkeit des ersten Zustandes größer ist als diese Zufallszahl. Ist das nicht der Fall, dann wird die Wahrscheinlichkeit des zweiten Zustandes auf die des ersten addiert und das Ergebnis wieder geprüft. Sobald die Summe größer ist als die Zufallszahl, wird der Index des letzten addierten Zustandes zurückgegeben, der dann das Messergebnis darstellt. Es ist klar, dass man fast immer den richtigen Zustand misst, wenn die Lösung eine Wahrscheinlichkeit nahe 1 hat, also die Summe stark vergrößert. Gäbe es immer nur eine Lösung, dann würde dieses Vorgehen ausreichen. Sobald mehrere Lösungen im Endzustand enthalten sind, muss man auch mehrmals messen, da man sonst nur eine spezielle Lösung finden könnte. Hierfür dient die Methode `examine()`, die insgesamt 100 einzelne Messungen vornimmt. Es gibt ein Array `percentage`, dessen Einträge angeben, wie oft ein bestimmter Zustand gemessen

wurde. Die m Zustände, die am häufigsten vorkommen, werden in das Array `states` kopiert, welches am Ende der Methode zurückgegeben wird und alle Lösungen enthält. Um einen realen Messvorgang korrekt zu simulieren, müsste bei jedem Aufruf von `measure()` der ganze Algorithmus erneut ablaufen, da die vorherige Messung den quantenmechanischen Zustand zerstört hat. Somit würde man nur genau *einen* Zustand messen, also nur *eine* mögliche Lösung. Um andere Lösungen zu entdecken, müsste die gesamte Prozedur wiederholt werden. In meiner Applikation kann man den quantenmechanischen Endzustand jedoch beliebig oft messen, ohne ihn zu verändern. Ich habe das bewusst so gewählt, um die Laufzeit bei großen Registerbreiten erträglich zu halten. Müsste der gesamte Algorithmus mehrfach ablaufen, so würde die benötigte Zeit mit der Zahl der Lösungen stark ansteigen. Nähere Erläuterungen dazu finden sich im Abschnitt 6.5.

6.4.14 `Grover.java` im Package `grover`

Hier wird der gesamte Algorithmus aus seinen Einzelteilen zusammengesetzt und ausgeführt. Alle hier vorhandenen Felder und Methoden sind statisch. Das Feld `n` speichert die Anzahl der Qubits für den Suchraum, `m` ist die Anzahl der Lösungen und im Array `s` werden die Lösungen hinterlegt. Alle Werte kann der Benutzer bei Programmstart eingeben, wofür die Methode `read()` dient. Dort wird auch geprüft, ob die eingegebenen Werte zulässig und sinnvoll sind. Hierbei muss beachtet werden, dass Indizes in Java immer mit 0 anfangen. Gibt es zum Beispiel 64 Elemente im Suchraum, dann läuft der Index von 0 bis 63. Wenn nötig, muss der Benutzer die Eingabe wiederholen oder das Programm benutzt vorgegebene Werte.
Ist dieser Schritt erledigt, wird die Methode `run()` aufgerufen, die den Algorithmus ausführt. Zuerst wird dazu der Startzustand $|00\ldots0\rangle$ aus n Qubits mittels Hadamard-Gattern in eine gleichmäßige Superposition gebracht. Danach erzeugt das Programm ein neues `GroverIteration`-Objekt `gi`, das k-mal mit sich selbst multipliziert wird. Die Zahl k berechnet die Methode `nrOfIterations()` anhand von n und m. So wird die Grover-Iteration also k-mal durchgeführt, wie es der Algorithmus verlangt. Es entsteht eine Matrix `iteration`, die, multipliziert mit dem Startzustand `start`, den gewünschten Endzustand `end` darstellt. Ein Measurement-Objekt wird erzeugt, das diesen Zustand untersucht und die m Indizes liefert, die Lösungen des Suchproblems sind. Zuletzt ruft die Methode `run()` die Methode `printResult()` auf, die die Ergebnisse auf die Konsole schreibt. Dabei werden die gesuchten Indizes, deren Wahrscheinlichkeiten und der zugehörige Zustand ausgegeben.
Bevor der Algorithmus in `run()` startet, wird die aktuelle Systemzeit in einer Variablen gespeichert. Wenn der Endzustand berechnet ist, wird dies ebenfalls gemacht. So kann die Laufzeit des Programms ermittelt werden, die mit der Anzahl der Qubits steigt und dem Benutzer angezeigt wird.
Ein typischer Programmablauf sieht zum Beispiel so aus:

```
Der Grover-Algorithmus
```

```
Im folgenden werden Sie gebeten, die Anzahl n der Qubits fuer den Suchraum
einzugeben. n sollte auf einem aktuellen PC nicht groesser als 8 sein, es sei
denn, Sie haben einen richtig schnellen Rechner. n muss groesser als 1 sein,
sonst koennen Sie gleich raten!
Bitte geben Sie n ein: 7
Der Suchraum besteht also aus 128 Elementen.

Wie viele Loesungen soll es geben?
Bitte eine Zahl zwischen 1 und 64: 3
Es gibt also 3 Loesungen.

Welche Loesung(en) soll das Orakel erkennen?
Bitte eine Zahl zwischen 0 und 127: 12
Bitte eine Zahl zwischen 0 und 127: 54
Bitte eine Zahl zwischen 0 und 127: 90
Als Loesung wurden die Elemente 12, 54, 90 gewaehlt.

Und los geht's!
Erstelle Matrix fuer die Grover-Iteration. Bitte einen Moment Geduld.
Fuehre 5 Iteration(en) durch. Das kann dauern...
Dauer: 4.313 sec.

Ergebnis:
Die 1. Loesung hat den Index 54 mit Wahrscheinlichkeit 0.3285.
Der Loesungszustand ist: |0110110>
Die 2. Loesung hat den Index 90 mit Wahrscheinlichkeit 0.3285.
Der Loesungszustand ist: |1011010>
Die 3. Loesung hat den Index 12 mit Wahrscheinlichkeit 0.3285.
Der Loesungszustand ist: |0001100>
```

6.5 Analyse

Die Simulation habe ich auf einem Pentium 4 Northwood 3.0 GHz mit 1024 MB RAM unter Windows 2000 mit Java 1.5 Beta 2 durchgeführt. Die Quelldateien, die kompilierten Dateien und die Dokumentation sind im Container `grover.jar`[8] zusammengefasst. Die Applikation kann über die Kommandozeile mit dem Befehl `java -jar -Xmx128m grover.jar` gestartet werden. Der Parameter `-Xmx128m` bewirkt, dass die Java Virtual Machine mehr Speicher für die Applikation reserviert als üblich. Damit wird verhindert, dass das Programm wegen Speichermangels abbricht, was auf meinem Rechner bei mehr als 8 Qubits passiert wäre.
Um zu untersuchen, wie sehr die Zahl der Qubits die Dauer des Programms beeinflusst, gibt es die Zeitmessung in der Klasse `Grover`. Die Tabelle 3 gibt einen Überblick für

[8] JAR = Java ARchive

die Fälle $n = 5$ bis $n = 9$.

Anzahl der Qubits	Matrizengröße	Zahl der Iterationen	Dauer in *sec*
5	32×32	4	0.079
6	64×64	6	0.516
7	128×128	8	5.422
8	256×256	12	70.05
9	512×512	17	818.02

Tabelle 3: Matrizengröße, Zahl der Grover-Iterationen und benötigte Zeit der Applikation bei wachsender Größe des Suchraums. Bei allen Durchläufen suchte das Programm nach genau einem Element.

Wie deutlich zu erkennen ist, kann der Grover-Algorithmus für kleine Suchräume mit Java noch sehr zügig simuliert werden. Jedoch nimmt die Laufzeit des Programmes um mehr als das Zehnfache zu, wenn man den Suchraum um ein Qubit erweitert. So rechnet die Simulation eine knappe Viertelstunde, wenn der Suchraum 512 Elemente, also 9 Qubits, umfasst. Größere Werte habe ich nicht mehr getestet. Allen Simulationen wurden direkt hintereinander ausgeführt, ohne dass nebenbei andere speicher- oder rechenintensive Anwendungen gestartet waren. Das Programm hat immer nach genau einem Element gesucht.
Bei wenigen Qubits werden CPU und Speicher kaum in Anspruch genommen. Ab 8 Qubits jedoch liegt die CPU-Belastung während einer Simulation recht konstant bei etwa 50%, wie eine Analyse durch den *Windows Task Manager* ergab. Das Programm benutzt bei 8 Qubits circa 40, bei 9 Qubits circa 130 Megabyte an Speicher. Zu Testzwecken habe ich auch den Fall $n = 10$ Qubits simuliert, die Simulation jedoch aus Zeitgründen nicht bis zum Ende laufen lassen. Der Speicherverbrauch lag hier bei etwa 500 Megabyte, die CPU-Belastung wiederum um 50%.
Die Simulation auf einem klassischen Computer kann also durchgeführt werden. Allerdings steigt der Ressourcenverbrauch stark an, wenn man die Zahl der Qubits erhöht. Bedenkt man, dass 10 Qubits = 1024 Elemente ein recht kleiner Wert für eine Datenbank ist, wird deutlich, dass ein klassischer Computer mit dieser Simulation sehr schnell überfordert ist.

