MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Computer Science and Information Theory

Optimizing memory usage in quantum algorithm simulation

Scientific Students' Association Report

Author:

Viktória Nemkin

Advisor:

dr. Katalin Friedl

2022

Contents

Kivonat

A piacon jelenleg elérhető kvantumalgoritmus-futtató keretrendszerek (IBM Qiskit, Google Cirq) a számításaikat a qubitek számában exponenciális méretű unitér mátrixokkal valósítják meg. Ennek következménye, hogy igen kis méretű bemenetek esetén is meglehetősen nagy mennyiségű memóriára van szükségük. Bár a meglévő rendszerek használnak bizonyos optimalizációs módszereket, ezek sokszor nem tudnak nagyságrendi javulást eredményezni (például a ritka mátrixos tárolási mód) vagy csak nagyon speciális algoritmusokra alkalmazhatók (Clifford-kapuk). A gyakorlatban ez azt jelenti, hogy az óriáscégekkel szemben egy átlagos felhasználó sok algoritmus esetében még viszonylag kis méretű bemeneteken sem tud ésszerű keretek között kísérletezni, az túl nagy hardverköltséggel járna.

A hardverszükséglet csökkenthető olyan algoritmussal, amely memóriát spórol, megnövekedett futásidőért cserébe. Például az unitér mátrixok éppen szükséges részmátrixai futás közben "on-the-fly" kiszámíthatóak, vagy akár a mátrixműveletek teljes egészükben helyettesíthetőek az azokkal ekvivalens hagyományos algoritmusokkal. Bár a korábban említett, a piacon elterjedt futtató keretrendszerek nyílt forráskódúak, sajnos az architektúrájuk szerves részét képezi az unitér mátrix tárolása, így azok bővítése ilyen irányban nem megoldható.

Dolgozatomban ezért egy ilyen memóriaoptimalizációs módszertan kidolgozásával és az ahhoz kapcsolódó, általános felhasználási körű kvantumalgoritmus-szimuláló keretrendszer megvalósításával foglalkozom. Bemutatom azokat a klasszikus algoritmus és architektúra tervezési lépéseket, melyek a rendszer alapját képezik, továbbá azt, hogy a keretrendszert hogyan lehet kvantumalgoritmusokkal kapcsolatos kutatások során felhasználni. A keretrendszer célja elsősorban az, hogy a kisebb erőforrással rendelkező felhasználók számára megnövelje a gyakorlati tesztek futathatóságának a korlátait és ezzel elősegítse az elméleti kutatómunkát. Ennek megfelelően az elkészült rendszert és a hozzá tartozó dokumentációt mindenki számára elérhetővé teszem open-source licenszelt formában az interneten.

Abstract

The quantum algorithm execution frameworks currently available on the market (IBM Qiskit, Google Cirq) implement their computations using unitary matrices of exponential size in the number of qubits. Consequently, they require large amounts of memory, even for small inputs. Although existing frameworks use some optimization methods, these often cannot provide improvements of an order of magnitude (e.g. sparse matrix storage mode) or are only applicable in special cases (Clifford gates). In practice, in contrast to a large company, the average user cannot experiment within reasonable limits, for many algorithms, even with relatively small inputs, as this would incur outstanding hardware costs.

Algorithms that save memory in exchange for increased runtime can reduce these hardware expenses. For example, any submatrix of the unitary matrix can be computed on-the-fly during runtime, or the equivalent conventional algorithm can replace the unitary matrix operation. Although the currently available frameworks are open-source, they store the unitary matrices in memory as an integral part of their architecture, making it impossible to incorporate these memory optimization techniques.

In my paper, I focus on developing these memory optimization methodologies and implementing them in a general-purpose quantum algorithm simulation framework. I present the classical algorithm and architecture design steps that form the basis of the system and demonstrate how this system can be used in quantum algorithm research. The framework is primarily intended to be used in a resource-constrained environment to enable running tests on a larger number of qubits, thus facilitating theoretical research. Accordingly, I will make the system and its documentation available to everyone in an open-source licensed form.

Chapter 1

Introduction

In this chapter I introduce the concept of Quantum Turing-machines and the so-called P versus NP problem, in particular its relations to bioinformatical problems, such as protein folding.

1.1 Quantum and classical computers

Originally the idea of a quantum computer was suggested by Richard Feynman in a 1982 article[9], where he explains that currently existing classical computers are ill-equipped to deal with the complexity of the calculations required to simulate quantum physics. His suggestion was to replace the current hardware standard with one, which works based on quantum physical phenomena, thus giving it the capability to simulate the very thing it is based on.

The current computational model we use for classical computers is called the Turing-machine. These new types of computers are so different from classical ones that they run based on a completely different set of rules. Following the work of many computer scientists (Benioff[3], Deutsch[7], and Bernstein and Vazirani[5]), the computational model for Quantum computers was mathematically defined in the late 1980s: the Quantum Turing machine.

In the classical world, on the Turing machine, mathematicians and computer scientists have been working on coming up with fast solutions to all kinds of algorithmic problems. Many of these problems have important real-life applications, but nobody has been able to come up with a fast solution to them. A subcategory of these unsolved problems is the ones where at least we are able to verify in a fast manner if a solution is correct, these are called the 'NP' problems. A simple way to use the verifier algorithm to solve a problem is to look at all of the possible solutions (the domain of the problem) and verify every one of them, until we find a correct solution. This runs in $O(N)$ linear time relative to the size of the problem's domain. The question is, can we do something faster? This is one of the famous Millennium Prize Problems set by the Clay Mathematics Institute a hundred years ago, the P versus NP problem. This problem has eluded computer scientists for a century.

In the quantum world, on the Quantum Turing machine, there exists a better method for the classical linear verifier search, which can do it in $O(\sqrt{N})$ time, relative to the size of the problem's domain. This algorithm is called Grover's search. It has also been proven by Bennett, Bernstein, Brassard, and Vazirani, that this is asymptotically tight[4].

1.2 Application of quantum algorithms in bioinformatics

An interesting area for algorithmic research is bioinformatics. Many problems here have significant impact on our everyday lives since discoveries in this area can help us solve many of today's major global problems, for example, aiding the creation of more effective medical treatments, advancing our understanding of genetic diseases, developing resistant crops to tackle a global food crisis or inventing novel technologies to reduce and revert environmental pollution. I am particularly interested in computer-aided drug design, where problems such as protein folding[6] and molecular docking[1] turn out to be NP-hard ones, which means that despite decades of effort, we have yet to come up with efficient solutions to them using classical computers.

In the past year I have been researching protein folding and how to implement it on a general-purpose quantum computer. I have ran into a significant problem: I was unable to run any experiments of usable size, mainly due to limitations in memory. Due to quantum parallelism, the memory requirements of running a quantum calculation simulation are super-exponential. In particular, there is one component in Qiskit, which seemed to come back in any form of model I have tried to implement: a quantum gate for taking the sum of n qubits, called the WeightedAdder class.

This component came to my attention, because a natural way to encode protein structures is by creating a 2D or 3D grid and laying the aminoacid chain down on it[8], as seen below.

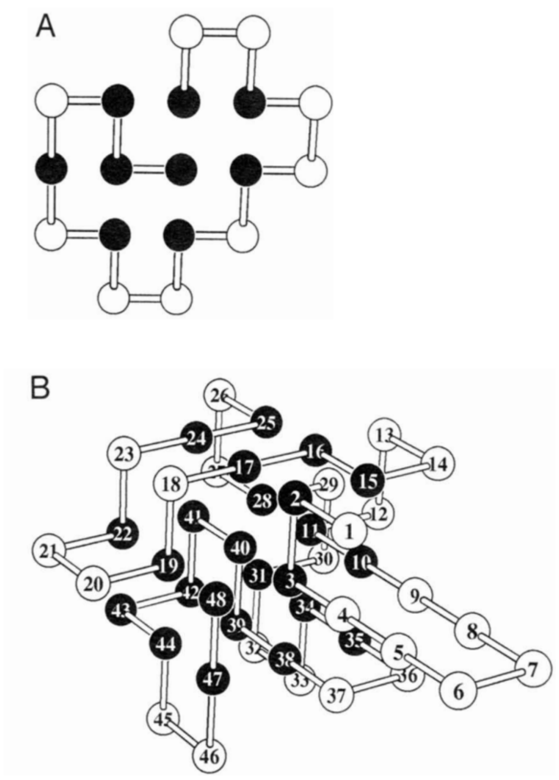


Figure 1.1: *HP model of protein folding[8]*

From a single vertex in a 3D grid, we can step in 4 or 6 directions: up, down, left, right and inwards and outwards in the 3D case. We can encode these naturally, using one-hot encoding, by introducing 6 bits of information. If we assume that the chain starts in the origin, then we can encode a chain shape by giving the directions of the $(n-1)$ steps it takes.

