KØBENHAVNS UNIVERSITET

Implementing Quantum Algorithms in SML

Caroline Kierkegaard and Mikkel Willén
Supervisor: Martin Elsman

January 30, 2025

```
--H---X-*-X-H-X-*-X-H-X-*-X-H-X-*-X-H-X-*-X-H-X-*-X-H--
        |       |       |       |       |       |
--H-----*---H-X-*-X-H---*---H-X-*-X-H---*---H-X-*-X-H--
        |       |       |       |       |       |
--H---X-*-X-H-X-*-X-H-X-*-X-H-X-*-X-H-X-*-X-H-X-*-X-H--
        |       |       |       |       |       |
--H-----*---H-X-*-X-H---*---H-X-*-X-H---*---H-X-*-X-H--
        |       |       |       |       |       |
--H---X-*-X-H-X-*-X-H-X-*-X-H-X-*-X-H-X-*-X-H-X-*-X-H--
        |       |       |       |       |       |
--H-Z---X-------X-------X-------X-------X-------X------
```

Abstract

Quantum computing promises significant advancements in computational power, enabling solutions to problems that are intractable for classical systems. Among the foundational quantum algorithms, Grover's search algorithm stands out for its ability to provide a quadratic speedup in searching unstructured databases. In this work, we implemented and simulated Grover's algorithm within the Standard ML (SML) framework, extending its capabilities to evaluate quantum algorithms on classical systems. Our implementation included the design and integration of key components such as the oracle and diffusion operators. The framework's accuracy was validated against the Quirk quantum simulator, showing consistent results. Experimental evaluation demonstrated the scalability of the implementation up to 16 qubits, highlighting both the promise and the limitations of classical simulation approaches. This project establishes a foundation for future extensions of the SML framework to support a broader range of quantum algorithms and provides insights into the challenges of hybrid quantum-classical computing.

Contents

1 Introduction

The rapid advancement of computer science has led to the emergence of new computational paradigms, transitioning from classical computation to quantum computation. In classical computing, information is stored and processed using bits, which can exist in one of two states: 0 or 1. Quantum computation, however, leverages principles of quantum mechanics to perform operations on data, offering new possibilities for solving problems that are intractable for classical computers.

As quantum computing continues to establish itself as a significant field in computer science, considerable efforts have been devoted to the development of quantum algorithms that demonstrate advantages over their classical counterparts. Among these, Grover's algorithm stands out as a key milestone. Proposed by Lov K. Grover [1], the algorithm provides a quadratic speed-up for searching an unstructured database of N elements. While classical search algorithms require $O(N)$ steps in the worst case to find a target element, Grover's algorithm accomplishes this task in $O(\sqrt{N})$ steps, making it significantly faster for large datasets.

In recent years, programming frameworks have emerged that allow for the simulation and analysis of quantum algorithms using classical computational resources. In this work, we would like examine the possibilities of implementing quantum algorithms in such a framework. Concretely, we present an implementation and simulation of Grover's quantum search algorithm in a Standard ML framework [3]. We aim to extend the framework, in order to see if it is possible to simulate and evaluate quantum algorithms efficiently by simulating Grover's algorithm.

This paper is organised as follows: Section 2 provides a brief overview of Grover's algorithm and its theoretical foundation. Section 3 discusses the implementation details using the Standard ML framework. Section 4 presents the evaluation of the results. Finally, Section 5 concludes with a summary and potential directions for future work.

2 Background

2.1 Quantum States

A quantum state provides a complete description of a quantum system. Unlike classical systems, where the state of an object is precisely defined, quantum states are expressed using probability amplitudes. These amplitudes encode the likelihood of different outcomes when a measurement is made.

In mathematical terms, the state of a quantum system is represented by a vector in a complex vector space called the Hilbert space. For a single qubit, the quantum state is a linear combination of the two basis states $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where α and β are complex numbers satisfying $|\alpha|^2 + |\beta|^2 = 1$. Here, $|\alpha|^2$ and $|\beta|^2$ represent the probabilities of measuring the system in states $|0\rangle$ and $|1\rangle$, respectively.

