

SAT-solving using Grover’s algorithm

Group name: Quantum Satisfiers

Group members:

Axel Kanne - jqc157

Stephen Hunter Barbella - qtl108

Victor Alexander Schmidt - rqc908

Zonghao Zou -cxr577

Project advisor: Fritz Henglein

Contents

1	Introduction	2
2	Background	2
2.1	Grover’s algorithm	2
2.1.1	Oracles	3
2.2	Tensor decompositions	3
2.3	Boolean Formulas	4
2.3.1	Algebraic Normal Form (ANF)	4
3	Constructing Oracles	5
3.1	Boolean and Phase Oracles	5
3.1.1	Qubits for oracles	5
3.2	Constructing Phase Oracles from Boolean Oracles	5
3.2.1	Creating a direct translation (the boolean/bit oracle)	5
3.2.2	Creating the phase oracle	6
3.3	Constructing Phase Oracles from ANF	6
4	Evaluating Quantum Circuits with Tensor Decomposition	7
4.1	Conventional Implementation	7
4.1.1	Data type representation	7
4.1.2	Evaluation implementation	7
4.2	ANF-Centered Implementation	10
4.2.1	Multi-Control Z Gates	10
4.2.2	Only 4 Qubits	10
4.3	Theoretical Analysis	11
4.4	Testing	12
4.5	Benchmarking	12
4.5.1	Simple oracle circuits	12
4.5.2	Complex oracle circuits	13
4.5.3	Oracle circuits with control-bit overlap	14
5	Problems	15
5.1	Measurement Problem	15
5.2	ANF-SAT is already P	16
6	Future Work	16
7	Conclusion	16

Abstract

Grover’s algorithm is a quantum algorithm for unstructured search that has a runtime of $O(\sqrt{n})$. This allows solving for boolean satisfiability (SAT-problem) in $O(2^{\sqrt{n}})$. The simulation of Grover’s algorithm on classical computer is heavily explored in literature. However, they mostly consider only a black box oracle and don’t exploit the structure in the oracle which might exists for specific applications of Grover’s, such as SAT-solving.

A recent paper [4] explored a highly efficient algorithm for simulating Grover’s Algorithm on a simple oracle using tensor decomposition. We hypothesize that there is a larger class of more complex oracles which have efficient tensor decomposition simulation. Here, we considered two ways of translating boolean expressions into oracles. The first solution is to directly translate the boolean expression into a boolean oracle and then into a phase oracle. The second solution is to convert the boolean expression into algebraic normal form (ANF) and then translate that into a phase oracle. Further exploration of ANF showed that there are certain boolean expressions in ANF which can be very efficiently simulated. For simulating Grover’s algorithm with these oracles, two different quantum circuit simulators were implemented which both use tensor decomposition for representing quantum state. One of these is a conventional implementation, and the other is specialized to phase oracles generated using ANF. Benchmarking these implementation showed that the specialized implementation was faster than the conventional one, and that the effectiveness of tensor decomposition relies on the structure and complexity of the oracle. However, our algorithms still falls short of traditional SAT solving methods since the boolean formulas we found to have efficient simulation are also easily solved using traditional methods.

1 Introduction

Satisfiability (SAT) solving lies at the heart of computational complexity. It was actually the first problem to be shown to be NP complete in the Cook-Levin Theorem [9]. Classical algorithms have a worst case exponential runtime when determining if an arbitrary boolean formula is satisfiable. However, Grover’s algorithm can fit very nicely within a SAT solver. It uses a boolean formula as a kind of oracle, which is exactly the type of circuit the algorithm finds a solution for.

When using tensor decompositions [4] as a form of quantum state tracking, we can efficiently simulate non-clifford operators. This allows for faster simulation on classical computing hardware. In this project two different quantum circuit simulators were implemented which both use tensor decomposition, were one was a general one for all types of circuits, and one was specialized to be used for Grover’s algorithm.

It turns out that a phase oracle can sometimes be trivial to construct. For example, if the boolean formula used to construct the phase oracle is in ANF already as it can be trivially translated into a sequence of quantum gates. Through entanglement and interference, these gates mark the correct input bit strings. The ANF oracle does this without the use of ancilla qubits. This is because these forms of oracle inject the formula directly into the phase of the circuit, which in turn limits tensor term rank.

In this project, boolean formulae have been solved using tensor decompositions and Grover’s algorithm. This is done by creating an oracle circuit from a given boolean formula ϕ and then evaluating the oracle using Grover’s algorithm. We do this to increase the amplitudes of inputs that evaluates ϕ to true. Finally, we measured and displayed the result to the user.

2 Background

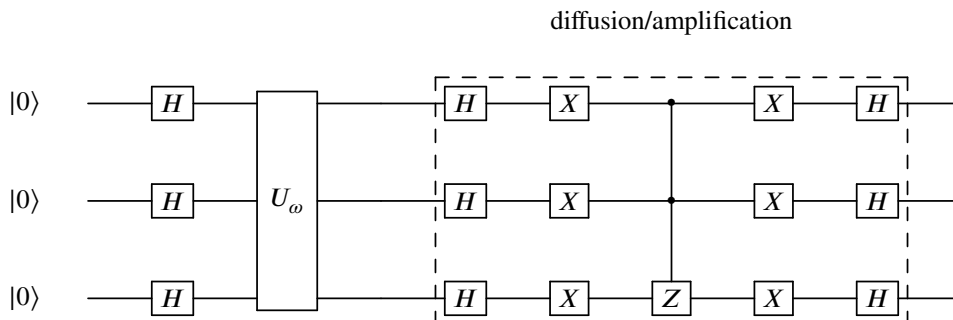
2.1 Grover’s algorithm

Grover’s algorithm is a quantum algorithm for finding (with high probability) the unique input to a black box function using just $O(\sqrt{n})$ calls, where n is the size of the domain [10]. Normally, in a non-quantum setting, this would take $O(n)$ calls to the function. Additionally, it was proven to be asymptotically optimal.

The algorithm can be broken down into three parts:

1. Initialize all qubits to the uniform superposition. $|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$
2. Perform the Grover’s iteration $r(N)$ times:
 - (a) Apply the operator U_ω
 - (b) Apply the Grover diffusion operator $U_s = 2|s\rangle\langle s| - I$
3. Measure the quantum state

Here is a 3-qubit circuit example of Grover’s [3, 4]:



The amplification part of the circuit is taken from [3, 4].

2.1.1 Oracles

An oracle function f is traditionally a black-box function, that takes some input bit string and returns `true` or `false`. The phase oracle U_ω used in Grover's algorithm has to encode all bit strings that succeed under the function $f \forall x. x \in \{0, 1\}^n, f(x_1 \dots x_n) = 1$, where n is the size of the input bit string, in the phase of the circuit; the phase oracle should flip the sign of the bit string, that returns `true` when input in f . The phase oracle U_f can be written as:

$$|x\rangle \xrightarrow{U_f} (-1)^{f(x)} |x\rangle$$

For instance, say f marks the bit string 111, so applying U_f should yield

$$\begin{aligned} |+++\rangle &= \frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle) \\ U_f |+++\rangle &= \frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle - |111\rangle) \end{aligned}$$

Later in this report, we will explore how to construct different types of phase oracles, using different gates to encode phase into a quantum circuit.

2.2 Tensor decompositions

One way of representing quantum state is using a *state vector*, which for n qubits is a vector of length 2^n of each amplitude for each computational basis state [6, p. 16]:

$$|\psi\rangle = \sum_{x \in \{0,1\}^n} \alpha_x \cdot |x\rangle \quad (1)$$

This representation is exponential in the number of qubits, and therefore using it in the simulation of quantum systems would have a high spatial complexity of $O(2^n)$. This is also the advantage of real quantum systems, where all the information of the state vector is in a way 'hidden' during computation [6, p. 16-17]. Another way of representing quantum state, is representing it as a tensor product of qubits:

$$|\psi\rangle = |q_1\rangle \otimes \dots \otimes |q_n\rangle \quad (2)$$

With this representation, each qubit would be a 2-dimensional vector, and for n qubits it would mean that the amount of values to store would be $2n$, with a spatial complexity of $O(n)$. The single tensor product of qubits is the state vector before being fully evaluated, since using the *kroenecker product* to evaluate the tensor product would result in the state vector. It would be interesting to ask whether it is possible to simulate quantum circuits by representing state using the tensor product representation rather than a state vector.