A chain like this is viable, when it doesn't cross over itself. A chain's optimality is assessed by counting how many pairs of various aminoacids are neighbouring each other. To answer both of these questions, we must be able to calculate relative distances between any two points of the chain. Using the directinal one-hot encoding model, these questions can be answered by taking the sum of some qubits.

Using these operations, we can create a quantum oracle, that assesses the optimality of a particular chain and use Grover's quantum search algorithm to find the best possible solution.

Unfortunately, while Qiskit itself is open-source, it's architecture (similarly to other quantum computing frameworks) is designed from the core to store the matrices of various operations (such as the WeightedAdder operation) in its memory and retrieve this information during simulation. This means that I am unable to correct this single operation in Qiskit.

1.3 Contents of this dissertation

In order to reduce the memory requirements for any quantum computation simulation, I have to be able to reduce storing large operation matrices in memory whenever I can. This requires a completely different architecture.

The scope of this report is designing and implementing this architecture, particularly solving the problem with WeightedAdder. While the original motivation for the focus on this specific component comes from protein folding, bioinformatics is out of scope for this paper. Instead, I will be taking a much simpler problem, a generic version of Sudoku, which also requires the WeightedAdder component and Grover-search to be solved.

The remaining chapters are structured as follows: In Chapter 2 I introduce Grover's search algorithm framework and solve a generalized version of the Sudoku puzzle with it. I iterate over the necessary components from this solution, the particular operators needed for the oracle and the amplitude amplification technique's implementation. In Chapter 3 I lay down the mathematical foundations for a quantum simulator framework's implementation, particularly the solution to applying a quantum operator to a subset of the registers in the system, then I introduce the quantum operators and their implementations in my system. Finally, I describe the architectural design patterns used in the system. In Chapter 4 I summarize the results of this paper and lay down my plans for the future.

Chapter 2

Grover's algorithm

Grover's search algorithm[13] is a quantum algorithm framework, that takes a user-defined solution verifier algorithm (the oracle) and turns it into a $\Theta(\sqrt{N})$[4] solver. This provides a quadratic speedup over the classical brute force equivalent.

2.1 Introduction to Grover's search algorithm framework

Many sources call this a database search algorithm, since in Grover's original paper it was described as such. However, the 'database' here is an abstract entity, that represents the entire domain of the problem, while the so-called 'marked' elements are the correct solutions in this domain, for which the oracle would return a 'YES' answer. Using the terms 'problem domain' instead of 'database', 'verifier algorithm' insead of 'oracle' and 'solutions' instead of 'marked elements' makes Grover's importance and connection to the P versus NP problem clearer and the details of the algorithm easier to understand.

Another common description of Grover's search algorithm is that it can solve 'unstructured search problems'. What they mean by this is that the algorithm doesn't construct a solution by iterating over partial solutions or improving a non-solution step-by-step. Constrast this with for example how Prim's minimum spanning tree algorithm iterates on partial solutions by connecting the remaining vertices of the graph one at a time. This requires knowledge of the graph and knowledge of how to build a minimal spanning tree one vertex at a time.

Grover doesn't need to know the structure of the original problem, the relationship between partial solutions or how to improve non-solutions. It only needs to know how to verify a solution. It starts by taking all of the entities from the problem's domain with uniform distribution.

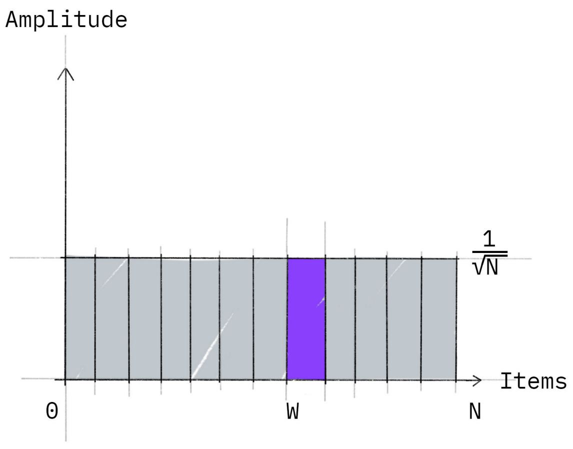


Figure 2.1: *Grover starts out with the uniform distribution[11]*

Then, it uses the verifier algorithm in a process to manipulate their probabilities until the correct entities' probabilities are very high, while the incorrect entities' probabilities are very low. This process is called amplitude amplification.

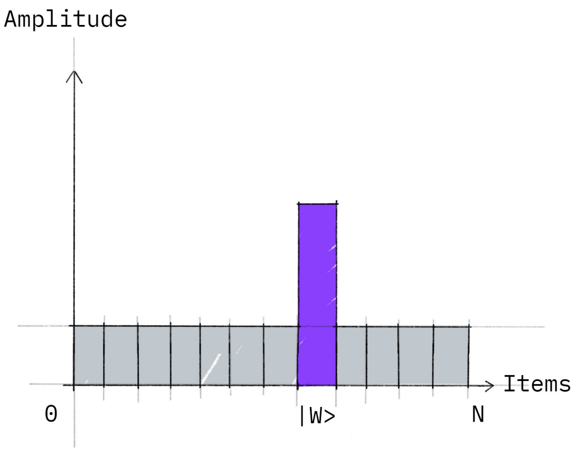


Figure 2.2: *Grover amplifies the amplitude(s) of the correct solution(s)[11]*

Finally, it samples from this probability distribution, which results in a correct solution entity with a high chance.

Working with a probability distribution over an exponentially large set of entities is only possible in a memory-efficient way on a quantum computer, thanks to the quantum physical nature of qubits.

A register of classical bits can only represent a single entity (encoded as a binary number), we would need separate registers to represent a set and we can only operate on the entire set in a linear fashion, one register at a time. In contrast, a register of quantum bits, or 'qubits' itself can represent a set of entities (a set of binary numbers) from the domain using the quantum physical phenomenon of superposition with a probability distribution over these elements.

The manipulation of these probabilities happens using quantum operators or gates, which are the basis of all quantum algorithms on gated general-purpose quantum computers.

However, we do not have access to this probability distribution or the high probability elements in it. The only thing we can do is read the register, which is an operation that samples a single entity from the current probability distribution in the register, destroying it in the process. We are unable to 'iterate' the contents of the register or know what the probability of the resulting element was from the sampling.

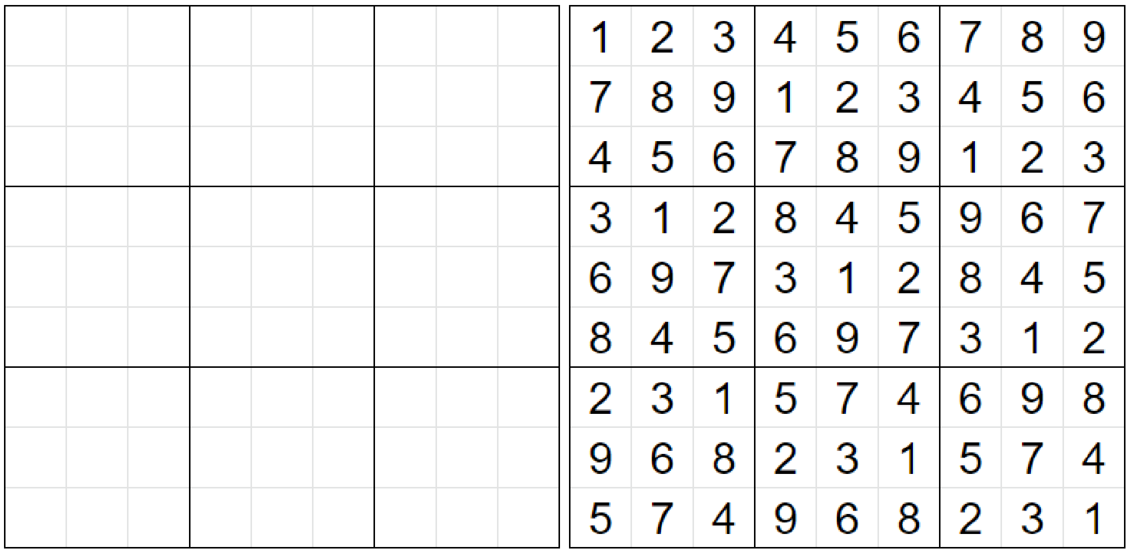
This is the reason why quantum parallelism is not as trivial as the name suggests: while we can run the computation itself in parallel, gaining access to the information that we stored in the register is difficult and destructive. Amplitude amplification is a technique that we use to fix this problem, however it requires $O(\sqrt{N})$ time, where N is the size of the problem's domain, where $N = 2^n$ if the quantum register has n qubits.

