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

Application of Quantum Computing in Bioinformatics

Master's Thesis

Author
Viktória Nemkin

Advisor
dr. Katalin Friedl

December 18, 2022

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Nemkin Viktória*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2022. december 18.

Nemkin Viktória
hallgató

Kivonat

A bioinformatika egy interdiszciplináris terület az informatika és a biológia határán. Fő célja biológiai kérdések megválaszolása, melynek első lépése, hogy azokat számítási problémákká alakítja, majd hatékony algoritmikus megoldásokat kínál rájuk. A bioinformatika kutatásának jelentős hatása van mindennapi életünkre, hiszen az ezen a területen elért eredmények segíthetnek megoldani napjaink jelentős globális problémáit: például lehetővé tehetik a hatékonyabb gyógyszerek felfedezését, a különböző genetikai betegségek megértését, olyan ellenálló termények kifejlesztését melyek hozzájárulhatnak a globális éhínség enyhítéséhez vagy új technológiák feltalálását a környezetszennyezés csökkentésére.

Sajnos a bioinformatika számos gyakorlati fontossággal bíró problémája algoritmuselméleti szempontból nehéz feladatnak bizonyul a klasszikus számítógépeken. Minden kutatási erőfeszítés ellenére ezekre a problémákra a mai napig nem sikerült kellően gyors, determinisztikus megoldást találni.

A kvantuminformatika az algoritmuselméleti kutatás szempontjából egy érdekes terület, hiszen a kvantumszámítógépek működése alapvetően más, mint a klasszikusaké, aminek az a következménye, hogy a klasszikus bonyolultságelmélet a kvantumvilágban másképpen alakul. Az egyik leghíresebb példa erre Shor prímfaktorizációs algoritmusa, amely egy nagy méretű kvantumszámítógépen gyorsan feltörhetné a ma elterjedt titkosításokat.

Ennek köszönhetően az elmúlt években fokozott figyelem irányult a kvantuminformatika területére mind különböző kormányzati szervek, mind globális nagyvállalatok részéről, akik jelentős támogatással szálltak be az új típusú, kvantumfizikai jelenségeken alapuló hardverek tervezésébe és a rajtuk futó szoftverek fejlesztésébe. Az Európai Bizottság például 2018-ban elindította a Quantum Flagship nevű nagyszabású, 10 évre szóló, 1 milliárd eurós finanszírozású kezdeményezését a kvantuminformatikai kutatások finanszírozására, amely a Horizont Európa keretében fut.

Bár a kvantuminformatika még mindig gyerekcipőben jár és a kvantumhardverek határai még nem ismertek, ez az újfajta számítási modell már sok új elméleti felfedezést tett lehetővé mind a klasszikus, mind a kvantumvilágban.

Dolgozatomban ismertetek néhány bioinformatikai problémát, különös tekintettel a fehérjehajtogatás feladatára és elmagyarázom Grover keresési algoritmusával és a kvantum sétákkal való kapcsolatát. Ezek után bemutatom a kvantuminformatika eszköztárát, ezen belül a kvantumsétákat és a Grover-féle keresőalgoritmust. Ismertetem a kvantumséták matematikai alapjait, olyan formában, mely a szakirodalomban kevésbé gyakori, melyből a megvalósítás természetes módon következik, majd az általam írt vizualizációs szoftver segítségével demonstrálom a tulajdonságaikat. Áttekintést adok a kvantumalgoritmusokkal való gyakorlati kísérletezés aktuális problémáiról, konkrétan a kvantum fehérjehajtogatással kapcsolatban, majd megtervezek és megvalósítok egy olyan keretrendszert, amely csökkentheti ezen problémákat.

Abstract

Bioinformatics is an interdisciplinary field between computer science and biology. Its main goal is to answer biological questions by transforming them into computational problems and providing efficient algorithmic solutions. Researching bioinformatics significantly impacts our everyday lives, as discoveries in this field could help us solve many of today's major global problems. Utilizing them, we could create novel medical treatments, advance our understanding of genetic diseases, develop resistant crops to tackle a global food crisis or invent new technologies to decrease environmental pollution.

Unfortunately, many practical problems in bioinformatics turn out to be computationally hard ones on classical hardware. Despite tremendous research effort to date, no sufficiently fast, deterministic solutions have been found to these problems.

Quantum informatics is a compelling field for algorithmic research since quantum computers work fundamentally differently from classical ones, which means that the already established classical problem complexities are different in the quantum world. One of the most famous examples is Shor's prime factorization algorithm, which could break modern-day encryption quickly on a large-scale quantum computer.

Due to this, governmental entities and global corporations are paying increased attention to quantum computing, and they are investing in quantum hardware and software development. For example, the European Commission started Quantum Flagship in 2018, a large-scale 10-year initiative with €1 billion in funding for quantum research, currently running under the Horizon Europe initiative.

While quantum computing is still in its early stages and the limits of quantum hardware are yet unknown, the availability of a different computational model has already made new theoretical discoveries possible in both the classical and the quantum worlds.

In my dissertation, I present a general overview of some computational problems in bioinformatics, particularly protein folding, and explain its connection to Grover's search algorithm and quantum walks. Then, I introduce the toolkit of quantum computation, specifically quantum walks and Grover's search algorithm. I present the mathematical framework for quantum walks, formulated in a way which is less common in literature, from which implementation follows naturally, and demonstrate their characteristics using the visualization software I have written. I describe the current practical problems with experimenting on quantum algorithms, specifically quantum protein folding, then design and implement a framework which can reduce some of these issues.

Chapter 1

Introduction

Richard Feynman originally suggested the idea of a quantum computer in a 1982 article[13], demonstrating that current computers experience an exponential slowdown when simulating quantum physical systems. In this paper, he proposes using a different type of computer: one that runs according to the laws of quantum physics, to simulate quantum physical systems without a slowdown.[21]

1.1 Computational models and the P versus NP problem

The mathematical computational model of classical computers is the Turing machine. The laws governing quantum computers are so fundamentally different from their classical relatives that they required defining a new model type. Following the work of many computer scientists (Benioff[4], Deutsch[11], and Bernstein and Vazirani[6]), the computational model for quantum computers was born in the late 1980s: the Quantum Turing machine.

In the past century researchers have been working on solving all kinds of classical algorithmic problems with critical real-life applications. They have defined various time complexity classes depending on how fast the current solutions are. Usually, the class P is considered fast enough.

For some problems, despite tremendous effort, researchers have yet to come up with a fast solver, however, they have found fast verifier algorithms. This means, that when a potential solution is suggested for a given task, they can quickly check its correctness. A problem like this is part of the complexity class NP.

To turn a fast verifier algorithm into a brute-force solver, we could search through all of the possible solutions - the domain of the problem - and verify all of them until we find one that passes. This runs in $O(N)$ linear time relative to the size of the problem's domain. The question is, can we do something faster? This is one of the famous unsolved Millennium Prize Problems set by the Clay Mathematics Institute a hundred years ago, the P versus NP problem.

In the quantum world, a better method exists for searching a problem's domain, which can do it in $O(\sqrt{N})$ time relative to the size of the domain. This algorithm is called Grover's search. It has also been proven by Bennett, Bernstein, Brassard, and Vazirani, that this is asymptotically tight[5].

1.2 Application of quantum algorithms in bioinformatics

A fascinating area for algorithmic research is bioinformatics, with many real-life applications. I am particularly interested in computer-aided drug design, which involves finding molecules of a particular shape that fit into molecular holes of a particular shape inside the human body. From this context, computational problems such as protein folding[10] and molecular docking[2] arise, and among many other bioinformatical problems, turn out to be NP-hard ones. This means that we have yet to devise efficient solutions to them in the classical world.

In the quantum world, Grover's search algorithm could give us a quadratic speedup over its classical brute-force counterpart. Furthermore, Grover's algorithm is an unstructured database search, meaning it does not rely on any information about the structure of the problem's domain.

For example, in the case of protein folding, the structure of the domain could be a simple rule for transforming one fold to another. We could portray this as a graph: the vertices representing all possible folds and the edges representing which folds transform into which others, based on our devised rule.

To boost Grover's speed even further, we could inject this structural information about the domain into it using the quantum equivalent of random graph walks, quantum walks.

1.3 Contents of this dissertation

The rest of this thesis is structured in the following way: In Chapter 2, I introduce 3 computational problems in bioinformatics: DNA sequencing, protein folding and molecular docking, and explain protein folding's connection to two quantum algorithms. In Chapter 3, I introduce the postulates of quantum mechanics, the basis of quantum computing. In Chapter 4, I discuss quantum random walks, in a bottom-up approach, starting from the simplest form and then generalizing it, and I present my improvemnts on these generalization techniques. In Chapter 5, I describe my simulator software's architecture and implementation details and present the results obtained from my simulation runs. In Chapter 6, I introduce Grover's search algorithm framework and solve a generalized version of the Sudoku puzzle with it. I iterate over the necessary components from this solution, the particular operators needed for the oracle and the amplitude amplification technique's implementation. In Chapter 7, I explain my approach for encoding candidate solutions in quantum protein folding and the memory usage problem I ran into. Then, I lay down the mathematical foundations and architectural design for the memory efficient quantum simulator framework I have written. In Chapter 8, I summarize the results of this dissertation and lay down my plans for the future.

Chapter 2

Bioinformatics

2.1 Computational problems in bioinformatics

In this chapter, I present an overview of some interesting and important bioinformatical problems, particularly protein folding, which is one of the central problems related to drug discovery[7].

2.1.1 DNA sequencing and the Human Genome Project

The complete genetic material of a human being is called the human genome, which is contained in the nucleus of every single cell of the human body and is replicated during cell division. The human genome consists of 46 chromosomes, each containing a single deoxyribonucleic acid, or DNA molecule for short. A DNA molecule is two twisting, paired strands (called the double helix structure) and comprises around 3.2 billion pairs of nucleotide bases. Nucleotides are usually denoted by the letters A, T, C and G, with A-T and C-G forming pairs. A sequence of nucleotides that together encode the synthesis of a specific protein or RNA is called a gene.

Research into the human genome started with the discovery of the double helix structure of the DNA molecule in 1953 by Francis Crick and James Watson. The Human Genome Project[9] was launched in 1990, an international research effort to identify the function of all 50,000 to 100,000 genes of the complete human genetic material and was completed in early 2022[39]. Besides injuries, nearly all human medical conditions are related to mutations at specific locations in the structure and function of the genetic material[9], so identifying changes in an individual's DNA sequence can be an indispensable tool for predicting and preventing disease while providing individualized care. This requires a quick sequencing of specific parts of a person's DNA and comparing them to a healthy variant.

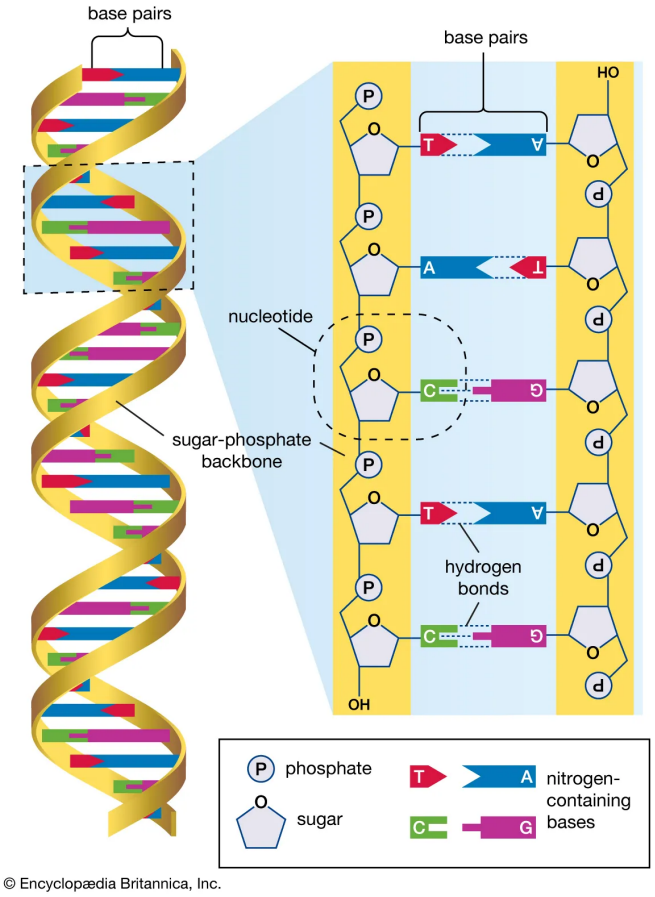


Figure 2.1: *DNA structure[18]*

From a computer science viewpoint, a simplified DNA model is a string consisting of the four letters from one of the strands of the physical molecule. (However, this model omits the 3D spatial structure.) Due to the limits of our technology, current methods enable us to read up to 1000 consecutive letters from a single sample. To sequence a longer string, we can break the molecule into sufficiently smaller fragments and sequence them individually. Unfortunately, in this process, the order in which the fragments originally occurred is lost. One possible approach to deal with this problem is to create copies of the DNA molecule we are interested in and randomly break each of these copies into fragments. With a high probability, the resulting fragments from different copies will overlap with each other. Then, the task becomes reconstructing the original string from these fragments while considering the possible errors in our biological measurements. The possible number of fragment orderings is enormously large, so a brute force algorithm does not suffice for solving this task.[9]

Between any two humans, the amount of genetic variation is estimated to be around 0.6%[35]. This enables us to use the targeted part of the genome sequence preassembled by the Human Genome Project and align our obtained DNA sequence fragments to the best fitting position to obtain the entire string of our DNA. Algorithms such as the Smith-Waterman algorithm and the Needleman-Wunsch algorithm can efficiently perform this sequence alignment task.

2.1.2 Protein folding

Proteins are one of the fundamental building blocks of the human body. They play an essential role in our immune system response, transportation of molecules throughout the body, the catalysation of metabolic reactions and signal transmission. Protein molecules consist of a single, long chain of amino acids. While several amino acid molecules exist in nature, only 20 of those are present in proteins.[7]

The 20 standard amino acids can be seen in Table 2.1. The most important property of these is their affinity to water: Some amino acids are polar (hydrophilic), commonly denoted by a P, which means they can establish hydrogen bonds with H_2O molecules. In contrast, others are hydrophobic (nonpolar), commonly denoted by an H, which do not establish these bonds with water.

Name	Abbreviation	Code	Polarity
Alanine	Ala	A	H
Valine	Val	V	H
Leucine	Leu	L	H
Isoleucine	Ile	I	H
Phenylalanine	Phe	F	H
Proline	Pro	P	H
Methionine	Met	M	H
Tryptophan	Trp	W	H
Arginine	Arg	R	P
Asparagine	Asn	N	P
Aspartic acid	Asp	D	P
Cysteine	Cys	C	P
Glutamic acid	Glu	E	P
Glutamine	Gln	Q	P
Glycine	Gly	G	P
Histidine	His	H	P
Lysine	Lys	K	P
Serine	Ser	S	P
Threonine	Thr	T	P
Tyrosine	Tyr	Y	P

Table 2.1: *Aminoacids in proteins*

The functionality and role of a protein are determined by its spatial structure, of which four levels are distinguished.

Primary structure: The sequence of amino acids in the protein.

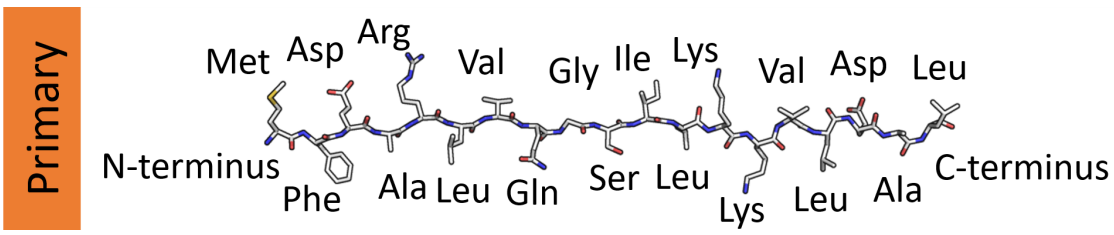


Figure 2.2: *Primary structure of the protein PCNA[33]*

Secondary structure: Created by the interactions between the atoms along the chain of amino acids, forming substructures (α-helices, β-sheets, loops).