The quantum state evolves deterministically according to the Schrödinger equation, but the act of measurement collapses the state into one of the possible basis states, a phenomenon unique to quantum systems.

2.2 Superposition

Superposition is a principle that allows a quantum system to exist simultaneously in multiple states. For a single qubit, the state $|\psi = \alpha|0\rangle + \beta|1\rangle$ indicates that the qubit is in a combination of the states $|0\rangle$ and $|1\rangle$, with probabilities determined by $|\alpha|^2$ and $|\beta|^2$.

The key feature of superposition is that it allows quantum systems to encode and process information in parallel. For example, and n-qubit system can represent a superposition of 2^n basis states:

$$|\psi\rangle = \sum_{x=0}^{2^n-1} c_x|x\rangle$$

where $|x\rangle$ represents a computational basis state, and c_x are complex coefficients. This capability enables quantum computers to evaluate multiple solutions simultaneously, providing a foundation for their computational advantage.

2.3 Quantum Entanglement

Entanglement is a phenomenon where the quantum states of two or more particles become correlated in such a way that the state of one particle cannot be described independently of the others, regardless of the distance between them. When particles are entangled, their measurement outcomes are linked even if they are spatially separated.

In quantum computing, entanglement is critical for implementing several key mechanisms. First, it enables quantum parallelism, where the states of multiple qubits are interdependent, allowing operations on one qubit to influence others in a coordinated way. This property is leveraged in quantum algorithms to process and propagate information efficiently across a quantum system.

Entanglement also plays a fundamental role in quantum error correction. Error correction protocols, such as the Shor codes, use entangled states to detect and correct errors in qubits without directly measuring and collapsing their states. This is essential for building fault-tolerant quantum computers, as it allows the system to preserve coherence over extended computations.

2.4 Quantum Gates

Quantum gates[2] are the building blocks of quantum circuits, analogous to classical logic gates in digital circuits. They operate on qubits and are represented as unitary transformations, meaning they are reversible and preserve the total probability of quantum states. These gates manipulate quantum states through operations such as rotation, superposition, and entanglement, enabling quantum algorithms to achieve computational advantages over classical counterparts.

The X-gate, also know as the quantum NOT gate, flips the state of a qubit. It swaps $|0\rangle$ and $|1\rangle$, analogous to the classical logical NOT operation. The X-gate is mathematically represented as:

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

When applied to a qubit, its effect is:

$$X|0\rangle = |1\rangle, \quad X|1\rangle = |0\rangle$$

The Z-gate performs a phase flip operation. It leaves $|0\rangle$ unchanged while flipping the phase of $|1\rangle$.

Mathematically, the Z-gate is represented as:

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

When applied to a qubit, it has the following effect:

$$Z|0\rangle = |0\rangle, \quad Z|1\rangle = -|1\rangle$$

The Hadamard gate is a cornerstone of quantum computing as it creates and manipulates superposition. It transforms a single qubit into an equal superposition of $|0\rangle$ and $|1\rangle$, distributing its amplitude evenly across both states. The H-gate is mathematically defined as:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

When applied to the computational basis states, its effects are:

$$H|0\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right), \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

The controlled-NOT gate, or CNOT, is a two-qubit gate where the first qubit acts as the control, and the second qubit is the target. The target qubit is flipped (X-operation) if the control qubit is in the state $|1\rangle$. Mathematically, the CNOT gate is represented in the basis $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ as:

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

The action of the CNOT gate on the states is as follows:

$$\text{CNOT}(|0\rangle \otimes |x\rangle) = |0\rangle \otimes |x\rangle$$

$$\text{CNOT}(|1\rangle \otimes |x\rangle) = |1\rangle \otimes X|x\rangle$$

where $x \in \{0, 1\}$.