6.6 Vergleich mit QuCalc

QuCalc[9] ist ein *Mathematica*-Paket, mit dem sich ein Quantencomputer simulieren lässt. Man kann beliebige Schaltkreise zusammenstellen und simulieren, weil alle benötigten Datentypen und Gatter gebrauchsfertig implementiert sind.
Ich habe mit *QuCalc* unter *Mathematica* 5.0 den Grover-Algorithmus simuliert, um herauszufinden, ob meine Applikation eventuell schneller oder langsamer arbeitet. Dazu habe ich den Algorithmus für 5 bis 9 Qubits laufen lassen und mit der *Mathematica*-Funktion `Timing[]` die Zeit für einen Durchlauf gemessen. Auch hier wurde nur

[9] http://crypto.cs.mcgill.ca/QuCalc/

nach einer Lösung gesucht. Außerdem hatte der Schaltkreis dieselbe Struktur wie mein Java-Programm.
Es ergeben sich bei beiden Simulationen qualitativ die gleichen Zeiten, beginnend bei etwa 100 Millisekunden für 5 Qubits bis zu einer Viertelstunde bei 9 Qubits. Auch bei *QuCalc* steigt die Laufzeit des Algorithmus also um etwa das Zehnfache, wenn der Suchraum um ein Qubit erweitert wird. Da QuCalc ebenfalls mit Matrizen arbeitet, überrascht dieses Ergebnis nicht. Unterschiede gibt es jedoch beim Speicherverbrauch, wo QuCalc mit etwa 35 Megabyte bei 9 Qubits deutlich genügsamer ist als mein Java-Programm. Hierfür wird die Speicherverwaltung von *Mathematica* 5.0 verantwortlich sein, die den Speicherplatzbedarf der verwendeten Datentypen (komplexe Zahlen) automatisch minimiert.

7 Ausblick

Niemand weiß, ob es in den nächsten Jahren und Jahrzehnten gelingen wird, die technischen Hürden beim Bau von Quantencomputern zu überwinden. Solange es keine praxistauglichen Quantencomputer mit einigen Dutzend bis hunderten von Qubits gibt, werden Simulationen auf klassischen Rechnern sinnvoll sein.
Es wäre sicherlich interessant, meine Applikation in einer anderen Programmiersprache zu schreiben. Anbieten würden sich C und C++ sowie Fortran. Da C++ auch objektorientiert ist, könnte man den Aufbau des Programmes übernehmen und müsste fast nur die Syntax anpassen. Erfahrungsgemäß sind C++-Programme schneller als in Java geschriebene Software, so dass sich die Laufzeit meiner Simulation wahrscheinlich verkürzen ließe. Der Speicherbedarf allerdings dürfte kaum sinken, wenn man auch in C++ mit 64-Bit Gleitkommazahlen arbeitet. Dadurch wäre die Simulation auch weiterhin auf relativ kleine Suchräume begrenzt.
Die Sprache C arbeitet sehr hardwarenah und ist deswegen vor allem schnell. Weil sie nicht objektorientiert ist, müsste die Struktur meiner Simulation von Grund auf verändert werden. Der Vorteil wäre ein hoher Geschwindigkeitszuwachs, da die rechenintensiven Matrixoperationen beschleunigt würden. C-Programme arbeiten in der Regel noch schneller als C++-Programme. Aber auch C kann nicht verhindern, dass sich der Speicherbedarf mit jedem Qubit verdoppelt.
Fortran ist auf das Rechnen mit Matrizen und komplexen Zahlen spezialisiert, weshalb es sich zur Simulation des Grover-Algorithmus bestens eignen sollte. Meine Kenntnisse über Fortran sind jedoch begrenzt, so dass ich mich für Java entschied. Fortran ist genau wie C eine prozedurale Sprache, also nicht objektorientiert, auch wenn es in neueren Versionen von Fortran Ansätze in diese Richtung gibt. Meine Simulation müsste ich für Fortran komplett umstrukturieren, wofür aber eine höhere Geschwindigkeit entschädigen könnte. Wahrscheinlich kann ein Fortran-Programm mit komplexen Matrizen Speicher sparender rechnen kann als ein Java/C++-Programm, da Fortran spezielle Speichertechniken und Lösungsstrategien für die Matrizenrechnung besitzt.
Die Simulation selbst hat einen Schwachpunkt: Man ist gezwungen, dem Programm vorab die Lösungen mitzuteilen, damit das richtige Orakel „erzeugt“ werden kann.

Zwar ist das Orakel im Grover-Algorithmus nur als Black Box beschrieben, aber in einer realen Simulation möchte der User eine echte Suche durchführen, das heißt, die Lösungen sollen vorher nicht bekannt sein. Dazu müsste man ein universelles quantenmechanisches Orakel implementieren, das Elemente in einer Datenbank anhand bestimmter Kriterien erkennen kann. Dies müsste eine unitäre Operation sein, die auf alle Datenbankeinträge parallel wirkt, und sie müsste sich mit vertretbarem Aufwand aus elementaren Gattern zusammensetzen lassen. Je nach Größe der Datenbank kann dadurch ein erheblicher zusätzlicher Hardwareaufwand entstehen. Erstaunlich ist, dass dieser Punkt in der Literatur häufig übergangen wird. Jedenfalls konnte ich keine Ausführungen dazu finden.

Anhang

A Eine sehr kurze Einführung in die Theoretische Informatik

A.1 Die Turingmaschine

Die Turingmaschine[10] ist ein mathematisches Konstrukt, um eine Klasse von berechenbaren Funktionen zu bilden.
Die Turingmaschine besteht aus vier Elementen:

- Ein Programm.
- Ein Speicherband, das sich wie ein Computerspeicher verhält. Es hat unendlich viele Feldern, in denen jeweils genau ein Zeichen gespeichert werden kann.
- Ein Schaltwerk mit endlich vielen Zuständen. Es steuert das Verhalten der Turingmaschine.
- Ein Programm-gesteuerter Lese- und Schreibkopf, der auf dem endlosen Speicherband ein Feld nach links oder rechts rücken, ein Zeichen lesen, schreiben oder löschen und stehen bleiben kann.

Turing zeigte, dass diese Maschine jedes algorithmisierbare (berechenbare) Problem lösen kann.

A.2 Algorithmen

Ein **deterministischer** Algorithmus ist ein Algorithmus, bei dem nur definierte und reproduzierbare Zustände auftreten. Anders gesprochen heißt das, auf eine Anweisung im Algorithmus folgt unter den gleichen Voraussetzung immer die gleiche Anweisung. Folglich können bei einem **nicht-deterministischen** Algorithmus nicht reproduzierbare und undefinierte Zustände auftreten. Zum Beispiel hat ein Algorithmus, der eine Zufallszahl liefert, ein nicht-deterministisches Verhalten.

A.3 Komplexitätsklassen

Die Menge aller Sprachen, die ein deterministischer Algorithmus (bzw. eine deterministische Turingmaschine) in Polynomialzeit lösen kann, bezeichnet man mit **P**.
NP ist die Menge aller Sprachen, die ein nicht-deterministischer Algorithmus (bzw. eine nicht-deterministische Turingmaschine) in Polynomialzeit lösen kann.
Der Begriff **NP-Vollständigkeit** klassifiziert eine Menge von Entscheidungsproblemen[11], die in gewisser Hinsicht besonders schwierig sind und für die angenommen wird,

[10] Benannt nach dem englischen Mathematiker Alan Turing, der diese Idee 1936 veröffentlichte.
[11] Probleme, für die zu einer gegebenen Eingabe als Lösung nur zwei Antworten (z. B. ja oder nein) vorgesehen sind

dass keine effizienten Algorithmen existieren, die diese lösen.
Probleme, die zu dieser Menge gehören, bezeichnet man als **NP-vollständig**. Dazu zählen unter anderem:

- Das Erfüllbarkeitsproblem der Aussagenlogik
- gerichteter bzw. ungerichteter Hamiltonkreis (auch Hamiltonpfad)
- Problem des Handlungsreisenden
- das Rucksackproblem (Knap-Sack-Problem)
- 3COLOR
- ...

