Tampere University

Juha Lilja

QUANTUM COMPUTING FOR AUTOMATED VEHICLE TRAJECTORY PLANNING

Bachelor's thesis
Faculty of Information Technology and Communication Sciences
Examiner: Taneli Riihonen
January 2023

ABSTRACT

Juha Lilja: Quantum Computing for Automated Vehicle Trajectory Planning
Bachelor's thesis
Tampere University
Bachelor's Programme in Computing and Electrical Engineering
January 2023

This work presents an algorithm for solving trajectory planning problem with quantum computing. Trajectory planning is an important problem in the field of automated vehicles, as it makes it possible for the vehicle to traverse in new environments without a preplanned route. It is a computationally expensive task, especially in dynamic environments with lots of moving objects, which is why using quantum computing for the problem is examined in this work. The presented algorithm is designed to scale for arbitrarily complex trajectory planning problems. The algorithm is implemented and tested with a quantum computing simulator and a quantum computer. However, due to limitations in currently available quantum hardware and the high computational cost of quantum computing simulation, the algorithm can only be tested with a simplified version of the problem. Simulation tests show promising results as the algorithm can solve the simplified problem as designed in varying test cases. Tests with a quantum computer show that major improvements in quantum hardware are needed to be able to use the algorithm for real-world applications.

Keywords: trajectory planning, quantum computing, automated vehicles

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Juha Lilja: Kvanttilaskenta autonomisten ajoneuvojen trajektorin laskennassa
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma
Tammikuu 2023

Tämä työ esittää kvanttilaskentaa käyttävän algoritmin trajektorin laskentaan. Trajektorin laskenta on keskeinen ongelma autonomisten ajoneuvojen suunnittelussa, sillä se mahdollistaa ajoneuvon liikkumisen muuttuvassa ympäristössä ilman etukäteen laskettua reittiä. Ongelma on laskennallisesti vaativa, etenkin kun ympäristössä on paljon liikkuvia esteitä, jotka tekevät ympäristöstä muuttuvan. Tämän takia tässä työssä tarkastellaan kvanttilaskennan hyödyntämistä ongelman ratkaisemisessa. Työssä esitetty algoritmi on suunniteltu skaalautumaan ongelman ratkaisemiseen trajektorin laskennan haastavuuden kasvaessa. Algoritmi toteutetaan ja toteutusta testataan kvanttitietokonesimulaattorilla sekä kvanttitietokoneella. Tällä hetkellä saatavilla olevien kvanttitietokoneiden rajoittuneisuuden sekä kvanttitietokoneen simuloinnin raskauden takia testausta voidaan kuitenkin tehdä ainoastaan yksinkertaistetulla ongelmalla. Simulaatiotestien tulokset ovat lupaavia, sillä algoritmi ratkaisee yksinkertaistetun ongelman suunnitellusti erilaisissa testitilanteissa. Kvanttitietokoneella tehdyt testit osoittavat, että huomattavia parannuksia kvanttitietokoneisiin tarvitaan, jotta algoritmia pystytään hyödyntämään.

Avainsanat: trajektorin laskenta, kvanttilaskenta, autonomiset ajoneuvot

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

PREFACE

This thesis has received funding from VTT Technical Research Centre of Finland. I want to thank my colleagues in VTT Automated Vehicles research team for sharing their knowledge. I also want to thank my supervisor Taneli Riihonen for giving valuable feedback throughout the writing process.

Tampere, 19th January 2023

Juha Lilja

CONTENTS

1. Introduction 1
2. Quantum Computing 2
    2.1 Quantum States and Information 2
    2.2 Quantum Gates and Circuits 4
    2.3 Grover's Algorithm 8
    2.4 Quantum Error Correction 10
    2.5 State-of-the-Art Quantum Computers 12
3. Automated Vehicle Trajectory Planning 14
    3.1 Decoupled Approach 14
    3.2 Direct Approach Using State Lattices 16
4. Quantum Computing Algorithm for Trajectory Planning 19
    4.1 Designing the Oracle 19
    4.2 Details of Running the Algorithm 22
5. Implementation of the Algorithm 24
    5.1 Circuit Implementation of the Algorithm 25
    5.2 Test Results 26
6. Conclusion 29
References 30
Appendix A: Code for Implementing the Quantum Circuit With Qiskit 34

LIST OF SYMBOLS AND ABBREVIATIONS

AV	automated vehicle
C-space	configuration-space in trajectory planning
$CNOT$	controlled NOT quantum gate
D	diffuser in Grover's algorithm
$MCNOT$	multi-controlled NOT quantum gate
NISQ	noisy intermediate-scale quantum
O	oracle in Grover's algorithm
QC	quantum computing
QEC	quantum error correction
RRT	rapidly-exploring random tree

1. INTRODUCTION

Trajectory planning is a major research topic within the field of automated vehicles (AV). It is a problem of finding a trajectory for the vehicle to traverse in its environment without colliding with obstacles. When the environment is complex with lots of static and moving obstacles, it is a computationally expensive task. However, trajectories should be computed fast in real-time as the environment might be constantly changing with time.

Quantum computing (QC) uses quantum phenomena to perform computations. The idea of QC was suggested by Feynman in 1982 when he stated that a quantum computer could be more efficient in simulating quantum mechanical systems than a classical computer [1]. Simulating quantum systems is one application where QC is expected to bring an advantage over classical computers. Other applications include optimization problems and cryptography. Currently, many companies around the world are investing in QC research and are developing quantum computers. Some companies are also offering cloud access to their quantum computers for developers to test and run their quantum programs. This has increased interest in the research on the topic.

This work presents a QC algorithm for AV trajectory planning. The algorithm is designed to scale for arbitrarily complex trajectory planning problems. Currently, the algorithm is greatly limited by the available quantum hardware. The algorithm is tested in this work with a QC simulator and quantum computer. Due to limitations in current quantum hardware and the high computational cost of QC simulation, testing can be performed only for a simplified case of the problem. Simulation tests in different simplified test cases show promising results, as the algorithm can solve the problem as designed. Tests on a quantum computer show, that even a simple version of the problem is too complex to solve with this algorithm on the quantum computer used. It is evident that major advancements in quantum hardware are needed in order to use this algorithm in real-world applications.

This thesis is structured as follows. Chapter 2 introduces QC and its principles and presents the current state of QC. In chapter 3 the problem of AV trajectory planning is introduced and current classical implementations are presented. In chapter 4 the algorithm proposed in this work is explained. In chapter 5 the algorithm is implemented and applied to a simplified version of the problem and the algorithm is tested on an IBM quantum computer and QC simulator. Chapter 6 gives a conclusion of the thesis.

2. QUANTUM COMPUTING

Quantum computers are expected to bring an advantage over classical computers in various different problems. These problems can be for example simulation of quantum systems, cryptography, and optimization problems [2][3][4]. Ideal quantum computer can perform any computation a classical computer can [5]. However, quantum computers are not expected to fully replace classical computers, but to bring advantage on specific problems, which are hard for classical computers to solve.

2.1 Quantum States and Information

A quantum computer is a quantum system and its state can be represented by quantum states. When a quantum system is measured, it can only exist in a certain finite number of states which can be called basis states. However, between measurements, the system can be in a state called superposition which has probabilities for the basis states the system can collapse into when measured. In QC, qubits are the basic unit of information and qubits can exist in quantum states. Vectors in a complex vector space can be used to represent a quantum state. However, to make working with quantum states more convenient, a bra-ket notation can be used. The bra-ket notation, sometimes also called Dirac notation, was introduced by Paul Dirac in 1939 [6]. In bra-ket notation, a ket $|a\rangle$ denotes a vector $\mathbf{a}$, which defines a quantum state in a complex vector space. A bra $\langle a|$ is a complex conjugate of $|a\rangle$ so

$$|a\rangle = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} a_1^* & a_2^* & \dots & a_n^* \end{pmatrix}^\dagger = \langle a|^\dagger \,, \tag{1}$$

where $\mathbf{a}^\dagger$ denotes conjugate transpose of vector $\mathbf{a}$, a_i^* denotes complex conjugate of a_i and $[a_1, \dots, a_n] \in \mathbb{C}$.

Operation

$$\langle a||b\rangle = \langle a|b\rangle = \begin{pmatrix} a_0^* & a_1^* \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = a_0^* b_0 + a_1^* b_1 \tag{2}$$

is an inner product of $|a\rangle$ and $|b\rangle$. Operation

$$|a\rangle \langle b| = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \begin{pmatrix} b_0^* & b_1^* \end{pmatrix} = \begin{pmatrix} a_0 b_0^* & a_0 b_1^* \\ a_1 b_0^* & a_1 b_1^* \end{pmatrix} \tag{3}$$

is an outer product of $|a\rangle$ and $|b\rangle$. Operation

$$|a\rangle |b\rangle = |ab\rangle = |a\rangle \otimes |b\rangle = \begin{pmatrix} a_0 \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \\ a_1 \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} a_0 b_0 \\ a_0 b_1 \\ a_1 b_0 \\ a_1 b_1 \end{pmatrix} \tag{4}$$

is a tensor product of $|a\rangle$ and $|b\rangle$.

As stated above, qubits are the basic unit of information in QC, and the state of a qubit can be described as a quantum state. Any quantum state can be represented as a linear combination of two basis states, that is vectors of a vector space. These basis vectors can be chosen arbitrarily as long as they form an orthonormal basis. In the case of qubits, states

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{5}$$

and

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{6}$$

are typically used as basis states and this is called computational basis. These states $|0\rangle$ and $|1\rangle$ correspond to states 0 and 1 in classical bits. A qubit can also be in a superposition between the basis states. A general state of a qubit can be described as a vector being a linear combination of basis vectors and having a unit length. The general state $|v\rangle$ of a qubit can therefore be expressed as

$$|v\rangle = c_0 |0\rangle + c_1 |1\rangle = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix}, \tag{7}$$

where $[c_0, c_1] \in \mathbb{C}$ and $|c_0|^2 + |c_1|^2 = 1$.

Measurement of a qubit can be done with respect to any orthonormal basis. If a qubit is in

superposition with respect to the basis used in measurement, the qubit's state collapses into one of the basis states [7]. This means that the superposition of a system cannot be measured without affecting the state of the system. The probability that a quantum state $|v\rangle$ will collapse to state $|u\rangle$ when measured is $|\langle u|v\rangle|^2$. So if a qubit in the general state is measured with respect to computational basis, the probability of it collapsing to state $|0\rangle$ is

$$|\langle 0|v\rangle|^2 = \left| \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \right|^2 = |c_0|^2 \tag{8}$$

and similarly probability of it collapsing to state $|1\rangle$ is $|c_1|^2$.

A quantum register is a system consisting of multiple qubits. The basis of a vector space of a quantum register is a tensor product of the basis states of individual qubits. Thus it can be written as

$$\{|00\ldots00\rangle, |00\ldots01\rangle, \ldots, |11\ldots10\rangle, |11\ldots11\rangle\} \tag{9}$$

or

$$\{|0\rangle, |1\rangle, \ldots, |2^n - 2\rangle, |2^n - 1\rangle\}, \tag{10}$$

where n is the number of qubits in the register. A general state of an n-qubit quantum register is

$$c_0 |00\ldots00\rangle + c_1 |00\ldots01\rangle + \cdots + c_{2^n-2} |11\ldots10\rangle + c_{2^n-1} |11\ldots11\rangle, \tag{11}$$

where $[c_0, \ldots, c_{2^n-1}] \in \mathbb{C}$ and $|c_0|^2 + \cdots + |c_{2^n-1}|^2 = 1$. Similarly, as with a one-qubit system, the probability that a multi-qubit system will end up in state $|i\rangle$ when measured is $|c_i|^2$. This is why coefficient c_i is also called probability amplitude. So for example a state of a two-qubit system consisting of qubits $|a\rangle$ and $|b\rangle$ can be written as

$$|a\rangle |b\rangle = |ab\rangle = c_{00} |00\rangle + c_{01} |01\rangle + c_{10} |10\rangle + c_{11} |11\rangle = \begin{pmatrix} c_{00} \\ c_{01} \\ c_{10} \\ c_{11} \end{pmatrix}, \tag{12}$$

where $[c_{00}, c_{01}, c_{10}, c_{11}] \in \mathbb{C}$ and $|c_{00}|^2 + |c_{01}|^2 + |c_{10}|^2 + |c_{11}|^2 = 1$.