One of the most important property of quantum registers is that they can even represent probability distributions, even ones where the individual qubits are **not independent**. This is called quantum entanglement.

The simplest forms of quantum entanglement are Bell states, which can occur between two qubits. In one of these Bell states, the probability distribution of our 2 qubit quantum registers is "00" with 50% probability and "11" with 50%. Reading the contents of just the first qubit will result in a 50% chance of reading a 0 and a 50% chance of reading a 1. However, once we know the result from the first qubit, we can be 100% sure, that when we sample the second qubit, we will get the same number as a result from it.

2.2 Showcasing the algorithm on a simple task

In the original Sudoku puzzle, we have a $(3^2 \cdot 3^2)$ table, that must be filled with numbers between 1 and 9. A correct solution to a puzzle is where each row, column and distinct $(3 \cdot 3)$ square has unique numbers.



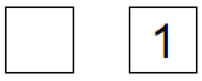
1	2	3	4	5	6	7	8	9
7	8	9	1	2	3	4	5	6
4	5	6	7	8	9	1	2	3
3	1	2	8	4	5	9	6	7
6	9	7	3	1	2	8	4	5
8	4	5	6	9	7	3	1	2
2	3	1	5	7	4	6	9	8
9	6	8	2	3	1	5	7	4
5	7	4	9	6	8	2	3	1

(a) *Empty* **(b)** *Solved*

Figure 2.3: *Sudoku puzzle*

In order demonstrate memory usage scaling, I generalize this Sudoku to a table of size $(n^2 \cdot n^2)$, where each row, column and $(n \cdot n)$ distinct subsquare of the table must be a unique number from the $[1, n^2]$ interval.

The only solution for $n = 1$ is trivial.

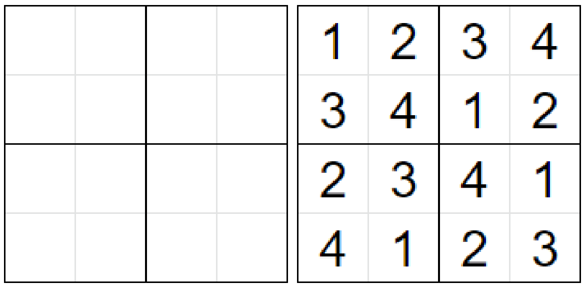


1

(a) *Empty* (b) *Solved*

Figure 2.4: *Sudoku puzzle* ($n = 1$)

An example solution for $n = 2$.



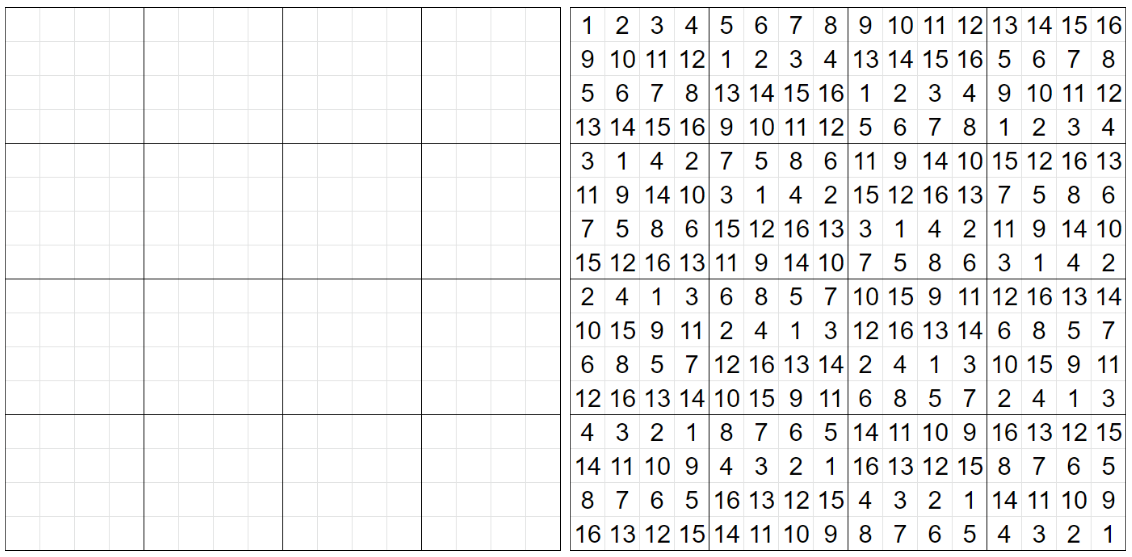
1	2	3	4
3	4	1	2
2	3	4	1
4	1	2	3

(a) *Empty* (b) *Solved*

Figure 2.5: *Sudoku puzzle* ($n = 2$)

$n = 3$ is normal Sudoku.

And an example for $n = 4$ is the following.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
9	10	11	12	1	2	3	4	13	14	15	16	5	6	7	8
5	6	7	8	13	14	15	16	1	2	3	4	9	10	11	12
13	14	15	16	9	10	11	12	5	6	7	8	1	2	3	4
3	1	4	2	7	5	8	6	11	9	14	10	15	12	16	13
11	9	14	10	3	1	4	2	15	12	16	13	7	5	8	6
7	5	8	6	15	12	16	13	3	1	4	2	11	9	14	10
15	12	16	13	11	9	14	10	7	5	8	6	3	1	4	2
2	4	1	3	6	8	5	7	10	15	9	11	12	16	13	14
10	15	9	11	2	4	1	3	12	16	13	14	6	8	5	7
6	8	5	7	12	16	13	14	2	4	1	3	10	15	9	11
12	16	13	14	10	15	9	11	6	8	5	7	2	4	1	3
4	3	2	1	8	7	6	5	14	11	10	9	16	13	12	15
14	11	10	9	4	3	2	1	16	13	12	15	8	7	6	5
8	7	6	5	16	13	12	15	4	3	2	1	14	11	10	9
16	13	12	15	14	11	10	9	8	7	6	5	4	3	2	1

(a) *Empty* (b) *Solved*

Figure 2.6: *Sudoku puzzle* ($n = 4$)

2.3 Designing a quantum solver for the Sudoku puzzle

In this section I first define the Sudoku problem's representation in a binary form, then design the verifier algorithm (quantum oracle) for the puzzle, finally I go over the remaining parts of Grover's framework and the amplitude amplification technique it uses.

2.3.1 Register definitions

The first step is to encode the problem using quantum registers. The size-n Sudoku table has n^2 rows and columns. Every cell in it is represented by a quantum register:

$$\text{cell}[i][j] = |0\dots010\dots0\rangle\,, \;\; \forall\,(0 \le i, j < n^2).$$

The length (qubits) of the register is n^2 and the number in the cell is represented using one-hot encoding. One-hot encoding means, that a number between 0 and $b-1$ is represented by a b bit register, each number corresponding to a single bit being 1 in the register:

$$\text{cell}[i][j][k] = \begin{cases} |1\rangle & \text{cell}_{(i,j)}\text{'s number is } (k+1) \\ |0\rangle & \text{otherwise} \end{cases}, \;\; \forall(\,0 \le i, j, k < n^2).$$

In Qiskit, once the qubit registers are added to a circuit, they can be indexed using a single dimensional index, such as:

$$\text{cell}[i][j][k] = \text{cell}[i \cdot n^4 + j \cdot n^2 + k], \;\; \forall\,(0 \le i, j, k < n^2).$$

2.3.2 Oracle operator

In this section I define the verification algorithm used by Grover's framework. This is done by creating constraints for the Sudoku cells' registers.

2.3.2.1 Constraint definitions

I will verify if a solution is correct, using uniqueness constraints. These constraints will all be using the same scheme, which I define as UNIQUE_ONE($[x_0, \dots, x_{n-1}]$) constraint, where $[x_0, \dots, x_{n-1}]$ is a list of single-dimensional indexes. If a UNIQUE_ONE($[x_0, \dots, x_{n-1}]$) constraint is applied, the qubits with indexes $[x_0, \dots, x_{n-1}]$ must contain exactly one $|1\rangle$.

The verifications are defined as follows.