A.4 Landau-Notation

Die Landau-Symbole beschreiben Mengen von Funktionen, die ähnliches Wachstumsverhalten haben. Drei dieser Symbole verwende ich in dieser Bachelorarbeit:
Die 'O'-Notation gibt eine obere Grenze für die Laufzeit an. Seien $f(n)$ und $g(n)$ zwei Funktionen. Dann meint '$f(n)$ ist $O(g(n))$', dass die Funktion $f(n)$ in der Klasse der Funktionen $O(g(n))$ ist. Das bedeutet, es gibt Konstanten c und n_0, so dass für alle Werte $n > n_0$ gilt: $f(n) \leq cg(n)$. Für genügend großes n stellt also die Funktion $g(n)$ eine obere Grenze für $f(n)$ dar. Somit kann uns die O-Notation angeben, wie sich ein Algorithmus im schlechtesten Fall (worst case scenario) verhält.
Man benutzt die Ω-Notation, um das Verhalten einer ganzen Klasse von Algorithmen zu beschreiben. Dann ist wichtig, dass man eine untere Grenze für die benötigten Ressourcen angeben kann. Eine Funktion $f(n)$ ist $\Omega(g(n))$, falls es Konstanten c und n_0 gibt, so dass für alle $n \leq n_0$ gilt: $cg(n) \leq f(n)$. Also ist $g(n)$ eine untere Grenze für $f(n)$.
Die Θ-Notation meint, dass $f(n)$ sich asymptotisch wie $g(n)$ verhält, bis auf ein paar Faktoren, die vernachlässigt werden können. Das bedeutet, $f(n)$ ist $\Theta(g(n))$, wenn es sowohl $O(g(n))$ als auch $\Omega(g(n))$ ist.

B Phasenabschätzung

Phasenabschätzung bezeichnet ein Verfahren, um den Eigenwert zu einem Eigenvektor eines unitären Operators abzuschätzen. Eine ausführliche Beschreibung findet sich in [Mos99]. Sei U der unitäre Operator und $|u\rangle$ der Eigenvektor mit Eigenwert $\exp^{2\pi i\varphi}$. Das Ziel ist nun, die Phase φ näherungsweise zu bestimmen.
Im folgenden möchte in in aller Kürze beschreiben, wie das Verfahren grundsätzlich funktioniert.
Der Algorithmus hat drei Voraussetzungen:

- eine Black Box, die eine bedingte unitäre Operation U^j für eine ganze Zahl j durchführt
- einen Eigenzustand $|u\rangle$ von U mit Eigenwert $\exp^{2\pi i\varphi_u}$
- und t Qubits, die mit $|0\rangle$ initialisiert werden. Hierbei ist $t = n + \lceil \log(2 + \frac{1}{2\epsilon}) \rceil$

Mit den folgenden Operationen lässt sich φ_u auf n Bit Genauigkeit bestimmen. Dabei bezeichnet $\tilde{\varphi_u}$ die Approximation von φ_u

1. Anfangszustand aus zwei Registern präparieren:
$$|0\rangle |u\rangle \tag{74}$$
2. Alle Zustände im ersten Register in eine gleichmäßige Superposition bringen:
$$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle |u\rangle \tag{75}$$
3. Black Box anwenden:
$$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle U^j |u\rangle = \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} \exp^{2\pi i j\phi_u} |j\rangle |u\rangle \tag{76}$$
4. Inverse Fouriertransformation anwenden:
$$\rightarrow |\tilde{\varphi_u}\rangle |u\rangle \tag{77}$$
5. Erstes Register messen:
$$\rightarrow \tilde{\varphi_u} \tag{78}$$

Das Schaltbild:

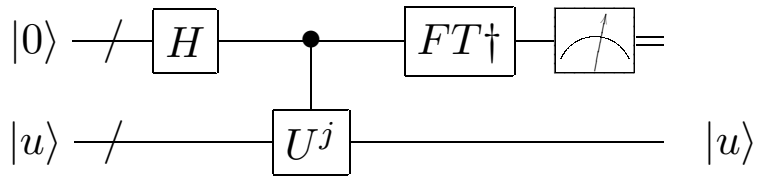


Abbildung 25: Schema des Schaltkreises für Phase estimation. Wie die inverse Fouriertransformation funktioniert, soll hier nicht näher beschrieben werden.

C Die Quelltexte

C.1 Das Package `math`

C.1.1 Die Klasse `Complex.java`

```
package math;
/**
 * @author Uwe Sander
 * @version 10.06.2004
 *
 * Diese Klasse reprasentiert  eine komplexe Zahl. Real- und Imaginaerteil sind
 * 64Bit-Gleitkommazahlen.
 */
public class Complex {
    //Realteil  und Imaginaerteil
    private double re, im;

    /**
     * Statische  Variable,  die eine reelle  1 darstellt ,  verpackt als  komplexe
     * Zahl. Dient als  Abkuerzung, damit fuer die reelle  1 kein Konstruktor
     * aufgerufen werden muss.
     */
    public static Complex eins = new Complex(1.0, 0.0);

    /**
     * Standard-Konstruktor, der Real- und Imaginaerteil mit 0 initialisiert.
     *
     */
    public Complex() {
        re = 0.0;
        im = 0.0;
    }

    /**
     * Konstruktor, der Real- und Imaginaerteil auf die uebergebenen Werte setzt.
     * @param re der Realteil der komplexen Zahl
     * @param im der Imaginaerteil der komplexen Zahl
     */
    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
```

```
/**
 * Konstruktor, der Real– und Imaginaerteil auf den gleichen Wert setzt.
 * @param val der neue Wert des Real– und Imaginaerteils.
 */
public Complex(double val) {
    this.re = val;
    this.im = val;
}

/**
 * Methode, um Real– und Imaginaerteil neu zu setzen
 * @param re der neue Realteil
 * @param im der neue Imaginaerteil
 */
public void set(double re, double im) {
    this.re = re;
    this.im = im;
}

/**
 * Methode, um Realteil neu zu setzen
 * @param re der neue Realteil
 */
public void setRe(double re) {
    this.re = re;
}

/**
 * Methode, um Imaginaerteil neu zu setzen
 * @param im der neue Imaginaerteil
 */
public void setIm(double im) {
    this.im = im;
}

/**
 * Methode, die den aktuellen Realteil  der komplexen Zahl liefert
 * @return der Realteil  der komplexen Zahl.
 */
public double getRe() {
    return re;
}
```

```
/**
 * Methode, die den aktuellen Imaginaerteil der komplexen Zahl liefert
 * @return der Imaginaerteil der komplexen Zahl
 */
public double getIm() {
   return im;
}

/**
 * Methode, um zwei komplexe Zahlen zu addieren, indem Real– und
 * Imaginaerteil getrennt addiert werden.
 * @param other die andere komplexe Zahl
 * @return die Summe der beiden komplexen Zahlen
 */
public Complex plus(Complex other) {
   double newRe = getRe() + other.getRe();
   double newIm = getIm() + other.getIm();
   return new Complex(newRe, newIm);
}

/**
 * Methode, um zwei komplexe Zahlen zu multiplizieren.
 * @param other die andere komplexe Zahl
 * @return das Produkt der beiden komplexen Zahlen
 */
public Complex mult(Complex other) {
   double newRe = getRe() * other.getRe() - getIm() * other.getIm();
   double newIm = getRe() * other.getIm() + getIm() * other.getRe();
   return new Complex(newRe, newIm);
}

/**
 * Methode, die den Betrag der komplexen Zahl zu berechnet. Der Betrag ist
 * die Wurzel aus der Summe des quadrierten Real– und Imaginaerteils.
 * @return der Betrag der komplexen Zahl.
 */
public double abs(){
   return Math.sqrt(re*re + im*im);
}

/**
 * Methode zur Darstellung der komplexen Zahl auf der Konsole.
 * @return eine Stringrepraesentation dieser komplexen Zahl
 */
```

```
    public String toString() {
        //numerische Genauigkeit
        double epsilon = Math.pow(10, -15);
        //Betraege von Real- und Imaginaerteil
        double r = Math.abs(im);
        double s = Math.abs(re);
        //Wenn ein Wert kleiner als epsilon ist, dann ist das auf Rundungsfehler
        //zurueckzufuehren; angezeigt wird 0.
        if (r < epsilon && s < epsilon) return "0.0";
        if (r < epsilon) return Double.toString(re);
        if (s < epsilon) return Double.toString(im) + "i";
        //Normalfall: zeige tatsaechlichen Wert an
        return "(" + re + " + "+ im + "i)";
    }
}
```