In quantum computing, a multiple-controlled NOT gate is an extension of the standard Controlled-NOT gate (or C^nX gate), where a target qubit is flipped if and only if all control qubits are in the $|1\rangle$ state. This gate generalises the concept of conditional operations, enabling conditional logic with more than one control qubit.

For a system with $n + 1$ qubits, where n are control qubits and 1 is the target qubit, the multiple-controlled NOT gate flips the target qubit $|t\rangle$ when the control qubits are all in the $|1\rangle$ state. If any control qubit is in $|0\rangle$, the target qubit remains unchanged. Mathematically, the action of a C^nX gate can be writte as:

$$C^nX|c_1, c_2, ..., c_n, t\rangle = \begin{cases} |c_1, c_2, ..., c_n, t \oplus 1\rangle & \text{if } c_1 = c_2 = ... = c_n = 1 \\ |c_1, c_2, ..., c_n, t\rangle & \text{otherwise} \end{cases}$$

Here, $c_1, c_2, ..., c_n$ are control qubits, t is the target qubit, and $\oplus$ denotes the XOR operation.

The matrix representation of a C^nX gate is $2^{n+1} \times 2^{n+1}$, and is a block-diagonal matrix. In the computational basis, it has the following structure:

$$C^nX = \begin{bmatrix} I_{2^n-1} & 0 \\ 0 & X \end{bmatrix}$$

where I_{2^n-1} is the identity matrix for the first 2^{n-1} rows, and X is the single-qubit NOT operation applied to the target when all control qubits are $|1\rangle$.

2.5 Grover's Algorithm

Grover's Algorithm [1] operates on a quantum system consisting of n qubits, where $N = 2^n$ represents the number of possible states. Initially, the quantum system is prepared in the equal superposition state:

$$|\psi_0\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$$

This state ensures that all N possibilities are equally probable, forming the basis for quantum parallelism.

The algorithm iteratively amplifies the amplitude of the target state using two key operations, the oracle and the diffusion operator. The oracle, denoted U_f, marks the target state $|x_t\rangle$ by flipping its amplitude. It is expressed the following way:

$$U_f|x\rangle = \begin{cases} -|x\rangle & \text{if } x = x_t \\ |x\rangle & \text{otherwise} \end{cases}$$

This operation leaves the amplitudes of all other states unchanged, but introduces a phase inversion for the target state.

The diffusion operator, denoted U_s, amplifies the probability amplitude of the marked state. This operator is defined as:

$$U_s = 2|\psi_0\rangle\langle\psi_0| - I$$

where $|\psi_0\rangle$ is the initial superposition state and I is the identity matrix. The diffusion operator increases the amplitude of the marked state by reflecting the state vector around the average amplitude of all states.

These two operations are applied iteratively, with the number of iterations T given by:

$$T = \left\lfloor \frac{\pi}{4}\sqrt{N} \right\rfloor$$

Each iteration enhances the probability amplitude of the target state while reducing the amplitudes of non-target states. After T iterations, the system's quantum state is:

$$|\psi_T\rangle \approx \sin((2T+1)\theta)|x_t\rangle + \cos((2T+1)\theta)|\psi_\perp\rangle$$

where $\theta = \arcsin\left(\frac{1}{\sqrt{N}}\right)$ and $|\psi_\perp\rangle$ represensts the superposistion of all non-target states.

Finally, the system is measured, collapsing the superposition to a single state. With high probability, the measurement yields the target state $|x_t\rangle$.

2.6 Ancilla Bits in Quantum Computing

In quantum computing, ancilla bits [2], or ancilla qubits, are auxiliary qubits introduced into a quantum circuit to assist with operations, often without encoding part of the problems primary data. Ancilla qubits are important in facilitating complex computations, ensuring reversibility, and encoding intermediate results in a manner that aligns with quantum mechanical principles.

Reversibility is a fundamental requirement in quantum computing due to the unitary nature of quantum gates. Consider a computation where intermediate results are calculated. Without ancilla qubits, overwriting quantum states would result in the loss of prior information, violating reversibility. Ancilla qubits provide a buffer, allowing intermediate results to be stored and reused while preserving the reversibility of the overall computation.