Cells shall be one-hot encoded

Every (i, j) row and column index pair corresponds to a cell. The qubits in this cells, indexed by k shall have a single $|1\rangle$ among them:

$$\text{UNIQUE\_ONE}([i \cdot n^4 + j \cdot n^2 + k]_{0 \le k < n^2}), \;\; \forall 0 \le i, j < n^2.$$

Numbers in each row shall be unique

For every i row and every k one-hot encoded number position, the k number should be present in the ith row exactly once, indexed by the j columns:

$$\text{UNIQUE\_ONE}([i \cdot n^4 + j \cdot n^2 + k]_{0 \leq j < n^2}), \quad \forall 0 \leq i, k < n^2.$$

Numbers in each column shall be unique

For every j column and every k one-hot encoded number position, the k number should be present in the jth column exactly once, indexed by the i rows:

$$\text{UNIQUE\_ONE}([i \cdot n^4 + j \cdot n^2 + k]_{0 \leq i < n^2}), \quad \forall 0 \leq j, k < n^2.$$

Numbers in each square shall be unique

In order to create this constraint, the row and column indexes must be taken apart into an inner and outer index:

$$\begin{aligned} i &= i_{outer} \cdot n + i_{inner}, \quad 0 \leq i_{outer}, i_{inner} < n, \\ j &= j_{outer} \cdot n + j_{inner}, \quad 0 \leq j_{outer}, j_{inner} < n. \end{aligned}$$

This way i_{outer} and j_{outer} index the squares the constraint is applied to, while i_{inner} and j_{inner} index their internal cells.

Then, for every square, indexed by the (i_{outer}, j_{outer}) pair and every k one-hot encoded number position, the k number should be present in the (i_{outer}, j_{outer}) square exactly once, indexed by the (i_{inner}, j_{inner}) cell index pairs:

$$\begin{aligned} \text{UNIQUE\_ONE}([(i_{outer} \cdot n + i_{inner}) \cdot n^4 + (j_{outer} \cdot n + j_{inner}) \cdot n^2 + k]_{0 \leq i_{inner}, j_{inner} < n}, \\ \forall 0 \leq i_{outer}, j_{outer} < n, \\ \forall 0 \leq k < n^2. \end{aligned}$$

2.3.2.2 Implementation of the UNIQUE_ONE constraint

In order to implement a UNIQUE_ONE($[x_0, \ldots, x_{n-1}]$) constraint, we use the WeightedAdder component from Qiskit. This takes n qubits and sums them up into a $\log_2(n)$ sized array. We want the sum to be exactly 1, which means that the output of the sum array should be equal to $|0 \ldots 01\rangle$. Adding a NOT gate to the least significant qubit, the output should be $|0 \ldots 0\rangle$. This can be tested using a multi-controlled NOT, or $M - CNOT$ gate.

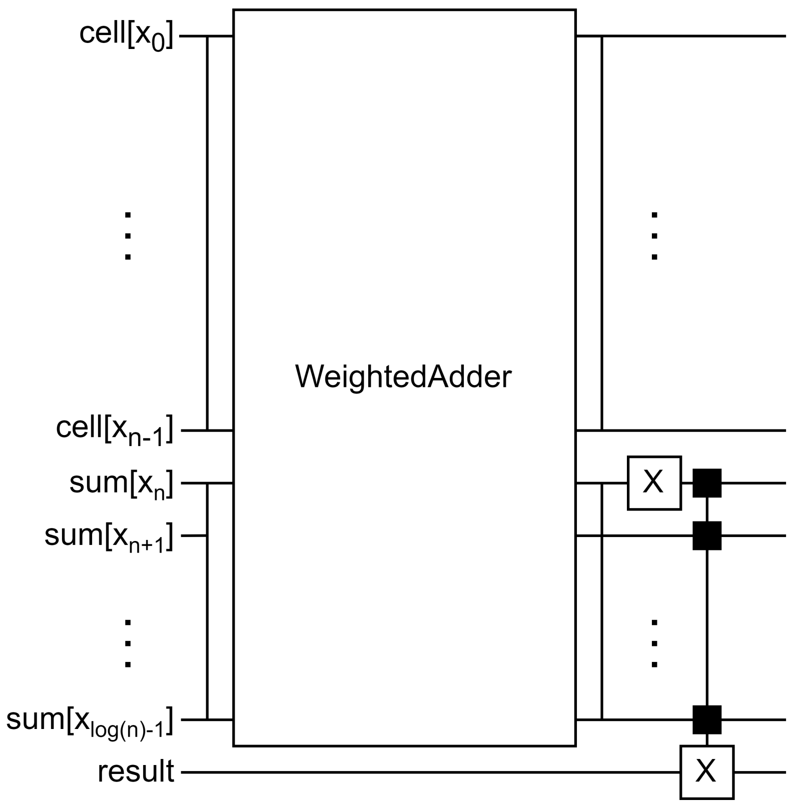


Figure 2.7: *UNIQUE_ONE constraint implementation*

For multiple UNIQUE_ONE constraints, the results can be aggregated using a final multi-controlled $CNOT$ gate, or using a single, common $M - CNOT$ gate for all of them.

In the end, this final $MCNOT$ operation is applied to a single, $|oracle\rangle$ qubit in the circuit.

2.3.3 Grover's framework: The amplitude amplification technique

This chapter is based on the chapter on Grover's algorithm from the Qiskit Texbook[11].

Let us denote the $cells[i][j][k]$ qubits with the $|cells\rangle$ qubit vector!

In the beginning, Grover initializes this vector to the uniform distribution:

$$|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle \,.$$

This is done by applying an n-dimensional Hadamard-matrix to the $|0 \ldots 0\rangle$ vector:

$$|s\rangle = H^{\otimes n} |0\rangle \,.$$

We can see this initial state in the following figure.

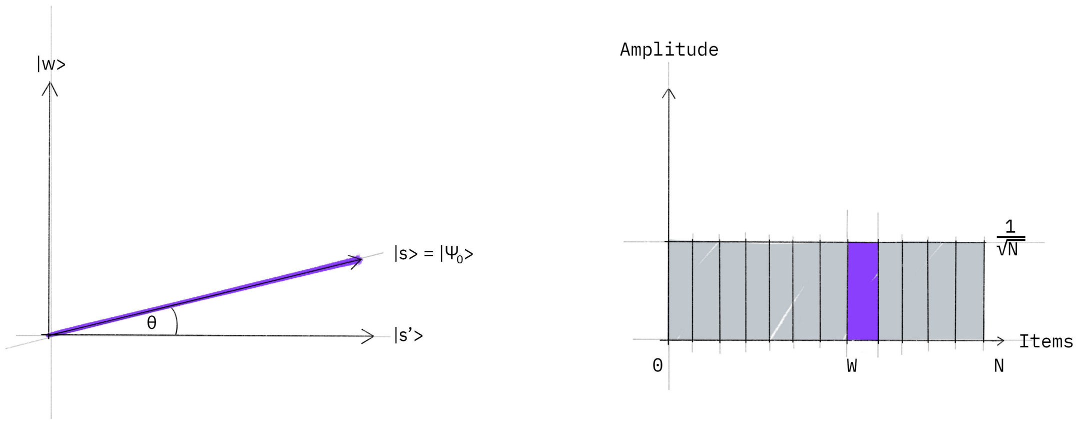


Figure 2.8: *Initialization[11]*

On the right hand side, the individual amplitudes are represented for all elements in the $|cells\rangle$ vector. This can be seen as an N-dimensional vector. Since we are only interested in what is the probability of sampling a correct solution, we can project this N-dimensional vector-space into a 2-dimensional one, where the dimensions correspond to the probability of a solution and a non-solution sampling. The x-axis, or $|s'\rangle$ represents non-solutions, while the y-axis, or $|w\rangle$ represents the solutions.

In order to create a Grover's oracle from the oracle function defined in the previous section, I initialize the $|oracle\rangle$ qubit to $|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. When the oracle circuit is applied to $|oracle\rangle = |-\rangle$, the phase kickback effect results in a negative amplitude multiplier exactly on the elements in the $|cells\rangle$ vector, which are solutions according to the oracle.

This effect can be seen on these figures:

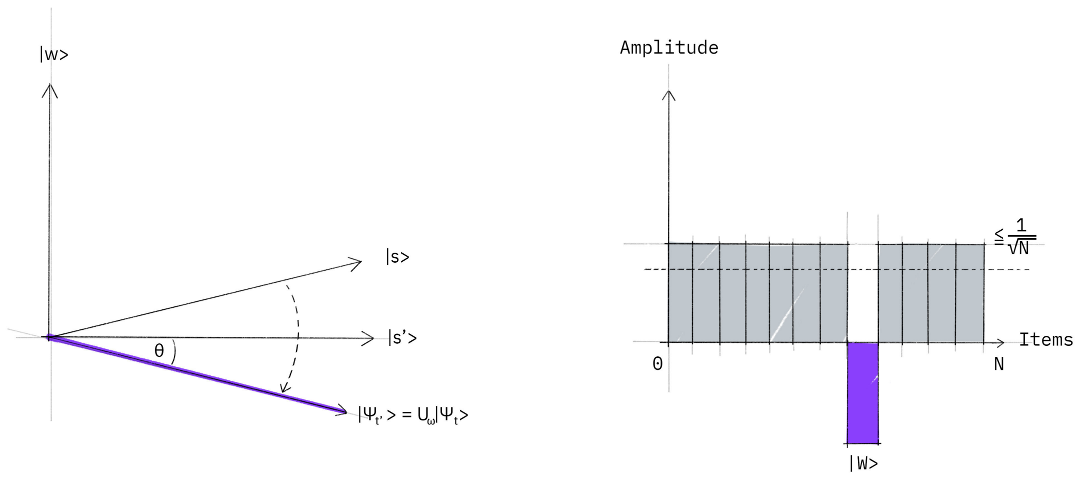


Figure 2.9: *Phase kickback[11]*

On the right-hand side, the solutions amplitudes are flipped. Since the solutions constutire the y-axis on the left-hand side, this results in a reflection over the x-axis.