C.1.2 Die Klasse `Matrix.java`

```
package math;

/**
 * @author Uwe Sander
 * @version 11.06.2004
 *
 * Die Klasse repraesentiert eine Matrix. Es sind einige Matrizenoperationen
 * implementiert, z. B. das Matrixprodukt und das Tensorprodukt.
 */
public class Matrix {

    /**
     * Die Elemente der Matrix werden in diesem Array gespeichert.
     * Es ist von aussen sichtbar.
     */
    public Complex[][] val;

    /**
     * Die Zeilenzahl der Matrix
     */
    protected int n;

    /**
     * Die Spaltenzahl der Matrix
     */
    protected int m;
```

```
/**
 * Standardkonstruktor, der eine 1x1-Matrix erzeugt, die den Wert 0 (als
 * komplexe Zahl gekapselt) enthaelt.
 *
 */
public Matrix(){
    this.n = 1;
    this.m = 1;
    val = new Complex[n][m];
    val[0][0] =  new Complex();
}

/**
 * Konstruktor, der eine 1x1-Matrix erzeugt, die den uebergebenen Wert
 * enthaelt.
 * @param c der Wert der 1x1-Matrix
 */
public Matrix(Complex c) {
    this();
    val[0][0] =  c;
}

/**
 * Dieser Konstruktor erzeugt eine nxm-Matrix, deren Elemente alle gleich 0
 * sind.
 * @param n Die Zeilenzahl der Matrix
 * @param m Die Spaltenzahl der Matrix
 */
public Matrix(int n, int m) {
    this.n = n;
    this.m = m;
    val = new Complex[n][m];

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            val[i][j] = new Complex();
        }
    }
}

/**
 * Liefert die Zeilenzahl der Matrix
 * @return die Anzahl der Zeilen in der Matrix
```

```
 */
public int getRows() {
   return n;
}

/**
 * Liefert die Spaltenzahl der Matrix
 * @return die Anzahl der Spalten in der Matrix
 */
public int getColumns() {
   return m;
}

/**
 * Liefert ein neues Matrix-Objekt mit dem Ergebnis der Matrixmultiplikation
 * this*M zurueck, wenn die Spaltenzahl von this und die Zeilenzahl von other
 * identisch sind; Sonst wird null zurueckgegeben.
 * @param other die Matrix, mit der diese Matrix multipliziert wird
 * @return das Produkt aus den Matrizen this und other
 */
public Matrix mult(Matrix other) {
   Matrix newMat;
   int l = 0; // Spaltenzahl this
   int n = 0; // Zeilenzahl M
   int m = 0; // Spaltenzahl M (== Zeilen-
   // zahl this )

   int k = 0; // Spaltenindex other = 0... n-1
   int i = 0; // Zeilenindex this =0...l-1

   Complex sum = new Complex(); // Produktmatrixelement c_ik
   int j = 0; // Laufindex bei c_ik

   if (getColumns() != other.getRows()) { // Spalten(this) != Zeilen(other)
      return null;
   } else {
      m = getColumns(); // Summationsindexgrenze
      l = getRows(); // Zeilenzahl this holen
      n = other.getColumns(); // Spaltenzahl other holen

      newMat = new Matrix(l, n); // Platz fuer Produktmatrix

      //sum = 0.0; // c_ik = 0
      for (i = 0; i < l; i++) { // Fuer alle Zeilen
```

```
            for (k = 0; k < n; k++) { // Fuer alle Spalten
                for (j = 0; j < m; j++) { // Spalte i (this )*Zeile k (other)
                    sum = sum.plus(val[i][j]. mult(other.val[j][ k]));
                    // aufsummieren
                }
                newMat.val[i][k] = sum; // c_ik retten
                sum = new Complex(); // naechstes Element
            } // alle Spalten
        } // alle Zeilen
    } // Matrizenmultiplikation moeglich?
    return newMat;
}

/**
 * Methode, die das Tensorprodukt dieser Matrix mit der uebergebenen Matrix
 * zurueckgibt.
 * @param other die andere Matrix
 * @return das Tensorprodukt aus this und other
 */
public Matrix tensor(Matrix other) {

    //Zeilenzahl der anderen Matrix
    int oRows = other.getRows();
    //Spaltenzahl der anderen Matrix
    int oCols = other.getColumns();
    //Zeilenzahl der neuen Matrix
    int noRows = this.n * oRows;
    //Spaltenzahl der neuen Matrix
    int noCols = this.m * oCols;
    Matrix newMat = new Matrix(noRows, noCols);
    //durchlaufe alle Elemente der beiden Matrizen und berechne die Eintraege
    //der neuen Matrix
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            for (int k = 0; k < oRows; k++) {
                for (int l = 0; l < oCols; l++) {
                    newMat.val[i * oRows + k][j * oCols + l] =
                        val[i][ j]. mult(other.val[k][ l]);
                }
            }
        }
    }
    return newMat;
}
```

```
    /**
     * Gibt einen n-zeiligen String zurueck, der zeilenweise die m Elemente der
     * Matrixzeile enthaelt.
     */
    public String toString() {
        String out = "" + n + "x" + m + "-Matrix\n";
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                out += "[" + val[i][j] + "]";
            }
            out += "\n";
        }
        return out;
    }
}
```

C.2 Das Package `gates`

C.2.1 Die Klasse `CNot.java`

```
package gates;

import math.Complex;
import math.Matrix;

/**
 * @author Uwe Sander
 * @version 12.06.2004
 *
 * Diese Klasse repraesentiert ein CNOT-Gatter.
 * Der Standard-Konstruktor erzeugt ein CNOT mit einem Kontroll-Qubit und einem
 * Ziel-Qubit. Es gibt einen weiteren Konstruktor, dem die Zahl der
 * Kontroll-Qubits uebergeben wird. Dadurch erhaelt man im Allgemeinen ein
 * generalisiertes Toffoli-Gatter, welches in dieser Applikation aber wie ein
 * CNOT-Objekt behandelt wird.
 *
 * Implementiert das Gate-Interface, weil es ein Gatter darstellt .
 */
public class CNot extends Matrix implements Gate {

    /**
     * Standard-Konstruktor, der ein CNOT-Gatter mit einem Kontroll-Qubit und
     * einem Ziel-Qubit erzeugt. Es handelt sich also um ein Zwei-Qubit-Gatter.
     * Dieser Konstruktor ruft den zweiten auf.
```

```
     */
    public CNot() {
        this(1);
    }

    /**
     * Konstruktor, der ein CNOT-Gatter erzeugt. Dieses Gatter hat ein Ziel-Qubit
     * und mehrere Kontroll-Qubits, deren Anzahl uebergeben wird.
     * Es wird eine Matrix der Groesse 2^controlQubits erstellt , die auf der
     * Hauptdiagonalen nur Einsen enthaelt, bis auf die letzten beiden Zeilen, wo
     * die beiden Einsen vertauscht werden.
     * @param controlQubits Anzahl der Kontroll-Qubits
     */
    public CNot(int controlQubits) {
        //Matrix erstellen
        super((int) Math.pow(2, controlQubits + 1),
                (int) Math.pow(2, controlQubits + 1));

        //Matrix fuellen
        for (int i = 0; i < n - 2; i++) {
            val[i][i] = Complex.eins;
        }
        //In die rechte untere Ecke kommt ein "X-Gatter".
        val[n - 2][n - 1] = Complex.eins;
        val[n - 1][n - 2] = Complex.eins;
    }
}
```

C.2.2 Die Klasse `Gate.java`

```
package gates;

/**
 * @author Uwe Sander
 * @version 10.06.2004
 *
 * Ein Marker-Interface, dass alle Gatter implementieren. Die Klasse ist leer ,
 * weil sie alle Quantengatter nur "markieren" soll (ohne eigene
 * Funktionalitaet ).
 */
public interface Gate{

}
```

C.2.3 Die Klasse `Hadamard.java`

```
package gates;

import math.Complex;
import math.Matrix;

/**
 * @author Uwe Sander
 * @version 12.06.2004
 *
 * Diese Klasse repraesentiert ein Hadamard-Gatter.
 * Der Standard-Konstruktor erzeugt ein Hadamard-Objekt, das auf ein Qubit
 * wirkt. Es wird folgende 2x2-Matrix erzeugt:
 *
 * 1/sqrt(2) * (1   1)
 *             (1 -1)
 *
 * Implementiert das Gate-Interface, weil es ein Gatter darstellt .
 */
public class Hadamard extends Matrix implements Gate {
    // Der Vorfaktor
    private static Complex f = new Complex(1.0 / Math.sqrt(2), 0.0);

    /**
     * Standard-Konstruktor, der eine 2x2-Matrix erstellt. Der letzte Eintrag ist
     * -1, alle anderen sind 1, jeweils multipliziert mit dem Vorfaktor.
     *
     */
    public Hadamard() {
        super(2, 2);
        val[0][0] = f.mult(Complex.eins);
        val[1][0] = f.mult(Complex.eins);
        val[0][1] = f.mult(Complex.eins);
        val[1][1] = f.mult(new Complex(-1.0, 0.0));
    }

    /**
     * Diese Methode berechnet das Tensorprodukt aus mehreren Hadamard-Gattern.
     * Die Anzahl der Qubits, auf die die Hadamard-Gatter wirken, kann uebergeben
     * werden und bestimmt, wie oft das Tensorprodukt ausgefuehrt wird.
     * @param size die Anzahl der Qubits
     * @return das Tensorprodukt aus size Hadamard-Gattern
     */
```

```
    public static Matrix hadamard(int size) {
        //Matrix mit einem Element: einer Eins.
        Matrix mat = new Matrix(Complex.eins);
        //berechnet size-faches Tensorprodukt von Hadamard-Gattern
        for (int i = 0; i < size; i++) {
            mat = mat.tensor(new Hadamard());
        }
        return mat;
    }
}
```