2.2 Quantum Gates and Circuits

The state of a qubit or a multi-qubit system can be changed with quantum gates, which are operations to quantum states. A quantum gate can be defined by a unitary matrix [8].

Matrix A is unitary when its conjugate transpose $A^\dagger$ is the inverse of A so that $AA^\dagger = I$. This requirement means that the quantum gates are always reversible. The operation of gate A is applied to a quantum state $|v\rangle$ by matrix multiplication $|v'\rangle = A|v\rangle$. This operation can then be reversed by computing $|v\rangle = A^\dagger |v'\rangle$ to obtain the original state.

As an example, one frequently used quantum gate operating on a single qubit is the Hadamard gate H. It is defined by a matrix

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} = \begin{pmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}\end{pmatrix}. \tag{13}$$

The effect of this to qubits in states $|0\rangle$ and $|1\rangle$ are

$$H|0\rangle = \begin{pmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}\end{pmatrix}\begin{pmatrix}1\\ 0\end{pmatrix} = \begin{pmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{pmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \tag{14}$$

and

$$H|1\rangle = \begin{pmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}\end{pmatrix}\begin{pmatrix}0\\ 1\end{pmatrix} = \begin{pmatrix}\frac{1}{\sqrt{2}}\\ -\frac{1}{\sqrt{2}}\end{pmatrix} = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle. \tag{15}$$

From these equations, it can be seen that the Hadamard gate changes the state of both of these qubits into a superposition where the possibilities of the qubits collapsing to either state $|0\rangle$ or state $|1\rangle$ is $\frac{1}{2}$. However, it must be noted that, although the probabilities are the same, the states $H|0\rangle$ and $H|1\rangle$ are different states. Because $H = H^\dagger$, applying H twice will result $HH|0\rangle = |0\rangle$ and $HH|1\rangle = |1\rangle$. States $H|0\rangle$ and $H|1\rangle$ are often also denoted as $|+\rangle$ and $|-\rangle$ respectively.

Quantum gates can also operate on more than one qubit. One frequently used gate operating on two qubits is a controlled NOT or $CNOT$ gate. A $CNOT$ gate is defined by matrix

$$\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}. \tag{16}$$

The two qubits the gate operates on are called as control qubit and target qubit. The target qubit is flipped if the control qubit is in state $|1\rangle$. If input qubits are only allowed to be in states $\{|0\rangle, |1\rangle\}$, the operation of the $CNOT$ gate is the same as classical XOR gate if the target qubit is considered as the output.

A similar gate to $CNOT$ with more than one control qubits can be designed and in that case, the gate is called multi-controlled $CNOT$ and it is referred to in this work

as $MCNOT$. The basic $MCNOT$ gate flips the target qubit if each of the control qubits is in state $|1\rangle$. However, in this work, an arbitrary basis state for the control register can be chosen to be the state which flips the target qubit, as this can easily be implemented by flipping control qubits in state $|0\rangle$ before and after the basic $MCNOT$ gate.

Now, a two-qubit system is considered. System is initialized as $|00\rangle$. Next, a Hadamard gate is applied to the first qubit. As a result, the system is in state

$$\begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{pmatrix}. \quad (17)$$

After that, a $CNOT$ gate is applied to the system. This will result in the system being in a state

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle . \quad (18)$$

When measuring the system, it has a probability of $\frac{1}{2}$ to be collapsed into state $|00\rangle$ and a probability of $\frac{1}{2}$ to be collapsed into state $|11\rangle$. The probabilities of the measurement resulting in $|01\rangle$ or $|10\rangle$ are both 0. Thus there is a correlation between the two qubits in the system. This correlation is called entanglement.

A group of qubits are said to be entangled when the state of the system they form cannot be described as a product of states of the individual qubits in the system [9]. In the case of the above example, it can be seen that there are no numbers a_0, a_1, b_0, b_1 which satisfy

$$\begin{aligned} (a_0 |0\rangle + a_1 |1\rangle) \otimes (b_0 |0\rangle + b_1 |1\rangle) &= a_0 b_0 |00\rangle + a_0 b_1 |01\rangle + a_1 b_0 |10\rangle + a_1 b_1 |11\rangle \\ &= \frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle , \end{aligned} \quad (19)$$

because if $a_0 b_1 = 0$, then $a_0 = 0 \vee b_1 = 0$ but if $a_0 = 0$, then $a_0 b_0 = 0 \neq \frac{1}{\sqrt{2}}$ and similarly if $b_1 = 0$, then $a_1 b_1 = 0 \neq \frac{1}{\sqrt{2}}$. An example of an unentangled two-qubit state would be

$$\frac{1}{2} |00\rangle + \frac{1}{2} |01\rangle + \frac{1}{2} |10\rangle + \frac{1}{2} |11\rangle , \quad (20)$$

because it could be written as

$$(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle) \otimes (\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle), \tag{21}$$

hence described as a product of the states of the individual qubits.

If two qubits are entangled, measurement on other qubit will immediately collapse the state of the other qubit also. This can be observed even if the entangled qubits are separated physically far away from each other and both qubits are measured so precisely at the same time, that there is not enough time for even light to travel between the qubits between the measurements [10].

To make meaningful computations, quantum gates are applied in sequence to create quantum circuits. Quantum circuits start with initializing all qubits, usually to $|0\rangle$ state. After that, a series of quantum gates are applied to create a desired computation, and finally, the qubits are measured to read out the result.

Previously in this section, a simple quantum circuit was used to create entanglement. This circuit consists of two quantum gates, H and $CNOT$. This circuit can be drawn as a circuit diagram as shown in figure 2.1. The first step is to initialize both qubits to $|0\rangle$. Next, a Hadamard gate is applied to qubit q_0 which is drawn as a box with the gate name inside. In the third step, a $CNOT$ gate is applied, q_0 as a control qubit and q_1 as a target qubit. Finally, both qubits are measured.

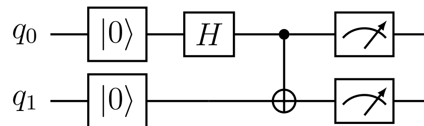


Figure 2.1. *Circuit diagram for entanglement circuit.*

A common way of drawing circuits is to denote the initial state of the qubits after the name of the qubit as shown in figure 2.2. A general operation on multiple qubits is drawn with a box across the qubit signals it operates on as is done with U in figure 2.2. An $MCNOT$ gate is drawn so that the target qubit marked with symbol $\oplus$ is flipped when each control qubit marked with a black dot is in state $|1\rangle$ and each control qubit marked with a white dot is in state $|0\rangle$. So for example in figure 2.2 the target qubit q_2 is flipped if and only if $|q_1 q_0\rangle = |01\rangle$ after U.

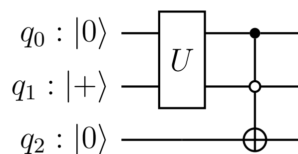


Figure 2.2. *Quantum circuit.*

2.3 Grover's Algorithm

This section introduces Grover's algorithm as an example of a useful quantum algorithm. It is also used in the trajectory planning algorithm of this work. Grover's algorithm is a quantum algorithm for unstructured search designed by Lov Grover in 1996 [11]. Grover's algorithm finds an item that satisfies a predefined condition from an unstructured list of N items. With one item as a solution, solving the problem on a classical computer would take $O(N)$ steps, while Grover's algorithm can find the solution in $O(\sqrt{N})$ steps [11]. This means that when the number of items N grows, the computation time grows linearly in classical computers, while with Grover's algorithm, the computation time grows only quadratically with respect to N. Grover's algorithm can also be generalized to problems with M solutions and it then takes $O(\sqrt{N/M})$ steps to find a solution [5]. Grover's algorithm is optimal for the problem, no other quantum algorithm can solve this problem faster [12].

Implementation of Grover's algorithm starts by initializing all qubits at the n-qubit input register x to state $|+\rangle$ so that the system is initially at state

$$\sum_{i=0}^{N} \frac{1}{\sqrt{N}} |i\rangle \tag{22}$$

which is a uniform superposition of all possible basis states. The size of the input register is n and thus there are $N = 2^n$ basis states from which M states are solutions for the problem. The goal is to find one item that is a solution. When a quantum gate is applied to this uniform superposition state, it will compute the result for all possible basis states simultaneously in one computation. However, this state cannot be measured without affecting the superposition of the system and there is an equal probability for each basis state to be measured. This is why in Grover's algorithm the next step is amplitude amplification. This amplifies amplitudes of solution states and thus increases the probability of measuring a solution state. Amplitude amplification is performed by applying oracle O and diffuser D operations.

The oracle O needs to be designed specifically for each problem. It can be constructed for any boolean function $f(x) : \{0,1\}^n \rightarrow \{0,1\}$, that takes as an input a value with n bits. The value of $f(x)$ should be 1 for each x that is a solution and 0 for every other x. As a simple example $f(x)$ could be a function that takes a 3-bit value as an input and outputs 1 if $x \geq 5$. This can be implemented with two $MCNOT$ gates as shown in figure 2.3. The creation of quantum circuits that implement more complex oracles can however be a challenging task, but there are some proposed solutions to automate the task [13]. If the target qubit is set to state $|-\rangle$ and x is in some general quantum state as in equation 11, the effect is that each coefficient c_i, where $f(i) = 1$, will become $-c_i$.

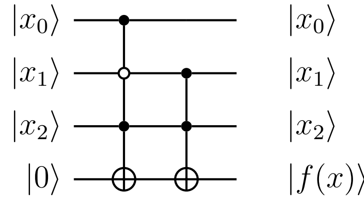
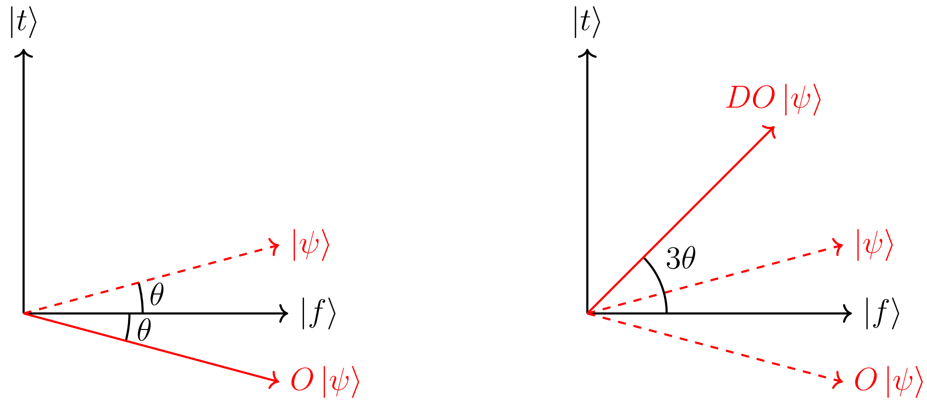


Figure 2.3. *Circuit of $f(x)$.*

The effect of the oracle can be represented geometrically. Let the superposition of all solution states be $|t\rangle$ and the superposition of all other states be $|f\rangle$. Vectors $|t\rangle$ and $|f\rangle$ are orthogonal and they span a plane in a vector space. All states the system can be in can be expressed as a linear combination of $|t\rangle$ and $|f\rangle$. The initial state of the system can be seen in figure 2.4a, where $|\psi\rangle$ denotes the initial state before amplitude amplification. Applying O reflects the system state about the $|f\rangle$. The effect can be seen in figure 2.4a.