This is explored in "Functional Quantum Circuit Simulation with Tensor Term Rank Reduction"[4], where *tensor decomposition* is used to represent quantum state in simulation of quantum circuits. In this representation, quantum state is expressed as a *tensor term*, which is a sum of tensor products:

$$\sum_{i=1}^r q_1 \otimes \dots \otimes q_n \quad (3)$$

Each summand is *pure tensor*, and the number of summands is the *tensor term rank*. To evaluate a single-qubit gate on a tensor term, one can just apply the gate to the target qubit of each pure tensor (here the j th qubit):

$$G_j \left(\sum_{i=1}^r q_{i,0} \otimes \dots \otimes q_{i,n} \right) = \sum_{i=1}^r q_{i,0} \otimes \dots \otimes G(q_{i,j}) \otimes \dots \otimes q_{i,n} \quad (4)$$

For control gates however, we will have two results, where the first is if the control-qubits are all 0, in which case the gate is not applied, and one were they are all 1, and the gate is applied. Applying the control gate on a pure tensor with n qubits with *target gate* G can be written as the following sum of pure tensors:

$$C^{n-1}(G)(\otimes_{i=1}^n q_i) = \left(\prod_{j=1}^{n-1} \langle 0 | q_j \rangle \right) \cdot (|0\rangle^{\otimes(n-1)} \otimes q_n) + \left(\prod_{j=1}^{n-1} \langle 1 | q_j \rangle \right) \cdot (|1\rangle^{\otimes(n-1)} \otimes G(q_n)) \quad (5)$$

$$= \alpha \cdot (|0\rangle^{\otimes(n-1)} \otimes q_n) + \beta \cdot (|1\rangle^{\otimes(n-1)} \otimes G(q_n)) \quad (6)$$

In equation 6, α is the product of the 0-amplitudes of all the control-qubits, while β is the product of the 1-amplitudes for all the control-qubits. However we can rewrite this by adding and subtracting the tensor-product with the control-qubits set to 1 multiplied with the control-qubits' 1 amplitudes:

$$C^{n-1}(G)(\otimes_{i=1}^n q_i) = \alpha \cdot (|0\rangle^{\otimes(n-1)} \otimes q_n) + \beta \cdot (|1\rangle^{\otimes(n-1)} \otimes G(q_n)) \quad (7)$$

$$= \alpha \cdot (|0\rangle^{\otimes(n-1)} \otimes q_n) + \beta \cdot (|1\rangle^{\otimes(n-1)} \otimes G(q_n)) + \beta \cdot (|1\rangle^{\otimes(n-1)} \otimes q_n) - \beta \cdot (|1\rangle^{\otimes(n-1)} \otimes q_n) \quad (8)$$

$$= \otimes_{i=1}^n q_i + \beta \cdot (|1\rangle^{\otimes(n-1)} \otimes G(q_n)) - \beta \cdot (|1\rangle^{\otimes(n-1)} \otimes q_n) \quad (9)$$

$$= \otimes_{i=1}^n q_i + \beta \cdot (|1\rangle^{\otimes(n-1)} \otimes (G(q_n) - q_n)) \quad (10)$$

In equation 9, we use the fact that $\alpha \cdot (|0\rangle^{\otimes(n-1)} \otimes q_n) + \beta \cdot (|1\rangle^{\otimes(n-1)} \otimes q_n) = \otimes_{i=1}^n q_i$, and in equation 10 we use the equality $v \otimes w_1 + v \otimes w_2 = v \otimes (w_1 + w_2)$. The effect the control gate has on the pure tensor can be described with equation 10 as an *additive* effect [4], where we will refer to $\beta \cdot (|1\rangle^{\otimes(n-1)} \otimes (G(q_n) - q_n))$ as a *correction term*. Using this equation means that evaluating a multi-control gate on a pure tensor, is the same as keeping the original pure tensor and adding the correction term derived from the original pure tensor. We can apply the control gate to a tensor term, by applying the control gate on each pure tensor of the tensor term, increasing the number of pure tensors and thereby the tensor term rank.

The spatial complexity of the tensor decomposition representation is dependent on the tensor term rank. With a rank of r and n qubits, the spatial complexity is $O(rn)$. This can end up being worse than the state vector representation if the rank is high enough. In particular, if we have an initial state of one single pure tensor with n qubits and m control gates are applied, we could end up doubling the number of pure tensors each time a control-gate is applied, leading to a rank of 2^m . The paper ([4]) introduces *simplification rules*, which can be used to reduce the tensor term rank dynamically. First one can represent the pure tensor using a *global phase*, here represented as α :

$$\sum_{i=1}^r \alpha_i \cdot (q_{i,0} \otimes \dots \otimes q_{i,n}) \quad (11)$$

This enables simplification of two pure tensors. If they are multiples of each other, we can simplify to one pure tensor with one global phase using the equality $v_1 \otimes w + v_2 \otimes w = (v_1 + v_2) \otimes w$:

$$\alpha \cdot (\otimes_{i=1}^n q_i) + \alpha' \cdot (\otimes_{i=1}^n q'_i) = (\alpha + \alpha') \cdot (\otimes_{i=1}^n q_i) \quad \text{if } (\otimes_{i=1}^n q_i) = (\otimes_{i=1}^n q'_i) \quad (12)$$

The other simplification rules presented in the paper are about the use of control gates $C^n(G)(\otimes_{i=1}^n q_i)$, since there are cases where adding a correction term is not needed:

- The n control qubits are guaranteed to be $|0\rangle^{\otimes n}$, so the product of the 0-amplitudes of the control qubits is equal to 1. In this case the pure tensor remains the same:

$$C^{n-1}(G)(\otimes_{i=1}^n q_i) = \otimes_{i=1}^n q_i \quad \text{if } \left(\prod_{j=1}^{n-1} \langle 0|q_j \rangle \right) = 1 \quad (13)$$

- The n control qubits are guaranteed to be $|1\rangle^{\otimes n}$, so the product of the 1-amplitudes of the control qubits sum to 1. Simply apply the target gate G to the pure tensor:

$$C^{n-1}(G)(\otimes_{i=1}^n q_i) = \otimes_{i=1}^{n-1} q_i \otimes G(q_n) \quad \text{if } \left(\prod_{j=1}^{n-1} \langle 1|q_j \rangle \right) = 1 \quad (14)$$

- G does not change the target qubit, in this case the pure tensor remains the same:

$$C^{n-1}(G)(\otimes_{i=1}^n q_i) = \otimes_{i=1}^n q_i \quad \text{if } G(q_n) = q_n \quad (15)$$

2.3 Boolean Formulas

Define a boolean formula ϕ as an algebraic expression of binary variables (atoms) $p \in \{\text{true}, \text{false}\}$ and logical operators. A boolean formula is then *well-formed* in the context of this report, if it can be constructed recursively from 16.

$$\phi := p \mid \neg \phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \oplus \phi_2 \quad (16)$$

The **satisfiability problem** (SAT Problem) asks if there exists some arbitrary assignment σ of the variables p_i s in ϕ , such that ϕ evaluates to **true** when the p_i s are substituted with $\sigma(p_i)$ in ϕ . The SAT problem is known to be NP-complete; no algorithm consistently solves the SAT in polynomial time - the problem is of exponential complexity $O(2^n)$ [9].

2.3.1 Algebraic Normal Form (ANF)

Definition 1. *Algebraic Normal Form is the representation of boolean formulae, where conjunctive atomic statements $p_1 \wedge p_2 \wedge \dots$ appear between the exclusive disjunctive logical operator. Formally, define ANF as a formula ϕ that can be constructed recursively from 17.*

$$\begin{aligned} \phi &:= \psi \mid \phi_1 \oplus \phi_2 \\ \psi &:= p \mid \text{true} \mid \psi_1 \wedge \psi_2 \end{aligned} \quad (17)$$

ANF is a canonical form of boolean expression, meaning two logically equivalent formulas will convert to the same ANF [8]. To convert a boolean formula to ANF, use the rewrite rules 18 to eliminate $\neg$ and $\wedge$. Note that $\llbracket \cdot \rrbracket$ denotes the transformation $\llbracket \phi \rrbracket \rightsquigarrow \phi_{\text{ANF}}$, for any well-formed boolean formula ϕ .

$$\begin{aligned} \llbracket p \rrbracket &\rightsquigarrow p \\ \llbracket \phi_1 \wedge \phi_2 \rrbracket &\rightsquigarrow \llbracket \phi_1 \rrbracket \wedge \llbracket \phi_2 \rrbracket \\ \llbracket \phi_1 \oplus \phi_2 \rrbracket &\rightsquigarrow \llbracket \phi_1 \rrbracket \oplus \llbracket \phi_2 \rrbracket \\ \llbracket \neg \phi \rrbracket &\rightsquigarrow \llbracket \phi \rrbracket \oplus \text{true} \\ \llbracket \phi_1 \vee \phi_2 \rrbracket &\rightsquigarrow \llbracket \phi_1 \rrbracket \oplus \llbracket \phi_2 \rrbracket \oplus (\llbracket \phi_1 \rrbracket \wedge \llbracket \phi_2 \rrbracket) \end{aligned} \quad (18)$$

The resulting formula derived from applying the rewrite rules, contains only atoms (p), constants (true), exclusive disjunctions ($\oplus$), and conjunctions ($\wedge$). However, these rules are not sufficient for ANF conversion, as exclusive disjunctions may be nested inside of conjunctions. The equivalences 19 are supplied for distributing conjunctions across exclusive disjunctions, and to eliminate (most) constants.

$$\begin{aligned}
\phi_1 \wedge (\phi_2 \oplus \phi_3) &= (\phi_1 \wedge \phi_2) \oplus (\phi_1 \wedge \phi_3) \\
(\phi_1 \oplus \phi_2) \wedge \phi_3 &= (\phi_1 \wedge \phi_3) \oplus (\phi_2 \wedge \phi_3) \\
\text{true} \wedge \phi &= \phi \\
\phi \wedge \text{true} &= \phi \\
\phi \oplus \phi &= \text{false} \\
\text{false} \oplus \phi &= \phi
\end{aligned} \tag{19}$$

Which is sufficient to convert boolean formulae to ANF, with $O(2^n)$ complexity.

3 Constructing Oracles

Given a well-formed boolean formula ϕ , how can an oracle circuit U_ϕ be created, such that U_ϕ marks exactly the solutions to ϕ ? Without knowing the solutions of ϕ itself, this is a non-trivial task.