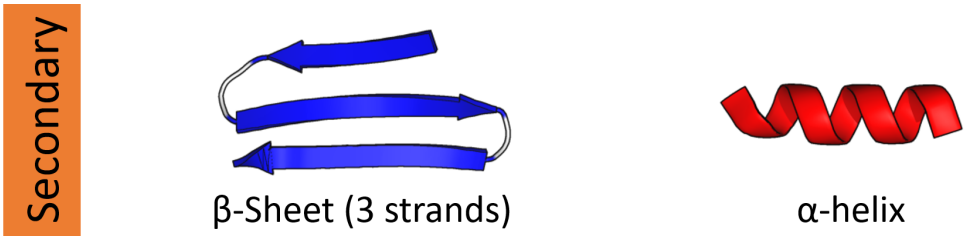


Figure 2.3: *Secondary structure of the protein PCNA[33]*

Tertiary structure: The spatial arrangement of all atoms within the chain. Secondary structure elements are grouped together as motifs, and functional units called domains.

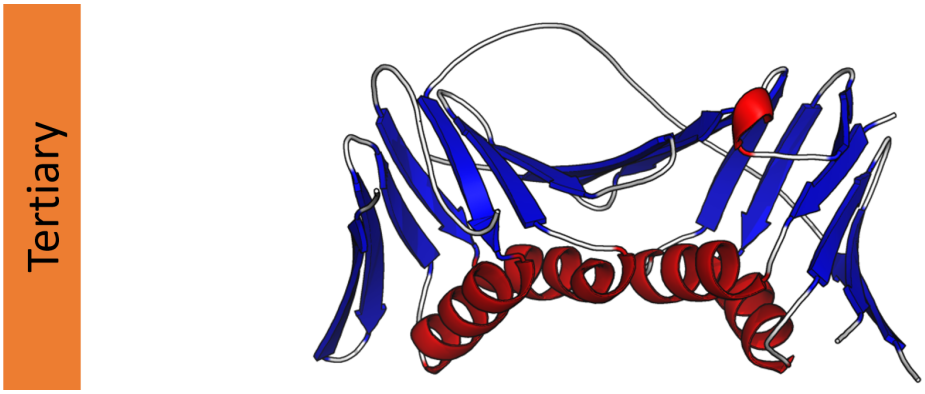


Figure 2.4: *Tertiary structure of the protein PCNA[33]*

Quaternary structure: Describes the composition of the whole protein from polypeptide subunits and potentially other molecular parts.

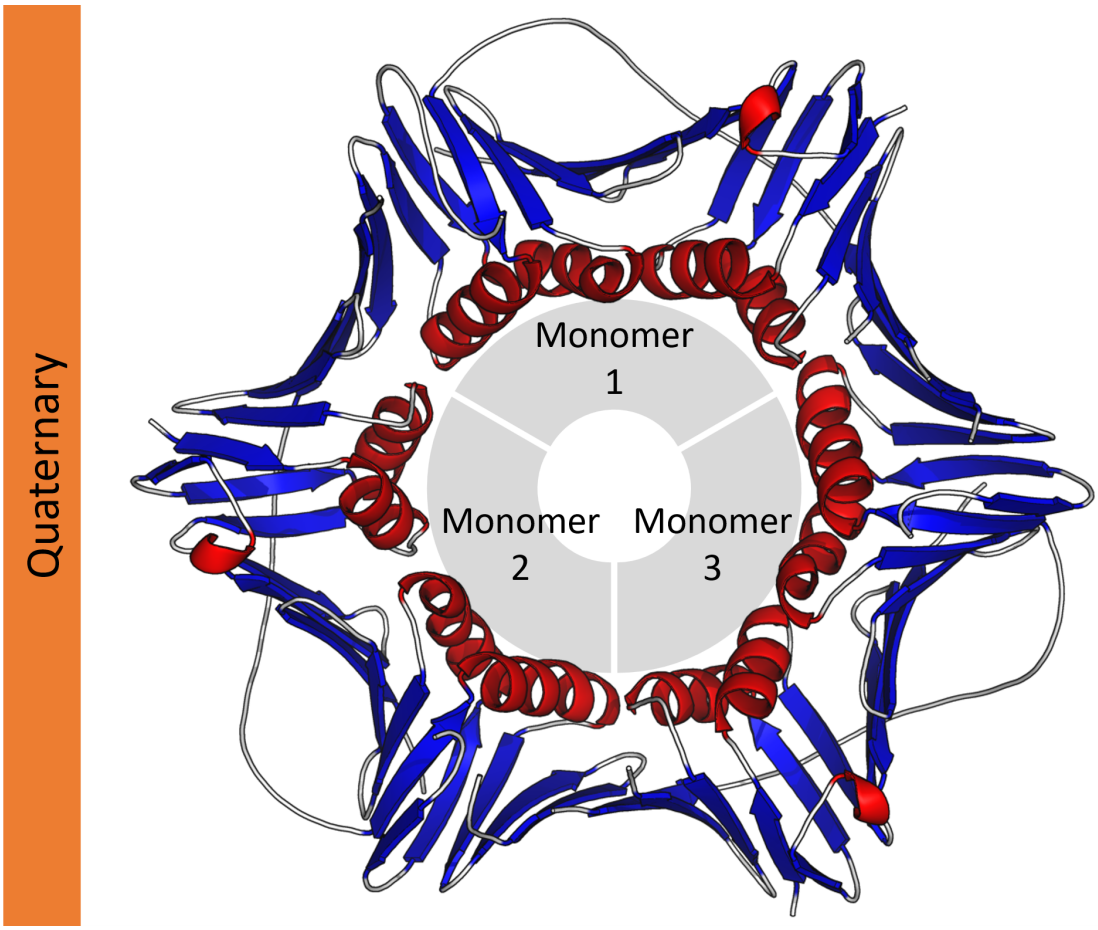


Figure 2.5: *Quaternary structure of the protein PCNA[33]*

For example, hemoglobin is a protein found in red blood cells whose function is to carry oxygen molecules throughout the blood vessels. The oxygen molecule binds to the heme groups found in the protein.

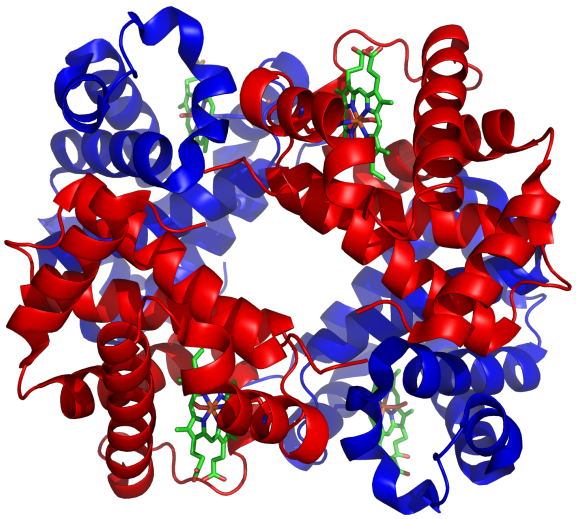


Figure 2.6: *Hemoglobin, the iron-containing heme groups are shown in green[37]*

To understand the function of a protein, we must determine its complete 3D structure. While techniques exist to measure a molecule's structure in real life, these are incredibly time-consuming and expensive to perform, even for a single molecule. Consequently, we would like to design algorithms that can predict the complete 3D structure of a protein based on its primary structure or its amino acid sequence. These algorithms are called protein folding algorithms.

In the molecular docking task, the candidate protein has already been selected with its 3D structure known. For the purpose of drug discovery, we are searching for the best possible protein as well, which is why databases of proteins with their predicted 3D structures are being worked on, such as https://www.uniprot.org/, or https://alphafold.ebi.ac.uk/ [32].

Modelling and predicting the complete four levels of protein structure is a complex task. Interestingly, simplified models exist that can be experimentally shown to achieve a reasonable approximation. One of these models was given by Dill et al. in 1995. (republished in 2008) [12].

The HP model replaces all protein's amino acids by their water affinity categories: polar (P) and hydrophobic (H), or 0 and 1. Then, the protein can be represented by a sequence consisting of letters H and P of the length equal to its amino acid number. Finally, this chain is embedded in either a two- or a three-dimensional grid. The letters must be placed in the grid's intersections so that consecutive letters are in adjoining intersections (either to the up, down, left or right directions and above or under in the 3D case) while the chain is not allowed to cross itself over.

A scoring of the placement is determined by the count of the all pairs of non-covalently bonded (non-consecutive) but neighbouring H letters on the grid. This captures the idea of energy minimization, in which the hydrophobic molecules tend to be close to each other and thus avoid exposure, while the polar molecules are neutral, since in the cell, all molecules are immersed in "water", the energy is best conserved by forming an external shell from their water-liking parts and hiding their water-disliking parts in the inside.

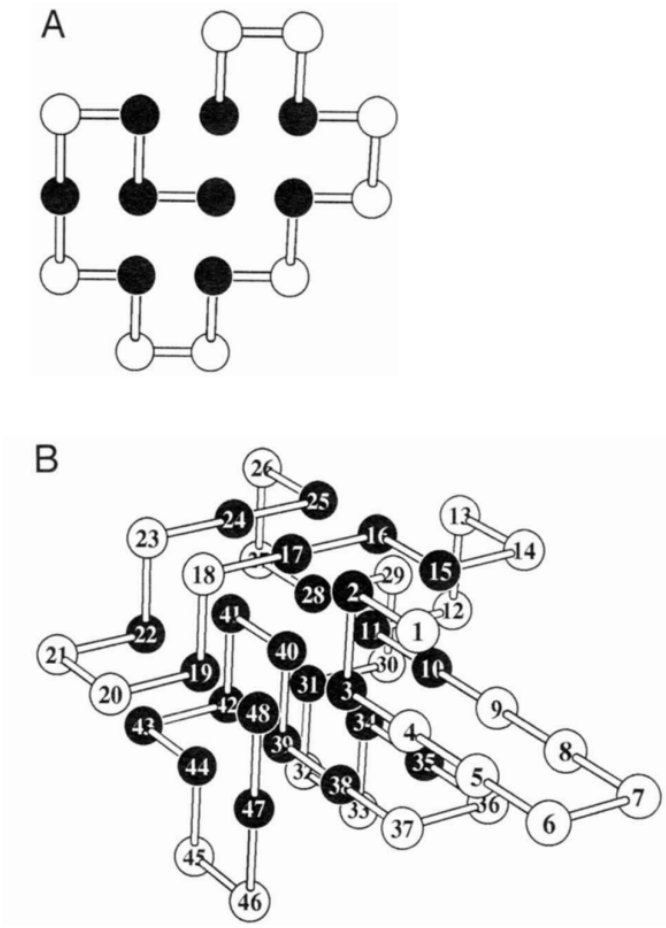


Figure 2.7: *Examples of a HP chain embedded in 2D and 3D space (H represented in black, P in white)[12]*

Even this simplified problem is proven to be NP-hard, since a reduction from the Hamiltonian-cycle problem can be shown [10]. We could achieve a slightly more complex model, by keeping the grid but reverting back to 20 possible letters, corresponding to the 20 possible amino-acids and employing a scoring matrix for all possible pairs of neighbours, however the two-dimensional HP model has been found to be the best match of the experimental results among all of these approximations. [10]

2.1.3 Molecular docking

Proteins are the primary agents of biological function, as they control the various chemical mechanisms that occur inside the cells. Proteins that act as biological catalysts are called enzymes. Catalysts facilitate various chemical reactions without being consumed in the process. These molecules have a binding site (a 'hole' on their 3D surface), into which only a specific other molecule fits, which is called a substrate. During the chemical reaction, the substrate is turned into other products. [15]

Many human diseases result from abnormal interactions between proteins. In order to prevent disease, we can stop (inhibit) these interactions from happening. This is done by blocking the binding site of the faulty enzyme with another molecule. Conventional

medicine uses giant molecules (antibodies) or tiny molecules (like aspirin) to achieve this. The next generation of protein therapeutics currently under development aims to find inhibitors within the family of smaller-sized proteins. [30]

In order to inhibit an enzyme's reaction from happening, we must find a protein that folds into a shape that fits into the enzyme's binding site to prevent it from catalyzing a reaction. The analogy of 'lock-and-key' was coined for this by Emil Fischer in 1894. He suggested that the 'lock' describes the enzyme's binding site, and the 'key' describes the missing molecule inhibitor, which has to fit into the 'lock'. [2] [25] The computational task of predicting whether a protein with a known 3D structure will fit inside the binding site of a specific enzyme is called molecular docking.

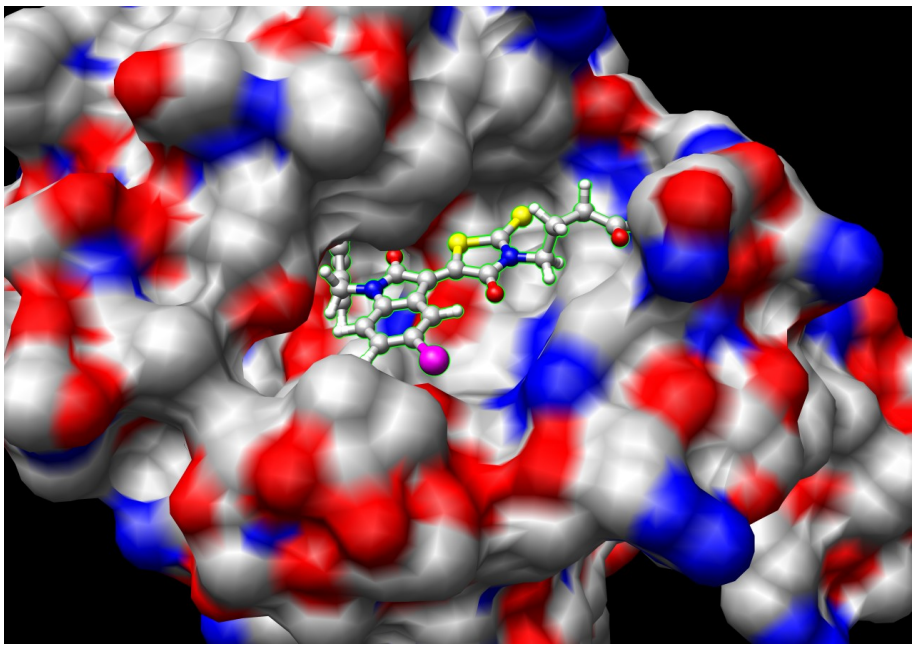


Figure 2.8: *Molecular docking[8]*

In the molecular docking problem, both the binding site's 3D structure and the protein's 3D structure are represented by a graph. The graph's vertices are the atoms on the surface. The edges represent a chemical bond between the corresponding atoms. The weight of a given edge represents the physical distance between the atoms. Omitting the edge weights, the problem is analogous to subgraph isomorphy (an NP-hard problem), the binding site's graph representation being a subgraph of the inhibitor molecule's entire surface in a graph representation. With the edge weights present, we can employ a measure that assigns a score to an isomorphy mapping of the vertices between the two graphs. Typically root mean square deviation (RMSD) is used, which can be defined for two sets of (ordered) points. The mapping achieving the highest score is the best theoretical fit for the molecule. [36]

2.2 Using quantum algorithms to solve bioinformatical problems

While some problems in bioinformatics, such as DNA sequence alignment, already have a fast enough solution, others, such as protein folding and molecular docking, turn out to be NP-hard ones. I have chosen to explore protein folding in more detail.