C.2.4 Die Klasse `Id.java`

```
package gates;

import math.Complex;
import math.Matrix;
/**
 * @author Uwe Sander
 * @version 16.06.2004
 *
 * Diese Klasse repraesentiert ein Identitaetsgatter , das ein Qubit nicht
 * veraendert. Der Standard-Konstruktor instanziiert ein Id-Objekt, das eine
 * 2x2-Einheitsmatrix erzeugt:
 *
 * (1 0)
 * (0 1)
 *
 * Implementiert das Gate-Interface, weil es ein Gatter darstellt .
 */
public class Id extends Matrix implements Gate {

    /**
     * Standard-Konstruktor, der die Matrix erzeugt.
     */
    public Id() {
        super(2, 2);
        val[0][0] = Complex.eins;
        val[1][0] = new Complex();
        val[0][1] = new Complex();
        val[1][1] = Complex.eins;
    }

    /**
```

```
     * Diese Methode berechnet das Tensorprodukt aus mehreren Id-Gattern.
     * Die Anzahl der Qubits, auf die die Id-Gatter wirken, kann uebergeben
     * werden und bestimmt, wie oft das Tensorprodukt ausgefuehrt wird.
     * @param size die Anzahl der Qubits
     * @return das Tensorprodukt aus size Id-Gattern
     */
    public static Matrix id(int size) {
        //Matrix mit einem Element: einer Eins.
        Matrix mat = new Matrix(Complex.eins);
        //berechnet size-faches Tensorprodukt von Id-Gattern
        for (int i = 0; i < size; i++) {
            mat = mat.tensor(new Id());
        }
        return mat;
    }
}
```

C.2.5 Die Klasse `Not.java`

```
package gates;

import math.Complex;
import math.Matrix;
/**
 * @author Uwe Sander
 * @version 12.06.2004
 *
 * Diese Klasse repraesentiert ein NOT-Gatter, das ein Qubit invertiert.
 * Der Standard-Konstruktor erzeugt ein Not-Objekt, das auf ein Qubit wirkt.
 * Es wird folgende 2x2-Matrix erzeugt:
 *
 * (0 1)
 * (1 0)
 *
 * Implementiert das Gate-Interface, weil es ein Gatter darstellt .
 */
public class Not extends Matrix implements Gate {

    /**
     * Standard-Konstruktor, der die Matrix erzeugt.
     */
    public Not() {
        super(2, 2);
        val[0][0] =  new Complex();
```

```
        val[1][0] =  Complex.eins;
        val[0][1] =  Complex.eins;
        val[1][1] =  new Complex();
    }

    /**
     * Diese Methode berechnet das Tensorprodukt aus mehreren NOT-Gattern.
     * Die Anzahl der Qubits, auf die die NOT-Gatter wirken, kann uebergeben
     * werden und bestimmt, wie oft das Tensorprodukt ausgefuehrt wird.
     * @param size die Anzahl der Qubits
     * @return das Tensorprodukt aus size NOT-Gattern
     */
    public static Matrix not(int size) {
        //Matrix mit einem Element: einer Eins.
        Matrix mat = new Matrix(Complex.eins);
        //berechnet size-faches Tensorprodukt von NOT-Gattern
        for (int i = 0; i < size; i++) {
            mat = mat.tensor(new Not());
        }
        return mat;
    }
}
```

C.3 Das Package `grover`

C.3.1 Die Klasse `Grover.java`

```
package grover;
import gates.Hadamard;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import math.Matrix;

/**
 * @author Uwe Sander
 * @version 14.06.2004
 *
 * Diese Klasse implementiert den Grover-Algorithmus. Ueber die Kommandozeile
 * koennen verschiedene Parameter eingegeben werden (Anzahl der Qubits fuer den
 * Suchraum, die Anzahl der Loesungen und die Elemente, die die Loesungen
 * darstellen  sollen ).
 * In der Methode run() wird der Schaltkreis zusammen gebaut, die Methode
```

```
 * measure() misst den Zustand am Ende des Algorithmus. Die Methoden read()
 * und printResult() dienen zum Einlesen der Parameter und Ausgeben der
 * Ergebnisse.
 */
public class Grover {
    //Die Anzahl der Qubits fuer den Suchraum
    private static int n;
    //Die Loesungen des Suchproblems werden in diesem Array gespeichert
    private static int[] s;
    //Die Anzahl der Loesungen.
    private static int m;

    /**
     * Diese Methode wird aufgerufen, wenn die Applikation gestartet wird.
     * @param args dieses Array wird nicht verwendet
     */
    public static void main(String[] args) {
        System.out.println("Der Grover-Algorithmus\n");
        //Benutzereingaben lesen
        read();
        //Los geht's
        run(n, s, m);
    }

    /**
     * Fuehrt den Grover-Algorithmus aus.
     * Zuerst wird ein Register mit n Qubits erstellt , welche anschliessend
     * in eine Superposition gebracht werden. Dann wird berechnet, wie oft die
     * Grover-Iteration durchgefuehrt werden muss und selbige auf das Register
     * angewendet. Zum Schluss messen wir den Zustand und erhalten mit hoher
     * Wahrscheinlichkeit die gesuchten Zustaende.
     * @param n die Anzahl der Qubits fuer den Suchraum
     * @param s die Loesung(en) des Suchproblems
     * @param m die Anzahl der Loesungen
     */
    public static void run(int n, int[] s, int m) {
        System.out.println("Und los geht's!");
        //Startzeit messen
        long startTime = System.currentTimeMillis();
        //Die Qubits fuer den Suchraum
        Qubit q = new Qubit();
        Register qreg = new Register(q, q);
        for (int i = 0; i < n - 2; i++) {
            qreg.add(q);
```

```
    }
    //Hadamard-Gatter angewandt auf alle Qubits
    Matrix h = Hadamard.hadamard(n);
    Matrix start = h.mult(qreg);
    String meldung = "Erstelle Matrix fuer die Grover-Iteration. ";
    if (n > 6)
        meldung += "Bitte einen Moment Geduld.";
    System.out.println(meldung);
    //Erstelle ein neues GroverIteration-Objekt
    GroverIteration gi = new GroverIteration(n, s);
    Matrix iteration = gi;
    //Grover-Iteration wird k-mal durchgefuehrt
    long k = nrOfIterations((int) Math.pow(2, n), m);
    meldung = "Fuehre " + k + " Iteration(en) durch. ";
    if (n > 6)
        meldung += "Das kann dauern...";
    System.out.println(meldung);
    int i = 0;
    while (i < k - 1) {
        iteration = iteration.mult(gi);
        i++;
    }
    //Iteration durchfuehren
    Matrix end = iteration.mult(start);
    //Endzeit messen
    long endTime = System.currentTimeMillis();
    System.out.println("Dauer: " + (endTime - startTime)/1000.0 + " sec.\n");

    //Endzustand messen
    Measurement measure = new Measurement(end, m);
    int[] solution = measure.examine();

    //Ergebnis ausgeben
    printResult(solution, Math.pow(end.val[solution[0]][0].abs(), 2));
}

/**
 * Berechnet die Anzahl der Grover-Iterationen, die noetig sind, bevor wir
 * messen koennen.
 * @param N die Anzahl der Elemente im Suchraum
 * @param M die Anzahl der Loesungen
 * @return die Zahl der benoetigten Iterationen
 */
private static long nrOfIterations(int N, int M) {
```

```
    double theta = Math.asin(2 * Math.sqrt(M * (N - M)) / N);
    long r =
        Math.round(Math.acos(Math.sqrt((double) M / (double) N)) / theta);
    return r;
}

/**
 * Liest Benutzereingaben von der Kommandozeile. Der Benutzer wird zuerst
 * nach der Anzahl der Qubits fuer den Suchraum gefragt. Danach muss der
 * Index, den das Orakel erkennen soll, angegeben werden.
 * Die Methode prueft, ob der Benutzer sinnvolle Werte angegeben hat.
 */
private static void read() {
    //Ein Reader fuer das Lesen von der Kommandozeile
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);

    try {
        System.out.print("Im folgenden werden Sie gebeten, die Anzahl n ");
        System.out.println("der Qubits fuer den Suchraum einzugeben.");
        System.out.print(
            "n sollte auf einem aktuellen PC nicht groesser als 8 sein, es sei denn,");
        System.out.println(" Sie haben einen richtig schnellen Rechner.");
        //solange n < 2, lese neues n ein
        while (n < 2) {
            System.out.println(
                "n muss groesser als 1 sein, sonst koennen Sie gleich raten!");
            System.out.print("Bitte geben Sie n ein: ");
            n = Integer.parseInt(br.readLine());
        }
        // bei ungueltiger Eingabe setze n auf 3
    } catch (NumberFormatException e) {
        n = 3;
        System.err.println ("Sie haben keine gueltige Zahl n eingegeben!");
        System.err.println ("n wird automatisch auf " + n + " gesetzt.");
        //hier ist etwas ganz schief gelaufen
    } catch (IOException e) {
        System.err.println ("Fehler! -> Programm wird beendet.");
        e.printStackTrace();
    }
    int N = (int) Math.pow(2, n);
    System.out.println(
        "Der Suchraum besteht also aus " + N + " Elementen.\n");
```

```
try {
    System.out.println("Wie viele Loesungen soll es geben? ");
    m = 0;
    //solange m kleiner als 1, lese neues m ein
    while (m < 1) {
        System.out.print(
            "Bitte eine Zahl zwischen 1 und " + N / 2 + ": ");
        m = Integer.parseInt(br.readLine());
    }
    //bei ungueltiger Eingabe setze m auf 1
} catch (NumberFormatException e) {
    m = 1;
    System.err.println ("Sie haben keine gueltige Zahl m eingegeben!");
    System.err.println ("m wird automatisch auf " + m + " gesetzt.");
    //        hier ist etwas ganz schief gelaufen
} catch (IOException e) {
    System.err.println ("Fehler! -> Programm wird beendet.");
    e.printStackTrace();
}
String str = "";
if (m > 1) str = "en";
System.out.println("Es gibt also " + m + " Loesung" + str + ".\n");

try {
    s = new int[m];
    System.out.println("Welche Loesung(en) soll das Orakel erkennen?");
    s[0] = -1;
    //alle Loesungen einlesen
    for (int i = 0; i < m; i++) {
        do {
            System.out.print(
                "Bitte eine Zahl zwischen 0 und " + (N - 1) + ": ");
            s[i] = Integer.parseInt(br.readLine());
            //pruefe, ob Zahl schon als Loesung vorhanden
            if (!checkInput(s[i], i)) {
                System.out.println(
                    "Loesung doppelt vorhanden. Bitte neu eingeben!");
                i--;
            }
        //solange s nicht innerhalb des Suchraumes liegt, lese neues s ein
        } while (s[i] < 0 || s[i] >= N);
    }
    //bei ungueltiger Eingabe setze s auf 2
} catch (NumberFormatException e) {
```

```
            s = new int[1];
            s[0] = 2;
            m = 1;
            System.err.println ("Sie haben keine gueltige Zahl s eingegeben!");
            System.err.println ("s wird automatisch auf " + s[0] + " gesetzt.");
            //hier ist  etwas ganz schief  gelaufen
        } catch (IOException e) {
            System.err.println ("Fehler! -> Programm wird beendet.");
            e.printStackTrace();
        }
        //Kleine Benutzerhilfe
        if (m > 1) {
            System.out.print("Als Loesung wurden die Elemente ");
            for (int i = 0; i < m; i++) {
                System.out.print(s[i]);
                if (i < m - 1)
                    System.out.print(", ");
            }
            System.out.println(" gewaehlt.\n");
        } else {
            System.out.println(
                "Als Loesung wurde das " + s[0] + ". Element gewaehlt.\n");
        }
    }

    /**
     * Diese Methode prueft, ob ein Benutzer eine Loesung doppelt angegeben hat.
     * Dazu wird das Array mit den bisher eingegebenen Loesungen durchlaufen und
     * jedes  Element auf Uebereinstimmung geprueft.
     * @param die gerade eingebene Zahl
     * @param der Index im Array, bei dem wir uns gerade befinden
     * @return true falls  das uebergebene Element noch nicht als Loesung
     *                     angegeben wurde
     */
    private static boolean checkInput(int input, int index) {
        for (int i = 0; i < index; i++) {
            if (s[i] == input) return false;
        }
        //kein Element doppelt
        return true;
    }

    /**
     * Methode, um die Ergebnisse des Grover-Algorithmus zusammenzufassen
```

```
     *
     * @param solution die Loesung des Suchproblems
     * @param prob die Wahrscheinlichkeit dieser Loesung
     */
    private static void printResult(int[] solutions, double prob) {
        System.out.println("Ergebnis:");
        for (int i = 0; i < m; i++) {
            System.out.print("Die " + (i + 1) + ". Loesung hat den Index ");
            System.out.print(solutions[i]);
            String StrProb = Double.toString(prob);
            StrProb = StrProb.substring(0, 6);
            System.out.println(" mit Wahrscheinlichkeit " + StrProb + ".");
            //Baue String, der den Loesungszustand in Dirac-Notation darstellt
            StringBuffer sb = new StringBuffer("|");
            sb.append(Integer.toBinaryString((solutions[i])));
            int z = 1;
            while (sb.length() < n + 1) {
                sb.insert (z++, '0');
            }
            sb.append(">");
            System.out.println("Der Loesungszustand ist: " + sb);
        }
    }
}
```