(a) *State of the system after applying O.* ***(b)*** *State of the system after applying D.*

Figure 2.4. *Grover's algorithm geometrical representation.*

Diffuser D is an operation that reflects the state $O\,|\psi\rangle$ about the original state $|\psi\rangle$. The geometric effect of D is shown in figure 2.4b. The diffuser can be constructed as shown in figure 2.5. This same construction can be used regardless of the problem, only the number of qubits the D operates on changes depending on the problem.

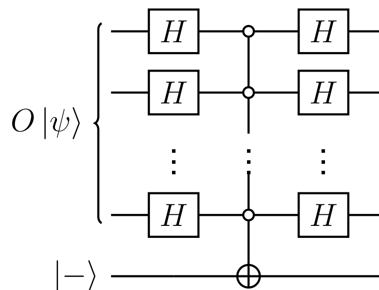


Figure 2.5. *Circuit of a diffuser operation.*

As a result, the new state $DO|\psi\rangle$ is now closer to the superposition of all solution states $|t\rangle$. The effect of this is that, when a measurement is made to the system, the result will be a solution state with a higher probability than initially. Applying O and D can be repeated multiple times before measurement to move the system state closer to $|t\rangle$. However, iterating too many times results in decreasing probability of measuring a solution state as the system state starts to get farther from $|t\rangle$. Equation

$$T = \left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{M}} \right\rfloor \tag{23}$$

defines an optimal number of iterations of applying O and D to maximize the probability of measuring a solution state [12][5].

2.4 Quantum Error Correction

Physical quantum systems are very sensitive to any interruptions from the surrounding environment, which affects and disturbs the state of the quantum system very easily. This is why quantum computers need to be well isolated from the surrounding environment. However, perfect isolation cannot be achieved in practice, and also there needs to be always a way to interact with the quantum computer. This interaction is needed to alter the qubit states and to read results. These are why some errors always happen and that is why quantum error correction (QEC) is needed to counter these errors.

Error correction is done by encoding additional redundant information to the system so that the correct result can be decoded even after errors have happened during the computation. In classical computing, a simple way to add redundancy would be to replace each bit with three bits corresponding to the original bit. So mapping $1 \to 111$ and $0 \to 000$. This way, the original bit can still be recovered even if one bit is flipped by perceiving 101 as 1 for example because it is more probable that there has been an error in only one bit rather than in two or more bits.

In QC, to recover from bit flips, a three qubit bit flip code can be used. This error-correction code is implemented so that basis states are encoded as $|0\rangle \to |000\rangle$ and $|1\rangle \to |111\rangle$. This can be done with two $CNOT$ gates. These resulting 3-qubit states are referred to as logical qubits. A general state of one logical qubit is then $c_0|000\rangle + c_1|111\rangle$. Bit flip errors can then be detected with two additional qubits which are called ancilla qubits. A circuit for this is shown in figure 2.6. In the circuit, the top three qubits are the data qubits representing one logical qubit. The bottom two qubits are ancilla qubits. First, all of the data qubits are coupled to the same state with two $CNOT$ gates. Gate E represents an error which flips 1 or 0 data qubits. Ancilla qubits are then coupled with data qubits so that ancilla$_0$ is coupled with data$_0$ and data$_1$ and ancilla$_1$ is coupled with data$_1$ and data$_2$. From this it follows that ancilla$_0$ is flipped if either data$_0$ or data$_1$ was flipped (but

not both) and similarly with ancilla$_1$, data$_1$ and data$_2$. Thus by measuring ancilla qubits it can be detected which data qubit was flipped if any. Possible outcomes are listed in table 2.1. Based on this information, the detected error can then be corrected by applying NOT gate to the erroneous qubit. If two or more qubits are flipped, this error code is not sufficient to correct that.

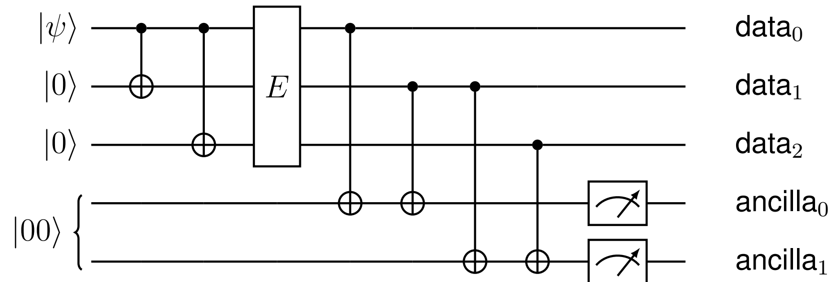


Figure 2.6. *Circuit implementation of a three qubit bit flip code.*

Table 2.1. *Error based on the state of the ancilla bits.*

ancilla	error
$\|00\rangle$	no error
$\|01\rangle$	data$_0$ flipped
$\|10\rangle$	data$_2$ flipped
$\|11\rangle$	data$_1$ flipped

This type of error coding only corrects bit flip errors, but any other types of errors to qubits are not corrected. For correcting arbitrary errors on a single qubit the Shor code can be used [14]. The Shor code encodes each qubit to nine qubits as

$$|0\rangle \rightarrow \frac{1}{2\sqrt{2}}(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)(|000\rangle + |111\rangle) \tag{24}$$

and

$$|1\rangle \rightarrow \frac{1}{2\sqrt{2}}(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)(|000\rangle - |111\rangle). \tag{25}$$

A quantum circuit implementing this encoding is presented in figure 2.7. Bit flip errors can be corrected as shown previously by comparing qubits inside three qubit groups. Similarly, phase errors can be detected and corrected by comparing phases between three qubit groups. Thus any error that is a combination of these errors can be corrected with this method as long as an error occurs in only one qubit. 7-qubit and 5-qubit codes have also been found [15][16].

In addition to error correction codes, a fault-tolerant gate design is used to reduce the propagating effects of errors. As a simple example of fault-tolerant gate design a $CNOT$ gate operating on data encoded by the three qubit code is shown in figure 2.8. In figure

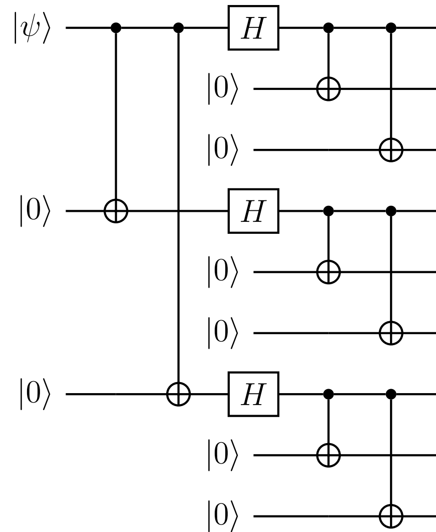
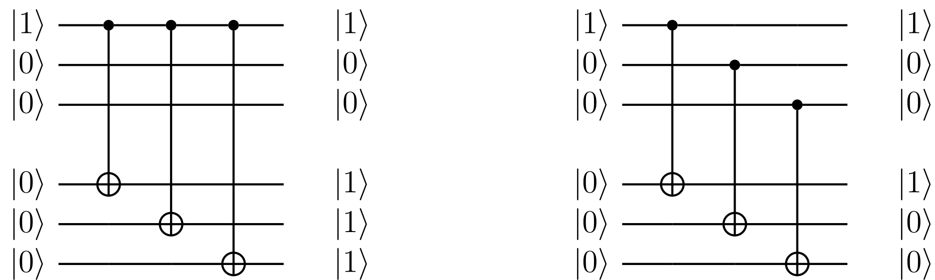


Figure 2.7. *The Shor code circuit. [5]*

2.8a is a non-fault-tolerant $CNOT$ gate and in figure 2.8b is a fault-tolerant $CNOT$ gate. Both are getting as input an erroneous state $|001\rangle$ which should be $|000\rangle$ but a bit flip error has happened to one qubit. As can be seen in 2.8a, the non-fault-tolerant gate cascades the one qubit error to three qubits while the fault-tolerant gate in 2.8b carries the error to only one additional qubit so that the original state can still be recovered.



(a) *A non-fault-tolerant $CNOT$ gate.* ***(b)*** *A fault-tolerant $CNOT$ gate.*

Figure 2.8. *Fault-tolerant gate design.*

As can be noted, QEC techniques multiply the number of qubits and gates needed for logical qubits. This is one reason why current quantum computers are generally not implementing QEC because the already limited number of qubits would become even more limited and the complexity of the circuits grow. An effect of this is that quantum computers with far more qubits are needed to make fault-tolerant quantum computers.

2.5 State-of-the-Art Quantum Computers

Quantum computers are currently in the era of noisy intermediate-scale quantum (NISQ) computers. The term NISQ is used for quantum computers operating in order of 100

qubits which are affected by noise and are not error corrected. However, many revolutionary quantum algorithms would require millions of qubits with error correction. Current NISQ devices do not present an advantage for practical problems over classical computers but it is under active research to create more algorithms for NISQ computers in various different problems.[4]

Two of the most promising technologies to build a quantum computer are trapped ion and superconducting technologies. However, it is not straightforward to scale these technologies and it is still not clear if these can be used to build error-tolerant quantum computers with millions of qubits. Major development for both quantum error correction techniques and hardware is needed to build implementations of such quantum computers.[17]

A trapped ion technology uses ions as qubits. A ground state and an excited state of an ion can be mapped to be the two basis states of a qubit. This approach is implemented by forming a linear crystal of ions by trapping them with electromagnetic traps and laser cooling them to near absolute zero temperature in a vacuum [18]. The state of the individual ions can then be manipulated by interacting with them by laser beams [19]. To scale this approach, multiple small ion traps and communication between those can be used [20]. Using microwaves to control the ions instead of lasers is also proposed to make scaling easier [21].

In superconducting technology, materials that become superconductors at low temperatures are used to create superconducting circuits. Components called Josephson junctions are then used in these circuits to create qubits. Three different types of qubits can be created in this way and the qubits can be controlled by adjusting voltage, current or magnetic flux depending on the qubit type [22]. The operation temperature for these circuits needs to be in the order of 10 mK [23].

3. AUTOMATED VEHICLE TRAJECTORY PLANNING

Trajectory planning in robotics refers to a problem of finding a trajectory from a starting point to some target point, which a robot can follow while avoiding collisions. The trajectory is a path combined with timing information, that is speeds at which the trajectory should be performed. Trajectory planning is an important issue in designing AVs because it provides a way for the vehicle to navigate in an unknown and changing environment.

The space in which the vehicle operates is referred to here as a configuration-space or C-space. The C-space is a space consisting of all different configurations or states a vehicle can be in. These configurations can be as simple as just x- and y-coordinates of the vehicle in a 2-dimensional space. Additional parameters can be included in configurations, which can be steering angle and velocity for example. The dimension of the C-space depends on the number of parameters chosen for configurations.

To account for obstacles in the vehicle's operating space, the C-space is further divided into two subspaces, C_{free} and C_{obs}. Subspace C_{free} contains all of the configurations which are free from obstacles and C_{obs} is a subspace of C-space that contains all of the configurations occupied by obstacles. The starting configuration which is the initial configuration the vehicle is starting from is referred to here as c_s. There can exist one or more target configurations which are configurations the vehicle is trying to achieve. A set of target configurations is referred to as C_t and a general target configuration belonging to C_t is c_t.

Methods for trajectory planning can be divided into two general categories, decoupled and direct methods [24]. In the decoupled method, a collision-free path without timing information is constructed first. After the path is found, the timing information is then added to transform the path into a trajectory. In the direct approach, these two steps are combined into one, and trajectory is calculated directly in one step.