In the context of Grover's Algorithm, ancilla qubits are often employed during the implementation of the oracle or other transformations. For example, consider an oracle designed to mark the target state by flipping its amplitude. The marking process may require additional qubits to encode the result of a comparison operation without directly altering the primary computational qubits.

Mathematically, if the oracle function $f(x)$ evaluates to 1 for the target state x_t and 0 otherwise, the oracle operation can be expressed as:

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

where $|a\rangle$ represents an ancilla qubit, initialised to $|0\rangle$, and $\oplus$ denotes a bitwise XOR. The ancilla qubit $|a\rangle$ stores the result of $f(x)$ temporarily during the computation. If $f(x) = 1$, the state of the ancilla qubits is flipped to $|1\rangle$; otherwise it remains unchanged. This allows the oracle to perform its operations without disturbing the quantum state of the main computation.

3 Design and implementation

We have implemented Grover's search algorithm in the SML framework [3]. The important building blocks of our implementation is the Oracle operation, the diffusion operation and the repetition of these. In this section, we will describe the design choices made, when implementing the algorithm. Our full implementation can be found on this github repository [4].

3.1 Design of the Oracle Operation

Algorithm 1: Oracle Function

```
Input: A target state, n qubits
# Flip target bits to map the target to $|1...1\rangle$
Combine I for 0's and X for 1's with oo
Apply a controlled not on the ancilla, where all the other bits are controls
Reverse the bit flip
```

The oracle begins by transforming the target state $|t\rangle$ into $|1...1\rangle$ using X gates, flipping any qubits set to 0 in the binary representation of the target. A controlled X gate is then applied to introduce a phase inversion for $|1...1\rangle$, effectively marking the target state. Finally, the X gates are reapplied to restore the state to its original configuration

This approach ensures that the oracle's action is confined to the target state, leaving all other states unchanged. It provides a general, reusable structure that can be adapted for different target states and qubit counts, seamlessly integrating into the broader Grover's algorithm framework.

To flip the bits of the target state to $|1...1\rangle$, the following function is used:

```
fun flipBits n numQubits : t =
    if n > (powInt 2 numQubits) - 1 then
        raise Fail ("Invalid target state: " ^ Int.toString n)
    else if n = 0 then
        if numQubits > 1 then
            flipBits 0 (numQubits - 1) ** X
        else
            X
    else if n = 1 then
        if numQubits > 1 then
            flipBits 0 (numQubits - 1) ** I
        else
            I
    else
        if n mod 2 = 0 then
            flipBits (n div 2) (numQubits - 1) ** X
        else
            flipBits (n div 2) (numQubits - 1) ** I
```

The controlled phase inversion is implemented using a controlled-not gate cascading over all qubits, targeting the ancilla qubit:

```
fun cxNot 1 = X
  | cxNot n = C (cxNot (n - 1))
```

The oracle function combines these components as follows:

```
fun oracleNaive n numQubits : t =
    let
        val flip = flipBits n (numQubits - 1) ** I
    in
        flip oo cxNot numQubits oo flip
    end
```

This implementation first flips the bits of the target state to transform it into $|1...1\rangle$. A controlled-not gate is then applied with all the flipped qubits as controls and the ancilla as the target, introductin the phase inversion. Finally, the bit flips are reversed to restore the qubits to their original configuration. This modular approach allows the oracle to adapt to any target state or qubit count.

3.2 Design of the Diffusion Operator

Algorithm 2: Diffusion Operator

```
Input: n qubits, an ancilla qubit
# Apply Hadamard gates to all qubits to create a superposition
Flip all qubits using X gates
Apply a controlled Z gate on the ancilla, controlled by all other qubits
Reverse the flips with X gates
Reverse the initial superposition by reapplying Hadamard gates
```

The diffusion function amplifies the probability amplitude of the target state by reflecting the quantum state about the average amplitude of all states. It uses a combination of quantum gates to enforce quantum interference and adjust amplitudes appropriately.