3.1 Boolean and Phase Oracles

3.1.1 Qubits for oracles

First and foremost, we need to define what qubits represent. The qubits that we apply the oracle (and the rest of Grover's algorithm) on needs to correlate with the boolean formula in some sense - otherwise, the whole simulation makes no sense at all. Consider a boolean formula ϕ containing unique variables $p_1, \dots, p_n$. For at direct $\phi \rightarrow U_\phi$ translation, assign a designated qubit $q_1, \dots, q_n$ to each atom in an a quantum circuit:

$$\begin{array}{ccc}
q_1 & \text{---} & \dots \\
\vdots & \text{---} & \dots \\
q_n & \text{---} & \dots
\end{array}$$

To keep track of which $p_i \mapsto q_i$, we simply refer to all p_i as their indexed q_i as their designated reference qubit.

Now, a quantum oracle can be generated with size directly correlating to the amount of unique variables in ϕ . All that remains is to convert the logical operators to gates, such that the phase (sign) of all valid assignments is flipped.

3.2 Constructing Phase Oracles from Boolean Oracles

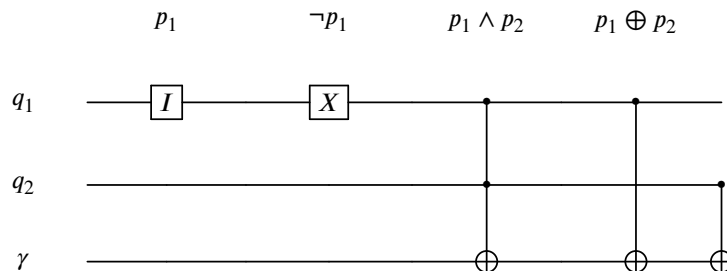
3.2.1 Creating a direct translation (the boolean/bit oracle)

Again, refer to ϕ as any well-formed boolean formula. We want to simulate each operator $\wedge, \vee, \oplus, \neg$ as a quantum gate. Going through each form of boolean expression 16, we want show an equivalent quantum gate operation.

First, define $\llbracket \cdot \rrbracket : \phi \rightarrow \text{program} \times \text{qubit}$. program is the resulting oracle: a sequence of in other words a sequence of Pauli gate with zero or more control qubits. And qubit is for storing the result, with $|1\rangle$ corresponding to the correct solution and $|0\rangle$ to an incorrect one.

$$\begin{aligned}
\llbracket p_i \rrbracket &= (I, q_i) \\
\llbracket \neg \phi \rrbracket &= \text{let } \llbracket \phi \rrbracket = (G, q) \text{ in } (X_{q_i} G, q) \\
\llbracket \phi_1 \wedge \phi_2 \rrbracket &= \text{let } \llbracket \phi_1 \rrbracket = (G_1, q_1), \llbracket \phi_2 \rrbracket = (G_2, q_2) \text{ in } (C_{q_1} C_{q_2} X_\gamma G_1 G_2, \gamma) \\
\llbracket \phi_1 \vee \phi_2 \rrbracket &= \llbracket \neg(\neg \phi_1 \wedge \neg \phi_2) \rrbracket \\
\llbracket \phi_1 \oplus \phi_2 \rrbracket &= \text{let } \llbracket \phi_1 \rrbracket = (G_1, q_1), \llbracket \phi_2 \rrbracket = (G_2, q_2) \text{ in } (C_{q_1} X_\gamma C_{q_2} X_\gamma G_1 G_2, \gamma)
\end{aligned}$$

Where γ represents a *fresh* ancilla qubit, that stores the intermediate result of a logical operator that acts on two subformulae. *Fresh* in this context, means that a new ancilla is needed for each binary logical operator in ϕ . For a better visual representation, a quantum circuit is as follows:



To argue why these representations are correct:

- Any atom p_i simply refers to its qubit q_i for any higher logical operation to reference. In of itself, simply referring to a specific atom has no semantical meaning in the context of a boolean formula, besides simply stating its existence.
- $\neg p_i$ creates an X_{q_i} gate, as $\neg$ and X have comparable logic, as the former flips a bit and the latter flips a qubit.
- $p_1 \wedge p_2$ becomes $C_{q_1} C_{q_2} X_\gamma$. This gate semantically says, flip γ if q_1 and q_2 is 1.
- $p_1 \oplus p_2$ becomes $C_{q_1} X_\gamma C_{q_2} X_\gamma$. This sequence of gates first flips γ if $q_1 = 1$, then flips γ again if $q_2 = 1$. This mimics $\oplus$, as if $q_1 = 1$ and $q_2 = 1$, then γ is flipped twice, causing no change, which is equivalent to $q_1 = 0$ and $q_2 = 0$. Only if one of q_1 and q_2 are equal to 1 is γ flipped.

3.2.2 Creating the phase oracle

We now have a way to translate a boolean formula ϕ directly into a circuit. However there is a problem: the phase oracle needs to mark the *phase* of solution bit-strings. Our direct translation effect, if run on a circuit, leaves qubits flipped and marks no phase at all.

To fix this, say we have created the direct translation $U_d \equiv \llbracket \phi \rrbracket$ that induces no phase change and flips the ancilla qubits. To fix the latter, simply reverse U_d and re-apply it to the circuit - as quantum gates are unitary, this will undo the flipping. This is known as uncomputation. However, before uncomputing, do phase-kickback on the result qubit a to mark the solution in the phase. This makes the sequence of gates the following:

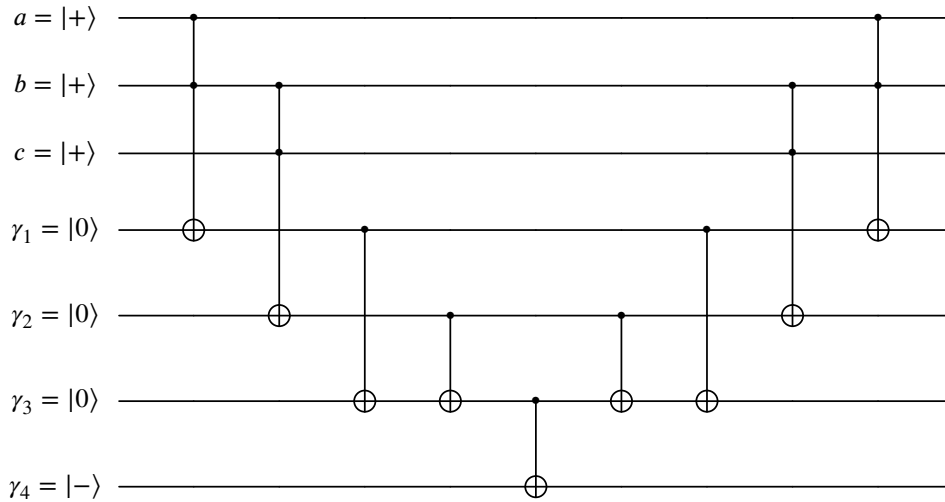
$$U_\phi = U_d^\dagger C_\phi X_a U_d \quad (20)$$

However, to do phase-kickback, ancilla a needs to be initialized to $|1\rangle$, such that $H|1\rangle = |-\rangle$ and the $C_\phi X_a$'s effect on a is a phase flip, since $X|-\rangle = -|-\rangle$. What this means for Grover's is that we mark the phase in a qubit that is unrelated to the formula ϕ .

Let us walk through an example:

$$\begin{aligned} \phi &= (a \wedge b) \oplus (b \wedge c) \\ U_d &= C_{\gamma_2} X_{\gamma_3} C_{\gamma_1} X_{\gamma_3} C_b C_c X_{\gamma_2} C_a C_b X_{\gamma_1} \\ U_\phi &= U_d^\dagger C_{\gamma_3} X_{\gamma_4} U_d \\ |+_a +_b +_c 0_{\gamma_1} 0_{\gamma_2} 0_{\gamma_3} -_{\gamma_4}\rangle &= |++000-\rangle \\ U_d |+_a +_b +_c 0_{\gamma_1} 0_{\gamma_2} 0_{\gamma_3} -_{\gamma_4}\rangle &= (|00+000\rangle + |10+000\rangle + |010000\rangle + |011011\rangle + |110101\rangle + |111110\rangle) \otimes |-\rangle \\ C_{\gamma_3} X_{\gamma_4} U_d |+_a +_b +_c 0_{\gamma_1} 0_{\gamma_2} 0_{\gamma_3} -_{\gamma_4}\rangle &= (|00+000\rangle + |10+000\rangle + |010000\rangle - |011011\rangle - |110101\rangle + |111110\rangle) \otimes |-\rangle \\ U_d^\dagger C_{\gamma_3} X_{\gamma_4} U_d |+_a +_b +_c 0_{\gamma_1} 0_{\gamma_2} 0_{\gamma_3} -_{\gamma_4}\rangle &= (|00+\rangle + |10+\rangle + |010\rangle - |011\rangle - |110\rangle + |111\rangle) \otimes |000-\rangle \end{aligned}$$

Coefficients have been omitted for brevity. Observe that the intermediate results stored in $\gamma_{1,2,3}$ have been uncomputed, but the phase in solutions $\{a \mapsto \text{true}, b \mapsto \text{true}, c \mapsto \text{false}\}$ and $\{a \mapsto \text{false}, b \mapsto \text{true}, c \mapsto \text{true}\}$ have been correctly marked with a phase flip. The circuit for U_ϕ looks like this:



3.3 Constructing Phase Oracles from ANF

It turns out sometimes there is a much simpler way of creating phase oracles. Assume we have a boolean formula on ANF 17, then creating the oracle is trivial. First, define $\llbracket \cdot \rrbracket : \text{ANF} \rightarrow \text{program}$, where a program is a sequence of (multi-controlled) Pauli gates, as seen previously.

$$\begin{aligned} \llbracket p_i \rrbracket &= Z_{q_i} \\ \llbracket p_i \wedge \dots \wedge p_n \rrbracket &= C^{\otimes n-1} Z_n \\ \llbracket \phi_1 \oplus \phi_2 \rrbracket &= \llbracket \phi_1 \rrbracket \circ \llbracket \phi_2 \rrbracket, \text{ where } \circ \text{ denotes the sequencing operation.} \end{aligned} \quad (21)$$

Each term of the ANF are independently true or false. To translate into a quantum circuit, use the $C^{\otimes n} Z$ gate; as the gate flips the polarity of the state, where all the control qubits and the Z 'ed qubit are 1, this effectively marks

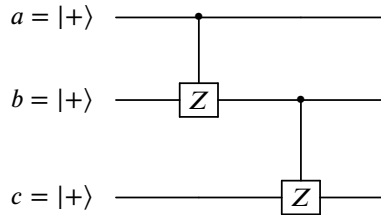
the state where *all* the qubits are 1: And as discussed earlier, our original set of qubits represent the variables themselves, a $C^{\otimes n}Z$ gate then simply marks the solution, where a term is true.

But this only holds for a single conjunctive formula in the ANF, right? Well, it turns out that repeatedly applying CZ gates flips the flipped states, such that e.g. $x_1 \oplus x_1 \wedge x_2$ flips qubit 1 and then entangles and flips qubit 1 and 2. Using tensor decompositions, tensor terms, and tensor sums, the example can be written out like this:

$$\begin{aligned}
C_0 Z_1 Z_0 |++\rangle &= C_0 Z_1 |--\rangle \\
&= |--\rangle - 2\left(-\frac{1}{2}\right) |11\rangle \text{ (uses 1)} \\
&\text{(expanding to statevector to show marked state)} \\
&= \frac{1}{\sqrt{2^2}}(|00\rangle + |01\rangle - |10\rangle - |11\rangle) + |11\rangle \\
&= \frac{1}{2}(|00\rangle + |01\rangle - |10\rangle + |11\rangle)
\end{aligned}$$

Observe carefully, that applying the CZ from the $x_1 \wedge x_2$ term of $x_1 \oplus x_1 \wedge x_2$ unmarks the term $|11\rangle$, which was also not a solution (this is what we want)!

Here is another simple representation of $\phi_{ANF} \rightarrow \text{program}$ for boolean formula $(a \wedge b) \oplus (b \wedge c)$:



Notice that this type of phase oracle uses no ancillae, but simply encodes the phase directly into the qubits. This is advantageous, as this limits the growth of the tensor term rank when applying non-clifford gates (CZ 's in our case).

4 Evaluating Quantum Circuits with Tensor Decomposition

This section will cover the implementation and evaluation (benchmarking and testing) of the implemented quantum circuit evaluators using tensor decomposition.

4.1 Conventional Implementation

In this section the conventional implementation of quantum simulation using tensor decomposition will be explained, which will be called the *general evaluator*. The general evaluator can evaluate any circuit using tensor decomposition with rank reduction, and uses the concepts explained in section 2.2.

4.1.1 Data type representation

The `Gate` type is split up in two categories: single gates `Sing` and control gates `Ctrl`. The gate types contains an `Op` type, which is the type of single-qubit gate to apply, which is either an identity, hardamard or Pauli gate. Single gates besides a single-qubit gate contains a set of integers, denoting indices for the qubits the single-qubit gate should be applied on. Control gates also have a set of integers denoting indices control qubits, and additionally it has an integer denoting the target qubit. The `Circuit` type is then just a list of gates, which is meant to be applied from left to right. All these types are shown in figure 1a. This way of representing gates makes it easy to find the target and control qubits to operate on, and makes it possible to keep the order of qubits the same throughout the simulation without the need for permuting the qubits. Additionally it simplifies application of gates, since only one type of gate is applied at a time.

A pure tensor has the type of `PureTensorMV`, which is a record type with the field `scalarMV`, the global phase of the pure tensor, and the field `qbsMV`, a mutable vector (as in data structure) with elements of type `Qubit`, which are 2-D math vectors in $\mathbb{C}^2$. A tensor term, of type `TensorMV` is then a list of pure tensors. These types are seen in figure 1b. By representing tensor terms as lists, one can apply a gate on a tensor term by mapping over the pure tensors in the list, and then collect the potential correction terms and append to the original list (after simplification). However the reason for using mutable vectors for representing the pure tensors, is to allow for read/write in constant time $O(1)$, which means that the running time of applying a gate to a pure tensor is proportional to the number of target and control qubits in the gate, which improves performance in case not all qubits are used as either control or target qubits in the gate. However using mutable vectors in haskell, which has destructive updates, requires some additional types to make it possible, which is why the type declarations of `PureTensorMV` and `TensorMV` are a bit more complex.

4.1.2 Evaluation implementation

The evaluator is defined by the following 5 functions:

- `applyGate`: This takes a gate and a pure tensor, and applies the gate on it, using relevant simplification rules if it is a control gate.

```

data Op
  = I | X | Y | Z | H
  deriving (Show, Eq)

data Gate =
  Sing Op (Set Int)
  | Ctrl Op (Set Int) Int
  deriving Show

type Circuit = [Gate]

newtype Qubit = Qubit (Vector C)

data PureTensorMV s = PT {
  scalarMV :: C,
  qbsMV :: V.MVector (PrimState (ST s)) Qubit
}

type TensorMV s = [PureTensorMV s]

```

(a) Haskell types for gates and circuits.

(b) Types for qubits, pure tensors and tensor terms.

Figure 1: Essential types used for the conventional evaluator

- `simpPureTensors`: This takes two pure tensors and checks if they can be combined into one, by checking if one is a multiple of the other.
- `simpTensor`: This takes a tensor term, and uses `simpPureTensors` as many times as possible.
- `evalCircuit`: This takes a circuit and a tensor term, and recursively applies the first gate of the circuit to the tensor term, until there are no more gates.
- `eval`: Takes a circuit and list of qubits, turns the list qubits into a pure tensor, and applies the circuit on it using `evalCircuit`.

The most interesting functions are the `simpPureTensors` and `applyGate` functions, which is where the simplification rules described in section 2.2 are used.

The code for the `applyGate` function is seen in figure 2. This function will be evaluated inside the `ST` monad, to enable modifying the mutable vectors of pure tensors, and will return a `Maybe` type, which will return `Nothing` if no correction term was generated, or a `Just` value containing a correction term.

In line 4-6 of figure 2 we handle `Sing` gates, which has a single-qubit gate `op` to apply to the qubits indices in `target_set`. This is done by modifying the qubits of the pure tensors at indices in the `target_set`, using `evalSingle` `op` to apply the type of single-qubit gate `op` on each qubit, which is done using matrix multiplication with the matrix representation of the gate on the qubit vector.

From line 7-32 we handle `Ctrl` gates, that again will have a single-qubit target gate `op`, `control_set` with the set of indices of the control bits, and a the target qubit `target`. From line 8-10, we retrieve the value of the target qubit into `tqb`, a list of values for the control bits into `ctrlL` and `res` has the result of evaluating the target gate on the target qubit.

In the condition on line 11-16, we check if any of the simplification rules shown in equations 13 and 15 can be used, which will result in no modification of the pure tensor and returning no correction term. The first disjunct checks if the simplification rule where the gate has no effect can be applied. This is done by checking if `res` is approximately equal to `tqb`, using the `≈` operator, which checks for equality between two qubits by comparing the magnitudes of their 0 and 1-amplitudes. The second disjunct calculates the product of the 0-amplitudes of the control bits ($\prod_{i \in \text{control_set}} \langle 0 | q_i \rangle$), by accessing the 0-amplitudes of `ctrlL` using `qfst` and multiplying them together with a fold, the result of which is compared to 1 to see if the simplification rule in equation 13 applies. Then if the aforementioned simplifications rules are not applicable, the product of 1-amplitudes of the control qubits is calculated at line 19 ($\prod_{i \in \text{control_set}} \langle 1 | q_i \rangle$), and is stored in the variable `beta`. If `beta` is equal to 1, then the simplification rule shown in equation 14 is applied, which means that the target gate will be applied directly on the pure tensor, and not return any correction term.