3.1 Decoupled Approach

In decoupled approach, the first step is to find an obstacle-free path with no timing information between c_s and c_t. This step is also called path planning. There are three major approaches to path planning, which are potential field based, sampling based, and

discrete methods [25].

Potential field based methods are constructing a potential field from the C-space. It is used for example in [26]. In this approach, a high potential is given to c_s and a low potential is given to c_t. All of the configurations in C_{obs} are assigned to have a high potential. A vehicle is then moving towards lower potential, which should lead it finally to c_t. However, target configuration is not always reached as there might exist one or more local minima where the vehicle might end up to. Different potential functions for creating the potential field can be used to try to avoid local minima to appear.

Sampling based methods take a random sample from the C-space and check connectivity between the random sample and previously selected samples. One commonly used method for this is a rapidly-exploring random tree (RRT) algorithm [27]. RRT algorithm constructs a graph G where vertices are configurations in C_{free} and edges between those are paths between configurations. First, the graph G contains only c_s as a vertex. New vertices are added by selecting a random configuration c_i from C_{free}, finding the nearest neighbor configuration for it from G vertices, and checking if an obstacle-free path exists between them. Finding a path between configurations can be done simply by checking if a straight line between them is free of obstacles, but more sophisticated methods can also be used. If a path is found, c_i is then added as a vertex to the G and it is connected with an edge to the nearest neighboring vertex. When a sample is found which is close enough to a target configuration and is connected to G via a path, a complete path can then be found by traversing back to c_s in G.

In the discrete method, the C-space is discretized as illustrated in figure 3.1, where each cell in a grid is a configuration with x- and y-coordinates as parameters and black cells are obstacles. A resulting discrete C-space can then be represented as a graph where configurations are vertices and neighboring vertices are connected by an edge. Any shortest path graph search algorithm such as A* can then be used to find the shortest path between c_s and c_t. This method is used for example in [28].

When an obstacle-free path is found, it then needs to be transformed into a trajectory. Depending on how the path is initially constructed it might include too tight curves for the vehicle to follow or even instantaneous changes to heading. Therefore, smoothing might be needed to make the path driveable. This can be done for example by representing the path with polynomial curves or Bézier curves which follow the original path as closely as possible while making sure that no too tight curves are used [29].

After smoothing, a velocity profile needs to be added to the path to transform it into a trajectory with timing information. The velocity profile can be computed by using an algorithm introduced in [30] and [31]. The algorithm first computes a velocity limit curve for the path which is done by assigning maximum velocities which can be used at each point in the path. After that, maximum and minimum accelerations are considered to create a

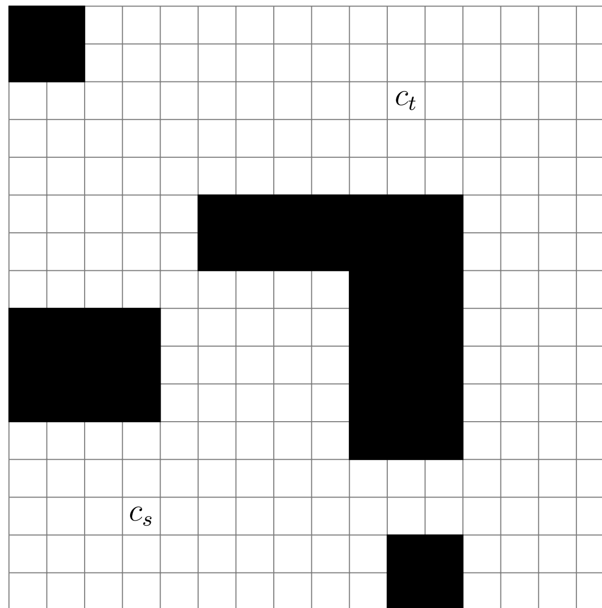


Figure 3.1. *Discretized C-space.*

velocity profile. The objective is to maximize velocity at each point of the path, so that the velocity stays below the velocity limit curve and that accelerations do not exceed maximum and minimum accelerations at any point. As a result, velocity information is added to the path which makes it a trajectory.

3.2 Direct Approach Using State Lattices

In the direct approach, a trajectory is created directly in one step without a path planning step. Similar approaches can be used for this as was used in path planning if the C-space includes velocity. Also other first-order differential parameters may be included in the C-space or even higher-order derivatives. These derivatives are included to make sure the resulting trajectory is driveable with the vehicle it is designed for. The disadvantage of adding more parameters is that each new parameter in configurations increases the dimensions of the C-space. The increased size of a C-space makes computation more expensive. That is why the use of QC for this problem is studied in this work.

RRT algorithm can be used for direct trajectory planning as for example in [32]. The approach for trajectory planning used in the algorithm of this work is like an extension to the grid based method of path planning and is based on state lattices. This approach is discussed for example in [33], [34], [35], [36], [37], [38] and described below based on these.

A state lattice can be viewed as a graph constructed so that each configuration (state) in a discrete C-space is a vertex of the graph and each configuration is connected to other configurations close to it with edges that represent trajectories. The trajectories which are acting as edges in state lattice are predesigned short trajectories, primitives.

Primitives are planned so that the vehicle can follow those accurately from one configuration to another. There is a finite number of these primitives and these can be reused from configuration to configuration to create a regular lattice. A simple example of a state lattice is presented in figure 3.2, where C-space is a three-dimensional space with x- and y-coordinates and heading as parameters. In the figure, the heading dimension is projected to the x-y-plane. The set of primitives is drawn with black lines and all of the gray lines are copies of these primitives, starting from different configurations. The points where the primitives connect to each other are the discrete configurations of C-space, between which the vehicle is traversing using the primitive trajectories.

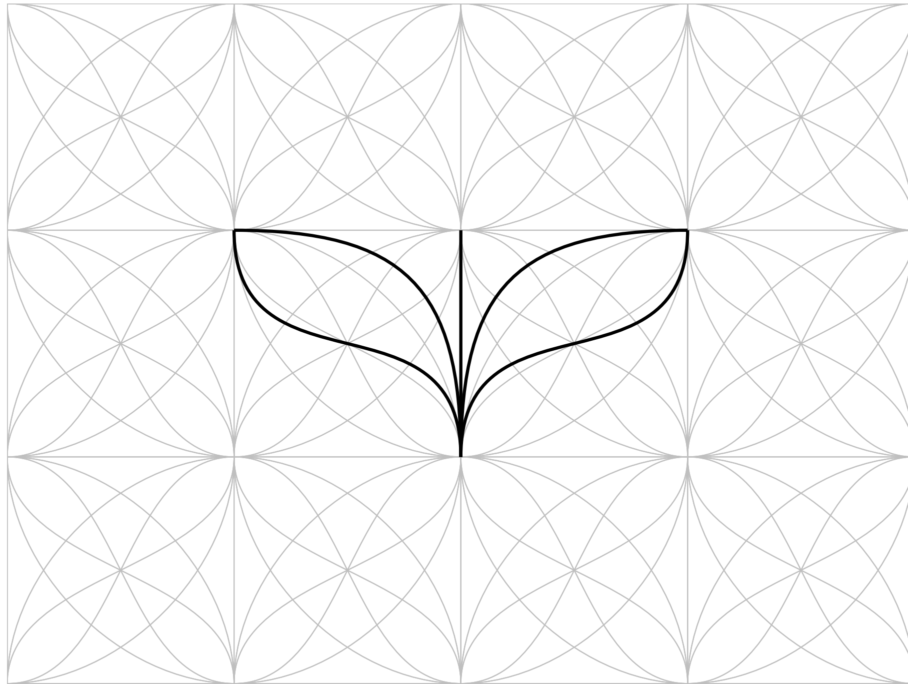


Figure 3.2. *An example of a simple state lattice.*

The actual trajectory from c_s to c_t is constructed from multiple primitive trajectories. From c_s, the vehicle can traverse any of the b primitives which are starting from c_s. This will lead the vehicle to a different configuration from where another primitive can be chosen to be traversed next and so on. This creates a tree-like structure of possible trajectories. As a result, there are b^d trajectories where b is the number of primitives starting from each configuration and d is the depth of the tree or the number of primitives the trajectory is constructed of. The task is then to find a valid trajectory from c_s to c_t, which does not collide with any obstacles, from all of the b^d possibilities. To check the collisions, primitives include information about which positions the vehicle collides with when traversing the trajectory. Classically, a graph search algorithm like A* can be used to find a valid trajectory from the tree graph. A QC algorithm for finding a valid trajectory is presented in chapter 4 of this work.

In the case of AVs, the environment the vehicle operates in usually includes moving ob-

stacles such as other vehicles and pedestrians. This makes the environment dynamic so that it is changing over time. To be able to work in dynamic environments, a spatiotemporal state lattice needs to be created by adding time dimension to the C-space [37]. For moving obstacles, predictions need to be made where they can move next, to be able to represent the obstacles in the spatiotemporal state lattice. These predictions of movements need to be constantly updated while new information about the dynamic obstacles' states is obtained.

Also, when considering AVs, the environment of the vehicle is usually not entirely known. Obstacles in the environment are detected by vehicles' sensors such as LiDARs, radars, and cameras. These sensors have limited range and visibility, and the information from the sensors is continuously updating. To address these uncertainties and the dynamic environment, trajectories must be recalculated frequently. This is why the computation of trajectory planning must be fast so that the vehicle has enough time to react to the changing and updating environment.

4. QUANTUM COMPUTING ALGORITHM FOR TRAJECTORY PLANNING

In this chapter, a quantum algorithm for trajectory planning is presented. This algorithm uses short primitive trajectories which are creating a state lattice as presented in section 3.2. From all of the possible trajectories constructed by d primitives, Grover's algorithm is used to find a valid trajectory from starting configuration c_s to target configuration c_t.

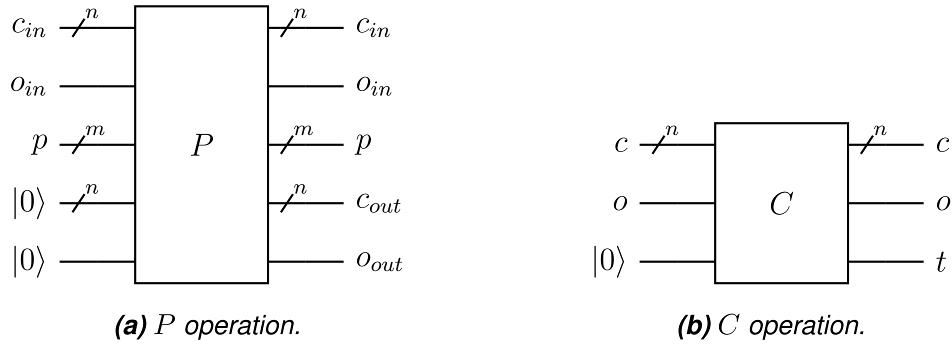
4.1 Designing the Oracle

To be able to apply Grover's algorithm to this problem, an oracle needs to be designed for this particular problem. The construction of an oracle uses ideas introduced in [39]. An algorithm presented for path planning in [40] is combined with state lattices to be able to apply it in direct trajectory planning and needed modifications for this are made. The main improvements are to make the algorithm less dependent on the choice of the exact number of primitives the trajectory consists of and collision checking is implemented differently for it to suit the trajectory planning problem.

First, the state lattice needs to be created as described in section 3.2. This defines the configuration-space in which the vehicle is operating and the primitive trajectories that are used to traverse between configurations. The trajectory is constructed by applying primitives one after another. This creates b^d possible trajectories starting from c_s, where b is the number of primitives starting from one configuration and d is the number of primitives that trajectories are constructed of.