Grover's search algorithm is a quantum algorithm that can solve problems classically in NP if a candidate solution can be encoded to a quantum register and the verifier algorithm can be translated to quantum gate logic. Applying multiple Grover searches, we can also give a solution that minimizes or maximizes a target function. In the case of protein folding, a candidate solution is a possible orientation of a chain in 3D space, and we are looking for the minimum energy configuration.

Quantum walks work by exploring the problem's domain, which can be represented in a graph form. The vertices are candidate solutions, and the edges connecting them are small transformations on them. In the case of protein folding, a possible small transformation is rotating the chain at a single point. This operation also has to be translated to quantum gate logic.

These two algorithms can be effectively combined. For example, the initialization step for Grover can be a quantum walk, or they can be applied in turns.

Chapter 3

Quantum computing

Algorithms in quantum computing are derived from the postulates of quantum mechanics. These fundamental rules define how a quantum computer operates and are essential for any discourse on quantum algorithms.

This introduction is based on the following books: Quantum Computing and Communications by Sándor Imre and Ferenc Balázs [3], Quantum Computing by Mika Hirvensalo [21] and Quantum Walks and Search Algorithms by Renato Portugal [29].

Postulate I. State space

The state of an isolated physical system can be described using a unit length *state vector* in a Hilbert space (i.e. complex linear vector space), or *state space*, equipped with an inner product.

Definition 3.1 (Qubit). A state vector in the 2 dimensional Hilbert space (H_2) is a qubit. The base vectors in this space are

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \text{ and } |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

A generic qubit is written in the form

$$a\,|0\rangle + b\,|1\rangle = \begin{pmatrix} a \\ b \end{pmatrix}$$

where $a, b \in \mathbb{C}$ and $|a|^2 + |b|^2 = 1$. ▪

Definition 3.2 (Superposition). A quantum system is said to be in *superpotion*, if its state vector is a linear combination of multiple basis states.

For example $a\,|0\rangle + b\,|1\rangle$ is in a superposition of the basis states $|0\rangle$ and $|1\rangle$, with probability amplitudes a and b.

Postulate II. Evolution

The time evolution of an isolated physical system is described using unitary transformation, which depends only on the starting and finishing time of the evolution.

A *quantum algorithm* is a sequence of unitary operators applied to an initial state.

Definition 3.3 (Unitary matrix). $\mathbf{U}$ is unitary if $\mathbf{U}^\dagger = \mathbf{U}^{-1}$ [20].

The following definitions are equivalent:

1. $\mathbf{U}$'s rows form an orthonormal basis of $\mathbb{C}^n$.
2. $\mathbf{U}$'s columns form an orthonormal basis of $\mathbb{C}^n$.
3. $\mathbf{U}$ is an isometry: it is injective and preserves length.
4. $\mathbf{U}$ preserves the inner product. ▪

Postulate III. Measurement

A quantum measurement is defined by a set of measurement operators $\{\mathbf{M}_m\}$, where m stands for the possible results of the measurement. The probability of measuring m if the system is in state $|v\rangle$ is

$$P(m|\,|v\rangle) = \langle v|\,\mathbf{M}_m^\dagger \mathbf{M}_m\,|v\rangle\,.$$

The state of the system after measuring m is then

$$|v'\rangle = \frac{\mathbf{M}_m\,|v\rangle}{\sqrt{\langle v|\,\mathbf{M}_m^\dagger \mathbf{M}_m\,|v\rangle}}.$$

The set of measurement operators have to satisfy the following *completeness relation*:

$$\sum_m \mathbf{M}_m^\dagger \mathbf{M}_m = I,$$

due to

$$1 = \sum_m P(m|\,|v\rangle) = \sum_m \langle v|\,\mathbf{M}_m^\dagger \mathbf{M}_m\,|v\rangle\,.$$

Projective measurement

To distinguish a set of orthonormal states $\{|\varphi_m\rangle\}$, the corresponding measurement operators can be produced as $\mathbf{P}_m = |\varphi_m\rangle\,\langle\varphi_m|$, with the following properties.

Property 3.1 ($\mathbf{P}_m$ is self adjoint).

$$\mathbf{P}_m^\dagger = \mathbf{P}_m$$

Since

$$\mathbf{P}_m^\dagger = (|\varphi_m\rangle \langle\varphi_m|)^\dagger = \langle\varphi_m|^\dagger |\varphi_m\rangle^\dagger = |\varphi_m\rangle \langle\varphi_m| = \mathbf{P}_m.$$

Property 3.2 ($\mathbf{P}_m$ is a projection).

$$\mathbf{P}_m\mathbf{P}_m = \mathbf{P}_m$$

Since

$$\mathbf{P}_m\mathbf{P}_m = (|\varphi_m\rangle \langle\varphi_m|)(|\varphi_m\rangle \langle\varphi_m|) = |\varphi_m\rangle (\langle\varphi_m|\varphi_m\rangle) \langle\varphi_m| = |\varphi_m\rangle 1 \langle\varphi_m| = |\varphi_m\rangle \langle\varphi_m| = \mathbf{P}_m.$$

Property 3.3 (The $\mathbf{P}_m$ are orthogonal).

$$m \neq n \Rightarrow \mathbf{P}_m\mathbf{P}_n = \mathbf{0}$$

Since

$$\mathbf{P}_m\mathbf{P}_n = (|\varphi_m\rangle \langle\varphi_m|)(|\varphi_n\rangle \langle\varphi_n|) = |\varphi_m\rangle (\langle\varphi_m|\varphi_n\rangle) \langle\varphi_n| = |\varphi_m\rangle 0 \langle\varphi_n| = \mathbf{0}.$$

From these properties follows, that the probability of measuring m in case of a projective measurement is

$$P(m||v\rangle) = \langle v| \mathbf{P}_m^\dagger\mathbf{P}_m |v\rangle = \langle v| \mathbf{P}_m\mathbf{P}_m |v\rangle = \langle v| \mathbf{P}_m |v\rangle = \langle v|\varphi_m\rangle \langle\varphi_m|v\rangle = |\langle\varphi_m|v\rangle|^2.$$

For example, the value of a qubit can be any unit length vector in H_2, however when we measure it, we will receive one of the base vectors of H_2. For $a|0\rangle + b|1\rangle$ we measure 0 with probability $|a|^2$ and 1 with probability $|b|^2$.

Postulate IV. Composite systems

The state space of an isolated composite physical system is the *tensor product* of the state spaces of the individual components. The current state vector of the composite system is the *tensor product* of the current state vectors of the individual systems.

If $V_1, \ldots, V_n$ are the state spaces of the individual systems, then $V_1 \otimes \cdots \otimes V_n$ is the composite state space and for $|v_1\rangle \in V_1, \ldots, |v_n\rangle \in V_n$ state vectors, $|v_1\rangle \otimes \cdots \otimes |v_n\rangle = |v_1, \ldots, v_n\rangle$ is the state vector of the composite system.

Definition 3.4 (Tensor product). The tensor product $\mathbf{A} \otimes \mathbf{B}$ of matrix $\mathbf{A}_{(r\times s)}$ and matrix $\mathbf{B}_{(t\times u)}$ is of size $(rt \times su)$ and is defined as follows [20]:

$$\text{For } \mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1s} \\ a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ a_{r1} & a_{r2} & \ddots & a_{rs} \end{pmatrix}, \text{ and } \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1u} \\ b_{21} & b_{22} & \dots & b_{2u} \\ \vdots & \vdots & \ddots & \vdots \\ b_{t1} & b_{t2} & \ddots & b_{tu} \end{pmatrix}$$

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \dots & a_{1s}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \dots & a_{2s}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{r1}\mathbf{B} & a_{r2}\mathbf{B} & \ddots & a_{rs}\mathbf{B} \end{pmatrix} \quad \blacksquare$$

and has the following properties:

Property 3.4 (Associativity).

$$(\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} = \mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C})$$

Property 3.5 (Mixed product property). If the corresponding matrices are compatible, then

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = (\mathbf{AC}) \otimes (\mathbf{BD}),$$

and as an immediate consequence, we obtain

$$(\mathbf{A} \otimes \mathbf{I})(\mathbf{I} \otimes \mathbf{B}) = \mathbf{A} \otimes \mathbf{B}.$$

Definition 3.5 (Quantum register). The composite system of n qubits is a *quantum register*, having the composite state space

$$H_2^{\otimes n} = H_2 \otimes H_2 \otimes ... \otimes H_2$$

and for $|q_{n-1}\rangle \in H_2, \dots, |q_0\rangle \in H_2$ individual state vectors, the composite state vector is

$$|q_{n-1}\rangle \otimes \cdots \otimes |q_0\rangle = |q_{n-1}, \dots, q_0\rangle . \quad \blacksquare$$

Definition 3.6 (Entangled state). Any state consisting of multiple qubits, that is not decomposable, i.e. that can not be written in the form of a composite system of qubits is *entangled*. $\blacksquare$

For example, the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is entangled, since it can not be written in the form

$$(a_0 |0\rangle + a_1 |1\rangle) \otimes (b_0 |0\rangle + b_1 |1\rangle) = a_0 b_0 |00\rangle + a_0 b_1 |01\rangle + a_1 b_0 |10\rangle + a_1 b_1 |11\rangle$$

since that would require from $a_0 b_0 = a_1 b_1 = \frac{1}{\sqrt{2}}$, for all coefficients to be non-zero and from $a_0 b_1 = a_1 b_0 = 0$ for either a_0 or b_1 and either a_1 or b_0 to be zero, which is a contradiction.

Chapter 4

Quantum walks

Current technology is yet to produce a significant number of qubits (quantum bits) in a quantum processor, but many believe the amount will increase over the years. The first practical quantum algorithms to be run on these processors are likely to be the ones that use qubits sparingly. Quantum walking, the generalized version of classical random walking, is exactly this kind of algorithm. The number of qubits required to run a quantum walk on a graph is logarithmic in the number of vertices, making it a promising technique for the near future. Furthermore, by extending the quantum walk algorithm with a Grover search, it can be used to perform searching, which could be used for solving many of the computationally hard bioinformatical problems.

4.1 Classical random walks

Before introducing quantum computation and specifically quantum walks, I first overview classical random walks, based on the book Probability and Computing, written by Michael Mitzenmacher and Eli Upfal [24].

A *random walk* is a stochastic process modeled by a particular type of Markov chain. While a variety of Markov chains exist, in this work, I use the following definition exclusively.

Definition 4.1 (Markov chain). A discrete time stochastic process $X_0, X_1, X_2, \ldots$ on a finite state space A is a Markov chain if it has the Markov property:

$$P(X_k = a_k \mid X_{k-1} = a_{k-1}, \ldots, X_0 = a_0) = P(X_k = a_k \mid X_{k-1} = a_{k-1}) \quad \forall a_0, \ldots, a_k \in A.$$

Without loss of generality, we can assume, that $A = \{0, 1, \ldots, n\}$.

If the Markov chain is homogenous (time-invariant), the probability of moving from state $i \in A$ to state $j \in A$ is independent of k, and thus can be shortened the following way:

$$P(X_k = j \mid X_{k-1} = i) = p_{j \leftarrow i} = p_{j,i} \quad \forall k \in \mathbb{Z}^+.$$

Where $p_{j,i}$ is called the *transition probability* between states i and j.

The *transition matrix* $\mathbf{P}$ is formed by the transition probabilities.

$$\mathbf{P}[j,i] = p_{j,i}$$

It follows, that for each column in $\mathbf{P}$, the sum is 1.

$$\sum_{j=0}^{n} \mathbf{P}[j,i] = 1 \quad \forall i \in \{0,\ldots,n\}$$

Let the *probability distribution* of the process in the t-th step be π_t. Then, π_t can be computed from the starting distribution π_0 using $\mathbf{P}$.

$$\pi_t = \mathbf{P}^t \pi_0$$

The *stationary distribution* (π) of the Markov chain is a distribution that does not change with a transition, i.e. $\pi = \mathbf{P}\pi$.

Markov chains can be represented using graphs. A directed, weighted graph $G(V,E)$ with weight function $w : E \to [0,1]$ represents a Markov chain, if $V = A$ and $w(i,j) = \mathbf{P}[j,i]$. If $\mathbf{P}[j,i] = 0$, then $\{i,j\} \notin E$.

A random walk on graph G starts from $X_0 = a_0$ and visits the vertices of the graph according to the states of the Markov-chain: $X_1 = a_1, X_2 = a_2, \ldots$.

Frequently studied characteristics of random walks are *hitting time* [38] and *mixing time* [24]. Informally, hitting time describes how quickly can a vertex be reached from another vertex in the graph, while mixing time expresses how fast the walk reaches the stationary distribution, where the starting vertex can no longer be identified.

Definition 4.2 (Hitting time). Let $h_{j,i}$ be the expected number of steps before node j is visited in a random walk starting from node i. Then, $h_{j,i}$ is given by the following recursive formula:

$$h_{j,i} = \begin{cases} 1 + \sum\limits_{k \in A} p_{j,k} h_{k,i} & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases}$$.

Definition 4.3 (Mixing time). The smallest time index of the Markov chain, where the total variational distance between the current and the stationary distribution is not greater than a given ε. This measure still depends on the starting distrubiton π_0, so we take the maximum over all of the possible π_0 distributions.

$$m(\varepsilon) = \max_{\pi_0}\{\min\{t : \sum_{j=0}^{n} |\pi_t[j] - \pi[j]| \leq \varepsilon\}\}$$.

4.2 From classical to quantum

In classical random walks, the walker moves from the current vertex via one of its outgoing edges, chosen randomly, weighted by the edge weights. This random choice can be interpreted as a (generalized) coin toss.

To formulate a quantum version of graph walking, we define the quantum coin, which will replace the classical concept of randomness with quantum superposition.

4.3 Formulating the Quantum coin

I used Renato Portugal's Quantum Walks and Search Algorithms [29] book as a reference for the different types of coins presented in this section.

A *quantum coin* is a quantum system, which behaves according to the postulates of quantum mechanics. It has a current state, represented by a state vector in a Hilbert space and a unitary time evolution operator, describing a coin toss.

After tossing the coin, the resulting coin state chooses the next step of the quantum walker. If there are d outgoing edges to choose from, then the coin's state space must have d orthonormal basis states, each corresponding to one of the possible edges. If the current state is one of the basis states, then the walker moves in that direction. However, in the quantum world, the coin can also be in a superposition, consisting of multiple basis states. This means that the walker will simultaneously move in all corresponding directions and occupy more than one vertex at the same time, resulting in the walker spreading over the graph in a superposition.

For the d dimensional coin state, the corresponding coin flip operator is a $(d \times d)$ dimensional unitary matrix. Based on what the transition operator is, several types of coins can be defined. The following ones are typically used in quantum walks.

4.3.1 Hadamard coin

The Hadamard coin is the most commonly used quantum coin. It is defined by the Hadamard-matrix as a transition operator:

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

If the starting coin state is $|0\rangle$, then flipping the coin once results in the following state:

$$\mathbf{H}\,|0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}}\,|0\rangle + \frac{1}{\sqrt{2}}\,|1\rangle\,.$$

If we measured the above coin, the probability of measuring 0 is

$$P(0 \mid \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)) = \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}.$$

Similarly, if the starting coin state is $|1\rangle$, then flipping the coin once results in the following state:

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

The probability of measuring 1 here is similarly

$$P(1 \mid \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)) = \left|-\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}.$$

An unexpected feature of this coin comes from the fact, that the Hadamard-matrix is Hermitian (self-adjoint), i.e. $\mathbf{H}^\dagger = \mathbf{H}$, while also unitary, i.e. $\mathbf{H}^\dagger = \mathbf{H}^{-1}$, which results in $\mathbf{H}^{-1} = \mathbf{H}$, thus $\mathbf{H}\mathbf{H} = \mathbf{I}$. This means, that after flipping the coin twice without measuring it, it will return the coin state to its origin. For example, starting from $|0\rangle$:

$$\mathbf{H}^2\,|0\rangle = \mathbf{H}\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{2}(|0\rangle + |1\rangle + |0\rangle - |1\rangle) = |0\rangle\,.$$

After the second flip, the probability of measuring $|0\rangle$ is 1, due to the destructive interference between the two $|1\rangle$ probability amplitudes, demonstrating a significant contrast between classical and quantum walks.