If no simplifications rules can be applied, a new correction term is created in lines 25-37. This is done by first making a copy of the pure tensor (line 26), and set all qubits in it that are part of `control_set` to be $|1\rangle$ (line 27). For calculating the correction we use a modified version of equation 10, where the target qubit is normalized using the normalizing value $\gamma = (G(q_t) - q_t)^\top (G(q_t) - q_t)$, to make comparison using equation 12 easier as is also done in [4]. This give us the following modified equation with normalization, with α being the global phase of the pure tensor and $\beta = \left(\prod_{j=1}^{n-1} \langle 1 | q_j \rangle \right)$:

$$C^{n-1}(G)(\alpha \cdot \otimes_{i=1}^n q_i) = \alpha \cdot \otimes_{i=1}^n q_i + \alpha \beta \gamma \cdot (|1\rangle^{\otimes(n-1)} \otimes ((G(q_n) - q_n))/\gamma) \quad (22)$$

```

1  applyGate :: Gate -> PureTensorMV s -> ST s (Maybe (PureTensorMV s))
2  applyGate gate (PT alph qbs) =
3      case gate of
4          Sing op target_set -> do
5              Prelude.mapM_ (VM.modify qbs (evalSingle op)) target_set
6              return Nothing
7          Ctrl op control_set target -> do
8              tqb <- VM.read qbs target
9              ctrlL <- Prelude.mapM (VM.read qbs) $ S.toList control_set
10             let res = evalSingle op tqb
11             if
12                 --  $C^{n-1}(G)(\otimes_{i=1}^n q_i) = \otimes_{i=1}^n q_i$  if  $G(q_n) = q_n$  (as seen in equation 15)
13                 res ~= tqb
14             ||
15                 --  $C^{n-1}(G)(\otimes_{i=1}^n q_i) = \otimes_{i=1}^n q_i$  if  $(\prod_{j=1}^{n-1} \langle 0|q_j \rangle) = 1$  (as seen in equation 13)
16                 (~= 1) (L.foldl (\acc q -> acc * qfst q) 1 ctrlL)
17             then return Nothing
18             else
19                 case L.foldl (\acc q -> acc * qsnd q) 1 ctrlL of
20                     --  $C^{n-1}(G)(\otimes_{i=1}^n q_i) = \otimes_{i=1}^n q_i \otimes G(q_n)$  if  $(\prod_{j=1}^{n-1} \langle 1|q_j \rangle) = 1$  (as seen in equation 14)
21                     beta | beta ~= 1 -> do
22                         VM.modify qbs (evalSingle op) target
23                         return Nothing
24                     --  $\beta = (\prod_{j=1}^{n-1} \langle 1|q_j \rangle)$ 
25                     beta -> do
26                         newPT <- VM.clone qbs
27                         Prelude.mapM_ (VM.modify newPT (setQubit (const 0) (const 1))) control_set
28                         tq <- VM.read newPT target
29                         let --  $G(q_t) - q_t$ 
30                             diff = evalSingle op tq - tq
31                             --  $\gamma = (G(q_t) - q_t)^T (G(q_t) - q_t)$ 
32                             gamma = sqrt $ dot (unQubit diff) (unQubit diff)
33                         VM.write newPT target (qubit (qfst diff / gamma) (qsnd diff / gamma))
34                         --  $C^{n-1}(G)(\alpha \cdot \otimes_{i=1}^n q_i) = \alpha \cdot \otimes_{i=1}^n q_i + \alpha \beta \gamma \cdot (|1\rangle^{\otimes(n-1)} \otimes ((G(q_n) - q_n))/\gamma)$ 
35                         -- (as seen in equation 22)
36                         -- Here only the correction term  $\alpha \beta \gamma \cdot (|1\rangle^{\otimes(n-1)} \otimes ((G(q_n) - q_n))/\gamma)$  is returned
37                         return $ Just $ PT (alph * beta * gamma) newPT

```

Figure 2: The applyGate function of the general evaluator, added with comments showing how the code relates to the theory explained in 2.2.

The code for the simpPureTensors is shown in figure 3, takes two pure tensors as an argument and will return the global phase of the second pure tensor *if* it is a multiple of the first. The returned global phase will be used by simpTensor to apply the simplification rule shown in equation 12. In line 3 of figure 3, the findNonEq function tests for equality between the two pure tensors, by using ~= between the qubits. findNonEq then will return Nothing if there are two or more qubits that are not equal to each other or return a Just value containing either an empty list, if all qubits were equal, or a list with the two qubits that were not equal to each other.

If the two qubits are equal, in line 6 we simply return the scalar of the second pure tensor. If all qubits except one are equal, we check if the non-equal qubits are multiples of each other using the /~ operator, which will return Nothing if they are not multiples of each other, or return the factor to multiply to make the second operand equal to the first. For normalized qubits, /~ will therefore return either Just 1 or Just -1. This returned factor, is then multiplied with the global phase of the second qubit in line 9 (if the qubits are multiples of each other).

```

1  simpPureTensors :: PureTensorMV s -> PureTensorMV s -> ST s (Maybe C)
2  simpPureTensors (PT _ qbs1) (PT s2 qbs2) = do
3      l <- findNonEq
4      case l of
5          -- simplify if puretensors are equal up to scalar
6          Just [] -> return $ Just s2
7          -- simplify if equal up to a scalar except one factor
8          Just [qb1, qb2] -> do
9              return $ (* s2) <$> qb2 /~ qb1
10         _ -> return Nothing
11  where findNonEq =
12      -- (...)

```

Figure 3: The simpPureTensors function of the general evaluator, with the code for the findNonEq left out.

4.2 ANF-Centered Implementation

One interesting thing about using phase oracles converted directly from ANF formulas is that the gates in our overall circuit is greatly restricted. In particular, the only gates which would be used are X gates, H gates, and $C^k(Z)$ gates. This allows us to formulate a tensor evaluator specifically tuned to these gates.

4.2.1 Multi-Control Z Gates

Note that the only control gate we would use are $C^k(Z)$ gates. To specialize our evaluator to ANF oracles, we derive a evaluation formula specific to multi-control Z gates.

Proposition 1. $C^{n-1}(Z)(\otimes_{i=1}^n q_i) = \otimes_{i=1}^n q_i - 2\bar{\beta} \cdot |1\rangle^{\otimes n}$ where $\bar{\beta} = \prod_{i=1}^n \langle 1|q_i\rangle$

Proof. Let $q_n = a|0\rangle + b|1\rangle$.

Then note $Z(q_n) = Z(a|0\rangle + b|1\rangle) = a|0\rangle - b|1\rangle$.

Thus, $(Z - I)(q_n) = (a|0\rangle - b|1\rangle) - (a|0\rangle + b|1\rangle) = -2b|1\rangle$

Then recall rule 10. We can specialize the generic gate G to Z in it to obtain

$$C^n(Z)(\otimes_{i=1}^n q_i) = \otimes_{i=1}^n q_i + \beta \cdot (|1\rangle^{\otimes(n-1)} \otimes (Z(q_n) - q_n)) \quad (23)$$

$$= \otimes_{i=1}^n q_i + \beta \cdot (|1\rangle^{\otimes(n-1)} \otimes (-2b|1\rangle)) \quad (24)$$

$$= \otimes_{i=1}^n q_i - 2\beta b |1\rangle^{n+1} \quad (25)$$

Then note that since $b = \langle 1|q_n\rangle$ and $\beta = \prod_{i=1}^{n-1} \langle 1|q_i\rangle$, βb is exactly $\bar{\beta} = \prod_{i=1}^n \langle 1|q_i\rangle$

□

Proposition 1 agrees with our prior intuition that multi-control Z gates phase flips only the $|1\dots 1\rangle$ basis by -1 , in a way which is independent of the position of the target. Furthermore, it allows us a simpler resulting tensor sum which is easier to dynamically simplify further.

4.2.2 Only 4 Qubits

We notice that the set $S = \{|0\rangle, |1\rangle, |+\rangle, |-\rangle\}$ is closed under the gates X and H , up to some phase flips, as demonstrated below:

$$X|0\rangle = |1\rangle \quad (26)$$

$$X|1\rangle = |0\rangle \quad (27)$$

$$X|+\rangle = |+\rangle \quad (28)$$

$$X|-\rangle = -|-\rangle \quad (29)$$

$$H|0\rangle = |+\rangle \quad (30)$$

$$H|1\rangle = |-\rangle \quad (31)$$

$$H|+\rangle = |0\rangle \quad (32)$$

$$H|-\rangle = |1\rangle \quad (33)$$

This combined with Proposition 1, which states $C^n Z|\psi\rangle = |\psi\rangle + \beta|1\rangle^n$, means that the vector space generated by S^n is closed under all gates in our circuit for ANF-oracles. Combined with the fact that our starting state $|0..0\rangle$ is in the span of S^n as well, means our entire quantum algorithm happens inside span S^n .

Thus, for an ANF-oracle Grover's circuit, we can actually represent our quantum state not as a formal tensor sum of pure tensors in $(\mathbb{C}^2)^n$ but rather as a formal sum of pure tensors in $\mathbb{C} \cdot S^n$. In particular, we can even represent it as a sum of pure tensors in $\mathbb{R} \cdot S^n$ because, though quantum states are Hilbert spaces over $\mathbb{C}$ in general, in the case of Grover's, only the real subspace is used.[1] The revised type definitions are seen in Figure 4.

```
data SingleGate
  = X | H
  deriving(Show)

data Gate
  = Only QubitPos SingleGate
  | MCZ [QubitPos]
  deriving(Show)

type Circuit = [Gate]

data Qubit =
  Ket0 | Ket1 | KetPlus | KetMinus

data PureTensor = PT {
  scalar :: Double,
  qbs :: V.Vector Qubit
}

type Tensor = [PureTensor]
```

(a) Haskell types for gates and circuits.

(b) Types for qubits, pure tensors and tensor terms.

Figure 4: Essential types used by the specialized evaluator

4.3 Theoretical Analysis

How successful is our evaluator at preventing exponential blow-up? The answer to this question is highly dependent on what kind of ANF equations we are considering. In general suppose we have an ANF equation

$$\bigoplus_{i=1}^n t_i \text{ where each } t_i \text{ is the conjunction of some variables}$$

Then the rank of our tensor after evaluating the oracle one time depends on the support (Definition 2) of the t_i s.