The function begins by applying Hadamard gates to all qubits, transforming the quantum state into a superposition where each basis state has equal amplitude. This prepares the system for the inversion operation. The X gates are then applied to flip all qubits, transforming $|0\rangle$ into $|1\rangle$ and vice versa. This step is followed a controlled X gate on the ancilla qubit, that inverts the phase of the $|1...1\rangle$ state, marking it for amplification.

After the phase inversion, the X gates are reapplied to revert the qubits to their original configuration. Finally, Hadamard gates are applied again to complete the inversion about the mean, restoring the system to a state where the target's amplitude is significantly amplified.

The Hadamard gates are implemented recursively:

```
fun hadamardI 1 = I
  | hadamardI n = H ** hadamardI (n - 1)
```

The X gates, used for flipping all qubits to prepare them for the phase inversion, are similarly defined:

```
fun notI 1 = I
  | notI n = X ** notI (n - 1)
```

The controlled X gate is also constructed using a recursive implementation:

```
fun cxNot 1 = X
  | cxNot n = C (cxNot (n - 1))
```

The diffusion function combines these components in a sequential manner:

```
fun diffusionNaive n numQubits : t =
    hadamardI numQubits oo
    notI numQubits oo
    cxNot numQubits oo
    notI numQubits oo
    hadamardI numQubits
```

This implementation reflects the quantum state about the mean amplitude by alternating between transformations in and out of the computational basis, ensuring the target state's probability amplitude is amplified. The function is modular, scalable, and integrates seamlessly into the broader structure of Grover's algorithm.

3.3 Implemetation of Grover's Algorithm

Grover's algorithm combines initialisation, iterative amplification, and measurement to search for a marked item in an unstructured database. The algorithm operates by repeatedly applying the oracle and diffusion functions to amplify the probability amplitude of the target state.

The algorithm starts by initialising the quantum state to $|0\rangle^{\otimes n+1}$, where n represents the number of qubits for the search, and an addional qubit is used as an ancilla:

```
fun initKets n m : ket =
    let
        val numQubits = n + m
    in
        ket (replicate(0, numQubits))
    end
```

This creates a quantum state with all qubits initialised to $|0\rangle$.

The algorithm applies Grover's iterations k times, where k is the optimal number of iterations for the given search space size:

```
val iterations = Real.floor (Math.pi / 4.0 * Math.sqrt (Real.fromInt (
   powInt 2 n)))
```

Each iteration consists of applying the oracle function followed by the diffusion function. These operations are sequentially composed:

```
fun repeatCircuit t n =
    if n < 1 then
        raise Fail "Number of iterations must be greater than 0"
    else if n = 1 then
        t
    else
        t oo repeatCircuit t (n - 1)
```

This function enables the repeated application of the oracle and diffusion operations.

The main function combines initialisation, Hadamard gates for superposition, and iterative amplification using the oracle and diffusion operators. It returns the final quantum circuit and the initialised quantum state:

```
fun groversNaive target n : t * ket =
    let
        val numQubits = n + 1
        val iterations = Real.floor (Math.pi / 4.0 * Math.sqrt (Real.
   fromInt (powInt 2 n)))
        val hadamardGates = hadamard numQubits
        val initAncilla = zAncilla numQubits
        val repetition = repeatCircuit (oracleNaive target numQubits oo
   diffusionNaive target numQubits) iterations
    in
        (hadamardGates oo initAncilla oo repetition, initKets n 1)
    end
```

The `measure_dist_ancilla` function simulates measuring a quantum state where some of the qubits are considered ancillary. It calculates the probability distribution of the remaining (non-ancillary) qubits, summing over the probabilities of all configurations of the ancillary qubits.