Definition 4.4 (2^n dimensional Hadamard-coin). A 2^n dimensional Hadamard-coin operator can be created by taking the tensor product of the 2 dimensional Hadamard-coin n times: $\mathbf{H}^{\otimes n}$.

4.3.2 Grover coin

The Grover coin originates from Grover's search algorithm, where it is applied as the diffusion operator.

Let $|D\rangle$ be the following state:

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

Using $|D\rangle$, the Grover coin is the following unitary matrix:

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

If $N = 2^n$, then $\mathbf{G}$ unrolls to the following representation:

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

4.3.3 Fourier coin

In contrast to the Hadamard and Grover coins, the Fourier coin can be of any size, not just a power of 2. A size N Fourier-coin, F_N is defined by the matrix of the Quantum Fourier Transform:

$$\mathbf{F}[k, l] = \frac{1}{\sqrt{N}} \omega^{kl}$$

where ω is the N-th root of unity,

$$\omega = e^{\frac{2\pi i}{N}}.$$

$\mathbf{F}$ unrolls to the following representation:

$$\mathbf{F} = \frac{1}{\sqrt{N}} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{pmatrix}$$

4.4 Quantum walks on the line

Kempe introduces quantum walks from a physicist's perspective in [22] using a particle characterised by its position on the line $|x\rangle$ and its spin state $|s\rangle$.

4.4.1 State space

Spin state

The spin state is in H_2 with the basis states spin up and down:

$$|\uparrow\rangle = |0\rangle,$$
$$|\downarrow\rangle = |1\rangle.$$

The spin state vector is then given by:

$$|s\rangle = s_0 |\uparrow\rangle + s_1 |\downarrow\rangle .$$

Position state

At the start of the walk the particle is at the origin $|0\rangle$ and the walking lasts for N steps. The position state is in $H_{(2N+1)}$ with the following basis vectors corresponding to the possible positions on the line.

$$\{|-N\rangle , |-(N-1)\rangle , \ldots , |-1\rangle , |0\rangle , |1\rangle , \ldots , |N-1\rangle , |N\rangle\}$$

I index the basis states using negative numbers to match the labels on the axis.

The position state vector is then given by:

$$|x\rangle = \sum_{i=-N}^{N} x_i |i\rangle .$$

Composite state

The composite state of the system, according to [PostulateIV] is then

$$|x\rangle \otimes |s\rangle .$$

4.4.2 Evolution

The particle travels on the line based on its current spin state:

- If the current spin state is $|0\rangle$, the particle moves to the left, i.e. from position $|i\rangle$ the particle travels to position $|i-1\rangle$.
- If the current spin state is $|1\rangle$, the particle moves to the right, i.e. from position $|i\rangle$ the particle travels to position $|i+1\rangle$.

This step is realised with the unitary matrix **S** which operates on the complete state of the system, $|x\rangle \otimes |s\rangle$ and is assembled from a left and a right shift operator acting on $|x\rangle$ and another operator for checking $|s\rangle$ compiled using tensor product.

Definition 4.5 (Left shift operator). To move from position $|i\rangle$ to the left ($|i-1\rangle$) the position vector is multiplied with the following **L** matrix, containing 1's above the diagonal. To keep **S** unitary, an unused transition must be added in the lower left corner (see Theorem 4.1).

$$\mathbf{L} = |N\rangle\langle -N| + \sum_{i=-(N-1)}^{N} |i-1\rangle\langle i| = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & & & \ddots & 1 \\ 1 & 0 & \cdots & & 0 \end{pmatrix} \tag{4.1}$$

For a given basis vector $|j\rangle$ only one of the summands in $\mathbf{L}$ is non-zero, where $i = j$, resulting in the required shift being performed.

$$\mathbf{L}|j\rangle = |j-1\rangle\langle j|j\rangle = |j-1\rangle \quad \blacksquare$$

Definition 4.6 (Right shift operator). To move from position $|i\rangle$ to the right ($|i+1\rangle$) the position vector is multiplied with the following $\mathbf{R}$ matrix, containing 1's under the diagonal. To keep $\mathbf{S}$ unitary, an unused transition must be added in the top right corner (see Theorem 4.1).

$$\mathbf{R} = |-N\rangle\langle N| + \sum_{i=-N}^{N-1} |i+1\rangle\langle i| = \begin{pmatrix} 0 & & \cdots & 0 & 1 \\ 1 & \ddots & & & 0 \\ 0 & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \\ 0 & \cdots & 0 & 1 & 0 \end{pmatrix} \tag{4.2}$$

For a given basis vector $|j\rangle$ only one of the summands in $\mathbf{R}$ is non-zero, where $i = j$, resulting in the required shift being performed.

$$\mathbf{R}|j\rangle = |j+1\rangle\langle j|j\rangle = |j+1\rangle \quad \blacksquare$$

Shift operator

Using matrixes L and R operating on the position register $|x\rangle$ only, we construct a unitary operator S, which operates on the composite state of the system, $|x\rangle \otimes |s\rangle$, executing matrix $\mathbf{L}$ on $|x\rangle$ only when $|s\rangle = |0\rangle$ and matrix $\mathbf{R}$ only when $|s\rangle = |1\rangle$.

$$\mathbf{S} = \mathbf{L} \otimes |0\rangle\langle 0| + \mathbf{R} \otimes |1\rangle\langle 1| \tag{4.3}$$

The execution logic is as follows:

$$\mathbf{S}(|x\rangle \otimes |s\rangle) =$$
$$(\mathbf{L} \otimes |0\rangle\langle 0| + \mathbf{R} \otimes |1\rangle\langle 1|)(|x\rangle \otimes |s\rangle) =$$
$$(\mathbf{L} \otimes |0\rangle\langle 0|)(|x\rangle \otimes |s\rangle) + (\mathbf{R} \otimes |1\rangle\langle 1|)(|x\rangle \otimes |s\rangle) = \ldots$$

using [TensorMixedProduct]:

$$\cdots = \mathbf{L}\,|x\rangle \otimes (|0\rangle\langle 0|s\rangle) + \mathbf{R}\,|x\rangle \otimes (|1\rangle\langle 1|s\rangle) =$$
$$|x-1\rangle \otimes s_0\,|0\rangle + |x+1\rangle \otimes s_1\,|1\rangle =$$
$$s_0\,|x-1,0\rangle + s_1\,|x+1,1\rangle\,.$$

- If the spin state was $|s\rangle = |0\rangle$ at the beginning, then $s_0 = 1$ and $s_1 = 0$, which means that the resulting system state is $|x-1,0\rangle$, which means that the particle shifted to the left, as designed.
- If the spin state was $|s\rangle = |1\rangle$ at the beginning, then $s_0 = 0$ and $s_1 = 1$, which means that the resulting system state is $|x+1,1\rangle$, which means that the particle shifted to the right, also as intended.

Furthermore, the spin state can be any mixed state $s_0\,|0\rangle + s_1\,|1\rangle$ as well. In this case the particle will shift *both* to the left and to the right, at the same time. When measured, the particle can be found in position $|x-1\rangle$ with probability $|s_0|^2$ and in position $|x+1\rangle$ with probability $|s_1|^2$.

In quantum graph walks, the walker can simultaneously explore multiple parallel paths in the graph, at the same time. With good design, this behaviour can be used to search the graph faster than in classical random graph walks.

Coin operator

To inject the quantum superposition into the walk, the particle's spin state is transformed using any 2 dimensional unitary matrix between shift operations. The Hadamard, Grover and Fourier coins mentioned earlier are commonly used as coin operators.

For any C operator on the coin register, the unitary transform for the composite system is defined as follows:

$$\hat{\mathbf{C}} = \mathbf{I} \otimes \mathbf{C}$$

since the coin operator does not modify the position register.

Evolution operator

Combining the shift operator and the coin operator together, we obtain the following evolution operator, defining one step of the quantum walk on the line. The step consists

of flipping the coin once, then applying the shifting the walker's position accordingly, as follows:

$$\mathbf{U} = \mathbf{S}\hat{\mathbf{C}} = \mathbf{S}(\mathbf{I} \otimes \mathbf{C})$$

4.4.3 Measurement

To measure the probability of the particle being at position $|i\rangle$, the projective measurement operator acting on $|x\rangle$ is defined as $\mathbf{P_i} = |i\rangle \langle i|$, in accordance with [PostulateIIIProjective].

Since the coin register need not be measured, we apply the identity operator on it, using $\mathbf{P_i} \otimes \mathbf{I}$ on the complete system to measure the particle's current position.

The probability of finding the particle in position i is:

$$P(i|\,|x\rangle) = \langle x, s|\,\mathbf{P_i} \otimes \mathbf{I}\,|x, s\rangle = \dots$$

using [TensorMixedProduct]:

$$\cdots = \langle x|\,\mathbf{P_i}\,|x\rangle\,\langle s|\,\mathbf{I}\,|s\rangle = \langle x|\,\mathbf{P_i}\,|x\rangle\,1 = \langle x|\,\mathbf{P_i}\,|x\rangle = \langle x|i\rangle\,\langle i|x\rangle = |\,\langle i|x\rangle\,|^2 = |x_i|^2$$

4.5 Generalization of Quantum Walks

After presenting quantum walking on the line, I review and extend two approaches to generalize it in this chapter.

1. **Section 4.5.1: Use multiple two-dimensional coins**: In [29], Renato Portugal shows the generalization of quantum walks on a line to a two-dimensional grid, using a method with 2 two-dimensional coins. In this work, I prove how his method reduces to effectively two synchronous independent walks on the x and y axes. Then I improve his technique by generalizing to arbitrarily large dimensional grids in a more memory-efficient way than what would naturally follow from his description.
2. **Section 4.5.2: Use a single higher dimensional coin**: In [29], Renato Portugal describes the generalization of a quantum walk on a line to an arbitrary undirected graph and gives the necessary condition for creating the unitary transition matrix without proof. In this work, I generalize to directed graphs and give proof of the generalization of the condition using directed graphs.

4.5.1 Generalization using multiple independent 2 dimensional coins

In [29], Renato Portugal defines the following method for Quantum Walking on a 2D grid:

Let the position state of the walker be $|x, y\rangle$ and the two coins $|c_x\rangle$, acting on the x coordinate and $|c_y\rangle$, acting on the y coordinate of the walker.

The shift operator moves the walker on the grid diagonally, according to the current state of the two coins, described by

$$\mathbf{S}\,|x,y\rangle\,|c_x\rangle\,|c_y\rangle = |x+(-1)^{c_x}, y+(-1)^{c_y}\rangle\,|c_x\rangle\,|c_y\rangle\,, \tag{4.4}$$

and the coin operator leaves the position state in place while flipping both coins at the same time, described by

$$\hat{\mathbf{C}} = \mathbf{I} \otimes \mathbf{C_4} = \mathbf{I} \otimes (\mathbf{C_2} \otimes \mathbf{C_2}). \tag{4.5}$$

Issues with this method

In 4.4, we can see how the matrix $\mathbf{S}$ will quickly increase in size, as further dimensions are added to the equation. In 2D, if the walker takes N steps, the size of $\mathbf{S}$ is

$$(2N+1)^2(2N+1)^2 2^2 2^2 = 16(2N+1)^4 = O(N^4).$$

To increase the dimension count, one would naturally append more coordinates to the composite position state and add further coins, for example in 3D $\mathbf{S}$ would become

$$\mathbf{S}\,|x,y,z\rangle\,|c_x\rangle\,|c_y\rangle\,|c_z\rangle = |x+(-1)^{c_x}, y+(-1)^{c_y}, z+(-1)^{c_z}\rangle\,|c_x\rangle\,|c_y\rangle\,|c_z\rangle\,,$$

For a dimension count d, the size of $\mathbf{S}$ is exponential in d.

$$((2N+1)^2)^d (2^2)^d = (4(2N+1)^2)^d = O(N^{2d}).$$

My improvements

Since in 4.4 the coordinates of the walker are updated independently by the separate coins, I was able to disassemble $\mathbf{S}$ into smaller matrices, using the properties of the tensor product.

To do this, in what follows, I first define $\mathbf{S}$ in 2D explicitly (as opposed to the implicit definition in 4.4, stating only how $\mathbf{S}$ updates the state of the system). I will be using the matrices $\mathbf{L}$ defined by Equation (4.1) and $\mathbf{R}$ defined by Equation (4.2).

Notice that $\mathbf{R}$ increases the walker's coordinates on the line, while $\mathbf{L}$ decreases them. This means, that on the y axis $\mathbf{R}$ acts by moving the walker up, and $\mathbf{L}$ acts by moving the walker down.

When the coins are in the state $|c_x, c_y\rangle = |0,0\rangle$, the walker moves up and to the right. This movement is captured by $\mathbf{S}_{0,0}$, acting on the position state:

$$\mathbf{S}_{0,0} = \mathbf{R} \otimes \mathbf{R}$$

The other three $\mathbf{S}_{c_x,c_y}$ matrices are defined similarly:

$$\begin{aligned}\mathbf{S}_{0,1} &= \mathbf{R} \otimes \mathbf{L} \\ \mathbf{S}_{1,0} &= \mathbf{L} \otimes \mathbf{R} \\ \mathbf{S}_{1,1} &= \mathbf{L} \otimes \mathbf{L}\end{aligned}$$

Then, I assemble $\mathbf{S}$ using $\mathbf{S}_{0,0}$, $\mathbf{S}_{0,1}$, $\mathbf{S}_{1,0}$ and $\mathbf{S}_{1,1}$ the following way:

- $\mathbf{S}_{0,0}$ acts only when $|c_x, c_y\rangle = |0,0\rangle$,
- $\mathbf{S}_{0,1}$ acts only when $|c_x, c_y\rangle = |0,1\rangle$,
- $\mathbf{S}_{1,0}$ acts only when $|c_x, c_y\rangle = |1,0\rangle$, and finally
- $\mathbf{S}_{1,1}$ acts only when $|c_x, c_y\rangle = |1,1\rangle$.