Definition 2. *The support of a boolean expression e is the set*

$$S = \{x | x \text{ appears in } e \text{ as a variable}\}$$

For example the support of $x_1 \wedge x_2 \wedge x_5$ is $\{x_1, x_2, x_5\}$

In particular, the rank should be equal to the size of the union closure (Definition 3) of $\{S_i | S_i \text{ is the support of } t_i\}$. This is a direct result of Proposition 2.

Definition 3. *The union closure of a family of sets $\mathcal{A}$ is*

$$\text{Ucl}(\mathcal{A}) = \left\{ \bigcup \mathcal{F} \mid \mathcal{F} \subseteq \mathcal{A}, \mathcal{F} \text{ finite}, \mathcal{F} \neq \emptyset \right\}.$$

Intuitively, the union closure of $\mathcal{A}$ is the collection of all sets you can build by repeatedly taking unions by sets in $\mathcal{A}$.

Proposition 2. *Given an oracle O converted from an ANF $\bigoplus_{i=1}^n t_i$ using the method in 3.3. After evaluating O , there will be a bijective correspondence between terms in our tensor and the sets in*

$$U = \text{Ucl}\{S_i | S_i \text{ is the support of } t_i\}$$

This correspondence is given by

$$f : (\beta \cdot \bigotimes_{k=0}^m q_k \text{ where } q_k = |1\rangle \text{ if } k \in I \text{ and } q_k = |+\rangle \text{ otherwise}) \mapsto \{x_i | i \in I\}$$

Proof. By 3.3, O should be a sequence of n multi-control Z gate, each on which operates on the qubits corresponding to the variables in S_i . Then we induct on n .

When $n = 0$, O is simply the identity function. Thus our tensor after evaluating O looks like the starting state $|+\dots+\rangle$, which corresponds to $\emptyset$ under f as desired.

Consider O with n gates, the first $n - 1$ makes a subcircuit O' . By induction hypothesis, after evaluating O' , our tensor terms $T = \{t_i\}$ should correspond to

$$U' = \text{Ucl}\{S_i | i \in 1..n - 1\}$$

That is if we take an arbitrary tensor term, it should look like

$$t_i = \bigotimes_{k=0}^m q_k \text{ where } q_k = |1\rangle \text{ if } k \in I \text{ and } q_k = |+\rangle \text{ otherwise}$$

up to some scalar.

Now we apply the $(n + 1)^{th}$ multi-control Z gate. For each t_i , the gate keep it and creates the correction term t'_i . If the t'_i s correspond to the set $\{S \cup S_n | S \in U'\}$, then we would have

$$f(\{t_i\} \cup \{t'_i\}) = U' \cup \{S \cup S_n | S \in U'\} = U$$

as desired.

To show that t'_i do in fact corresponds to the set as desired. Consider some t'_i , it is up to a scalar t_i but with the qubits corresponding to S_n set to $|1\rangle$ (Proposition 1). So the qubits which are set to $|1\rangle$ in t'_i are either the qubits which are already set to $|1\rangle$ in t_i (which corresponds to some $S \in U'$ by inductive hypothesis), or the qubits which corresponds to S_n . In other words, they are exactly the qubits corresponding to $S \cup S_n$. And all other qubits in t'_i are $|+\rangle$. Thus we have the t'_i as desired.

However, our map is not necessarily bijective. It is possible that we have some terms t'_a and t'_b which maps to the same S' . But this problem is resolved by the simplification rule. Since for t'_a and t'_b to both map to the same set, they must be equal except for the scalar beta by the definition of f . And if two tensor terms are equal up to a scalar, they will be combined as per simplification rule 12. $\square$

However, this only considers one oracle evaluation. But in Grovers, generally we do oracles evaluation again and again. It turns out that it doesn't actually matter how many iterations we run.

Corollary 2.1. *The tensor rank remains constant for any number of iteration of the oracle*

Proof. This is a corollary of Proposition 2, which says that our tensor terms corresponds to union closures. In the proof of the proposition, we also showed that taking a multi-control Z gate over the qubits corresponding to S on a tensor term t corresponds to taking the union of S and the set S_t corresponding to t . But since $S, S_t \in U$ already and U is a union closure, $S_t \cup S$ is also already in U because union closures are closed under unions. So, there must already exists a tensor term which is equal to the new correction term up to a scalar, and they can be combined with rule 12. $\square$

The last thing we need to deal with is the diffusion step, which only creates correction terms of the form $\beta |1\dots 1\rangle$ no matter what the input tensor is. Thus, all of these correction will collect into a single term $\bar{\beta} |1\dots 1\rangle$. And the diffusion steps will only grow the tensor term by 1 throughout the computation.

4.4 Testing

An automated test suite was designed to test that the two evaluators worked as expected. For the general evaluator, this was done by testing the following directly:

- Test that single qubit gates and control gates applications are simulated correctly.
- Test that the multiple-of simplification rule for tensor term rank reduction is correctly applied.
- Test that the control-gate simplification rules for tensor term rank reduction are correctly applied.
- Test that the final tensor term rank produced by evaluating different circuits matches the expected rank.
- Test that circuits where ordering of the gates does not matter gives the same result no matter the order.

For the specialized evaluator, property-based testing was done using the *QuickCheck*¹ haskell package. This was done for testing conversion of ANF expressions, checking the result of Grover's algorithm and testing that the tensor term rank was correct for different circuits.

It is also tested that all possible single solution oracles give the correct bitstring in simulation of Grover circuits for both evaluators from 2 to 4 qubits. The constructed oracles consist of one control-Z gate with 0 or more X gates, flipping the qubits that should be 0 in the solution bitstring. These are similar to the oracles used for benchmarking in section 4.5.1.

4.5 Benchmarking

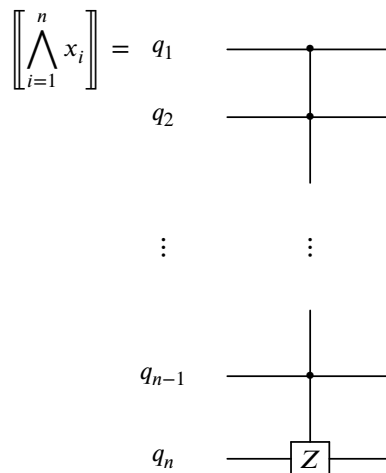
The benchmarking library *criterion*² was used, which function to be benchmarked several times to estimate performance, and one can then inspect estimates like the mean execution time. Besides the tensor decomposition evaluators, the *HQP* library's quantum simulator³ with state-vector semantics was used to compare performance with the evaluators. Benchmarking was done by running Grover's algorithm on different number of qubits, and using different oracles with structures that affect the final tensor term rank.

Measurement in the implemented libraries for resulting tensor terms is currently exponential, since this involves actually calculating the state vector and finding the outcome with the highest probability. This is discussed further in section 5.1. Therefore benchmarking only looks at the Grover circuit simulation itself, but excludes performance of doing a measurement.

The number of iterations of Grover's algorithm with a search space of size N and M solutions can be calculated using the following equation $\left\lceil \frac{\pi}{4} \sqrt{\frac{N}{M}} \right\rceil$ [5], which means that for n qubits and a single solution, the number of iterations required is $\left\lceil \frac{\pi}{4} \sqrt{2^n} \right\rceil$. In the oracles used for benchmarking in the next sections, the number of iterations used assumes that there is a single solution, the worst case, even though this is only the case for the oracle used in section 4.5.1. This makes it easier to compare the efficiency of simulation between the different oracle structures, however the number of iterations will not match the number of solutions for the oracle.

4.5.1 Simple oracle circuits

The first oracle for benchmarking is single-solution phase oracle that uses a single control-Z gate, which for n control bits will be oracles of the form $C^{n-1}(Z)$, which would be the same as solving the boolean expression $\bigwedge_{i=1}^n x_i$:



As we can see in figure 5 the two evaluators outperform the HQP quantum simulator. This is also expected since the tensor term rank will be 2 when the oracle consists of only a single control-Z gate, and as proved in corollary 2.1 it will remain constant, which means that for n qubits, representing the resulting tensor term in the evaluators will have spatial complexity of $O(n)$, as opposed to the state vector in HQP which will be $O(2^n)$.

We also see that the specialized evaluator is faster than the general one, by what looks like a constant factor. The reason for the specialized evaluator being faster could be because the specialized evaluator does not work

¹<https://hackage.haskell.org/package/QuickCheck>

²<https://hackage.haskell.org/package/criterion>

³<https://github.com/diku-dk/atp125-pub>

with complex numbers but instead reals, and it does not use matrix semantics to evaluate the result of applying single-qubit gates on qubits, which means that working on qubits in general is less computationally expensive for the specialized evaluator than the general evaluator. Interestingly, even though the general evaluator uses mutable vectors, whereas the specialized uses immutable vectors, the specialized evaluator is still faster. This is probably because constant time read/writes does not make a difference when size of the vectors are relatively small.