The oracle for this problem is constructed from two main operations or gates which are named here as P and C operations. The P operation computes a configuration which is achieved when starting from an input configuration and traversing a primitive. The P operation also performs collision checking. The C operation checks if the input configuration is a target configuration and if the whole trajectory leading to it is collision-free. In addition to these, the oracle also needs inverse operations $P^\dagger$ and $C^\dagger$ of P and C. High-level circuit representations of P and C are presented in figure 4.1.

A P operation applies a new primitive to the end of a trajectory and checks if it collides with an obstacle. It operates on 3 logical input registers and 2 logical output registers. As



(a) *P operation.* ***(b)*** *C operation.*

Figure 4.1. *High-level circuit representations of P and C operations.*

every quantum operation must be reversible, the number of input and output qubits must be equal and each possible input must map to unique output, from where an original input must be deducible. To ensure this, the logical output registers are initialized as $|0\rangle$ and fed as an input to P. Logical input registers come as an output from P unchanged. Logically, three topmost registers of P shown in figure 4.1a are considered as inputs and two registers at the bottom are considered as outputs. Input registers are configuration c_{in} represented in n qubits, qubit o_{in} which includes information if a trajectory is free of obstacles before this operation, and a primitive p represented in m qubits. As an output from the P operation, comes the resulting configuration c_{out} and a qubit o_{out} which includes information if a trajectory is free of obstacles after this operation. Configuration c_{out} is computed based on c_{in} and p, so that when starting from the input configuration c_{in} and traversing the primitive p, the resulting configuration is c_{out}. Operation P also checks if traversing the primitive p from c_{in} collides with any obstacles and based on that it sets $|o_{out}\rangle = |0\rangle$ if the primitive does not collide or $|o_{out}\rangle = |1\rangle$ if the primitive collides. If $|o_{in}\rangle = |1\rangle$ it means that the trajectory has collided before traversing this new primitive and in that case o_{out} is set to $|1\rangle$ regardless of if the trajectory collided with the new primitive.

The inputs of P can also be in a superposition. The effect of that is, that the P operation computes simultaneously all primitives p is in superposition of, from all configurations c_i is in superposition of. The outputs c_o and o_o are then in a superposition of all configurations that can be reached based on the inputs.

A C operation checks if a target configuration c_t is reached without colliding with any obstacles. It takes as an input a configuration c and a qubit o, which includes the information if a trajectory is free of obstacles. These are the two topmost registers in figure 4.1b. As an output, C operation gives t which is set to $|1\rangle$ if $c \in C_t$ and $|o\rangle = |0\rangle$, meaning that the trajectory leading to configuration c is free of obstacles. Otherwise t is set to $|0\rangle$.

An oracle O is then created from P and C operations and their inverse operations $P^\dagger$ and $C^\dagger$. Figure 4.2 shows the circuit of an oracle in a simple case of $d = 2$, that is, trajectories are created of two primitives. All of the registers c_i and o_i as well as t are initialized

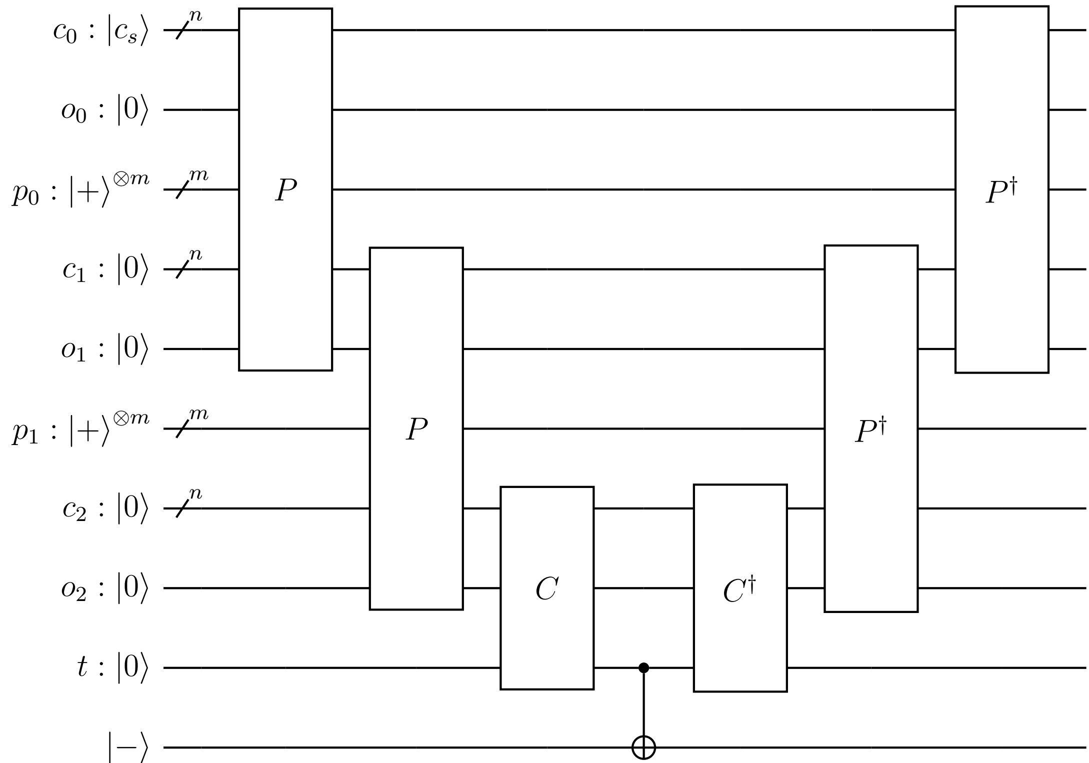


Figure 4.2. *Circuit of an oracle.*

to state $|0\rangle$, except c_0 which is set to the initial starting configuration c_s. All primitive registers p_i are initialized to an equal superposition of all computational basis states of a quantum register of width m, which is denoted in the figure as $|+\rangle^{\otimes m}$. With these inputs, the first P operation thus computes every possible resulting configuration a vehicle can be in when starting from c_s and traversing any of the primitives and checks if resulting trajectories collide. These outputs from the first P operation are then given as an input to another P operation with a new quantum register p_1 for primitives, which is in an equal superposition of all primitives. The second P operation then outputs a superposition of all possible configurations after two primitives together with collision information in c_2 and o_2. More P operations can be added to the circuit in a similar way to create longer trajectories.

The output of a last P operation, in this case c_2 and o_2, is then given as an input to C operation. The C operation sets $|t\rangle = |1\rangle$ for each input configuration that is a target configuration and that has $|o_2\rangle = |0\rangle$. After the C operation, a $CNOT$ gate with target qubit in state $|-\rangle$ is applied. The effect of this is, that coefficients for states where $|t\rangle = |1\rangle$ are set as negative as explained in section 2.3. To expand this effect to every other qubit, the inverse operations $C^\dagger$ and $P^\dagger$ of C and P are applied. The result of this is, that primitive registers p_i are in an equal superposition of all primitives, but combinations of primitives that are leading to a target configuration without colliding are marked with a negative coefficient.

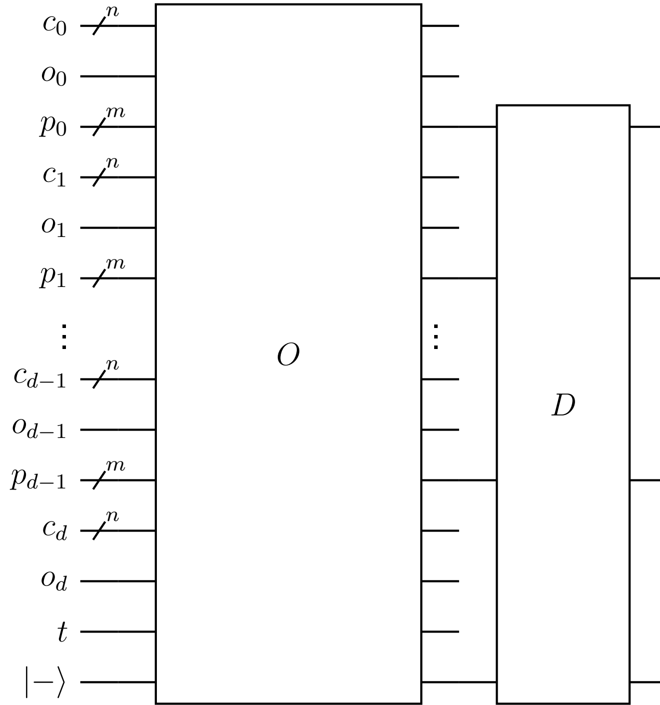


Figure 4.3. *Oracle and diffuser.*

4.2 Details of Running the Algorithm

After the oracle, a diffuser D, which is constructed as explained in section 2.3, is applied to the primitive registers p_i as shown in figure 4.3. Based on b^d and the number v of valid obstacle-free trajectories to a target configuration, the operation of applying O and D is then ideally repeated $T = \left\lfloor \frac{\pi}{4}\sqrt{b^d/v} \right\rfloor$ times based on equation 23 [12][5]. However, although the total number of trajectories b^d is known, the number v of valid obstacle-free trajectories to c_t is usually not known. Because of this, the exact value for T cannot be calculated. The algorithm should therefore be run as described in [41] and below.

1. Set $m = 1$ and $1 < \lambda < \frac{4}{3}$.
2. While $m \leq \sqrt{N}$ repeat steps (i) - (iii).
   (i) Choose an integer $0 < k \leq m$ uniformly at random.
   (ii) Use k as a number of iterations and run the algorithm.
   (iii) If the output is a valid solution, terminate, otherwise set $m \leftarrow \lambda m$.

The time complexity still stays as $O(\sqrt{N/M})$ [41].

The number of P operations used, defines the maximum length of trajectories. If too few P operations are used, it might be that no valid trajectory that reaches c_t exists. However, using too many P operations increases the cost of computation and may lead to unnecessarily long trajectories. Thus, finding the exact number of primitives a trajectory should consist of to reach a target is a problem that needs to be considered. The num-

ber of primitives needed could be estimated in an obstacle-free space by calculating the distance between c_s and c_t configurations. However, obstacles in the environment are affecting this estimate, since going around obstacles increases the number of primitives needed. Some way of calculating an estimate for the number of P operations to apply based on information about target configurations and obstacles is therefore needed, but determining how this should be calculated is left out of the scope of this work. However, one possible solution is to first choose a relatively small number of P operations and if the resulting trajectory from running the algorithm is invalid, the number of P operations could then be increased when running the algorithm again so that there is then a higher probability for a valid trajectory to exist.

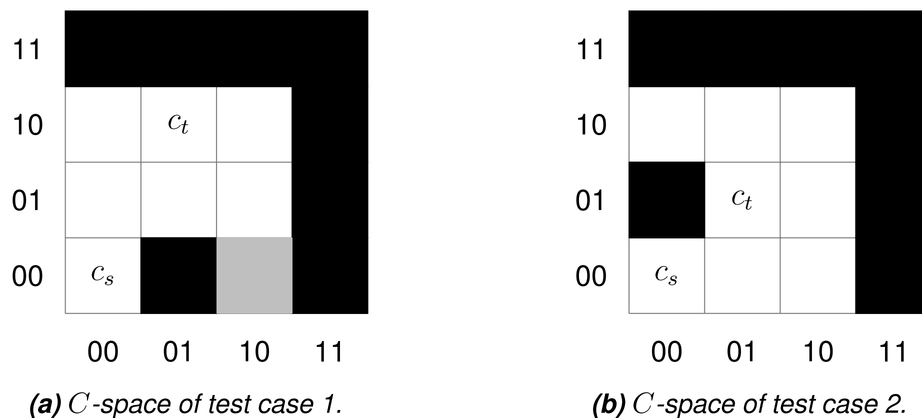
It should be noted, that the set of primitives should also include a null primitive, which is a primitive that does not change the configuration if it is an input to a P operation. Null primitives are needed because the algorithm with d P operations gives as a result only trajectories with exactly d primitives. The c_t however, may be reached also with less than d primitives. If a null primitive is included in a resulting trajectory, it should be dismissed. This ensures that resulting trajectories can consist of less than d non-null primitives. One consequence of this is, that the algorithm is actually more likely to return trajectories with many null primitives. This is because the position of null primitives within the trajectory does not change the actual trajectory followed by a vehicle because null primitives are discarded. If a trajectory consisting of d primitives includes one null primitive, there exists d other trajectories consisting of the same sequence of primitives, except with the null primitive at a different position within the trajectory. Therefore, there is a greater probability that a trajectory including many null primitives is coming as a result, than that a trajectory with no null primitives becomes as a result. This effect might actually be preferred because fewer non-null primitives suggest a more optimal trajectory. It is, however, not guaranteed, that fewer non-null primitives equal a more optimal trajectory.