Using the same method as in Equation (4.3) I arrive at:

$$\begin{aligned}\mathbf{S} = \mathbf{S}_{0,0} \otimes |0,0\rangle \langle 0,0| + \\ \mathbf{S}_{0,1} \otimes |0,1\rangle \langle 0,1| + \\ \mathbf{S}_{1,0} \otimes |1,0\rangle \langle 1,0| + \\ \mathbf{S}_{1,1} \otimes |1,1\rangle \langle 1,1|\end{aligned}$$

After substituting the $\mathbf{S}_{c_x,c_y}$ matrices in:

$$\begin{aligned}\mathbf{S} = (\mathbf{R} \otimes \mathbf{R}) \otimes |0,0\rangle \langle 0,0| + \\ (\mathbf{R} \otimes \mathbf{L}) \otimes |0,1\rangle \langle 0,1| + \\ (\mathbf{L} \otimes \mathbf{R}) \otimes |1,0\rangle \langle 1,0| + \\ (\mathbf{L} \otimes \mathbf{L}) \otimes |1,1\rangle \langle 1,1|\end{aligned}$$

Let us name the coin state $|0\rangle$ heads and the coin state $|1\rangle$ tails. Then, using the following equalities and introducing matrices $\mathbf{H}$ and $\mathbf{T}$, as shorthands:

$$\begin{aligned}|0,0\rangle \langle 0,0| &= (|0\rangle \langle 0|) \otimes (|0\rangle \langle 0|) = \mathbf{H} \otimes \mathbf{H} \\ |0,1\rangle \langle 0,1| &= (|0\rangle \langle 0|) \otimes (|1\rangle \langle 1|) = \mathbf{H} \otimes \mathbf{T} \\ |1,0\rangle \langle 1,0| &= (|1\rangle \langle 1|) \otimes (|0\rangle \langle 0|) = \mathbf{T} \otimes \mathbf{H} \\ |1,1\rangle \langle 1,1| &= (|1\rangle \langle 1|) \otimes (|1\rangle \langle 1|) = \mathbf{T} \otimes \mathbf{T}\end{aligned}$$

I arrive at:

$$\begin{aligned}\mathbf{S} = (\mathbf{R} \otimes \mathbf{R}) \otimes (\mathbf{H} \otimes \mathbf{H}) + \\ (\mathbf{R} \otimes \mathbf{L}) \otimes (\mathbf{H} \otimes \mathbf{T}) + \\ (\mathbf{L} \otimes \mathbf{R}) \otimes (\mathbf{T} \otimes \mathbf{H}) + \\ (\mathbf{L} \otimes \mathbf{L}) \otimes (\mathbf{T} \otimes \mathbf{T})\end{aligned}$$

Then, using [TensorMixedProduct] I inflate the equation with (appropriately sized) $\mathbf{I}$ matrices:

$$\begin{aligned}\mathbf{S} = ((\mathbf{R} \otimes \mathbf{I})(\mathbf{I} \otimes \mathbf{R})) \otimes ((\mathbf{H} \otimes \mathbf{I})(\mathbf{I} \otimes \mathbf{H})) + \\ ((\mathbf{R} \otimes \mathbf{I})(\mathbf{I} \otimes \mathbf{L})) \otimes ((\mathbf{H} \otimes \mathbf{I})(\mathbf{I} \otimes \mathbf{T})) + \\ ((\mathbf{L} \otimes \mathbf{I})(\mathbf{I} \otimes \mathbf{R})) \otimes ((\mathbf{T} \otimes \mathbf{I})(\mathbf{I} \otimes \mathbf{H})) + \\ ((\mathbf{L} \otimes \mathbf{I})(\mathbf{I} \otimes \mathbf{L})) \otimes ((\mathbf{T} \otimes \mathbf{I})(\mathbf{I} \otimes \mathbf{T}))\end{aligned}$$

At this point, I introduce a few aliases to make the equation more manageable. Notice, how for example $\mathbf{I} \otimes \mathbf{R}$ is acting on the 2D position state, but only updating the y coordinate to move the walker upward. This gives a way to naturally define $\mathbf{S}_{\text{up}} = \mathbf{I} \otimes \mathbf{R}$, and similarly:

$$\begin{aligned}\mathbf{S}_{\text{right}} &= \mathbf{R} \otimes \mathbf{I} \\ \mathbf{S}_{\text{left}} &= \mathbf{L} \otimes \mathbf{I} \\ \mathbf{S}_{\text{up}} &= \mathbf{I} \otimes \mathbf{R} \\ \mathbf{S}_{\text{down}} &= \mathbf{I} \otimes \mathbf{L}\end{aligned}$$

At the same time, for example $\mathbf{I} \otimes \mathbf{H}$ is acting on the composite coin state, but only checking if the second coin's state is heads, which is the coin for the y axis. This gives a way to naturally define $\mathbf{H}_y = \mathbf{I} \otimes \mathbf{H}$, and similarly:

$$\begin{aligned}\mathbf{H}_x &= \mathbf{H} \otimes \mathbf{I} \\ \mathbf{T}_x &= \mathbf{T} \otimes \mathbf{I} \\ \mathbf{H}_y &= \mathbf{I} \otimes \mathbf{H} \\ \mathbf{T}_y &= \mathbf{I} \otimes \mathbf{T}\end{aligned}$$

Substituting all of the aliases:

$$\begin{aligned}\mathbf{S} = (\mathbf{S}_{\text{right}}\mathbf{S}_{\text{up}}) \otimes (\mathbf{H}_x\mathbf{H}_y) + \\ (\mathbf{S}_{\text{right}}\mathbf{S}_{\text{down}}) \otimes (\mathbf{H}_x\mathbf{T}_y) + \\ (\mathbf{S}_{\text{left}}\mathbf{S}_{\text{up}}) \otimes (\mathbf{T}_x\mathbf{H}_y) + \\ (\mathbf{S}_{\text{left}}\mathbf{S}_{\text{down}}) \otimes (\mathbf{T}_x\mathbf{T}_y)\end{aligned}$$

Then, using [TensorMixedProduct] again, I arrive at:

$$\begin{aligned}\mathbf{S} = (\mathbf{S}_{\text{right}} \otimes \mathbf{H}_x)(\mathbf{S}_{\text{up}} \otimes \mathbf{H}_y) + \\ (\mathbf{S}_{\text{right}} \otimes \mathbf{H}_x)(\mathbf{S}_{\text{down}} \otimes \mathbf{T}_y) + \\ (\mathbf{S}_{\text{left}} \otimes \mathbf{T}_x)(\mathbf{S}_{\text{up}} \otimes \mathbf{H}_y) + \\ (\mathbf{S}_{\text{left}} \otimes \mathbf{T}_x)(\mathbf{S}_{\text{down}} \otimes \mathbf{T}_y)\end{aligned}$$

Then using the distributive property of matrix multiplication with respect to matrix addition I finally arrive at:

$$\mathbf{S} = ((\mathbf{S}_{\text{right}} \otimes \mathbf{H}_x) + (\mathbf{S}_{\text{left}} \otimes \mathbf{T}_x))((\mathbf{S}_{\text{up}} \otimes \mathbf{H}_y) + (\mathbf{S}_{\text{down}} \otimes \mathbf{T}_y))$$

We can see from the equation above, that $\mathbf{S}$ is actually the product of two shift operators, one only acting on the x coordinate using the first coin's state, the other acting only on the y coordinate, according to the second coin's state.

$$\begin{aligned}\mathbf{S}_x &= (\mathbf{S}_{\text{right}} \otimes \mathbf{H}_x) + (\mathbf{S}_{\text{left}} \otimes \mathbf{T}_x) \\ \mathbf{S}_y &= (\mathbf{S}_{\text{up}} \otimes \mathbf{H}_y) + (\mathbf{S}_{\text{down}} \otimes \mathbf{T}_y)\end{aligned}$$

$$\mathbf{S} = \mathbf{S}_x \mathbf{S}_y$$

This proves, that the walk on the 2D grid decomposes into two independent line walks on the axes, since $\mathbf{S}_x$ only touches the x coordinate and the first coin, while $\mathbf{S}_y$ only touches the y coordinate and the second coin and there is no entanglement between the registers of the x and y axes. Using this fact, we can simply simulate two independent quantum walks on the line in parallel, or sequentially, using the same registers, which wastly decreases the memory needs of the algorithm. Running in parallel, the memory needs is now $d(4(2N+1)^2) = O(dN^2)$, or running sequentially $(4(2N+1))^2 = O(N^2)$, however the latter uses dN steps, instead of N, and d measurements, instead of 1.

4.5.2 Generalization using a single higher dimensional coin

Using this method, we generalize quantum walking to arbitrary directed graphs. To do so, we first generalize to regular graphs, then discuss how non-regular graphs can be regularized.

In a d-regular graph, the vertices have d neighbours, meaning the walker must choose from d possible directions at every step. Thus the coin is d dimensional. Using this idea as a starting point, we can reverse engineer the generalized walk from the walk on the line by starting from the evolution operator, which assumes nothing about the given graph or coin.

$$\mathbf{U} = \mathbf{S}\hat{\mathbf{C}} = \mathbf{S}(\mathbf{I} \otimes \mathbf{C})$$

In this equation, $\mathbf{C}$ can be any d-dimensional unitary matrix. The definition of $\mathbf{S}$ requires more thought, since $\mathbf{S}$ has to encode the graph's structure, while also satisfying [PostulateII].

In one dimension, $\mathbf{S}$ was defined the following way:

$$\mathbf{S} = \mathbf{L} \otimes |0\rangle \langle 0| + \mathbf{R} \otimes |1\rangle \langle 1| .$$

$|0\rangle \langle 0|$ and $|1\rangle \langle 1|$ were matrices that checked the current state of the coin and "activated" the transition $\mathbf{L}$ or $\mathbf{R}$ accordingly. In d dimensions, the coin has d possible states, i.e. "sides"

$$\{|0\rangle , |1\rangle , \ldots , |d-1\rangle\},$$

which means $\mathbf{S}$ will be constructed using d transition matrices, describing the graph's structure

$$\mathbf{S} = \mathbf{S}_0 |0\rangle \langle 0| + \mathbf{S}_1 |1\rangle \langle 1| + \cdots + \mathbf{S}_{d-1} |d-1\rangle \langle d-1| .$$

In the 1 dimensional case $\mathbf{L}$ and $\mathbf{R}$ described stepping to the left and to the right, which are the directed edges of the line graph and $\mathbf{L} + \mathbf{R}$ is the adjacency matrix of the line graph. To generalize this, $\mathbf{S}_0 + \mathbf{S}_1 + \cdots + \mathbf{S}_{d-1}$ is going to be the adjacency matrix of the d-regular graph.

The question is how do we construct the matrices $\mathbf{S}_0, \mathbf{S}_1, \ldots, \mathbf{S}_{d-1}$ from a given adjacency matrix of a d-regular graph? It turns out, that in order for $\mathbf{S}$ to satisfy [PostulateII], there are strict rules on how these $\mathbf{S}_i$ matrices can be defined.

It seems to be a well-known fact in the literature, that for d-regular undirected graphs with a valid edge coloring using d colors, a possible choice for the sides of the coin correspond to the colorsets of the edges. This means, that the S_i adjacency matrix contains all of the edges that have the ith color assigned to them, in both directions (i.e. S_i is symmetric). This is also mentioned in [29], however no proof is given in this book or any other books and articles I have found during research.

In the following section, I present a more generalized theorem for directed d-regular graphs formulated and proven by me. Then I discuss how the special case of the theorem for undirected graphs gives the edge coloring as a result.

Theorem 4.1. Given a coined quantum walk on a directed, d-regular graph G, in the shift operator of the walk: $\mathbf{S} = \sum_{i=0}^{d-1} \mathbf{S}_i \otimes |i\rangle \langle i|$, assuming the $\mathbf{S}_i$ are nonnegative, real matrices, it follows that they are **permutation** matrices. ▪

Proof.

According to [PostulateII], $\mathbf{S}$ must be unitary.

According to [Unitary] the columns of $\mathbf{S}$ form an orthonormal basis. This means, that the inner product of any two different columns is 0.

Let $\mathbf{S}$ be a matrix of size $(N \times N)$. Then

$$(\mathbf{S}\,|k\rangle)^{\dagger}(\mathbf{S}\,|j\rangle) = 0 \quad \forall j \neq k,\ 0 \leq j,k \leq N.$$

Since both the $\mathbf{S}_i$ matrices and the $|i\rangle\langle i|$ matrices contain only non-complex, non-negative values, this means that $\mathbf{S}$ contains also only non-complex, non-negative values. Thus the only way the inner product of two different columns can be 0 is if the columns don't contain non-zero values in the same row.

From this observation follows, that for each individual S_i matrix, no two different columns can contain non-zero values in the same row. If there were two different columns in S_i, that contained non-zero values in the same row, then that would result in $\mathbf{S}_i \otimes |i\rangle\langle i|$ containing two different columns containing non-zero values in the same row, which would result in $\mathbf{S}$ containing two different columns containing non-zero values in the same row (since no matrices contain negative or complex values), which is a contradiction.

Using [Unitary], the rows of $\mathbf{S}$ also form an orthonormal basis. With similar reasoning, it can be proven that for each individual S_i matrix, no two different rows can contain non-zero values in the same column.

From these two observations follows, that each matrix $\mathbf{S}_i$ contains exactly one non-zero value in each row and also each column. Adding the fact, that the rows and columns of $\mathbf{S}$ are normalized, means that this non-zero value must always be a 1, resulting in the $\mathbf{S}_i$ being **permutation** matrices.

□.

When this theorem is applied to undirected graphs, the adjacency matrices $\mathbf{S}_i$ can be contructed to be symmetric (since the graph's adjacency matrix is also symmetric) and in this case they correspond to a valid edge coloring using d colors (since if $\mathbf{S}_i[j,k] = 1$, then also $\mathbf{S}_i[k,j] = 1$ due to symmetry, and the $\{k,j\}$ edge has the color i, while no other edges of vertex j or k have the ith color, since $\mathbf{S}_i$ is a permutation matrix).

Applying Vizing's theorem to d-regular graphs, the graphs can be categorized into two classes:

- **Class 1 d-regular graphs**: Their edge-chromatic number is d. The construction defined in this section works for these graphs.
- **Class 2 d-regular graphs**: Their edge-chromatic number is $d+1$. [29] seems to state, that for these types of graphs, the position-coin notation does not give a way to construct a quantum walk on them (and the arc notation shall be used), however using Theorem (4.1) if we extend the method to directed graphs, it can be possible to do so. For example, the triangle is 2-regular, but its edge-chromatic number is 3. However, if we direct the edges both ways, we suddenly arrive at two cycles of length 3, for which the adjacancy matrices are permutation matrices, allowing for the construction of a unitary evolution operator.

Chapter 5

Quantum walk simulator software

In this chapter, I present the simulator software I wrote. I discuss the currently available solutions, why I chose to write the software, the architecture, the components and design patterns I used, the challenges I faced during the development and the solutions I found.

5.1 Software design

I developed the software using Python 3, based on the Strategy design pattern. It supports graphs commonly found in the literature while also providing a method for combining them, facilitating experimentation on several kinds of regular graphs. This composition is also the foundation of the quantum walk. It can simulate classical and quantum walks on the same graphs and produce a report file detailing the results. In the quantum case, the characteristics of the walk are also dependent on the type of coin used to generate the probabilities, which can be defined in several ways. The program includes the Hadamard, Grover, and Fourier coins and can easily be extended with others.

Running several simulations, I compared the behavior of classical and quantum walks and demonstrated the quantum characteristics expected from the theoretical literature, the ballistic nature of the Hadamard walk, and the cyclic property of quantum walks.

5.1.1 Currently available solutions

Since publicly accessible quantum computers currently only have around 5-10 qubits, it is not viable at the moment to run quantum walks on a real quantum computer. Hence why, when I started researching quantum walks, I quickly began looking into simulator software. While there are many of these currently available, most of them have at least one of the following issues:

1. Not maintained and developed anymore: the last commit was years ago.
2. Written in a low-level language, like C++, in a script-like fashion, with a prominent focus on memory and performance optimization while neglecting readability, modularity and extensibility.
3. Works exclusively on a specific type of graph, for example, n-dimensional lattices only.

4. Unable to compare and contrast classical and quantum walks on the same graph, running only quantum simulations.

5. Hard to understand as a novice.

There is no general, open-source solution available that is designed and developed using sound software engineering practices and an architecture that allows for experimentation with different kinds of graphs with both classical and quantum simulations available.

I intend my solution to be valuable for research purposes while also providing a readable open-source codebase for college students to study the algorithm.

5.1.2 Architecture

The architecture of my simulator program employs the Strategy design pattern, which is described in the following way:

"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." [17]

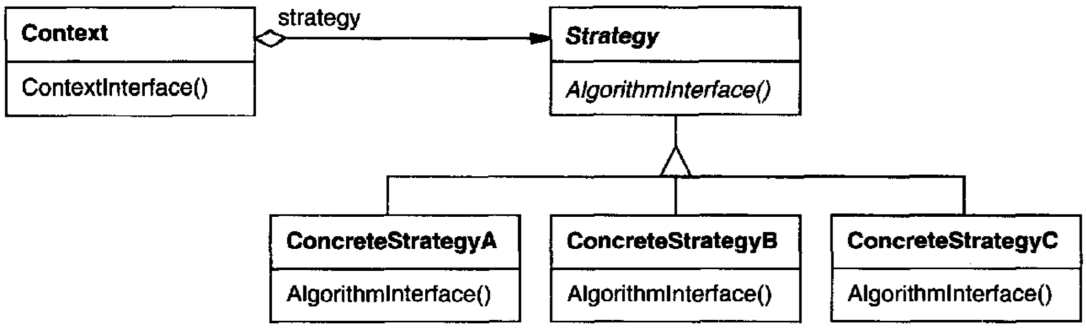


Figure 5.1: *UML diagram for the Strategy design pattern from [17]*

This is a great design pattern for research purposes since it facilitates experimentation with various algorithms for the same purpose. It also makes the code easily readable, as the Strategy interface provides an abstraction layer between the Context and the concrete implementation.