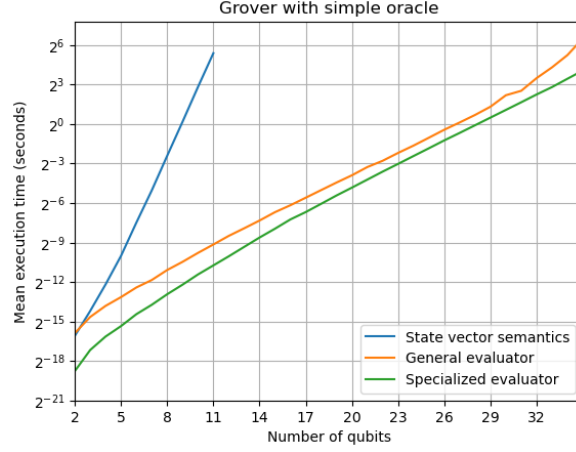
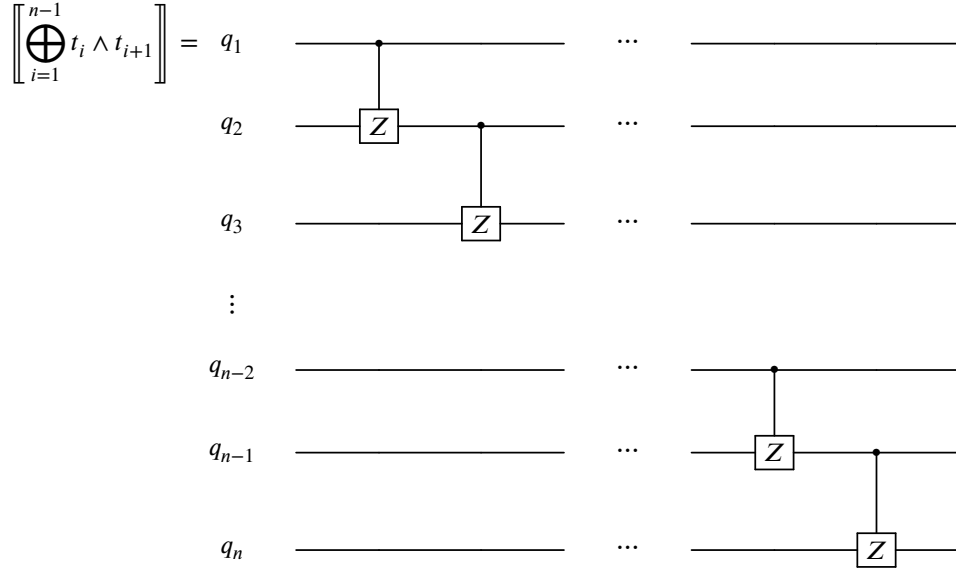


Figure 5: Benchmarking results of simulating Grover circuits using a simple single-solution oracle consisting of one single control-Z gate. As can be seen the simulation the HQP simulation was stopped after 11 qubits, while the evaluators were stopped after 35 qubits.

4.5.2 Complex oracle circuits

In the previous section, the oracle was very simple. This leads to a Grover circuit resulting in a low tensor term rank of 2 that could be effectively evaluated using tensor decomposition. However, what if the oracle is complex and leads to exponential tensor term rank? To benchmark this, we created oracle circuits, where for any two gates with control bit sets S_1 and S_2 we have that $S_1 \not\subseteq S_2 \wedge S_2 \not\subseteq S_1$, which means that any union between two or more sets of control bits for the gates will always give a new unique set, resulting in an size of the union closure that is exponential in the number of qubits. This can be achieved with the following oracle circuit, which represents the boolean expression $\bigoplus_{i=1}^{n-1} t_i \wedge t_{i+1}$ for n literals (similar to the example seen in section 3.3):



We see from the benchmark results shown in figure 6, that the HQP simulator is faster than the two evaluators. This is expected since the tensor term rank is exponential in the number of qubits, which means that the spatial complexity of the evaluators will be $O(2^n \cdot n)$, since there are 2^n pure tensors each of size n , whereas the state vector in HQP will still have an $O(2^n)$ spatial complexity. This also further solidifies that simulation of Grover's algorithm using tensor decomposition depends on the structure of the oracle.

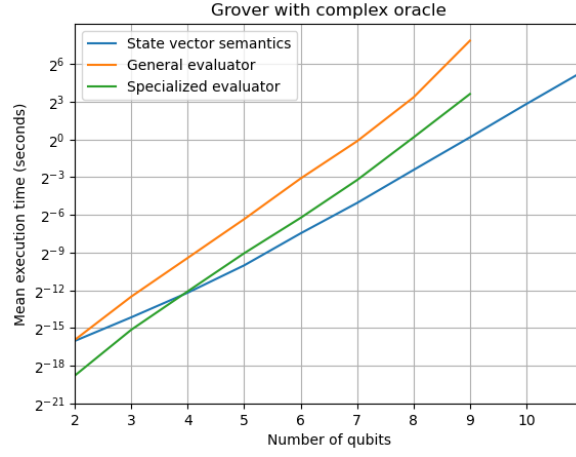
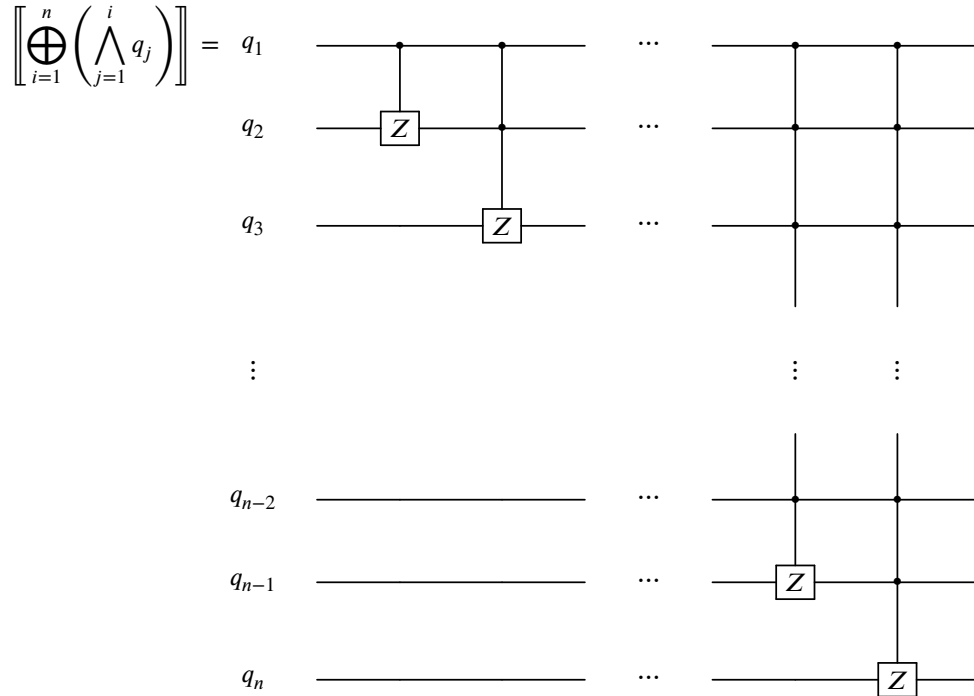


Figure 6: Benchmarking results of running Grover's algorithm using a complex oracle which result in exponential rank.

4.5.3 Oracle circuits with control-bit overlap

The last case is where the complexity of the oracle is high in terms of number of control-gates, but where the control-Z gates have overlapping control bits that makes it optimal for the tensor evaluators. The oracles here represent boolean expressions of the form $\bigoplus_{i=1}^n \left(\bigwedge_{j=1}^i q_j \right)$ for n literals, which gives oracles of the following form:



This means that for any two gates with control bit sets S_1 and S_2 , we have that $S_1 \subseteq S_2 \vee S_2 \subseteq S_1$. Thus, by Proposition 2, for n qubits, there will be n control-Z gates and the oracle will lead to a tensor term rank of n .

We see in figure 7 that here the evaluators outperforms the HQP state vector simulator once again. This makes sense since now the spatial complexity of the evaluators should be $O(n^2)$, which is still better than the HQP state vector. However we do also see that using this oracle results in a slower running time than the simple oracle used in section 4.5.1.

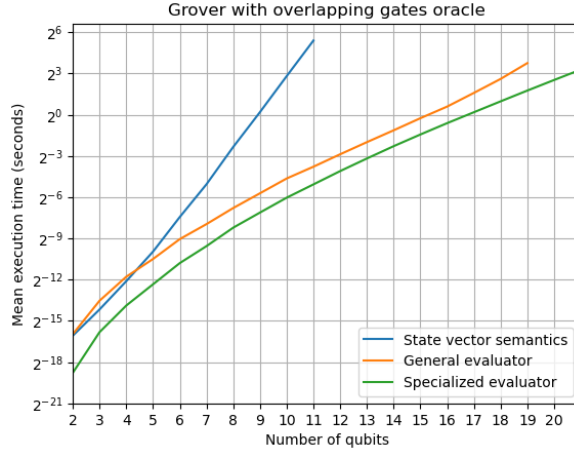


Figure 7: Benchmarking results of running grovers algorithm using a complex oracle which result in linear tensor term rank.

5 Problems

5.1 Measurement Problem

Notice that in this report, we have focused on evaluating quantum state expressed as tensor decomposition, but did not touch on measuring the resulting quantum state. One naive way is to expand it into a State Vector and then calculate the probability of each bit string using the state vector. However, this is exponential in runtime and defeats the complexity benefit of using a tensor decomposition.