So far, this algorithm does not provide any guarantee about the optimality of the resulting trajectory, although null primitives explained previously may somewhat improve the optimality. The result of the algorithm is a random valid trajectory with equal probability for all valid trajectories. The optimality of the result can be increased by running the algorithm multiple times as presented in [42]. After the first valid solution is found, a cost is calculated for it. A cost could be, for example, a distance or time it takes to traverse the trajectory. This cost is then used as a baseline and C operation is altered so, that it compares costs for all solution trajectories to this baseline and only marks trajectories that have lower costs than the baseline. The algorithm is then run again with this modified C operation and the same inputs as in first time. This time, the result will be a valid trajectory with a smaller cost than what the original result had if one exists. This can then be repeated again, using the cost of the most recent result trajectory as a baseline. This way, a more optimal solution could be found.

5. IMPLEMENTATION OF THE ALGORITHM

In this chapter, the circuit implementation of P and C operations from chapter 4 is explained. The algorithm is also implemented using Qiskit, which is an open-source framework for constructing and working on quantum circuits with Python [43]. The algorithm is then tested with a matrix product state simulator and IBM Falcon quantum processor.

Because simulating a quantum computer on a classical computer is expensive and because the Falcon processors cannot perform very complicated computations, the problem needs to be simplified for testing. The C-spaces for two test cases are presented in figure 5.1. The C-space is a 4 by 4 grid with x- and y-coordinates as parameters, both represented with two bits as shown in figure 5.1. The configurations are written so that the two most significant bits represent the x-coordinate and the two least significant bits represent the y-coordinate so that for example the target configuration c_t in test case 1 is 0110. Configurations filled with black are obstacles.



(a) *C-space of test case 1.* ***(b)*** *C-space of test case 2.*

Figure 5.1. *C-spaces of simulation test cases.*

Primitives are represented with two bits and are shown in figure 5.2 as when starting from the highlighted configuration marked with black borders, primitive is written to the resulting configuration. Primitives are chosen to be so that 00 is null primitive, 01 is moving left, 10 is moving right and 11 is moving up. There is no down primitive as only 4 primitives could be represented with two bits and the null primitive was chosen to be included in the set of primitives. As a result, in the figure 5.1a, the configuration filled with gray cannot be achieved when starting from c_s in the figure.

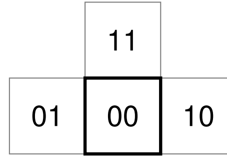


Figure 5.2. *Primitives of a simplified problem.*

The simplicity of a resulting quantum circuit is further enhanced by omitting the checking of movements, that are going outside the C-space. To account for this, obstacles are placed in each configuration where x- or y-coordinate is 11, as shown in figure 5.1. This prevents moving from the edge of the C-space to the opposite edge with one primitive.

This simplified version of the problem becomes a path planning problem as no velocities are considered. It is the same problem as in [40]. However, there are some differences in the implementation, most notably a difference in how collision checking is implemented and a different set of primitives is used.

5.1 Circuit Implementation of the Algorithm

The circuit implementation of a P operation is shown in figure 5.5. The circuit is divided into sections in the figure so that the first four sections are for applying the four primitives and the last section is for collision checking. Quantum registers are mapped so that the least significant qubit is at the top. The circuit is designed using $MCNOT$ gates as presented in [44] stage 1.

The circuit for applying primitives is presented in the four first sections of figure 5.5. As an example, applying primitive 01, which is the second section in the figure, is explained here. Qubits of output configuration register c_{out} are target qubits and are controlled by qubits of input configuration c_{in} and primitive p registers. Register c_{out} is always initialized to state $|0\rangle$ before P operation. For each $MCNOT$ gate in the second section, primitive qubits activate controls when in state $|01\rangle$. As primitive 01 does not change the y-coordinate, qubits c_{out0} and c_{out1} are set to the same state as c_{in0} and c_{in1} correspondingly. Qubit c_{out2} is set to inverse of state c_{in2}. Qubit c_{out3} is set to $|1\rangle$ if $|c_{in2}\rangle = |c_{in3}\rangle$.

After the output configuration is calculated, the collision checking is done as shown in the last section of figure 5.5. First o_{out} is set to $|1\rangle$ if $|o_{in}\rangle = |1\rangle$. For each obstacle one $MCNOT$ gate is then applied where controls are set based on the obstacle's x- and y-coordinates. For example, the second gate in the collision checking section is for checking if the output configuration is 0100. The circuit in figure 5.5 is constructed based on obstacles in test case 1 shown in figure 5.1a.

The C operations for both test cases are shown in figure 5.3. Because in this case there is only one target configuration, the C operation consists of only one $MCNOT$ gate.

Therefore, there is no need for an additional $CNOT$ gate as is presented in figure 4.2. The same effect on input qubits is achieved with just the C operation when t qubit is initialized to $|-\rangle$.

(a) *Test case 1 C operation.* ***(b)*** *Test case 2 C operation.*

Figure 5.3. *Circuits of C operations.*

Diffuser D is implemented as shown in figure 2.5 with $2d$ control qubits as all the primitive qubits are fed as an input. The circuit of $P^\dagger$ is constructed by applying the gates from P in inverted order. With these operations implemented, the complete circuit can be constructed as shown in figure 4.2.

5.2 Test Results

The algorithm for the simplified problem was implemented with Qiskit and the code is shown in appendix A. The algorithm was run on a matrix product state simulator [45] and IBM Falcon quantum processor. The simulator simulated an ideal quantum computer with no noise. The test results are presented in this section.

Simulation test case 1 is presented in figure 5.1a. Starting configuration $c_s = 0000$, $c_t = 0110$, $C_{obs} = \{0100, 0011, 0111, 1011, 1111, 1110, 1101, 1100\}$ and number of used primitives for each path (number of P operations) $d = 3$. The number of iterations t of how many times O and D were applied was adjusted to see the effect it has on the performance of the algorithm. With each value of t, the algorithm was run $10\,000$ times. Results of how many times the algorithm returned a valid obstacle-free path to c_t can be seen in figure 5.4. As there are two valid obstacle-free paths from c_s to c_t with three primitives and the number of all different sets of primitives is 4^3, the optimal number for t based on equation 23 is $\left\lfloor \frac{\pi}{4}\sqrt{4^3/2} \right\rfloor = 4$. As can be seen from the figure 5.4, when using the optimal number of iterations $t = 4$, only 4 of the resulting paths were invalid. With $t = 3$ and $t = 5$, the result was still a valid path in over $85\ \%$ of runs. When $t = 2$ or $t = 6$, the percentage of valid results decreased to under $60\ \%$.

The C-space in simulation test case 2 is shown in figure 5.1b. In this case $c_s = 0000$, $c_t = 0101$ and $C_{obs} = \{0001, 0011, 0111, 1011, 1111, 1110, 1101, 1100\}$. The number of primitives for paths was set to $d = 4$. There are 4 valid paths with length 4 and 1 valid path with length 2. As the path with length 2 must include two null primitives when using

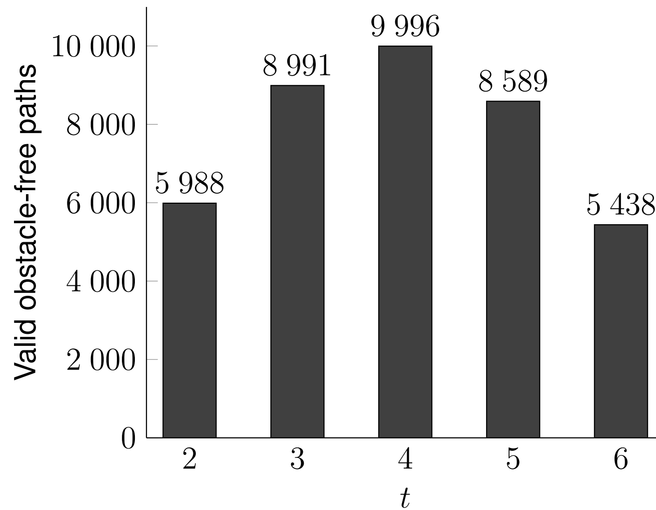


Figure 5.4. *Simulation test case 1 results.*

a total of four primitives, there are 6 unique sequences of primitives, which result in this path. Therefore, there should be a greater probability of the result having length 2, than length 4. The number of iterations used was $t = \left\lfloor \frac{\pi}{4}\sqrt{4^4/10} \right\rfloor = 3$. The algorithm was run $10\ 000$ times and result counts can be seen in table 5.1, where n is the number of times the primitive sequence came as a result. From table 5.1a it can be seen, that each valid set of primitives was a result with roughly equal probability. Therefore, the resulting path had length 2 with greater probability than length 4, as can be seen from table 5.1b.

Table 5.1. *Simulation test case 2 results.*

(a) *Valid sets of primitives.*

primitives	n	primitives	n
$10, 10, 01, 11$	952	$00, 10, 00, 11$	992
$10, 01, 10, 11$	971	$10, 00, 00, 11$	995
$10, 10, 11, 01$	946	$00, 10, 11, 00$	955
$10, 11, 10, 01$	961	$10, 00, 11, 00$	985
$00, 00, 10, 11$	1 024	$10, 11, 00, 00$	926

(b) *Valid paths by length.*

path length	n
2	5 877
4	3 830
any	9 707

The algorithm was also tested on a quantum computer. The processor used in tests was *ibm_algiers*, which is one of IBM Falcon processors. It was possible to run only the simplest form of the problem with $c_s = 0000$, $c_t = 0001$, $d = 1$, $t = 1$, and no obstacles. Measurement results were completely random and the correct path was not coming as a result any more than any other result, so even this simplest form of a problem turned out to be too complex. This is probably due to the circuit containing too many gates so that the computation takes too long. The quantum state of the system is therefore affected by random noise at the time it takes for the computation to finish, which destroys the desired quantum state.