Finally, another reflection is performed, which reflexts over the average amplitude in the current superposition. For non-solution elements this decreases their overall probability, while the flipped solution elements gain probability.

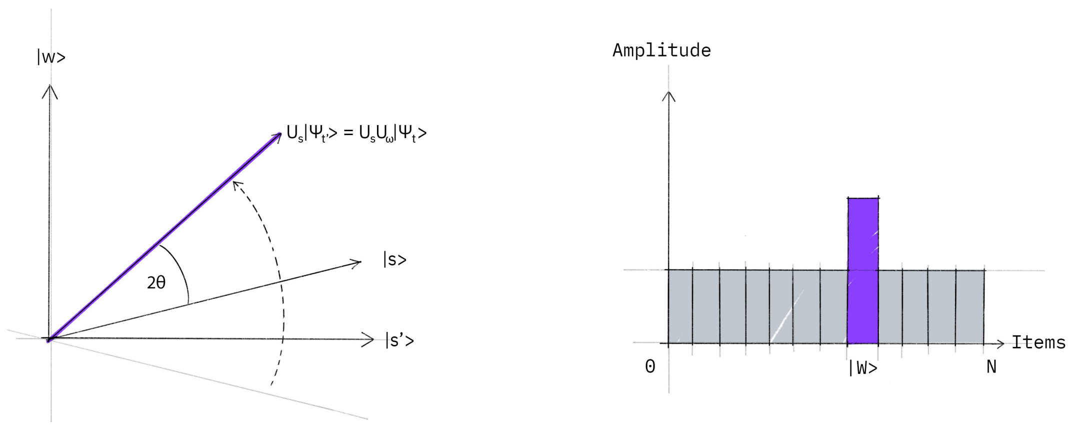


Figure 2.10: *Reflect over the average amplitude[11]*

On the left-hand side this can be represented by a reflextion over the initial (uniform) distribution, the $|s\rangle$ vector. This operation is called the diffuser operator, which is implemented by a Grover matrix.

Together, these two reflections constitute a rotation towards the $|w\rangle$ solution axis with a degree that depends on the size of the search space (N) and the number of solutions (M). In order to reach the $|w\rangle$ axis as close as possible, the rotation must be performed $\sqrt{\frac{N}{M}}$ times.

In order to recompute the oracle on the new search space, first the old results must be erased from the ancilla (sum) qubits in the system. Since the WeightedAdder operator works internally with $CNOT$ gates, erasing the result can be done by applying the same circuit in reverse order.

2.4 The tools needed to implement Grover's algorithm

In this capter I have introduced Grover's algorithm and how to use it to solve a generalized version of the Sudoku puzzle.

In order to use this framework, 4 operators must be implemented in the system: the Hadamard (for the phase-kickback), the Grover (the diffuser operator), the Sum (WeightedAdder) and the Multi-controlled NOT gate (for the oracle bit).

Chapter 3

Simulation of quantum algorithms in a memory-efficient way

In this chapter I will examine the architecture and implementation of the framework that is capable of running simulations of gated general-purpose quantum computation.

3.1 Design goals

For an n qubit register, the register itself must be stored using 2^n complex numbers (the probability amplitudes of each of the 2^n 0/1 variations), however the size of the matrix that is applied to it is $(2^n)^2$, which is considerably larger.

Qiskit uses a lot of memory, because it stores every single quantum operator matrix in memory. Furhermore, even if the operation is the same, if it is applied multiple times, individual instances of the matrix are created. This is extremely wasteful.

While it uses some techniques to reduce the memory allocations, such as sparse matrix representation, this cannot fundamentally get around the issue, that the architecture itself does not allow flexibility of operator representation.

Instead of storing the matrices in-memory I will be designing a system where operators can be created without the need for a matrix representation at all, or when that is not possible the currently used column of the matrix can be generated "on-the-fly" for application.

3.2 Quantum registers

The first step in the implementation process is designing the inner workings of the quantum registers. In order to represent an n qubit register, we must store 2^n complex numbers, the probability amplitudes of each of the possible 0/1 bit representations, as follows:

$$\begin{aligned}
|0\dots000\rangle &\rightarrow c_0\\
|0\dots001\rangle &\rightarrow c_1\\
|0\dots010\rangle &\rightarrow c_2\\
|0\dots011\rangle &\rightarrow c_3\\
&\dots\\
|1\dots111\rangle &\rightarrow c_{2^n-1},
\end{aligned}$$

where $c_0, \dots, c_{2^n-1} \in \mathbb{C}$.

When multiple registers are present in the system, handling operators that are only applied to some of the registers becomes problematic. Since qubits can be entangled, every single new register added to the system multiplies the amount of storage required for the probability amplitudes.

3.2.1 General solution

Let there be r registers in the system,

$$\{R_0, \dots, R_{r-1}\}$$

and let n_i be number of qubits in register R_i, where $0 \leq i < r$.

The total number of qubits in the system is therefore

$$n = \sum_{i=0}^{r-1} n_i.$$

Then, let U be an operation, a unitary (square) matrix, that is applied to k of the registers:

$$\{R_{r_0}, \dots, R_{r_{k-1}}\}, \quad 0 \leq r_j < r, \quad 0 \leq j < k, \quad k \leq r.$$

The number of qubits U must operate on is therefore

$$m = \sum_{j=0}^{k-1} n_{r_j}.$$

which means that matrix U is of size $(M \times M)$, where

$$M = 2^m = \prod_{j=0}^{k-1} 2^{(n_{r_j})}.$$

The complex probability amplitudes for all possible register contents in the system are stored in a single array C of complex numbers. The size of C is

$$N = 2^n = \prod_{i=0}^{r-1} 2^{n_i}$$

and its contents are

$$C = [C[0], \ldots, C[N-1]] \in \mathbb{C}^N.$$

Let us introduce binary indexing sequences on C. A sequence of qubits

$$|b_{n-1}, b_{n-2}, \ldots, b_2, b_1, b_0\rangle$$

is a binary indexing sequence on C and it corresponds to

$$C[|b_{n-1}, b_{n-2}, \ldots, b_2, b_1, b_0\rangle] = C[B],$$

where

$$B = \sum_{i=0}^{n-1} b_i \cdot 2^i.$$

A binary indexing sequence is partitioned by the registers in the following way:

$$|b_{n-1}, b_{n-2}, \ldots, b_2, b_1, b_0\rangle = |R_{r-1}|R_{r-2}| \ldots |R_2|R_1|R_0\rangle\,.$$

Similarly, a single cell of matrix U can be indexed using 2-dimensional binary indexing sequences. Matrix U is indexed by the following two m dimensional qubit sequences:

$$|a_{m-1}, a_{n-2}, \ldots, a_2, a_1, a_0\rangle$$

and

$$|b_{m-1}, b_{n-2}, \ldots, b_2, b_1, b_0\rangle$$

and it corresponds to

$$U[|a_{m-1}, a_{n-2}, \ldots, a_2, a_1, a_0\rangle][|b_{m-1}, b_{n-2}, \ldots, b_2, b_1, b_0\rangle] = U[A][B],$$

where

$$A = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

and

$$B = \sum_{i=0}^{n-1} b_i \cdot 2^i.$$

A 2-dimensional binary indexing sequence on matrix U can also be partitioned by the registers U is applied to, in the following ways:

$$|a_{m-1}, a_{m-2}, \ldots, a_2, a_1, a_0\rangle = |R_{r_{k-1}}|R_{r_{k-2}}| \ldots |R_{r_2}|R_{r_1}|R_{r_0}\rangle$$

and

$$|b_{m-1}, b_{m-2}, \ldots, b_2, b_1, b_0\rangle = |R_{r_{k-1}}|R_{r_{k-2}}| \ldots |R_{r_2}|R_{r_1}|R_{r_0}\rangle .$$

To implement the application of matrix U to registers $\{R_{r_0}, \ldots, R_{r_{k-1}}\}$ in the system, the C array must be rearranged, so that U can be applied to continuous subsequences of C.