C.3.2 Die Klasse `GroverIteration.java`

```
package grover;

import gates.Hadamard;
import math.Matrix;

/**
 * @author Uwe Sander
 * @version 14.06.2004
 *
 * Eine Grover-Iteration besteht aus folgenden Operatoren:
 * - Orakelaufruf
 * - Hadamard
 * - bed. Phasenverschiebung
 * - Hadamard
 */
public class GroverIteration extends Matrix {
    //Die Registerbreite (inkl. Orakel-QuBit)
```

```
    private int regSize;
    //Die Loesungen des Suchproblems
    private int solution[];

    //Alle benoetigten  Gatter-Operationen
    private Oracle o;
    private Matrix whs;
    private PhaseShift ps;

    /**
     * Konstruktor, der die  Operatoren in der richtigen  Reihenfolge zusammenfuegt
     * und die Gesamtmatrix erstellt.
     * @param regSize die Registerbreite  = Anzahl der Qubits
     * @param solution die Loesungen des Suchproblems
     */
    public GroverIteration(int regSize,  int[]  solution ) {
        super((int) Math.pow(2, regSize), (int) Math.pow(2, regSize));
        this.regSize = regSize;
        this.solution = solution;
        calcMatrix();
    }

    /**
     * Methode, die die Gesamtmatrix aufstellt. Die drei Matrizen für das Orakel,
     * die Hadamard-Gatter und die Phasenverschiebung werden multipliziert.
     */
    private void calcMatrix() {
        //Orakel erzeugen
        o = new Oracle(regSize, solution);
        //Hadamard-Gatter auf alle QuBits erzeugen
        whs = Hadamard.hadamard(regSize);
        //Phasenverschiebung erzeugen
        ps = new PhaseShift(regSize);
        //Gesamtmatrix berechnen
        val = whs.mult(ps).mult(whs).mult(o).val;
    }
}
```

C.3.3 Die Klasse `Measurement.java`

```
package grover;

import math.Matrix;
```

```
/**
 * @author Uwe Sander
 * @version 23.06.2004
 *
 * Die Klasse Measurement repraesentiert einen Messvorgang an einem
 * quantenmechanischen Zustand. Die Methode measure() gibt ein zufaelliges
 * Element des zu messenden Zustandes zurueck, wobei die Wahrscheinlichkeiten
 * natuerlich beachtet werden. Die Methode examine() untersucht den Zustand
 * daraufhin, welche Elemente bei mehreren Messungen am haeufigsten auftreten
 * (Importance sampling). Die m Elemente mit den hoechsten Wahrscheinlichkeiten
 * werden zurueckgegeben, wobei m die Zahl der Loesungen des Grover−Algorithmus
 * ist .
 */
public class Measurement {
    //Die Anzahl der Loesungen
    private int m;
    //Die Matrix, die den zu messenden Zustand repraesentiert
    private Matrix matrix;

    /**
     * Konstruktor, der die Matrix und die Zahl der Loesungen uebergeben bekommt
     * @param matrix die zu messende Matrix
     * @param m die Zahl der Loesungen
     */
    public Measurement(Matrix matrix, int m) {
        this.matrix = matrix;
        this.m = m;
    }

    /**
     * Misst den Zustand, der durch matrix repraesentiert wird. Es werden die
     * Wahrscheinlichkeiten aller Zustaende nacheinander summiert. Sobald die
     * Summe groesser ist als eine zufaellige Zahl z (0 < z < 1), wird der
     * Zustand zureuckgegeben, der als letztes addiert wurde.
     * @return der Index des wahrscheinlichsten Zustandes
     */
    private int measure() {
        double sum = 0.0;
        double abs = 0.0;
        int result = 0;
        double random = Math.random();
        int i;
        for (i = 0; i < matrix.getRows(); i++) {
            abs = matrix.val[i][0]. abs();
```

```
            sum += abs * abs;
            if (random < sum)
                return i;
        }
        return i;
    }

    /**
     * Diese Methode fuehrt 100 Messungen durch und erstellt eine Liste, in der
     * die Wahrscheinlichkeiten aller Elemente verzeichnet sind. Am Ende
     * werden die Werte mit hohen Wahrscheinlichkeiten zurueckgegeben.
     * @param matrix die Matrix, welche den Endzustand repraesentiert
     * @return die Loesung(en) des Suchproblems mit hoher Wahrscheinlichkeit
     */
    public int[] examine() {
        int size = matrix.getRows();
        //Array fuer Wahrscheinlichkeiten
        int[] percentage = new int[size];
        //Die Anzahl der Messungen
        int nr = 100;
        //Erstellt ein Array, dessen Eintraege angeben, wie oft welcher Index bei
        //100 Messungen vorkommt.
        while (nr > 0) {
            int result = measure();
            for (int j = 0; j < size; j++) {
                if (result == j) {
                    percentage[j]++;
                }
            }
            nr--;
        }

        //Array, das die Loesungen enthalten wird
        int[] states = new int[m];
        //Pickt die m Eintraege mit der hoechsten Wahrscheinlichkeit heraus
        for (int p = 0; p < m; p++) {
            int marker = 0;
            for (int l = 0; l < percentage.length; l++) {
                //Wenn wir eine hoehere Wahrscheinlichkeit haben als bisher, dann
                //merken wir uns das Element
                if (percentage[l] > percentage[marker]) {
                    marker = l;
                    states[p] = l;
                }
```

```
            }
            //loesche die schon abgearbeitete Loesung aus dem Array
            percentage[marker] = 0;
        }
        //Die endgueltigen Loesungen
        return states;
    }
}
```