5.1.3 Language choice

With the specified goals and the architecture in mind, I needed a language that is object-oriented, easily readable by beginners and has extensive capabilities for using complex numbers, linear algebra and plotting. For these purposes, I choose the Python language. Python is concise, it reads like pseudocode and has libraries such as NumPy, SciPy and Matplotlib, and so on, covering all areas of data science. Furthermore, it is well-known and extensively used by researchers with no software engineering background, allowing for easier collaboration.

5.1.4 High level design

The source code of the software can be divided into three parts:

- Graph models
- Simulators
- Running, configuration and result collection

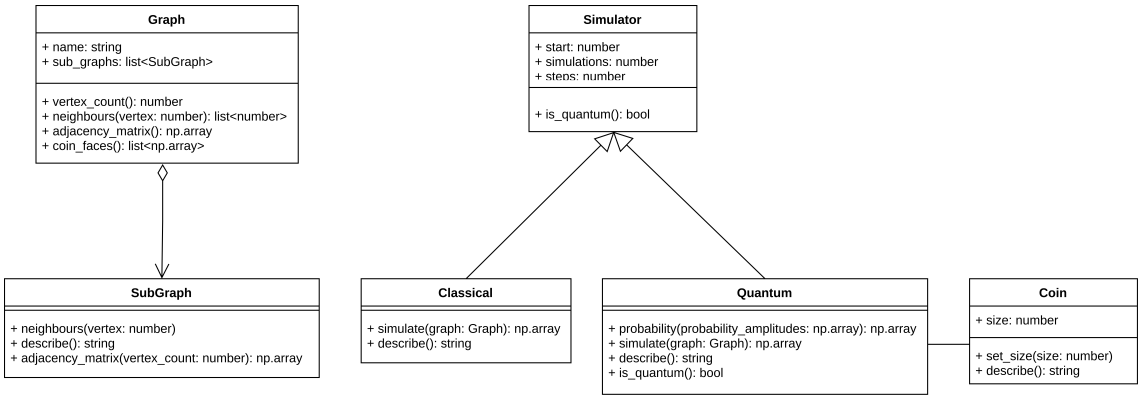


Figure 5.2: *UML diagram for the Graph models and Simulators*

On the UML diagram above, the SubGraph class is a Strategy, with the following ConcreteStrategy implementations:

- BinaryTree
- Bipartite
- Circle
- Grid
- Hypercube
- Path
- Random

Each of these employs an oracle that calculates neighbouring vertices on-the-fly.

Furthermore, the Simulator is also a Strategy, implemented by the Classical and the Quantum classes, the latter using the Coin Strategy, implemented by the Hadamard, Grover and Fourier (DFT) classes.

(For a cleaner diagram, I did not picture the SubGraph and Coin implementations.)

5.1.5 Graph models

I ran several experiments on various graphs while researching quantum graph walks, including paths, circles, bipartite graphs, hypercubes, and grids. Initially, I directly generated and stored their adjacency matrices, however, I quickly ran into memory scaling issues with this approach. Furthermore, in quantum research, graphs are typically built like 'Legos', glueing together a few common types, which was challenging to do with my original approach.

To combat these issues, I switched from the adjacency matrix representation to the oracle representation. The oracle is a function that returns the neighbours of a given vertex. Since I was using common graphs, I could calculate neighbouring indexes on-the-fly without storing anything about these graphs and only querying what is needed at the current step, dramatically reducing the memory requirements of the graph models.

5.1.6 Simulators

I implemented a classical and a quantum simulator class. The quantum simulator can currently simulate directed k regular graphs, however since the permutation matrix decomposition, or in the undirected case, the edge coloring of the matrix is an NP-complete problem, in the current setup, the graph oracle must be implemented in a way that returns the neighbours in the same color order for all inputs. Since the human programmer designs the oracle, this is not a critical limitation at the moment. I have implemented a check as a safety guard to ensure the resulting shift matrices are unitary in case an error is made while coding one of the oracles.

5.1.7 Running, configuration and result collection

Using the above classes, I developed a framework in which experimental runs can be configured very quickly. The results of the run are collected in an aggregated Latex document, using Matplotlib for creating various graphics. It contains the given graph, the named type of the subgraphs, the adjacency matrices, the distribution results of the simulations, including empirical hitting and mixing times and the eigenvalues and eigenvectors of the evolution operators. In the following chapter, I present several examples collected from these Latex reports of my experiments.

5.1.8 Source code availability

My simulator software is available under the open-source MIT license on my personal Github account, under the following link:

`https://github.com/nemkin/quantum-walk`

5.2 Presentation of poperties of quantum walks

In this chapter, I review classical and quantum walking on three specific graphs, for which interesting results can be observed. I discuss the evolution of the probability distributions and the hitting and mixing times for classical and quantum walks with the Hadamard, Grover and Fourier (DFT) coins.

5.2.1 Walks on the line

The first graph to be reviewed is the line (with 100 vertices), using the adjacency matrix below. It is important to note, that an extra edge has to be added to connect the two ends of the line (see Theorem (4.1) for details).

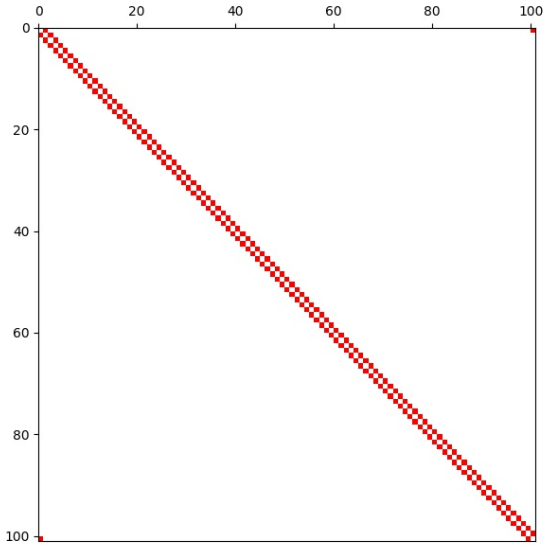


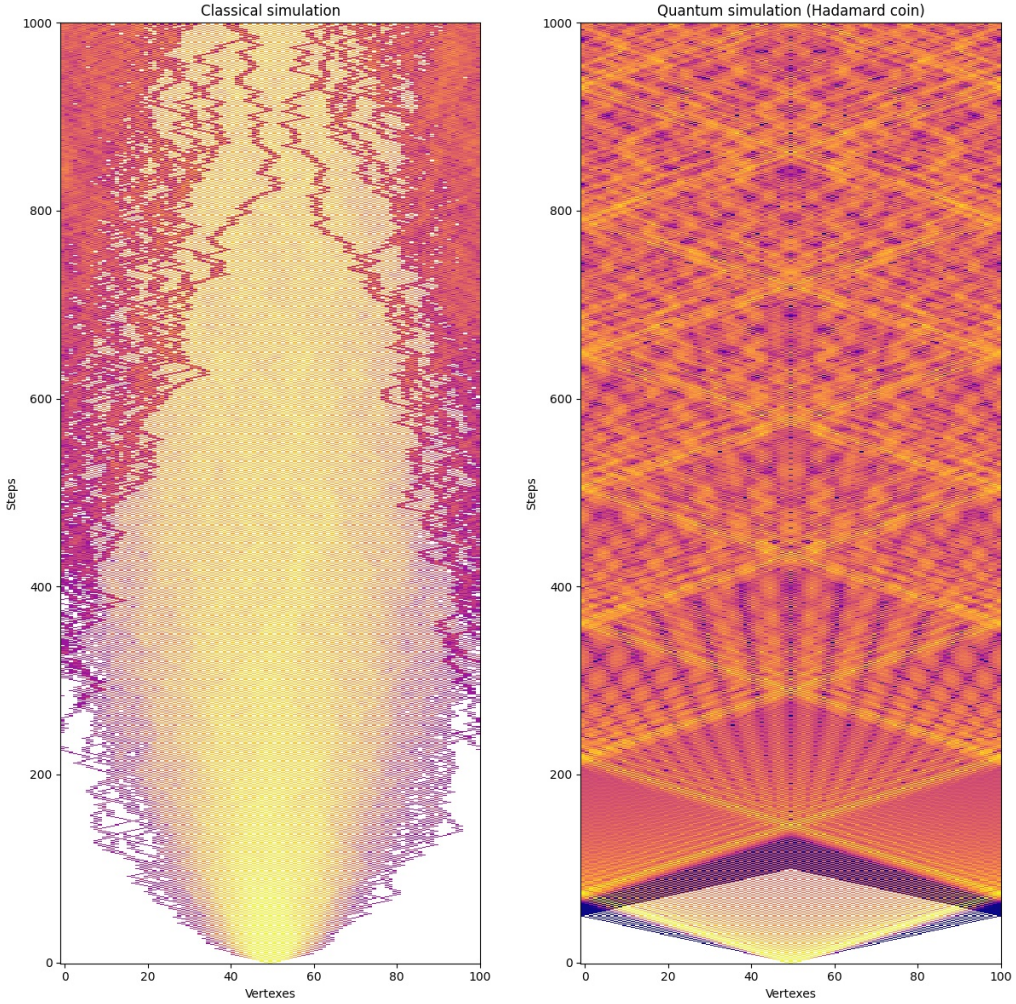
Figure 5.3: *Adjacency matrix of the line*

In the following pictures, we can see the changes in the probability distribution during the walk. The x axis contains the vertices, and the y axis contains the steps. The walker starts from the centre, and in the classical case, multiple runs are done to arrive at a probability distribution, while in the quantum case, a single walker is enough, as it spreads in superposition over the graph.

The ballistic nature of the walk can be seen from steps 0 to 50, where the bright yellow diagonals represent a strong probability concentration spreading to the two ends of the line. When the probability bumps reach the sides, they cross over and travel to the opposite ends.

From steps 50 to 200, we can see secondary, tertiary, and further yellow bumps travelling alongside the main ones. These reach the ends slower and cross over each other later. This results in a beautiful weaved pattern in the picture.

Since the line is a 2-regular graph, 2 dimensional coins are used. The 2 dimensional Hadamard-coin and Fourier-coin are identical, while the 2 dimensional Grover-coin results in the walker not moving away from the starting position, hence why only the Hadmard coin is shown in the distributions.



(a) *Classical walk* **(b)** *Quantum walk with the Hadamard coin*

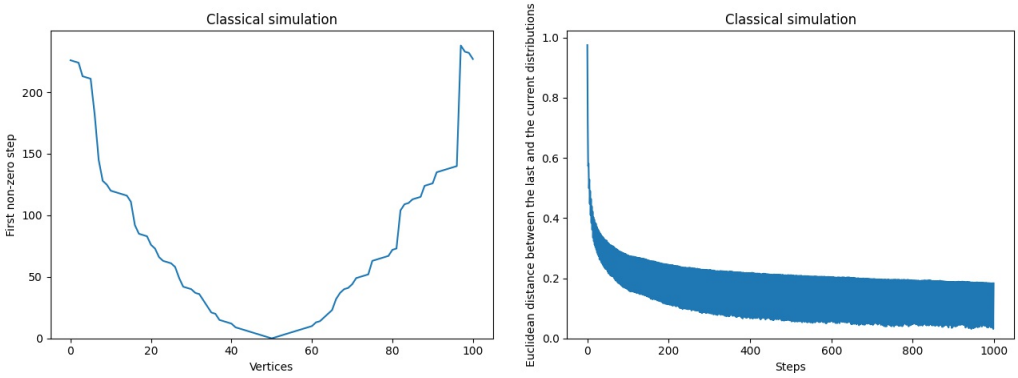
Figure 5.4: *Probability distribution of classical and quantum walks on the line*

I have empirically measured hitting and mixing times for the different types of walks. Hitting time is the expected number of steps to reach a specific vertex from the starting point. For this, I have plotted the number of steps it took to first reach a particular vertex from the starting point. Mixing time is the number of steps it takes before reaching the stationary distribution with ε error. For this, I have plotted the Euclidean difference between the walk's current and the end distribution.

In the following pictures, we can see the classical hitting and mixing times. Since my classical simulator approximates the distribution by running multiple walkers, the hitting time is slightly asymmetric. We can see from comparing the classical and the quantum hitting times that the quantum walk spreads faster than the classical one.

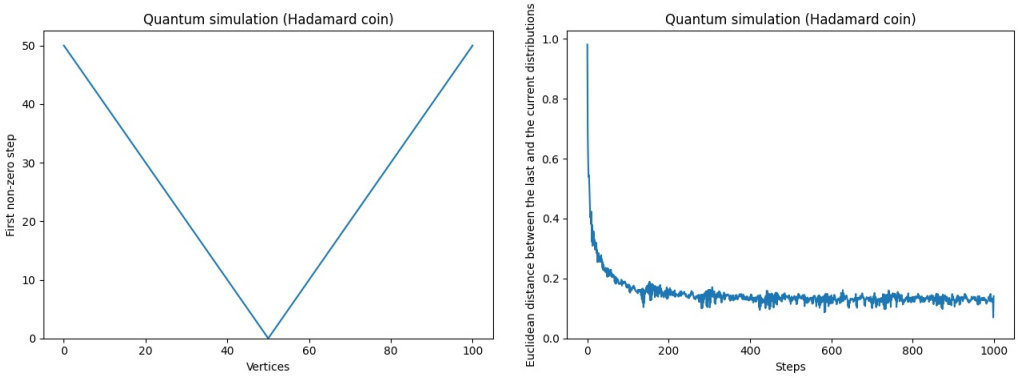
On the classical mixing time, we can see that the walker has not reached the stationary distribution. This is because until the walker reaches the ends of the line, the graph is essentially bipartite with two different limiting distributions and only after crossing over to the other side can the walk spread uniformly. This can also be seen in the probability distribution image above. At around step 300, the colour of the image intensifies at the sides. Before that, the distribution alternates between odd and even indexes having 0 probability, resulting in a chessboard pattern of white and colorful rectangles.

The quantum walk is mixing much better, as can be seen by the mixing time and the distribution image as well.



(a) *Classical hitting time* **(b)** *Classical mixing time*

Figure 5.5: *Classical hitting and mixing times on the line*



(a) *Hadamard hitting time* **(b)** *Hadamard mixing time*

Figure 5.6: *Quantum (Hadamard) hitting and mixing times on the line*

5.2.2 Walks on the grid

The second graph reviewed is the 2 dimensional grid (with $4 \times 4 = 16$ vertices), using the adjacency matrix below.