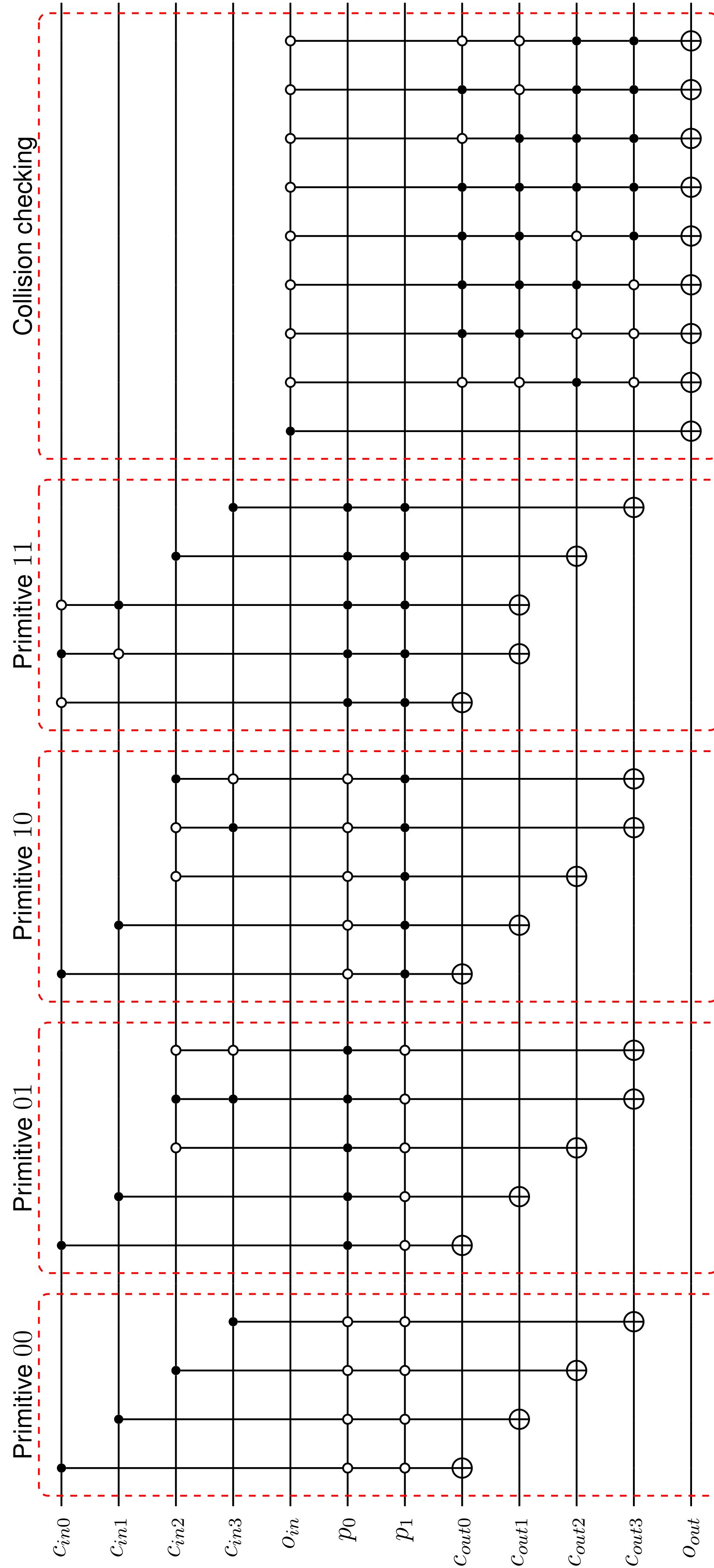


Figure 5.5. *Circuit of a P operation.*

6. CONCLUSION

This work presented an algorithm for solving trajectory planning problem with quantum computing. The algorithm was implemented for a simplified problem and tested with a QC simulator and with quantum computer.

Simulation tests showed promising results. The algorithm was able to solve the simplified problem as designed in different test cases. However, as QC simulation is a costly task for classical computers, it was not possible to evaluate how the algorithm would perform when the complexity increases.

Tests on real quantum hardware showed not that good results. It was not possible to solve even a simple problem with the designed algorithm. This was probably because the resulting quantum circuit was too complex so the computation took too much time and the quantum computer used for testing was not able to sustain the quantum state uninterrupted for this long. There need to be major advancements in QC hardware before the algorithm has the possibility to be useful in real-world applications. When the QC hardware improves, further tests could be carried out to see how the algorithm is performing.

Further work could be done for optimizing the construction of a quantum circuit the algorithm produces. This would improve the performance of the algorithm, especially when the complexity of the problem increases. This work did not discuss about how to design and choose the primitive trajectories used for creating a state lattice. This would also be a topic for further research, as the choice of primitives has a major impact on the resulting trajectories.

REFERENCES

[1] Richard P. Feynman. "Simulating Physics with Computers". In: *International journal of theoretical physics* 21.6-7 (1982), pp. 467–488. ISSN: 0020-7748. DOI: 10.1007/BF02650179.

[2] Seth Lloyd. "Universal Quantum Simulators". In: *Science* 273.5278 (Aug. 1996), p. 1073. ISSN: 00368075.

[3] R. Horodecki, S. Ya. Kilin, and J. Kowalik. *Quantum Cryptography and Computing: Theory and Implementation*. IOS Press, Incorporated, 2010. ISBN: 978-1-60750-547-1.

[4] Kishor Bharti et al. "Noisy Intermediate-Scale Quantum Algorithms". In: *Reviews of Modern Physics* 94.1 (Feb. 2022), p. 015004. ISSN: 0034-6861, 1539-0756. DOI: 10.1103/RevModPhys.94.015004.

[5] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge ; New York: Cambridge University Press, 2000. ISBN: 978-0-521-63235-5 978-0-521-63503-5.

[6] P. a. M. Dirac. "A New Notation for Quantum Mechanics". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 35.3 (July 1939), pp. 416–418. ISSN: 1469-8064, 0305-0041. DOI: 10.1017/S0305004100021162.

[7] Eleanor G. Rieffel et al. *Quantum Computing: A Gentle Introduction*. Cambridge, UNITED STATES: MIT Press, 2011. ISBN: 978-0-262-29539-0.

[8] Mika Hirvensalo, Th. Bäck, and A. E. Eiben. *Quantum Computing*. Berlin, Heidelberg, GERMANY: Springer Berlin / Heidelberg, 2001. ISBN: 978-3-662-04461-2.

[9] Ryszard Horodecki et al. "Quantum Entanglement". In: *Reviews of Modern Physics* 81.2 (June 2009), pp. 865–942. DOI: 10.1103/RevModPhys.81.865.

[10] Juan Yin et al. "Bounding the Speed of 'spooky Action at a Distance'". In: *Physical Review Letters* 110.26 (June 2013), p. 260407. ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.110.260407. arXiv: 1303.0614 [quant-ph].

[11] Lov K. Grover. "A Fast Quantum Mechanical Algorithm for Database Search". In: *arXiv.org* (Nov. 1996).

[12] Christof Zalka. "Grover's Quantum Searching Algorithm Is Optimal". In: *Physical Review A* 60.4 (Oct. 1999), pp. 2746–2751. ISSN: 1050-2947, 1094-1622. DOI: 10.1103/PhysRevA.60.2746. arXiv: quant-ph/9711070.

[13] Raphael Seidel et al. *Automatic Generation of Grover Quantum Oracles for Arbitrary Data Structures*. Oct. 2021. DOI: 10.48550/arXiv.2110.07545. arXiv: 2110.07545 [quant-ph].

[14] P. W. Shor. "Scheme for Reducing Decoherence in Quantum Computer Memory". In: *Physical review. A, Atomic, molecular, and optical physics* 52.4 (1995), R2493–R2496. ISSN: 1050-2947. DOI: 10.1103/PhysRevA.52.R2493.

[15] Andrew Steane. "Multiple Particle Interference and Quantum Error Correction". In: *arXiv.org* (May 1996). DOI: 10.1098/rspa.1996.0136.

[16] E. Knill et al. "Benchmarking Quantum Computers: The Five-Qubit Error Correcting Code". In: *Physical Review Letters* 86.25 (June 2001), pp. 5811–5814. DOI: 10.1103/PhysRevLett.86.5811.

[17] Mark Horowitz and Emily Grumbling. *Quantum Computing: Progress and Prospects*. Washington, D.C., UNITED STATES: National Academies Press, 2019. ISBN: 978-0-309-47970-7.

[18] C. Monroe and J. Kim. "Scaling the Ion Trap Quantum Processor". In: *Science* 339.6124 (Mar. 2013), pp. 1164–1169. DOI: 10.1126/science.1231298.

[19] J. I. Cirac and P. Zoller. "Quantum Computations with Cold Trapped Ions". In: *Physical Review Letters* 74.20 (May 1995), pp. 4091–4094. DOI: 10.1103/PhysRevLett.74.4091.

[20] D. Kielpinski, C. Monroe, and D. J. Wineland. "Architecture for a Large-Scale Ion-Trap Quantum Computer". In: *Nature* 417.6890 (June 2002), pp. 709–711. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature00784.

[21] Bjoern Lekitsch et al. "Blueprint for a Microwave Trapped Ion Quantum Computer". In: *Science Advances* 3.2 (Feb. 2017), e1601540. DOI: 10.1126/sciadv.1601540.

[22] He-Liang Huang et al. *Superconducting Quantum Computing: A Review*. Nov. 2020. DOI: 10.48550/arXiv.2006.10433. arXiv: 2006.10433 [quant-ph].

[23] Anton Frisk Kockum and Franco Nori. "Quantum Bits with Josephson Junctions". In: *Fundamentals and Frontiers of the Josephson Effect*. Ed. by Francesco Tafuri. Springer Series in Materials Science. Cham: Springer International Publishing, 2019, pp. 703–741. ISBN: 978-3-030-20726-7. DOI: 10.1007/978-3-030-20726-7_17.

[24] Howie Choset et al. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. Cambridge, UNITED STATES: MIT Press, 2005. ISBN: 978-0-262-25591-2.

[25] Eugene Kagan. *Autonomous Mobile Robots and Multi-Robot Systems: Motion-Planning, Communication and Swarming*. 1st edition. Hoboken, New Jersey ; Wiley, 2020. ISBN: 978-1-119-21316-1.

[26] Yadollah Rasekhipour et al. "A Potential Field-Based Model Predictive Path-Planning Controller for Autonomous Road Vehicles". In: *IEEE Transactions on Intelligent Transportation Systems* 18.5 (May 2017), pp. 1255–1267. ISSN: 1558-0016. DOI: 10.1109/TITS.2016.2604240.

[27] Steven M. LaValle. "Rapidly-Exploring Random Trees: A New Tool for Path Planning". In: (1998).

[28] Mansoor Davoodi et al. “Multi-Objective Path Planning in Discrete Space”. In: *Applied Soft Computing* 13.1 (Jan. 2013), pp. 709–720. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2012.07.023.

[29] David González et al. “A Review of Motion Planning Techniques for Automated Vehicles”. In: *IEEE Transactions on Intelligent Transportation Systems* 17.4 (Apr. 2016), pp. 1135–1145. ISSN: 1558-0016. DOI: 10.1109/TITS.2015.2498841.

[30] Zvi Shiller and Hsueh-Hen Lu. “Computation of Path Constrained Time Optimal Motions With Dynamic Singularities”. In: *Journal of Dynamic Systems, Measurement, and Control* 114.1 (Mar. 1992), pp. 34–40. ISSN: 0022-0434. DOI: 10.1115/1.2896505.

[31] J.-J. E. Slotine and H. S. Yang. “Improving the Efficiency of Time-Optimal Path-Following Algorithms”. In: *1988 American Control Conference*. June 1988, pp. 2129–2134. DOI: 10.23919/ACC.1988.4790076.

[32] Steven M. LaValle and James J. Kuffner. “Randomized Kinodynamic Planning”. In: *The International Journal of Robotics Research* 20.5 (May 2001), pp. 378–400. ISSN: 0278-3649. DOI: 10.1177/02783640122067453.

[33] Maxim Likhachev and Dave Ferguson. “Planning Long Dynamically Feasible Maneuvers for Autonomous Vehicles”. In: *The International journal of robotics research* 28.8 (2009), pp. 933–945. ISSN: 0278-3649. DOI: 10.1177/0278364909340445.

[34] Mihail Pivtoraiko and Alonzo Kelly. “Efficient Constrained Path Planning via Search in State Lattices”. In: *International Symposium on Artificial Intelligence, Robotics, and Automation in Space*. Munich Germany, 2005, pp. 1–7.