The function operates in two parts. First, it reduces the probability vector recursively by summing over pairs of entries corresponding to the ancillary qubits. Each recursive step effectively removes one ancillary qubit by aggregating its probabilities. Next, the reduced probability vector is formatted into a distribution of main qubit states, represented as pairs of quantum states (kets) and their corresponding probabilities.

```
fun measure_dist_ancilla_helper (d: real vector) (num_ancilla: int) : real vector =
    if num_ancilla < 0 then
        raise Fail "You cannot have a negative number of ancilla bits"
    else if num_ancilla = 0 then
        d
    else
        let
            val len = Vector.length d
            val halflen = len div 2
            val v2 = Vector.tabulate (halflen, fn i => Vector.sub(d, 2 * i) + Vector.sub(d, 2 * i + 1))
        in
            measure_dist_ancilla_helper v2 (num_ancilla - 1)
        end

fun measure_dist_ancilla (s: state) (num_ancilla: int) : dist =
    let
        val v = dist s
        val v2 = measure_dist_ancilla_helper v num_ancilla
        val len = log2 (Vector.length v2)
    in
        Vector.mapi (fn (i, p) => (toKet(len, i), p)) v2
    end
```

The function starts by computing the probability distribution of the entire quantum state using the `dist` function. This produces a vector where each element represents the probability of a specific basis state. To marginalise ancillary qubits, the `measure_dist_ancilla_helper` function recursively sums over adjacent entries in the probability vector. Each recursion corresponds to removing one ancillary qubit by aggregating probabilities.

Once the ancillary qubits are marginalised, the reduced vector is converted into a more interpretable format. The function `toKet(len, i)` transforms each index into its binary representation, corresponding to the quantum state of the main qubits. This creates a list of pairs, where each pair contains a ket and its associated probability.

The modular design ensures flexibility, allowing it to handle any number of ancillary qubits or main qubits. By summing over ancillary outcomes, this function effectively focuses the measurement results on the main qubits, ensuring that the final distribution is accurate and meaningful for analysis.

4 Evaluation and Results

We successfully extended the SML framework with the functionalities used by Grover's search algorithm. We have performed experiments to test the performance of the quantum search algorithm.

To test the range of our implementation, we have performed experiments on circuit with a range of number of qubits increasing from 1 to 16 qubits, seen in table 1.

The results of our experiments demonstrate the successful implementation and scaling behavior of Grover's algorithm within the SML framework. For smaller qubit numbers, the probability of the correct state (ProbCorrect) aligns closely with the theoretical efficiency of Grover's algorithm, reaching near-optimal levels. However, as the number of qubits increases, we observe a gradual decline in this probability, stabilizing around 50% for 16 qubits. This trend reflects the increasing complexity of the state space, which grows exponentially with the number of qubits, and the inherent limitations of Grover's algorithm.

The behavior of incorrect states further supports these observations. While the probabilities of individual incorrect states (ProbIncorrect) diminish as the qubit count increases, the cumulative probability of all incorrect states (SumProbIncorrect) grows correspondingly.

numQubits	ProbCorrect	ProbWrong	SumProbWrong
1	0.5	0.5	0.5
2	1	0	0
3	0.9453125	0.0078125	0.0546875
4	0.908447265625	0.006103515625	0.0915527344
5	0.896936535835	0.00332462787628	0.1030634642
6	0.816377019397	0.00291465048576	0.1836229806
7	0.683735462787	0.00249027194656	0.31626453721
8	0.650349994728	0.00137117649126	0.34965000527
9	0.554456476626	0.000871905133804	0.44554352337
10	0.558355923306	0.000431714639975	0.44164407669
11	0.532238224051	0.000228510882242	0.46776177594
12	0.542708431802	0.000111670712625	0.45729156819
13	0.521155601617	5.84598215581E-05	0.47884439838
14	0.519292732029	2.93418340945E-05	0.48070726797
15	0.515622768257	1.47824711369E-05	0.48437723174
16	0.507572313746	7.51396484708E-06	0.49242768625