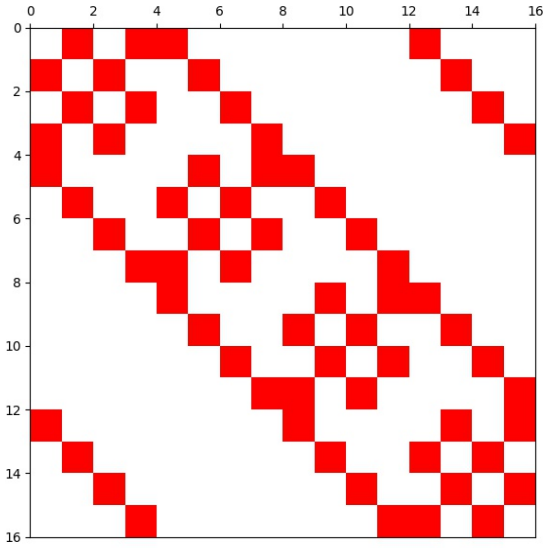
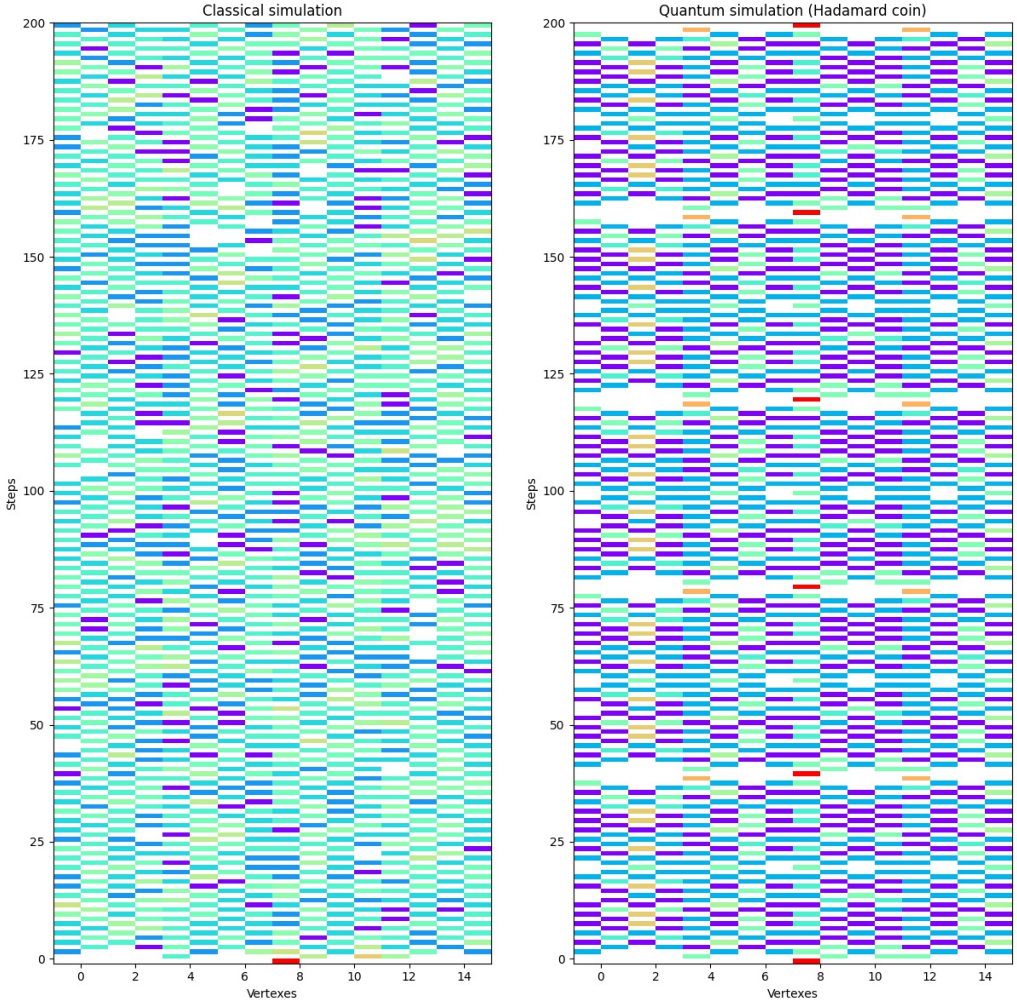


Figure 5.7: *Adjacency matrix of the grid*

The following 4 images contain the classical, the quantum Hadamard, the quantum Grover and the quantum Fourier walks on the grid. The classical walk quickly spreads over the graph since all vertices are close to each other (as opposed to the line, where the maximum distance is large).

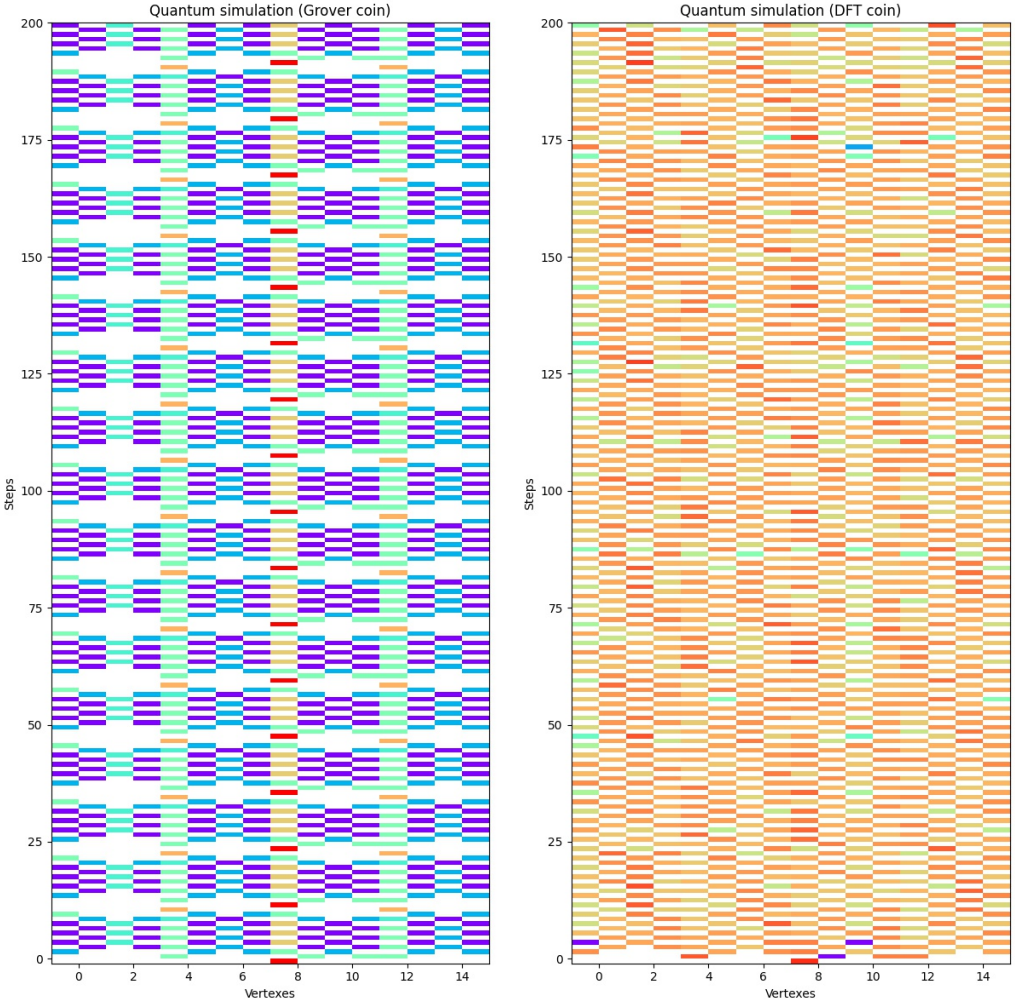
In the quantum case, using the Hadamard and Grover coins, an important quality of the quantum walks can be distinctly observed: quantum walks are periodic since the eigenvalues of the evolution operator are complex roots of unity. Furthermore, by choosing a vertex count that is a power of 2, I was able to create an evolution operator that has specific eigenvalues that result in the walker returning to its starting position with 100% probability (see the repeated red rectangles in the images), showing the cyclic nature of the quantum walk.

The Fourier coin mixes the state much better, resulting in no specific order in that image.



(a) *Classical walk* **(b)** *Quantum walk with the Hadamard coin*

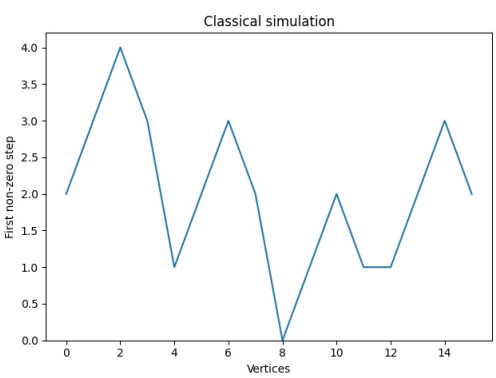
Figure 5.8: *Probability distribution of classical and quantum walks on the grid*



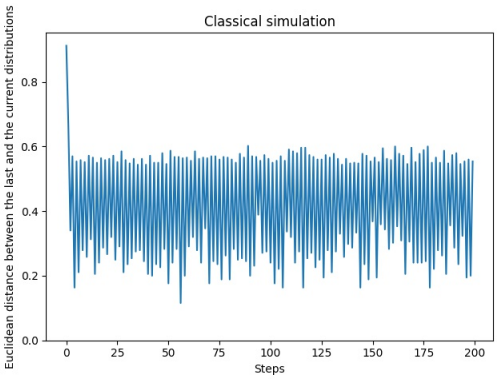
(a) *Quantum walk with the Grover coin* **(b)** *Quantum walk with the Fourier coin*

Figure 5.9: *Probability distribution of quantum walks on the grid*

Interestingly, the hitting times of the 4 walks are similar. This is probably due to the fact, that the graph is small, which allows the classical walk to spread just as quickly as its quantum counterpart. We can see, that neither of the walks reached a stationary distribution. In the quantum case, we know that the walks are periodic, so there is no stationary distribution, while in the classical case, the graph is bipartite.

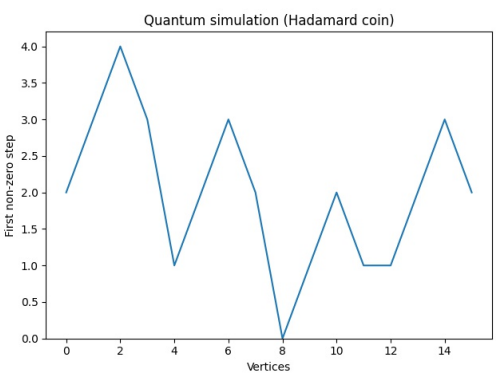


(a) *Classical hitting time*

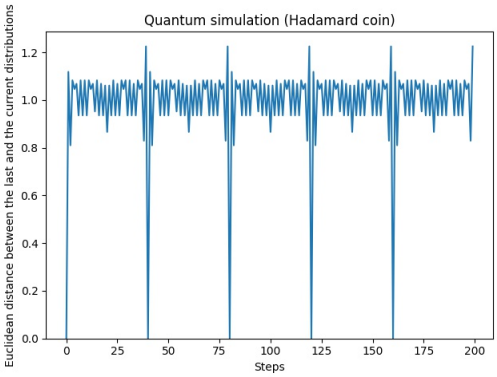


(b) *Classical mixing time*

Figure 5.10: *Classical hitting and mixing times on the grid*

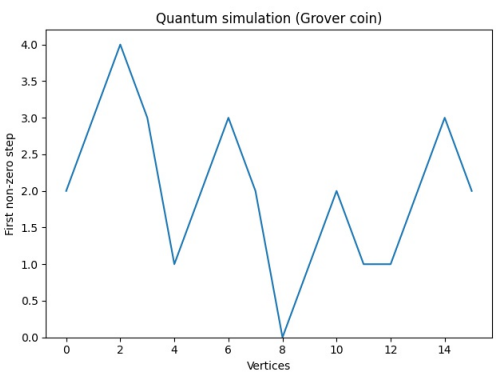


(a) *Hadamard hitting time*

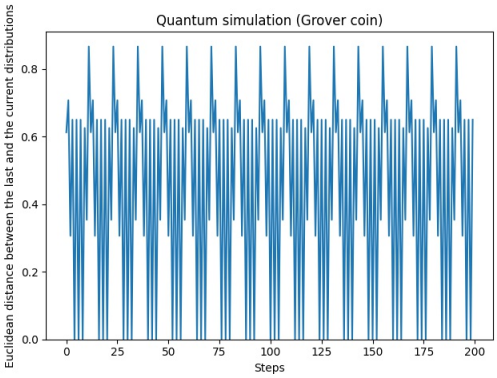


(b) *Hadamard mixing time*

Figure 5.11: *Quantum (Hadamard) hitting and mixing times on the grid*

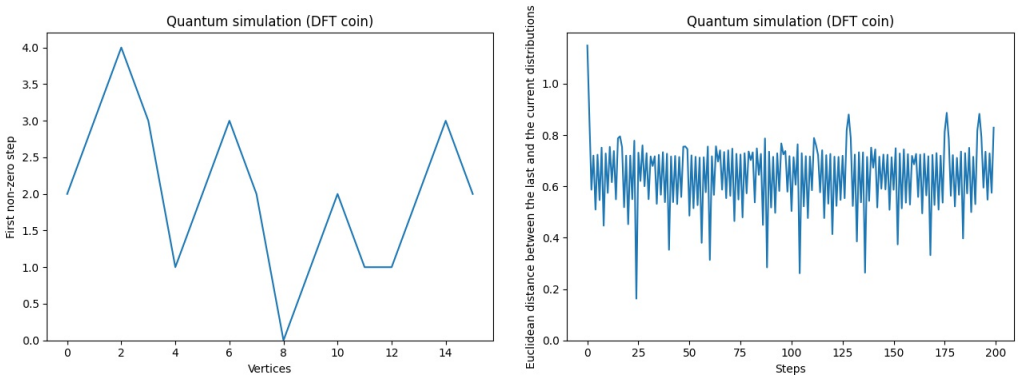


(a) *Grover hitting time*



(b) *Grover mixing time*

Figure 5.12: *Quantum (Grover) hitting and mixing times on the grid*



(a) *Fourier hitting time* **(b)** *Fourier mixing time*

Figure 5.13: *Quantum (Fourier) hitting and mixing times on the grid*

5.2.3 Walks on hypercube

The third graph reviewed is the 4 dimensional boolean hypercube (with $2^4 = 16$ vertices), using the adjacency matrix below.