C.3.4 Die Klasse `Oracle.java`

```
package grover;

import math.Complex;
import math.Matrix;

/**
 * @author Uwe Sander
 * @version 12.06.2004
 *
 * Diese Klasse repraesentiert das Orakel, das auf beliebig viele Qubits wirken
 * kann. Der Zustand, der die Loesung darstellt, wird "markiert", indem sein
 * Vorzeichen geaendert wird. Alle andere Zustaende bleiben unveraendert.
 */
public class Oracle extends Matrix {
    // Die Registerbreite
    private int regSize;
    //Die Loesung des Suchproblems
    private int[] solution;

    /**
     * Konstruktor der ein Orakel fuer einen Suchraum der Groesse regSize
     * erstellt . Die Loesung des Suchproblems wird uebergeben und dann die
     * entsprechende Matrix der Groesse 2^regsize erstellt .
     * @param regSize die Registerbreite = Anzahl der Qubits
     * @param solution die Loesung zum Suchproblem
     */
    public Oracle(int regSize, int[] solution) {
        super((int) Math.pow(2, regSize), (int) Math.pow(2, regSize));
        this.regSize = regSize;
        this.solution = solution;
        calcMatrix();
    }
```

```
    /**
     * Methode, die die Gesamtmatrix fuer das Orakel aufstellt.
     * Es wird eine Einheitsmatrix gebildet , die an den entsprechende Stellen
     * eine -1 statt einer 1 hat.
     */
    private void calcMatrix() {
        //Erschaffe einen Einheitsmatrix
        for (int i = 0; i < getColumns(); i++) {
            val[i][i] = Complex.eins;

        }
        //Die Eintraege der Matrix, die mit den Loesungszustaenden multipliziert
        //werden, bekommen ein anderes Vorzeichen.
        //Beachte: Array-index beginnt bei 0, d.h. 1. Zustand = 0. Eintrag in der
        //Matrix
        Complex c = new Complex(-1.0, 0.0);
        for (int i = 0; i < solution.length; i++) {
            val[solution[i]][solution[i]] = c;
        }
    }
}
```

C.3.5 Die Klasse `PhaseShift.java`

```
package grover;

import gates.CNot;
import gates.Hadamard;
import gates.Id;
import gates.Not;
import math.Complex;
import math.Matrix;

/**
 * @author Uwe Sander
 * @version 12.06.2004
 *
 * Diese Klasse repraesentiert eine bedingte Phasenverschiebung auf beliebig
 * vielen Qubits. Der Zustand |00...0> wird nicht verändert, alle anderen
 * Zustände wechseln das Vorzeichen. Dieser Operator ist aus elementaren Gattern
 * (X, H, I, CNOT) aufgebaut.
 */
public class PhaseShift extends Matrix {
    //Die Groesse des Registers, auf das dieser Operator wirken soll
```

```
private int regSize;

/**
 * Standard-Konstruktor, der ein PhaseShift-Objekt fuer einen Suchraum mit
 * 4 Elementen (= 2 Qubits) erzeugt.
 *
 */
public PhaseShift() {
    this(2);
}

/**
 * Konstruktor, der ein PhaseShift-Objekt fuer regSize Qubits erzeugt.
 * Erstellt die Matrix der Groesse 2^regsize.
 * @param regSize Die Registerbreite = Anzahl der Qubits
 */
public PhaseShift(int regSize) {
    super((int) Math.pow(2, regSize), (int) Math.pow(2, regSize));
    this.regSize = regSize;
    calcMatrix();
}

/**
 * Methode, die die bedingte Phasenverschiebung mit elementaren Gattern
 * implementiert und die Matrix entsprechend fuellt.
 * Die Implementation orientiert sich an dem Beispiel-Schaltkreis aus
 * "Quantum Computation and Quantum Information", Nielsen/Chuang, S. 256.
 *
 */
public void calcMatrix() {
    Hadamard h = new Hadamard();
    Id id = new Id();
    //Elementargatter fuer mehrere QuBits erstellen
    Matrix nots = Not.not(regSize);
    CNot cnot = new CNot(regSize - 1);
    // Die komplexe Zahl i
    Complex i = new Complex(0, 1);
    //Dies ist das Gatter i*I
    Matrix mat = new Matrix(2, 2);
    mat.val[0][0] = i;
    mat.val[1][1] = i;
    for (int j = 0; j < regSize - 2; j++) {
        mat = mat.tensor(id);
    }
```

```
        mat = mat.tensor(h);
        //Gesamtmatrix berechnen
        val = nots.mult(mat).mult(cnot).mult(mat).mult(nots).val;
    }
}
```

C.3.6 Die Klasse `Qubit.java`

```
package grover;

import math.Complex;
import math.Matrix;

/**
 * @author Uwe Sander
 * @version 10.06.2004
 *
 * Diese Klasse repraesentiert ein Qubit.
 * Ein Qubit ist von der Form alpha*|0> + beta*|1>. alpha und beta sind
 * komplexe Zahlen. Die Klasse ist eine Unterklasse von Matrix, d. h. ein Qubit
 * ist eine 2x1-Matrix. val[0][0] ist alpha, val[1][0] ist beta.
 */
public class Qubit extends Matrix {

    /**
     * Standard-Konstruktor, der ein Qubit mit alpha = 1 und beta = 1
     * initialisiert .
     *
     */
    public Qubit() {
        this(new Complex(1.0, 0.0), new Complex(0.0, 0.0));
    }

    /**
     * Konstruktor, der ein Qubit instanziiert und alpha und beta auf die
     * uebergebenen Werte setzt, falls die Wahrscheinlichkeiten sich zu 1
     * addieren.
     * @param alpha die komplexe Zahl alpha
     * @param beta die komplexe zahl beta
     */
    public Qubit(Complex alpha, Complex beta) {
        n = 2;
        m = 1;
        if (testProb(alpha, beta)) {
```

```
        val = new Complex[n][m];
        val[0][0] =  alpha;
        val[1][0] =  beta;
    } else {
        System.err.println(
            "Achtung! Die Gesamtwahrscheinlichkeit ist ungleich als 1!");
    }

}

/**
 * Konstruktor, der ein Qubit instanziiert  und alpha und beta auf die
 * uebergebenen reellen-Werte setzt, welche in komplexe Zahlen mit
 * Imaginaerteil=0 umgewandelt werden.
 * @param alpha die reelle Zahl alpha
 * @param beta die relle Zahl beta
 */
public Qubit(double alpha, double beta) {
    this(new Complex(alpha, 0.0), new Complex(beta, 0.0));
}

/**
 * Methode, um alpha und beta neu zu setzen. Prüft, ob dadurch die
 * Gesamtwahrscheinlichkeit groesser als 1 wird.
 * @param alpha das neue alpha
 * @param beta das neue beta
 */
public void set(Complex alpha, Complex beta) {
    if (testProb(alpha, beta)) {
        val[0][0] =  alpha;
        val[1][0] =  beta;
    } else {
        System.err.println(
            "Achtung! Die Gesamtwahrscheinlichkeit ist ungleich als 1!");
    }
}

/**
 * Liefert  alpha zurueck
 * @return die komplexe Zahl alpha
 */
public Complex getAlpha() {
    return val[0][0];
}
```

```
/**
 * Liefert beta zurueck
 * @return die komplexe Zahl beta
 */
public Complex getBeta() {
   return val[1][0];
}

/**
 * Setzt alpha auf den uebergebenen Wert
 * @param complex der neue Wert fuer alpha
 */
public void setAlpha(Complex alpha) {
   if (testProb(alpha, getBeta())) {
      val[0][0] = alpha;
   } else {
      System.err.println(
         "Achtung! Die Gesamtwahrscheinlichkeit ist ungleich als 1!");
   }
}

/**
 * Setzt beta auf den uebergebenen Wert
 * @param complex der neue Wert fuer beta
 */
public void setBeta(Complex beta) {
   if (testProb(beta, getAlpha())) {
      val[1][0] = beta;
   } else {
      System.err.println(
         "Achtung! Die Gesamtwahrscheinlichkeit ist ungleich als 1!");
   }
}

/**
 * Testet, ob die Gesamtwahrscheinlichkeit zweier komplexer Zahlen 1 ist.
 * Dabei werden eventuelle numerische Ungenauigkeiten berücksichtigt.
 * @param c1 die erste komplexe Zahl (in der Regel alpha)
 * @param c2 die zweite komplexe Zahl (in der Regel beta)
 * @return <tt>true<tt>, falls die Wahrscheinlichkeit gleich 1 ist,
 *         <tt>false<tt> sonst.
 */
private boolean testProb(Complex c1, Complex c2) {
```

```
        //Die Genauigkeit, mit der wir rechnen.
        double epsilon = Math.pow(10, -15);
        //Die Quadrate der Absolutwerte
        double sum = Math.pow(c1.abs(), 2) + Math.pow(c2.abs(), 2);
        //Liegt die Summe in einem kleinen Intervall um 1?
        return ((sum > 1.0 - epsilon) && (sum < 1.0 + epsilon));
    }

    /**
     * Methode zur Darstellung der komplexen Zahl auf der Konsole.
     * @return eine Stringrepraesentation dieses Qubits
     */
    public String toString() {
        return val[0][0] + "|0> + " + val[1][0] + "|1>";
    }
}
```