This can be done via a bit-mapping on the binary indexing sequences. The qubits corresponding to the registers $\{R_{r_0}, \ldots, R_{r_{k-1}}\}$ are moved to the lower end of the sequence, while the rest of the registers to the upper end.

Let us index the registers U is not applied to with $\{s_0, \ldots, s_{n-k-1}\}$, so that

$$\{0, \ldots, n-1\} = \{r_0, \ldots, r_{k-1}\} \dot\cup \{s_0, \ldots, s_{n-k-1}\}.$$

Then, the binary index sequence mapping (BISM) is defined as

$$\mathbf{BISM} : |R_{r-1}|R_{r-2}| \ldots |R_2|R_1|R_0\rangle \to |R_{s_{r-k-1}}| \ldots |R_{s_0}|R_{r_{k-1}}| \ldots |R_{r_0}\rangle .$$

The **BISM** function can be used to define the permutation on the C array, by mapping

$$C'[\mathbf{BISM}(B)] = C[B], \quad 0 \le B < N.$$

The C' array's binary indexing sequences can now be partitioned into an upper and lower region, such as

$$\left|b'_{n-1},\ldots,b'_m|b'_{m-1},\ldots b'_0\right\rangle,$$

where the lower region's indexing sequences correspond 1-to-1 to the U unitary matrix operation's second dimension.

With everything set up, we can now define the application of U to C'. Let the resulting register contents be R', where

$$R'[|b'_{n-1},\ldots,b'_k|a'_{k-1},\ldots a'_0\rangle] = \sum_{\substack{0\leq i<m\\ \forall b_i\in\{0,1\}}} U[|a'_{m-1},\ldots,a'_0\rangle][|b'_{m-1},\ldots,b'_0\rangle]\cdot C'[|b'_{m-1},\ldots,b'_0\rangle]. \tag{3.1}$$

During this application, we can see that the matrix is read in a column-by-column fashion, which means that we only need to generate a single column of U.

In cases where the operation is a mapping or aggregation itself (such as the WeightedAdder from the Grover's search), it can be performed without generating the column itself, the same permutation logic is used, but instead of generating an entire row of U, the $\left|a'_{m-1},\ldots,a'_0\right\rangle$ "output index" is calculated based on the $\left|b'_{m-1},\ldots,b'_0\right\rangle$ "input index" by a function

$$u:\left|b'_{m-1},\ldots,b'_0\right\rangle\to\left|a'_{m-1},\ldots,a'_0\right\rangle,$$

which then replaces the matrix multiplication as follows:

$$R'[|b'_{n-1},\ldots,b'_k|a'_{k-1},\ldots a'_0\rangle] = \sum_{\substack{0\leq i<m\\ \forall b_i\in\{0,1\}\\ u(\left|b'_{m-1},\ldots,b'_0\right\rangle)=\left|a'_{m-1},\ldots,a'_0\right\rangle}} C'[|b'_{m-1},\ldots,b'_0\rangle]. \tag{3.2}$$

These equations are the basis of the memory-efficiency of this framework.

Finally, the R' array must be inverse-permuted back to the original order of the indexing qubits

$$R[B]=R'[\mathbf{BISM}(B)],\quad 0\leq B<N.$$

3.2.2 Presenting the solution on an example

For example when 3 registers are present, R_0 consisting of 1 qubit, R_1 consisting of 1 qubits and R_2 consisting of 2 qubits, then the binary indexing sequence of the amplitude registers is the following: $|R_{2,0},R_{2,1};R_{1,0};R_{0,0}\rangle$.

The probability amplitudes stored are the following:

$$\begin{array}{llll}
|00,0,0\rangle \rightarrow & c_0 & |10,0,0\rangle \rightarrow & c_8 \\
|00,0,1\rangle \rightarrow & c_1 & |10,0,1\rangle \rightarrow & c_9 \\
|00,1,0\rangle \rightarrow & c_2 & |10,1,0\rangle \rightarrow & c_{10} \\
|00,1,1\rangle \rightarrow & c_3 & |10,1,1\rangle \rightarrow & c_{11} \\
& & & \\
|01,0,0\rangle \rightarrow & c_4 & |11,0,0\rangle \rightarrow & c_{12} \\
|01,0,1\rangle \rightarrow & c_5 & |11,0,1\rangle \rightarrow & c_{13} \\
|01,1,0\rangle \rightarrow & c_6 & |11,1,0\rangle \rightarrow & c_{14} \\
|01,1,1\rangle \rightarrow & c_7 & |11,1,1\rangle \rightarrow & c_{15}.
\end{array}$$

The simplest solution would be to apply a "no-operation" operator, or the identity matrix to the remaining registers, however this will not scale well memory-wise with the number of registers increasing in the system.

Instead I implemented the register handling in a way that allowed me to skip storing "no-operation" matrices in the memory completely. In order to apply an operator to only some registers in the system, the probability amplitudes are re-arranged in a way so that a continuous section of memory corresponds to a column of the matrix. This way, the matrix operation can be applied to sections of probability amplitudes iteratively.

For example, if we apply a 3 qubit operator to the registers R_0 and R_2, then the previous table is rearranged so that the bits corresponding to R_0 and R_2 are pushed towards the least significant bit in the following way:

$$|R_{2,1}, R_{2,1}; R_{1,0}; R_{0,0}\rangle \rightarrow |R_{1,0}; R_{2,0}, R_{2,1}; R_{0,0}\rangle .$$

$$\begin{array}{ll}
|00,0,0\rangle \rightarrow |0,00,0\rangle' \rightarrow c'_0 \rightarrow c_0 & |10,0,0\rangle \rightarrow |0,10,0\rangle' \rightarrow c'_4 \rightarrow c_8 \\
|00,0,1\rangle \rightarrow |0,00,1\rangle' \rightarrow c'_1 \rightarrow c_1 & |10,0,1\rangle \rightarrow |0,10,1\rangle' \rightarrow c'_5 \rightarrow c_9 \\
|00,1,0\rangle \rightarrow |1,00,0\rangle' \rightarrow c'_8 \rightarrow c_2 & |10,1,0\rangle \rightarrow |1,10,0\rangle' \rightarrow c'_{12} \rightarrow c_{10} \\
|00,1,1\rangle \rightarrow |1,00,1\rangle' \rightarrow c'_9 \rightarrow c_3 & |10,1,1\rangle \rightarrow |1,10,1\rangle' \rightarrow c'_{13} \rightarrow c_{11} \\
& \\
|01,0,0\rangle \rightarrow |0,01,0\rangle' \rightarrow c'_2 \rightarrow c_4 & |11,0,0\rangle \rightarrow |0,11,0\rangle' \rightarrow c'_6 \rightarrow c_{12} \\
|01,0,1\rangle \rightarrow |0,01,1\rangle' \rightarrow c'_3 \rightarrow c_5 & |11,0,1\rangle \rightarrow |0,11,1\rangle' \rightarrow c'_7 \rightarrow c_{13} \\
|01,1,0\rangle \rightarrow |1,01,0\rangle' \rightarrow c'_{10} \rightarrow c_6 & |11,1,0\rangle \rightarrow |1,11,0\rangle' \rightarrow c'_{14} \rightarrow c_{14} \\
|01,1,1\rangle \rightarrow |1,01,1\rangle' \rightarrow c'_{11} \rightarrow c_7 & |11,1,1\rangle \rightarrow |1,11,1\rangle' \rightarrow c'_{15} \rightarrow c_{15}.
\end{array}$$

Then, the 3 qubit operator $U^{8\times8}$, which is an (8×8) matrix can be applied iteratively in the following way:

1. Apply $U^{8\times8}$ to the probability amplitudes corresponding to $R_{1,0} = |0\rangle$:
   $[c_0, c_1, c_4, c_5, c_8, c_9, c_{12}, c_{13}] = [c'_0, c'_1, c'_2, c'_3, c'_4, c'_5, c'_6, c'_7]$.

The resulting vector is the first half of the complete result:
$[r'_0, r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7]$.

2. Apply $U^{8\times 8}$ to the probability amplitudes corresponding to $R_{1,0} = |1\rangle$:
$[c_2, c_3, c_6, c_7, c_{10}, c_{11}, c_{14}, c_{15}] = [c'_8, c'_9, c'_{10}, c'_{11}, c'_{12}, c'_{13}, c'_{14}, c'_{15}]$.
The resulting vector is the second half of the complete result:
$[r'_8, r'_9, r'_{10}, r'_{11}, r'_{12}, r'_{13}, r'_{14}, r'_{15}]$.

3. Iterate over all values for the untouched register R_1 and aggregate the results:
$[r'_0, r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7, r'_8, r'_9, r'_{10}, r'_{11}, r'_{12}, r'_{13}, r'_{14}, r'_{15}]$..