In particular, we are interested in an efficient algorithm which implements a "greedy measurement function" (Definition 4) on quantum states expressed as tensor decomposition. Intuitively, the greedy measurement function is a deterministic analogue of the true probabilistic measurement function, but it always only outputs the most likely measurement.

Definition 4. Given a quantum state

$$q = \sum_{k \in \{0,1\}^n} \alpha_k |k\rangle$$

The greedy measurement function g is defined as

$$g(q) = \operatorname{argmax}_{k \in \{0,1\}^n} |\alpha_k|$$

This problem is quite simple on tensor decomposition of rank 1 (i.e. pure tensors).

Proposition 3. There is a linear complexity algorithm which implements greedy measurement on quantum states expressed as pure tensors.

Proof. Let $v_i \in \mathbb{C}^2$ for $i = 1, \dots, n$, and consider the pure tensor $\bigotimes_1^n v_i$. Let this tensor corresponds to the quantum state

$$\sum_{k \in \{0,1\}^n} \alpha_k |k\rangle$$

We can obtain the following from the definition of the Kronecker product on column vectors:

$$\alpha_k = \alpha_{k_1 \dots k_n} = \prod_{i=1}^n v_i[k_i].$$

Our goal is to find

$$\bar{k} = \operatorname{argmax}_{k \in \{0,1\}^n} |\alpha_k|$$

Using multiplicativity of the complex modulus, we have

$$|\alpha_k| = \prod_{i=1}^n |v_i[k_i]|.$$

Each factor $|v_i[k_i]|$ is a nonnegative real number. Since multiplication by a nonnegative real preserves order on nonnegative reals, the product $\prod_{i=1}^n |v_i[k_i]|$ is maximized if and only if each factor $|v_i[k_i]|$ is individually maximized. Therefore, for each $i \in \{1, \dots, n\}$, we may choose

$$\bar{k}_i \in \operatorname{argmax}_{b \in \{0,1\}} |v_i[b]|.$$

Then let $\bar{k} = (\bar{k}_1, \dots, \bar{k}_n)$. This determines a maximizing bit string $\bar{k}$ in $O(n)$ time. $\square$

However, this only works when our tensor product only has one term. For even 2 term, the problem becomes NP-hard. [7]

But our position is also not hopeless: we're not trying to solve the general problem. In particular, we're dealing with very special class of quantum states which must look something like

$$\sum_{k \in \{0,1\}^n} \alpha_k |k\rangle \text{ where } \forall k. (a_k \approx 0 \text{ or } a_k \approx 1)$$

Reaching this state is the entire point of Grover's. Thus, we may expect there might be some specialized algorithm which can preform greedy measurement efficiently on this restricted set of states.

5.2 ANF-SAT is already P

Considering that much of our focus in this project is on applying Grover's Algorithm on solving ANF boolean formulas, we should ask how interesting of a problem is solving ANF formulas. And the disappointing answer is that solving a single ANF formula by itself is not very interesting and we have very simple linear complexity algorithms.

Proposition 4. *There exists a linear time algorithm to find solution to some boolean formula expressed in ANF.*

Proof. Consider an ANF formula

$$f = \oplus_{i=1}^n t_i \text{ where } t_i = \wedge_{j=1}^{m_i} x_{k_{ij}}$$

We can just pick some $t_{\bar{i}}$ with minimum degree (i.e $m_{\bar{i}} < m_i \forall i$). Then we just take $x_{k_{\bar{i}1}} \dots x_{k_{\bar{i}m_{\bar{i}}}}$ to be *true* and everything else to be *false*. Since $t_{\bar{i}}$ has minimum degree, all other terms t_i with $i \neq \bar{i}$ must have more variables and thus are not satisfied. Thus the whole formula is reduced to $true \oplus false \oplus \dots \oplus false = true$. $\square$

Thus, although the general SAT problem is NP-complete, SAT restricted to formulas in algebraic normal form (ANF-SAT) is solvable in polynomial time. This is because converting an arbitrary SAT instance into ANF may require an exponential blow-up in size, since it involves repeatedly distributing conjunction over exclusive-or, potentially an exponential number of times.

6 Future Work

Though ANF-SAT is a uninteresting problem due to being in P, there are variants of ANF-SAT that are more interesting. One such variant is the satisfiability of a *system* of ANF formulas. This problem is more commonly known through its algebraic characterization as solving a system of polynomial equations in

$$\mathbb{F}_2[x_1, x_2, \dots, x_n],$$

where $\mathbb{F}_2$ denotes the finite field with two elements.

The equivalence between ANF formulas and polynomials in $\mathbb{F}_2[x_1, x_2, \dots, x_n]$ is straightforward: we identify *false* and *true* with 0 and 1 $\in \mathbb{F}_2$, respectively, and observe that $\oplus$ and $\wedge$ correspond to addition and multiplication in $\mathbb{F}_2$.

Solving a system of polynomial equations over $\mathbb{F}_k[x_1, x_2, \dots, x_n]$ is NP-complete for any finite field $\mathbb{F}_k$, even when restricted to polynomials of degree at most two. This degree-two case is known as the *Multivariate Quadratic* (MQ) problem and is heavily studied in cryptography.[2] If our ANF-SAT algorithm could be extended to certain systems of ANF equations, it could be of interest to cryptographers.

7 Conclusion

We have successfully shown that there are multiple approaches for constructing oracles to use in in Grover's algorithm for SAT solving. Both solutions have their pros and cons. The biggest pro to using the boolean oracle is that we do not have to convert the boolean expression to another form, since it can directly be translated into a quantum circuit. The biggest con of this though is that some gates are less optimal and require the use of ancilla qubits. On the other hand, the biggest pro to using the phase oracle solution is that you do not have any ancilla qubits and the oracle is only made up of control Z gates and Z gates. The biggest con as we discussed above is that converting to ANF is NP, however, there are many cases where converting does not run in the worst case. An interesting area for further research would be to determine how difficult an expression is to convert to ANF and to determine at runtime which oracle to use.

Benchmarking of the two implemented quantum circuit evaluators using tensor decomposition show that tensor decomposition can be used to simulate a high number of qubits for Grover's algorithm, even if the oracle is more complex, excluding measurement. However the effectiveness of using tensor decomposition is highly dependent on the structure of the oracle, which have an influence on the tensor term rank, and can lead to worse performance than simulation using a state vector. More specifically, the best case is when the control bits between control gates in the oracles completely overlap, leading to a linear increase in rank for each gate rather than exponential. Furthermore we see that specializing the evaluator to work only for ANF generated oracles using only control-Z gates increases runtime performance, as it is more simple and less computationally expensive than the general evaluator.

One problem we were not able to solve was measurement of the final tensor term in non-exponential time. While a rank 1 tensor term is possible to measure, in general doing a measurement on a tensor term of higher rank has an exponential runtime. However there might some way to measure the class of quantum states generated from Grover's algorithm specifically.

References

- [1] 3Blue1Brown. *But what is quantum computing? (Grover's Algorithm)*. <https://www.youtube.com/watch?v=RQWpF2Gb-gU>. YouTube video, accessed January 15, 2026. 2025.
- [2] Emanuele Bellini, Rusydi H. Makarim, Carlo Sanna, and Javier Verbel. “An Estimator for the Hardness of the MQ Problem”. In: *Progress in Cryptology - AFRICACRYPT 2022*. Ed. by Lejla Batina and Joan Daemen. Cham: Springer Nature Switzerland, 2022, pp. 323–347. ISBN: 978-3-031-17433-9.
- [3] C. Figgatt, D. Maslov, K. A. Landsman, N. M. Linke, S. Debnath, and C. Monroe. “Complete 3-Qubit Grover search on a programmable quantum computer.” In: (2017). DOI: <https://doi.org/10.1038/s41467-017-01904-7>.
- [4] Fritz Henglein, Thobias Seth Larsen, Mads Rehof, and Sjúrdur Skorheim. “Functional Quantum Circuit Simulation with Tensor Term Rank Reduction”. Unpublished manuscript.
- [5] Sihyung Lee and Seung Yeob Nam. “Finding All Solutions with Grover’s Algorithm by Integrating Estimation and Discovery”. In: *Electronics* 13.23 (2024). ISSN: 2079-9292. DOI: 10.3390/electronics13234830. URL: <https://www.mdpi.com/2079-9292/13/23/4830>.
- [6] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information*. eng. 10th anniversary ed. Cambridge: Cambridge University Press, 2010. ISBN: 0-511-99400-1.
- [7] Nicholas D. Sidiropoulos, Paris A. Karakasis, and Aritra Konar. “Minimizing Low-Rank Models of High-Order Tensors: Hardness, Span, Tight Relaxation, and Applications”. In: *IEEE Transactions on Signal Processing* 72 (2024), pp. 129–142. ISSN: 1941-0476. DOI: 10.1109/tsp.2023.3338062. URL: <http://dx.doi.org/10.1109/TSP.2023.3338062>.
- [8] Wikipedia contributors. *Algebraic normal form* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 14-January-2026]. 2025. URL: https://en.wikipedia.org/wiki/Algebraic_normal_form.
- [9] Wikipedia contributors. *Cook–Levin theorem* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 17-January-2026]. 2025. URL: https://en.wikipedia.org/w/index.php?title=Cook%E2%80%9993Levin_theorem&oldid=1314492129.
- [10] Wikipedia contributors. *Grover’s Algorithm* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 14-January-2026]. 2026. URL: https://en.wikipedia.org/wiki/Grover%27s_algorithm.