Table 1: Testing simulation of Grover's on increasing number of qubits

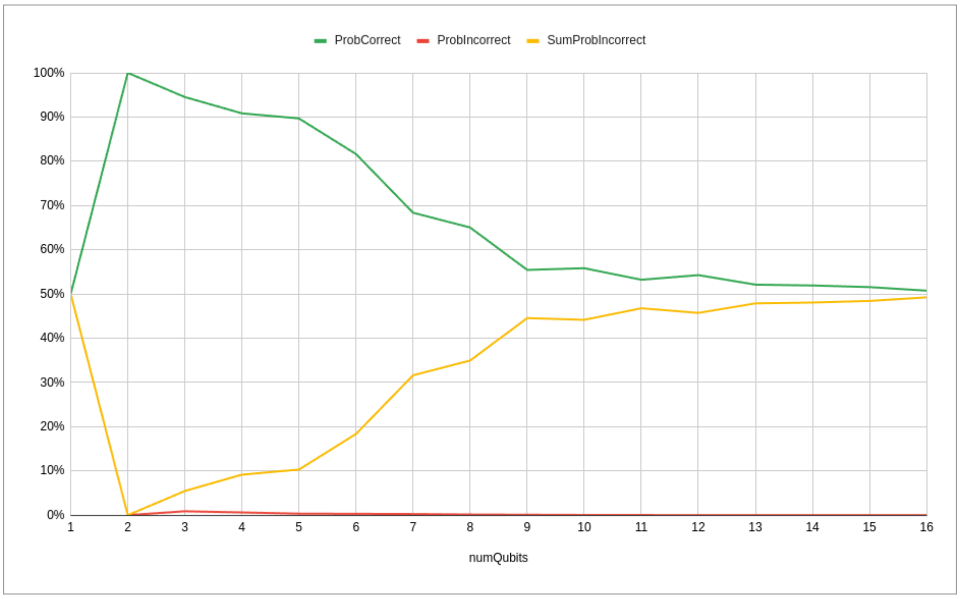

Figure 1: Graph over probabilities when simulating Grover's Algorithm

4.1 Comparison with the Simulation Tool Quirk

In order to verify the correctness of the probability distribution resulting from our simulation, we have compared the results to another simulation tool. We have used the tool called Quirk, in which one can simulate a circuit. We have simulated circuits up to 5 qubits, since bigger circuits were quite large. See references for the circuits [5] [6] [7] [8] [9].

As the results show in table 2, our SML implementation finds the correct state with the exact same probability as the Quirk tool.

numQubits	ProbCorrect(SML)	ProbCorrect(Quirk)	ProbWrong(SML)	ProbWrong(Quirk)
1	0.5	0.5	0.5	0.5
2	1	1	0	0
3	0.9453125	0.9453125	0.0078125	0.0078125
4	0.908447265625	0.908447265625	0.006103515625	0.006103515625
5	0.896936535835	0.896936535835	0.00332462787628	0.00332462787628

Table 2: Simulation results from Grover's in SML and Quirk

4.2 Comparison of `eval` and `interp`

The SML framework provides two ways of evaluating quantum circuits: the `eval` function and the `interp` function. The results show in table 3, that the two functions compute exactly the same probability distributions over the states.

numQubits	ProbCorrect(Interp)	ProbCorrect(Eval)	ProbWrong(Interp)	ProbWrong(Eval)
1	0.5	0.5	0.5	0.5
2	1	1	0	0
3	0.9453125	0.9453125	0.0078125	0.0078125
4	0.908447265625	0.908447265625	0.006103515625	0.006103515625
5	0.896936535835	0.896936535835	0.00332462787628	0.00332462787628
6	0.816377019397	0.816377019397	0.00291465048576	0.00291465048576
7	0.683735462787	0.683735462787	0.00249027194656	0.00249027194656

Table 3: Testing simulation of Grover's with the `eval` and `interp` functions

4.3 Evaluating the Extension of the Framework

In this work, we aimed to successfully extend the SML framework such that it could be used to simulate and evaluate quantum algorithms. As a concrete example of an algorithm we extended the framework with the building blocks of Grover's algorithm. As shown with the experiments above, we successfully implemented the functionalities of the algorithm and obtained correct results compared to another quantum simulation tool.