4. Revert the mapping to the original indexes:
$[r'_0, r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7, r'_8, r'_9, r'_{10}, r'_{11}, r'_{12}, r'_{13}, r'_{14}, r'_{15}] =$
$[r_0, r_1, r_4, r_5, r_8, r_9, r_{12}, r_{13}, r_2, r_3, r_6, r_7, r_{10}, r_{11}, r_{14}, r_{15}]$.

3.3 Quantum operators

In order to implement Grover's algorithm the following operators are needed: Hadamard, Grover (diffuser matrix), Sum (WeightedAdder), and Multi-Controlled NOT.

3.3.1 Hadamard

The Hadamard matrix is defined as follows. First, the (2×2) H matrix is the following:

$$\mathbf{H} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

A $(2^n \times 2^n)$ dimensional Hadamard matrix can be created by taking the tensor product of the (2×2) H matrix n times: $\mathbf{H}^{\otimes n}$.

From this equation the jth column of the ith row of the $(2^n \times 2^n)$ matrix can be defined by taking the **BITWISE_AND** between the i and j indexes in binary form, then counting the set bits in that selector, to decide which cells should get a negative multiplier, as follows:

$$H[i][j] = \frac{1}{\sqrt{2^n}} \cdot (-1)^{\mathbf{COUNT\_BITS}(i_{②}\ \mathbf{BITWISE\_AND}\ j_{②})}.$$

This equation is directly implemented and a single column of H is generated on-the-fly when H is applied, using equation (3.1).

3.3.2 Grover

The Grover matrix is the diffusion operator from Grover's algorithm. It is defined using the $\mathbf{H}^{\otimes n}$ matrix, as follows.

Let's define the register $|D\rangle$ to be the following:

$$|D\rangle = \mathbf{H}^{\otimes n} |0\rangle = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} |i\rangle .$$

Then G is the following matrix:

$$\mathbf{G} = 2 |D\rangle \langle D| - \mathbf{I}.$$

If $N = 2^n$, then $\mathbf{G}$ can be represented as follows:

$$\mathbf{G} = \begin{pmatrix} \frac{2}{N} - 1 & \frac{2}{N} & \dots & \frac{2}{N} \\ \frac{2}{N} & \frac{2}{N} - 1 & \dots & \frac{2}{N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{2}{N} & \frac{2}{N} & \ddots & \frac{2}{N} - 1 \end{pmatrix} .$$

It is straightforward to implement G, since the the jth column is $\frac{2}{N}$, except for the jth cell, where it is $\frac{2}{N} - 1$. This matrix is also generated on-the-fly, column-by-column, using equation (3.1).

3.3.3 Sum

The sum operator is one, that can be represented directly using equation (3.2), by defining the

$$u : |b_{m-1}, \dots, b_0\rangle \to |a_{m-1}, \dots, a_0\rangle$$

function.

First, the m qubits of the opetor are partitioned into two parts: input and output

$$m = s + \lceil \log_2(s) \rceil,$$

since the sum of s qubits can be represented on $\lceil \log_2(s) \rceil$ bits.

Then, u is defined as

$$u : |0, \dots, 0, b_{s-1} \dots, b_0\rangle \to |\mathbf{COUNT\_BITS}(b_{s-1} \dots, b_0), b_{s-1} \dots, b_0\rangle .$$

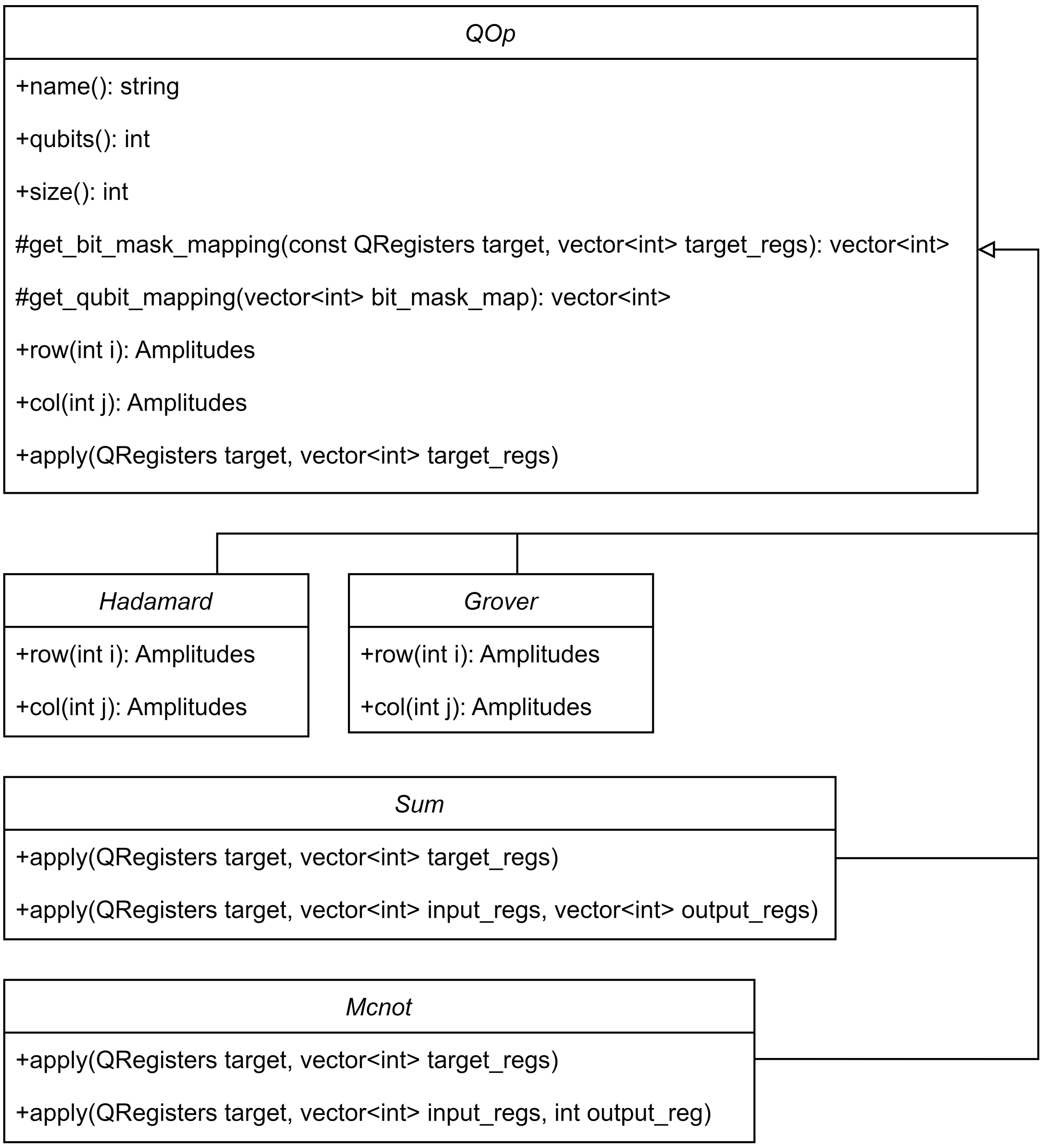
3.3.4 Multi-controlled NOT

Similarly to **Sum**, a Multi-controlled NOT operator is defined using an u function, which applies the *NOT* operator to its most significant bit, when any of the other bits are set.

$$u : |b_{m-1}, b_{m-2} \ldots, b_0\rangle \to |(\vee(b_{m-2} \ldots, b_0) \oplus b_{m-1}), b_{m-2} \ldots, b_0\rangle .$$

3.4 Implementation and design patterns

In the UML diagram below, we can see the part of the system that deals with these memory-efficient operators.



QOp
+name(): string
+qubits(): int
+size(): int
#get_bit_mask_mapping(const QRegisters target, vector<int> target_regs): vector<int>
#get_qubit_mapping(vector<int> bit_mask_map): vector<int>
+row(int i): Amplitudes
+col(int j): Amplitudes
+apply(QRegisters target, vector<int> target_regs)
Hadamard
+row(int i): Amplitudes
+col(int j): Amplitudes
Grover
+row(int i): Amplitudes
+col(int j): Amplitudes
Sum
+apply(QRegisters target, vector<int> target_regs)
+apply(QRegisters target, vector<int> input_regs, vector<int> output_regs)
Mcnot
+apply(QRegisters target, vector<int> target_regs)
+apply(QRegisters target, vector<int> input_regs, int output_reg)

Figure 3.1: *Strategy and Visitor pattern*

The goal of this framework is to allow the user to define quantum operators in a memory-efficient way. There are two types of operators: the ones that are columnwise generated on-the-fly and the u function operators, directly interacting with the binary indexing sequences of the registers.