[35] Thomas M. Howard et al. “State Space Sampling of Feasible Motions for High-Performance Mobile Robot Navigation in Complex Environments”. In: *Journal of Field Robotics* 25.6-7 (2008), pp. 325–345. ISSN: 1556-4967. DOI: 10.1002/rob.20244.

[36] Matthew McNaughton et al. “Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice”. In: *2011 IEEE International Conference on Robotics and Automation*. May 2011, pp. 4889–4895. DOI: 10.1109/ICRA.2011.5980223.

[37] Julius Ziegler and Christoph Stiller. “Spatiotemporal State Lattices for Fast Trajectory Planning in Dynamic On-Road Driving Scenarios”. In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Oct. 2009, pp. 1879–1884. DOI: 10.1109/IROS.2009.5354448.

[38] Aleksandr Kushleyev and Maxim Likhachev. “Time-Bounded Lattice for Efficient Planning in Dynamic Environments”. In: *2009 IEEE International Conference on Robotics and Automation*. May 2009, pp. 1662–1668. DOI: 10.1109/ROBOT.2009.5152860.

[39] Luís Tarrataca and Andreas Wichert. “Problem-Solving and Quantum Computation”. In: *Cognitive Computation* 3.4 (Dec. 2011), pp. 510–524. ISSN: 1866-9964. DOI: 10.1007/s12559-011-9103-6.

[40] Antonio Chella et al. “A Quantum Planner for Robot Motion”. In: *Mathematics (Basel)* 10.14 (2022), pp. 2475–. ISSN: 2227-7390. DOI: 10.3390/math10142475.

[41] Michel Boyer et al. “Tight Bounds on Quantum Searching”. In: *Fortschritte der Physik* 46.4-5 (1998), pp. 493–505. ISSN: 0015-8208. DOI: 10.1002/(SICI)1521-3978(199806)46:4/5<493::AID-PROP493>3.0.CO;2-P.

[42] W. P. Baritompa, D. W. Bulger, and G. R. Wood. “Grover’s Quantum Algorithm Applied to Global Optimization”. In: *SIAM Journal on Optimization* 15.4 (2005), p. 15. ISSN: 10526234. DOI: 10.1137/040605072.

[43] A-tA-v et al. *Qiskit: An Open-Source Framework for Quantum Computing*. 2021. DOI: 10.5281/zenodo.2573505.

[44] Ahmed Younes and Julian Miller. *Automated Method for Building CNOT Based Quantum Circuits for Boolean Functions*. Apr. 2003. DOI: 10.48550/arXiv.quant-ph/0304099. arXiv: quant-ph/0304099.

[45] Guifre Vidal. “Efficient Classical Simulation of Slightly Entangled Quantum Computations”. In: *Physical Review Letters* 91.14 (Oct. 2003), p. 147902. ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.91.147902. arXiv: quant-ph/0301063.

APPENDIX A: CODE FOR IMPLEMENTING THE QUANTUM CIRCUIT WITH QISKIT

```
from qiskit import *

def p_operation(obstacles):
    """
    Constructs a circuit of a P operation

    :param obstacles: list[string], list of obstacles as a string
   of 0s and 1s
    :return: QuantumCircuit, P operation circuit
    """

    p = QuantumCircuit(name='P')

    # initial configuration
    in_configuration = QuantumRegister(4, 'c_in')
    # collision in qubit, 1 if trajectory has collided before the
   gate, 0 otherwise
    in_collision = QuantumRegister(1, 'o_in')
    # primitive trajectory to apply
    primitive = QuantumRegister(2, 'p')
    # configuration after traversing the primitive
    out_configuration = QuantumRegister(4, 'c_out')
    # collision out qubit, 1 if trajectory has collided after the
   gate, 0 otherwise
    out_collision = QuantumRegister(1, 'o_out')

    p.add_register(in_configuration)
    p.add_register(in_collision)
    p.add_register(primitive)
    p.add_register(out_configuration)
    p.add_register(out_collision)

    # primitive 00
    p.x(primitive)
    p.mcx([in_configuration[0], primitive[0], primitive[1]],
   out_configuration[0])
```

```
33      p.mcx([in_configuration[1], primitive[0], primitive[1]],
        out_configuration[1])
34      p.mcx([in_configuration[2], primitive[0], primitive[1]],
        out_configuration[2])
35      p.mcx([in_configuration[3], primitive[0], primitive[1]],
        out_configuration[3])
36      p.x(primitive)
37
38      # primitive 01
39      p.x(primitive[1])
40      p.mcx([in_configuration[0], primitive[0], primitive[1]],
        out_configuration[0])
41      p.mcx([in_configuration[1], primitive[0], primitive[1]],
        out_configuration[1])
42      p.x(in_configuration[2])
43      p.mcx([in_configuration[2], primitive[0], primitive[1]],
        out_configuration[2])
44      p.x(in_configuration[2])
45      p.mcx([in_configuration[2], in_configuration[3], primitive[0],
        primitive[1]], out_configuration[3])
46      p.x(in_configuration[2])
47      p.x(in_configuration[3])
48      p.mcx([in_configuration[2], in_configuration[3], primitive[0],
        primitive[1]], out_configuration[3])
49      p.x(in_configuration[2])
50      p.x(in_configuration[3])
51      p.x(primitive[1])
52
53      # primitive 10
54      p.x(primitive[0])
55      p.mcx([in_configuration[0], primitive[0], primitive[1]],
        out_configuration[0])
56      p.mcx([in_configuration[1], primitive[0], primitive[1]],
        out_configuration[1])
57      p.x(in_configuration[2])
58      p.mcx([in_configuration[2], primitive[0], primitive[1]],
        out_configuration[2])
59      p.mcx([in_configuration[2], in_configuration[3], primitive[0],
        primitive[1]], out_configuration[3])
60      p.x([in_configuration[2], in_configuration[3]])
61      p.mcx([in_configuration[2], in_configuration[3], primitive[0],
        primitive[1]], out_configuration[3])
62      p.x(in_configuration[3])
63      p.x(primitive[0])
64
65      # primitive 11
66      p.x(in_configuration[0])
67      p.mcx([in_configuration[0], primitive[0], primitive[1]],
```

```
    out_configuration[0])
68      p.mcx([in_configuration[0], in_configuration[1], primitive[0],
    primitive[1]], out_configuration[1])
69      p.x([in_configuration[0], in_configuration[1]])
70      p.mcx([in_configuration[0], in_configuration[1], primitive[0],
    primitive[1]], out_configuration[1])
71      p.x(in_configuration[1])
72      p.mcx([in_configuration[2], primitive[0], primitive[1]],
    out_configuration[2])
73      p.mcx([in_configuration[3], primitive[0], primitive[1]],
    out_configuration[3])
74
75      # check collisions
76      p.cx(in_collision, out_collision)
77      p.x(in_collision)
78      for obs in obstacles:
79          for i in range(len(obs)):
80              if obs[-i-1] == '0':
81                  p.x(out_configuration[i])
82          p.mcx([in_collision, out_configuration[0],
    out_configuration[1], out_configuration[2],
    out_configuration[3]], out_collision)
83          for i in range(len(obs)):
84              if obs[-i-1] == '0':
85                  p.x(out_configuration[i])
86      p.x(in_collision)
87
88      return p
89
90  def c_operation(target_configuration):
91      """
92      Constructs a circuit of a C operation
93
94      :param target_configuration: string, target configuration as a
    string of 0s and 1s
95      :return: QuantumCircuit, C operation circuit
96      """
97
98      c = QuantumCircuit(name='C')
99
100     # configuration
101     configuration = QuantumRegister(4, 'c')
102     # collision qubit
103     collision = QuantumRegister(1, 'o')
104     # result = 1 if input configuration equals the target
    configuration, 0 otherwise
105     result = QuantumRegister(1, 't')
106
```

```
107     c.add_register(configuration)
108     c.add_register(collision)
109     c.add_register(result)
110 
111     # apply NOT to each 0 qubit in target configuration for MCNOT
        gate
112     for i in range(len(target_configuration)):
113         if target_configuration[-i-1] == '0':
114             c.x(configuration[i])
115 
116     # collision qubit is 0 if trajectory has not collided
117     c.x(collision)
118 
119     c.mcx([configuration[0], configuration[1], configuration[2],
        configuration[3], collision], result)
120 
121     c.x(collision)
122 
123     # uncompute the first step
124     for i in range(len(target_configuration)):
125         if target_configuration[-i-1] == '0':
126             c.x(configuration[i])
127 
128     return c
129 
130 def diffuser(depth):
131     """
132     Constructs a circuit of a diffuser
133 
134     :param depth: int, depth of the tree graph i.e. how many
        primitives are applied
135     :return: QuantumCircuit, diffuser circuit
136     """
137 
138     d = QuantumCircuit(2*depth + 1, name='D')
139 
140     d.h(range(2*depth))
141     d.x(range(2*depth))
142     d.mcx(list(range(2*depth)), 2*depth)
143     d.x(range(2*depth))
144     d.h(range(2*depth))
145 
146     return d
147 
148 def circuit(start_configuration, target_configuration, obstacles,
        depth, iterations):
149     """
150     Constructs the full circuit for the algorithm
```

```
151
152         :param start_configuration: string, start configuration as a
        string of 0s and 1s
153         :param target_configuration: string, target configuration as a
        string of 0s and 1s
154         :param obstacles: list[string], list of obstacles as a string
        of 0s and 1s
155         :param depth: int, depth of the tree graph i.e. how many
        primitives are applied
156         :param iterations: int, number of iterations to apply the
        amplitude amplification
157         :return: QuantumCircuit, full circuit
158         """
159
160         qc = QuantumCircuit()
161
162         # start configuration
163         start_configuration_register = QuantumRegister(4, 'c_0')
164         qc.add_register(start_configuration_register)
165         # collision in for the first P gate
166         collision_0 = QuantumRegister(1, 'o_0')
167         qc.add_register(collision_0)
168
169         # other registers for P gates
170         primitive_registers = []
171         for j in range(depth):
172             primitive = QuantumRegister(2, 'p_' + str(j))
173             primitive_registers.append(primitive)
174             configuration = QuantumRegister(4, 'c_' + str(j+1))
175             collision = QuantumRegister(1, 'o_' + str(j+1))
176
177             qc.add_register(primitive)
178             qc.add_register(configuration)
179             qc.add_register(collision)
180
181         # result qubit for C gate
182         result = QuantumRegister(1, 't')
183         qc.add_register(result)
184
185         # set initial states according to the start configuration
186         for i in range(len(start_configuration)):
187             if start_configuration[-i-1] == '1':
188                 qc.x(start_configuration_register[i])
189
190         # apply hadamard gate to primitive qubits
191         for primitive in primitive_registers:
192             qc.h(primitive)
193
```

```
194     # set initial state to the result qubit
195     qc.x(result)
196     qc.h(result)
197 
198     p = p_operation(obstacles).to_gate()
199     c = c_operation(target_configuration).to_gate()
200     d = diffuser(depth).to_gate()
201 
202     # construct the circuit
203     for j in range(iterations):
204         # apply P gates
205         for j in range(depth):
206             qc.append(p, range(7*j, 7*j+12))
207 
208         # apply C gate
209         qc.append(c, range(7*depth, 7*depth+6))
210 
211         # apply inverse P gates
212         for j in range(depth):
213             qc.append(p.inverse(), range(7*(depth-1-j),
    7*(depth-1-j)+12))
214 
215         # apply diffuser
216         primitive_qubits = []
217         for j in range(depth):
218             primitive_qubits.append(5+j*7)
219             primitive_qubits.append(6+j*7)
220         qc.append(d, primitive_qubits)
221 
222     # set result qubit back to 0
223     qc.h(result)
224     qc.x(result)
225 
226     return qc
```