Some of the implemented functionalities are specific to Grover's algorithm. However, other functionalities are very relevant for multiple other quantum algorithms. Ideally, the framework could be extended with multiple building blocks, making it easy to put them together and simulate a variety of algorithms.

5 Future Work

This project lays the groundwork for further exploration and enhancement of quantum algorithm simulation in SML. Several avenues for future work have been identified to extend the scope and improve the utility of the framework:

- **Implementing Additional Quantum Algorithms:** Expanding the repertoire of implemented algorithms will provide a broader evaluation of the framework's flexibility and performance. Algorithms such as Shor's algorithm or the Quantum Fourier Transform can serve as benchmarks for the framework's capabilities.
- **Comparison with Other Simulations:** Conducting a detailed comparison with other simulation frameworks, such as those implemented in Quantum Computer Language (QCL) or QRISP, will help assess the accuracy, efficiency, and usability of the SML implementation.
- **Optimisation of Algorithm Circuits:** Further optimisation of the quantum circuits, particularly for Grover's algorithm, could enhance performance by reducing the classical computational overhead and improving the simulation of quantum operations.
- **Implementing an Oracle for the SAT Problem:** Developing a custom oracle function tailored to solve the SAT problem will extend the practical applicability of the framework. This implementation would demonstrate how the framework handles non-trivial oracle designs and supports problem-specific modifications.
- **Extending Support for Multiple Ancilla Bits:** While the framework has been extended to handle one ancilla bit, future work could focus on generalizing this capability to support multiple ancilla bits. This would allow for the simulation of more complex algorithms and operations that rely on additional ancillary qubits.

By pursuing these directions, the project can contribute to advancing the field of hybrid quantum-classical computing and further validate the utility of the SML-based simulation framework.

6 Conclusion

This project successfully demonstrated the implementation and simulation of Grover's algorithm within the SML framework, providing a practical approach to exploring quantum algorithms on classical computational systems. By extending the framework with functionalities essential to Grover's algorithm, such as the oracle and diffusion operators, we validated its correctness through comparisons with theoretical expectations and external simulation tools like Quirk. The results revealed high fidelity in small-scale simulations and highlighted the scaling challenges associated with increasing qubit counts.

Beyond specific insights into Grover's algorithm, this work established a foundation for further developments in quantum algorithm simulation. The modular nature of the implementation allows for potential extensions to other algorithms, enhancing the versatility and applicability of the SML framework. This project underlines the growing potential of hybrid quantum-classical computing and serves as a stepping stone for future efforts in advancing quantum algorithm research and simulation technologies.

References

[1] L. K. Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing (STOC '96)*. New York, NY, USA: ACM, 1996, pp. 212–219. DOI: 10.1145/237814.237866.

[2] C.P. Williams. "Explorations in Quantum Computing". In: Springer, London, 2011. Chap. Quantum Gates. DOI: https://doi.org/10.1007/978-1-84628-887-6_2.

[3] M. Elsman. "atpl-sml-quantum". In: *Github repository* (2025). URL: https://github.com/diku-dk/atpl-sml-quantum.

[4] C. Kierkegaard and M. Willén. "atpl2024". In: *Github repository* (2025). URL: https://github.com/mikkelwillen/atpl2024/.

[5] *Quirk with 1 qubit*. URL: http://bit.ly/42ctVMg.

[6] *Quirk with 2 qubits*. URL: https://bit.ly/3PANYMG.

[7] *Quirk with 3 qubits*. URL: https://bit.ly/4hasLVG.

[8] *Quirk with 4 qubits*. URL: https://bit.ly/4hddp2S.

[9] *Quirk with 5 qubits*. URL: https://bit.ly/3PCKNUV.