The code for register handling, the **BISM** operator and the permutation of the probability amplitudes is common amongst all operators. When these operators are being used, they must be callable from the same interface, to ensure that they are interchangeable and can be inherited from.

In order to achieve this, I have implemented the Strategy design pattern. The intent of this pattern, according to the Design Patterns book[12], is to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. In this pattern, a common interface is defined for all operators, with the implementation dependent on the specific operator. This common interface will later allow me to pontentially generate circuit diagrams for an implemented quantum algorithm.

The `QOp` class is the base class of all of the others. The `Hadamard`, `Grover`, `Sum` and `Mcnot` classes all inherit from it. In particular, the `Hadamard` and `Grover` classes only redefine the `row` and `column` methods. The generic implementation of the apply method in `QOp` then uses these methods to apply the operator on the QRegisters.

The `get_bit_mask_mapping` and `get_qubit_mapping` methods deal with the probability amplitude permutation and the **BISM** mapping. They are protected, so inherited classes can make use of them too.

The `Sum` and `Mcnot` required me to implement a form of inversion-of-control. In the first iteration, a third class (an `Orchestrator`) received both the registers and the operator it should apply to them, iteratively generated the necessary rows from the operator and applied them onto the registers. This was poblematic, because this type of control flow made it difficult to implement the `Sum` and `Mcnot` operators, since they calculate their result without relying on an explicit representation row format.

The knowledge of how an operator should be applied to the registers should be given to the operator itself, since the framework relies on clever, operator-specific memory-efficient implementations to function. This is exactly what the Visitor pattern is used for. According to the Design Patterns book[12] we can *"Use the Visitor pattern when many distinct and unrelated operations need to be performed on objects in an object structure, and [we] want to avoid "polluting" their classes with these operations. Visitor lets [us] keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them."*

Chapter 4

Summary of results

My goal for the future is to explore quantum solutions to classically NP-hard problems and their connection to Grover's search algorithm. I am interested in computer-aided drug design, particularly the NP-hard problems of protein folding and molecular docking. While researching protein folding, I found a simplified model that is still NP-hard but can be implemented on a gated general-purpose quantum computer. However, I ran into a hard memory limit since the largest computable protein chain contained at most four amino acids, on which the problem is trivial.

I looked into various open-source quantum computational frameworks, notably Qiskit and how I might reduce their memory requirements. These frameworks focus on a different goal: to allow the programming of quantum computers their respective vendors sell, which means that simulation, especially memory-efficient simulation, is not their primary concern. Their implementation uses sparse-matrix representation of each unitary matrix and allocates the resources for each individual instance of them. This made it very difficult and expensive to use them for my usecase, which is testing my quantum algorithms for protein folding on even relatively small inputs.

The primary goal of this framework is to reduce memory-usage of simulation while trading in runtime. For research purposes, it is acceptable to wait for example a few days for a simulation of protein folding runs on a relatively high-end PC, however it is not cost-effective to buy terabytes of memory or rent a memory-optimized virtual machine from the cloud.

In this dissertation, I have developed the mathematical framework for implementing general-purpose software for gated quantum computer simulations. These developments have been:

- The logic of handling the probability amplitudes in the current set of registers and applying operators to a subset of these registers using qubit mapping permutations on their binary indexing sequences.
- The architecture allows individual tricks for memory-efficient operator implementation, such as on-the-fly generation and the u function method.
- The building blocks for Grover's algorithm's implementation, the Hadamard, Grover, Sum and MCnot operators.

4.1 Source code availability

The source code for the framework is available at the following link under the open-source MIT License:

https://github.com/nemkin/qmem

4.2 Plans for the future

My goals for the future with this framework is to finish the implementation of Grover's algorithm by connecting the implemented building blocks.

In addition, I would like to introduce unit testing for the individual components of the software. Since all of these operators rely heavily on custom implementation, it is important that their correctness is verified. In particular, I would like to explore methamorphic testing, in which the operators are tested by verifying if they admit to certain mathematical properties, such as the self-adjointness of the Hamilton-operator.

Furthermore, I would like to extend the available operators in the framework so that other types of algorithms can be implemented in it as well.

Bibliography

[1] Tripathi A and Bankaitis Va. Molecular Docking: From Lock and Key to Combination Lock. *Journal of Molecular Medicine and Clinical Applications*, 2(1), 2018. ISSN 25750305. doi: 10.16966/2575-0305.106. URL `https://www.sciforschenonline.org/journals/molecular-biology-medicine/IJMBM-2-106.php`.

[2] Ferenc Balázs and Sándor Imre. *Quantum computing and communications: an engineering approach*. Wiley, Hoboken, N.J., 2013. ISBN 9781118725474.

[3] Paul Benioff. Models of Quantum Turing machines. *Fortschritte der Physik*, 46(4-5):423–441, June 1998. ISSN 0015-8208, 1521-3978. doi: 10.1002/(SICI)1521-3978(199806)46:4/5<423::AID-PROP423>3.0.CO;2-G. URL `http://arxiv.org/abs/quant-ph/9708054`. arXiv:quant-ph/9708054.

[4] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and Weaknesses of Quantum Computing. *SIAM Journal on Computing*, 26(5):1510–1523, October 1997. ISSN 0097-5397, 1095-7111. doi: 10.1137/S0097539796300933. URL `http://epubs.siam.org/doi/10.1137/S0097539796300933`.

[5] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing - STOC '93*, pages 11–20, San Diego, California, United States, 1993. ACM Press. ISBN 9780897915915. doi: 10.1145/167088.167097. URL `http://portal.acm.org/citation.cfm?doid=167088.167097`.

[6] Pierluigi Crescenzi, Deborah Goldman, Christos Papadimitriou, Antonio Piccolboni, and Mihalis Yannakakis. On the Complexity of Protein Folding. *Journal of Computational Biology*, 5(3):423–465, January 1998. ISSN 1066-5277, 1557-8666. doi: 10.1089/cmb.1998.5.423. URL `http://www.liebertpub.com/doi/10.1089/cmb.1998.5.423`.

[7] David Deutsch. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400(1818):97–117, July 1985. ISSN 0080-4630. doi: 10.1098/rspa.1985.0070. URL `https://royalsocietypublishing.org/doi/10.1098/rspa.1985.0070`.

[8] Ken A. Dill, Sarina Bromberg, Kaizhi Yue, Hue Sun Chan, Klaus M. Ftebig, David P. Yee, and Paul D. Thomas. Principles of protein folding - A perspective from simple exact models. *Protein Science*, 4(4):561–602, December 2008. ISSN 09618368, 1469896X. doi: 10.1002/pro.5560040401. URL `https://onlinelibrary.wiley.com/doi/10.1002/pro.5560040401`.

[9] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6-7):467–488, June 1982. ISSN 0020-7748, 1572-9575. doi: 10.1007/BF02650179. URL `http://link.springer.com/10.1007/BF02650179`.

[10] Caroline Figgatt, Dmitri Maslov, Kevin A. Landsman, Norbert M. Linke, Shantanu Debnath, and Christopher Monroe. Complete 3-Qubit Grover search on a programmable quantum computer. *Nature Communications*, 8(1):1918, December 2017. ISSN 2041-1723. doi: 10.1038/s41467-017-01904-7. URL `http://www.nature.com/articles/s41467-017-01904-7`.

[11] Harkins Frank et al. Qiskit/qiskit: Qiskit 0.39.0, October 2022. URL `https://qiskit.org/textbook/ch-algorithms/grover.html`.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley professional computing series. Addison-Wesley, 1995. ISBN 9780201633610.

[13] Lov K. Grover. A fast quantum mechanical algorithm for database search. November 1996. URL `http://arxiv.org/abs/quant-ph/9605043`. arXiv:quant-ph/9605043.

[14] Paul R. Halmos. *Finite-dimensional vector spaces.* Undergraduate texts in mathematics. Springer-Verlag, New York, 1974. ISBN 9780387900933.

[15] Mika Hirvensalo. *Quantum computing.* Springer, Berlin; New York, 2001. ISBN 9783540407041.

[16] Viktória Nemkin. Simulation of quantum walks on a classical computer, 2021. URL `https://tdk.bme.hu/VIK/ViewPaper/Kvantumsetak-szimulacioja-klasszikus`.

[17] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information.* Cambridge University Press, Cambridge ; New York, 2000. ISBN 9780521632355 9780521635035.

[18] Renato Portugal. *Quantum Walks and Search Algorithms.* Quantum Science and Technology. Springer, Berlin; New York, 2018. ISBN 9783319978130.

[19] Miklos Santha. Quantum walk based search algorithms. In Manindra Agrawal, Dingzhu Du, Zhenhua Duan, and Angsheng Li, editors, *Theory and Applications of Models of Computation*, pages 31–46, Berlin, Heidelberg, 2008. Springer. ISBN 9783540792284.