C.3.7 Die Klasse `Register.java`

```
package grover;

import math.Matrix;

/**
 * @author Uwe Sander
 * @version 11.06.2004
 *
 * Diese Klasse repraesentiert ein Quanten-Register, das aus mehreren Qubits
 * besteht. Ein Register ist eine nx1-Matrix.
 * Man kann Qubits hinzufügen und ein neues Register aus einem bereits
 * vorhandenem erzeugen.
 */
public class Register extends Matrix {

    //Die Anzahl der QuBits in diesem Register
    private int size;

    /**
     * Konstruktor, der ein neues Register aus zwei Qubits erzeugt.
     * @param q1 das erste Qubit
     * @param q2 das zweite Qubit
     */
    public Register(Qubit q1, Qubit q2) {
        val = q1.tensor(q2).val;
```

```
    size = 2;
    n = 4;
    m = 1;
}

/**
 * Konstruktor, der ein neues Register aus einem zweiten erzeugt
 * @param qreg das andere Register
 */
public Register(Register qreg) {
    val = qreg.val;
    size = qreg.getSize();
    n = qreg.n;
    m = 1;
}

/**
 * Fuegt dem Register ein Qubit hinzu
 * @param q das neue Qubit
 */
public void add(Qubit q) {
    val = (this.tensor(q)).val;
    size++; //Die Groesse erhoeht sich 1
    n *= 2;  //Die Dimension der Matrix verdoppelt sich
}

/**
 * Liefert die Anzahl der Qubits in diesem Register.
 * @return die Groesse des Registers
 */
public int getSize() {
    return size;
}

/**
 * Methode zur Darstellung des Registers auf der Konsole.
 * @return eine Stringrepraesentation dieses Registers
 */
public String toString() {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < n; i++) {
        sb.append("(" + val[i][0] + ")");
        if (i < n-1 ) sb.append("\n");
    }
```

```
        return sb.toString();
    }
}
```

Literatur

[Aha99] D. Ahanorov. Quantum computation. In D. Stauffer, editor, *Annual Reviews of Computational Physics VI*. World Scientific, 1999. *arXive e-print quant-ph/9812037*.

[BBBV97] C. H. Bennett, E. Bernstein, G. Brassard, and U. V. Vazirani. Strengths and weaknesses of quantum computing. *SIAM Journal on Computing*, 26(5):1510–1523, 1997. *arXive e-print quant-ph/9701001*.

[BBC+95] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. H. Margolus, P. W. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457–3467, 1995. *arXive e-print quant-ph/9503016*.

[BBHT98] M. Boyer, G. Brassard, P. Høyer, and A. Tapp. Tight bounds on quantum searching. *Fortschritte der Physik - Progress of Physics*, 46(4–5):493–505, 30, 1998. *arXive e-print quant-ph/9605034*.

[BCDP96] D. Beckman, A. N. Chari, Sr. Devabhaktuni, and J. Preskill. Efficient networks for quantum factoring. *Physical Review A*, 54(2):1034–1063, 1996. *arXive e-print quant-ph/9602016*.

[BHT98] G. Brassard, P. Høyer, and A. Tapp. Quantum counting. *ArXiv Quantum Physics e-prints*, 1998. *arXive e-print quant-ph/9805082*.

[BMP+99] P. O. Boykin, T. Mor, M. Pulver, V. Roychowdhury, and F. Vatan. On universal and fault-tolerant quantum computing. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 486–494, 1999. *arXive e-print quant-ph/9906054*.

[CFH97] D. Cory, A. Fahmy, and T. Havel. Ensemble quantum computing by nuclear magnetic resonance spectroscopy. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 94, pages 1634–1639, 1997.

[CZ95] J. I. Cirac and P. Zoller. Quantum computations with cold trapped ions. *Physical Review Letters*, 74(20):4091–4094, 1995.

[Deu85] D. Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society London Series A*, 400:97–117, 1985.

[Deu89] D. Deutsch. Quantum computational networks. *Proceedings of the Royal Society London Series A*, 425:73–90, 1989.

[Die82] D. Dieks. Communication by epr devices. *Physics Letters A*, 92(6):271–272, 1982.

[Div95] D. P. Divincenzo. Quantum computation. *Science*, 270:255–261, 1995.

[Fey82] R. P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6&7):467–488, 1982.

[Fey85] R. P. Feynman. Quantum mechanical computers. *Optics News*, 11:11–20, 1985.

[Gro96] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 212–219, 1996. *arXive e-print quant-ph/9605043*.

[Gro97] L. K. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Physical Review Letter*, 79(2):325–328, 1997. *arXive e-print quant-ph/9706033*.

[Gro02] Lov K. Grover. Tradeoffs in the quantum search algorithm. *ArXiv Quantum Physics e-prints*, January 2002. *arXive e-print quant-ph/0201152*.

[KMW02] D. Kielpinski, C. R. Monroe, and D. J. Wineland. Architecture for a large-scale ion-trap quantum computer. *Nature*, 417:709–711, 2002.

[LD98] D. Loss and D. P. DiVincenzo. Quantum computation with quantum dots. *Physical Review A*, 57(1):120–126, 1998. *arXive e-print cond-mat/9701055*.

[LMP03] C. Lavor, L. R. U. Manssur, and R. Portugal. Grover's Algorithm: Quantum Database Search. *ArXiv Quantum Physics e-prints*, January 2003. *arXive e-print quant-ph/0301079*.

[Mos98] M. Mosca. Quantum searching, counting and amplitude amplification by eigenvector analysis. In R. Freivalds, editor, *Proceedings of International Workshop on Randomized Algorithms*, pages 90–100, 1998.

[Mos99] M. Mosca. *Quantum Computer Algorithms*. PhD thesis, University of Oxford, 1999.

[NC00] M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, 2000.

[NPT99] Y. Nakamura, Yu. A. Pashkin, and J. S. Tsai. Coherent control of macroscopic quantum states in a single-cooper-pair box. *Nature*, 398(6730):786–788, 1999.

[RG02] T. Rudolph and (Dr Strange) Lov K. Grover. Quantum searching a classical database (or how we learned to stop worrying and love the bomb). *ArXiv Quantum Physics e-prints*, January 2002. *arXive e-print quant-ph/0206066*.

[Sch95] B. Schumacher. Quantum coding. *Physical Review A*, 51:2738–2747, 1995.

[Sho94] P. W. Shor. Algorithms for quantum computation: Discrete log and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. Institute of Electrical and Electronic Engineers Computer Society Press, 1994.

[Sim94] D. R. Simon. On the power of quantum computation. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 116–123, Los Alamitos, CA, 1994. Institute of Electrical and Electronic Engineers Computer Society Press.

[Sim97] D. R. Simon. On the power of quantum computation. *SIAM Journal on Computing*, 26(5):1474–1483, 1997.

[Vin95] D. P. Di Vincenzo. Two-bit gates are universal for quantum computation. *Physical Review A*, 51(2):1015–1022, 1995. *arXive e-print cond-mat/9407022*.

[VSB+01] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang. Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414:883–887, 2001. *arXive e-print quant-ph/0112176*.

Danksagung

Die Bachelorarbeit entstand in der Arbeitsgruppe „Makroskopische Systeme und Quantentheorie“ im Fachbereich Physik an der Universität Osnabrück.
Ich möchte mich bei allen bedanken, die mich während dieser Arbeit unterstützt haben.
Besonderer Dank gilt natürlich Nils Rosemann für die hervorragende und fruchtbare Zusammenarbeit während der vier Monate.
Für die interessante Themenstellung und die ausgezeichnete Betreuung dieser Arbeit danke ich Herrn Prof. Heinz Jürgen Schmidt, von dem ich viele Ideen und Anregungen bekommen habe, auch in sprachlicher und stilistischer Hinsicht.
Nils Israel danke ich für das unvermeidliche Korrekturlesen.
Zu guter Letzt danke ich meinen Eltern, die viel dazu beigetragen haben, mich für die Naturwissenschaft zu begeistern, und ohne die mein Studium gar nicht möglich wäre.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich das Kapitel 4 „Quantencomputer“ gemeinsam mit Nils Rosemann verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die anderen Teile meiner Bachelorarbeit habe ich selbstständig und ohne fremde Hilfe verfasst. Ich habe zuvor noch keine Bachelorprüfung der Fachrichtung Physik abgelegt.

Osnabrück, den 21. Juli 2004.

Uwe Sander