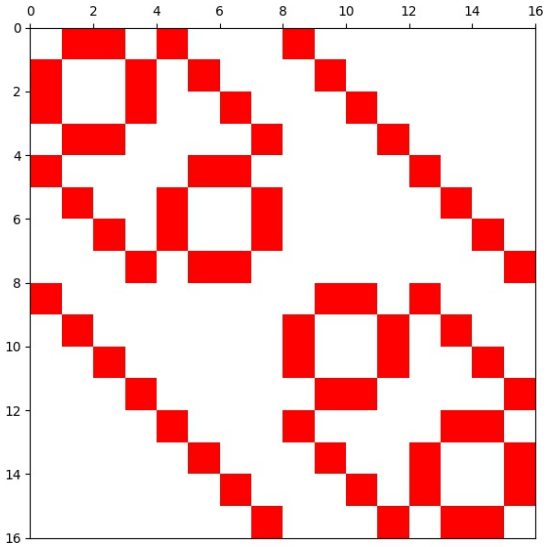
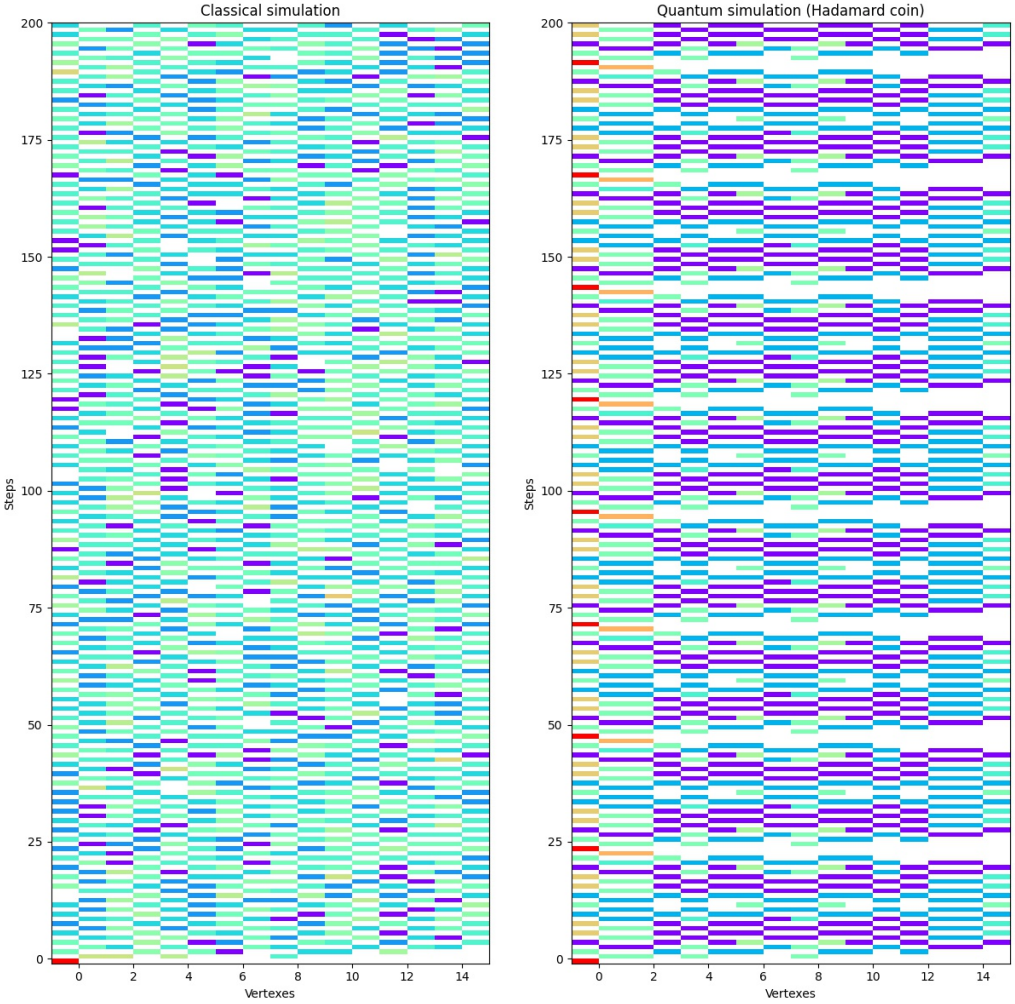


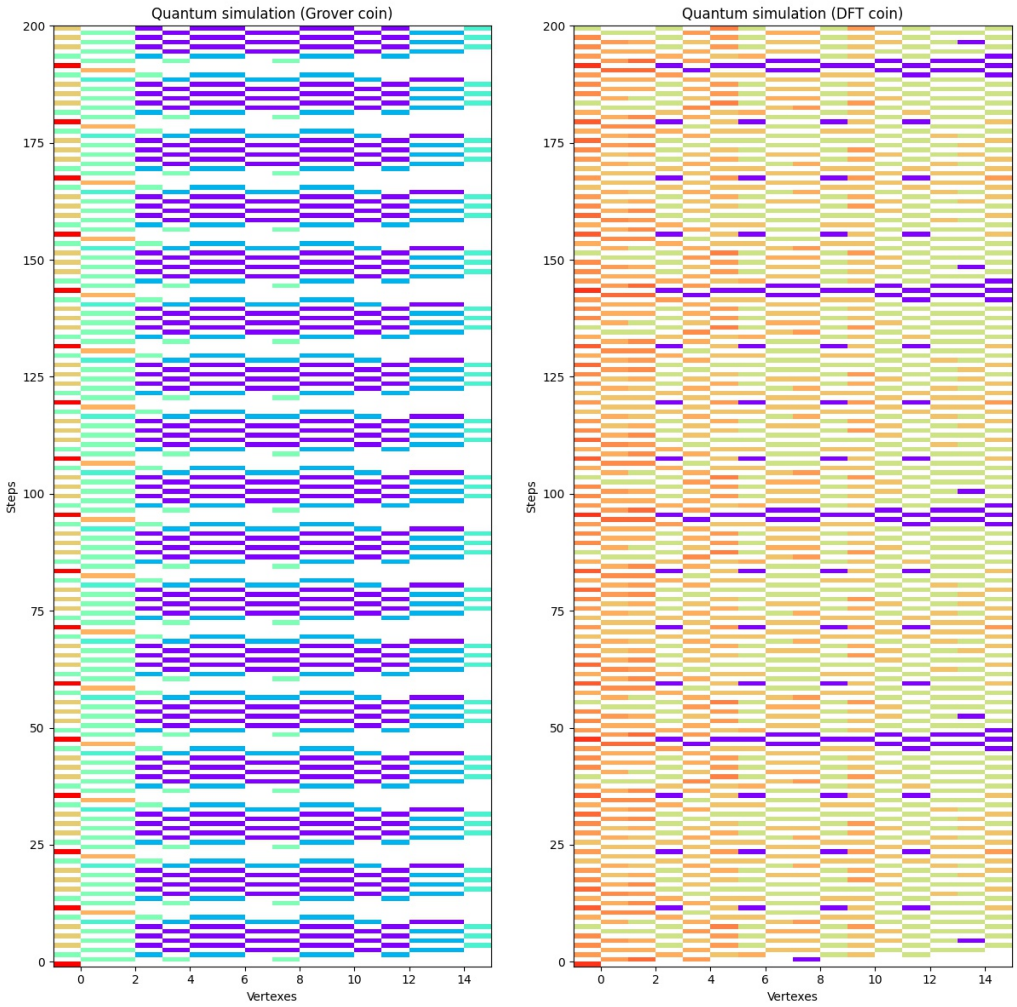
Figure 5.14: *Adjacency graph of the hypercube*

Similarly to the grid, the walks are recurrent (cyclic) in the Hadamard and Grover case. Interestingly in this case the periodicity can be visibly observed with the Fourier coin, however the walk does not return to its original starting point with 100% probability. This is due to the eigenvalues of the evolution operator just being slightly off, so there is no small exponent for which the evolution operator is the identity.



(a) *Classical walk* **(b)** *Quantum walk with the Hadamard coin*

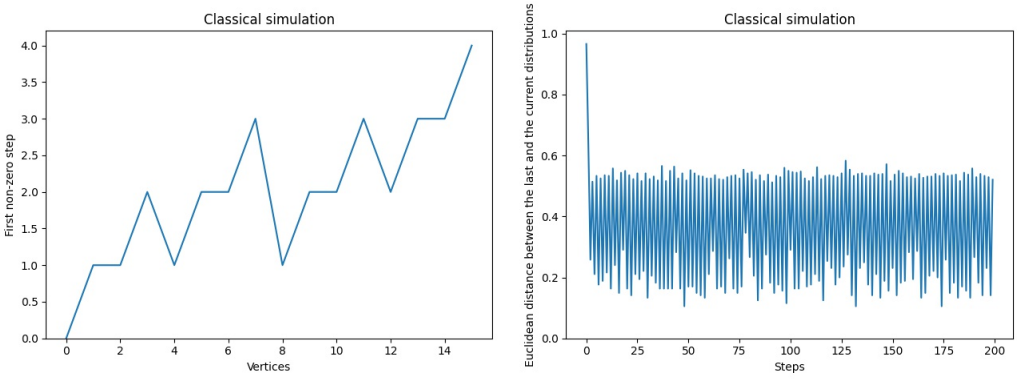
Figure 5.15: *Probability distribution of classical and quantum walks on the hypercube*



(a) *Quantum walk with the Grover coin* **(b)** *Quantum walk with the Fourier coin*

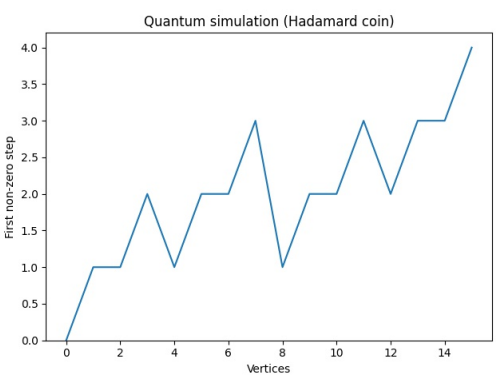
Figure 5.16: *Probability distribution of quantum walks on the hypercube*

Similarly to the grid, the hitting times are identical, since the graph is small and the classical walk's disadvantage is not visible. The hypercube is also bipartite, resulting in no stationary distribution for the classical walk either.

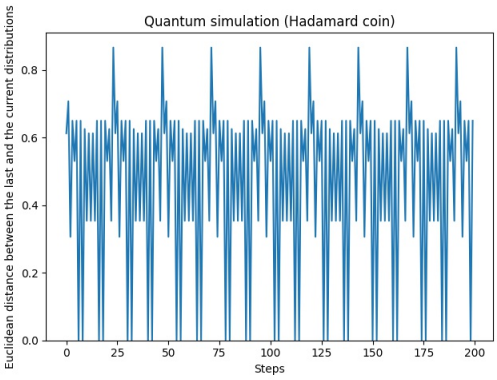


(a) *Classical hitting time* **(b)** *Classical mixing time*

Figure 5.17: *Classical hitting and mixing times on the hypercube*

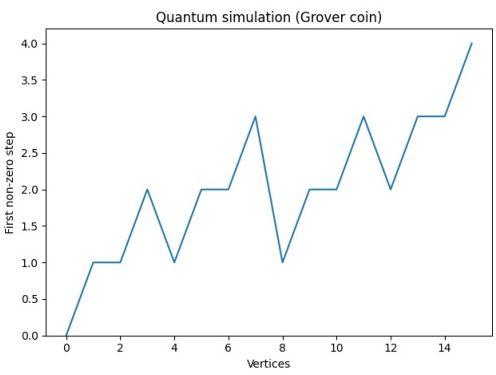


(a) *Hadamard hitting time*

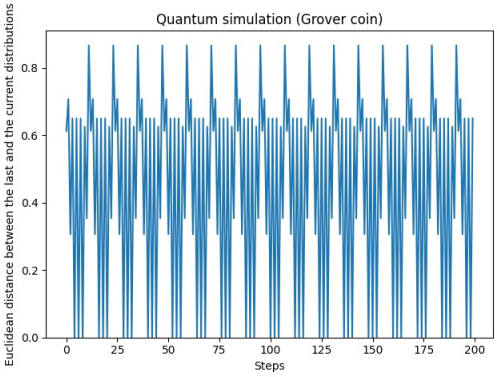


(b) *Hadamard mixing time*

Figure 5.18: *Quantum (Hadamard) hitting and mixing times on the hypercube*

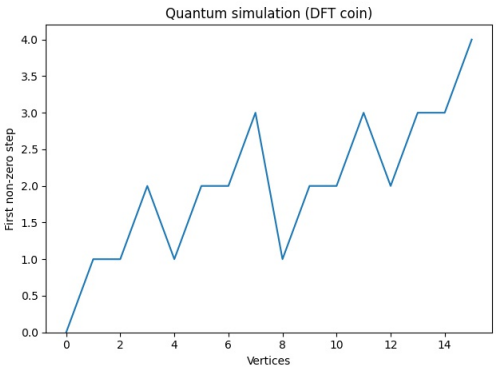


(a) *Grover hitting time*

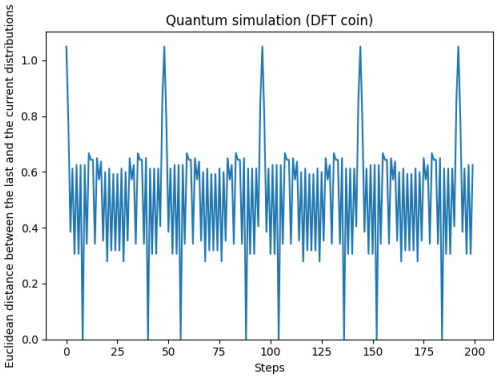


(b) *Grover mixing time*

Figure 5.19: *Quantum (Grover) hitting and mixing times on the hypercube*



(a) *Fourier hitting time*



(b) *Fourier mixing time*

Figure 5.20: *Quantum (Fourier) hitting and mixing times on the hypercube*

Chapter 6

Grover's quantum search algorithm

Grover's search algorithm[19] is a quantum algorithm framework, that takes a user-defined solution verifier algorithm (the oracle) and turns it into a $\Theta(\sqrt{N})$[5] solver. This provides a quadratic speedup over the classical brute force equivalent.

6.1 Introduction to Grover's search algorithm framework

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

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

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

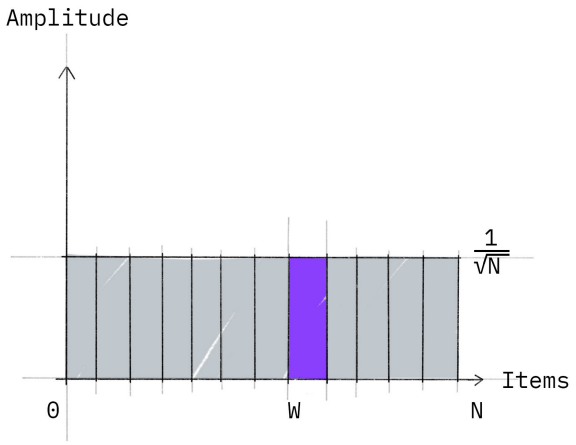


Figure 6.1: *Grover starts out with the uniform distribution[16]*

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

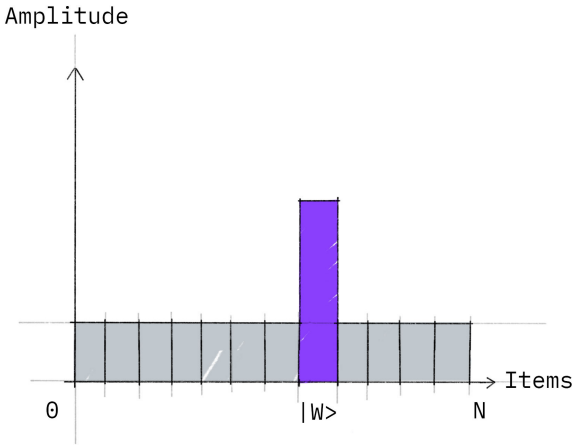


Figure 6.2: *Grover amplifies the amplitude(s) of the correct solution(s)[16]*

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

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

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

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

However, we do not have access to this probability distribution or the high probability elements in it. The only thing we can do is read the register, which is an operation that

samples a single entity from the current probability distribution in the register, destroying it in the process. We are unable to 'iterate' the contents of the register or know what the probability of the resulting element was from the sampling.

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

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

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

6.2 Showcasing the algorithm on a simple task

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

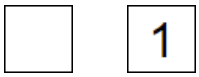
1	2	3	4	5	6	7	8	9
7	8	9	1	2	3	4	5	6
4	5	6	7	8	9	1	2	3
3	1	2	8	4	5	9	6	7
6	9	7	3	1	2	8	4	5
8	4	5	6	9	7	3	1	2
2	3	1	5	7	4	6	9	8
9	6	8	2	3	1	5	7	4
5	7	4	9	6	8	2	3	1

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

Figure 6.3: *Sudoku puzzle*

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

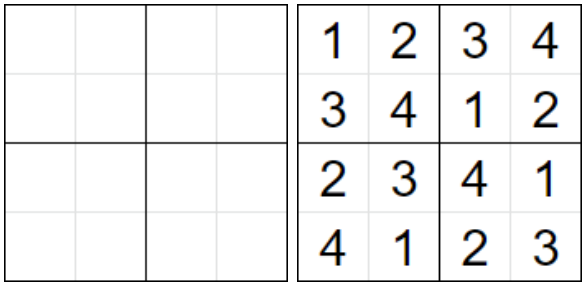
The only solution for $n = 1$ is trivial.



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

Figure 6.4: *Sudoku puzzle* ($n = 1$)

An example solution for $n = 2$.

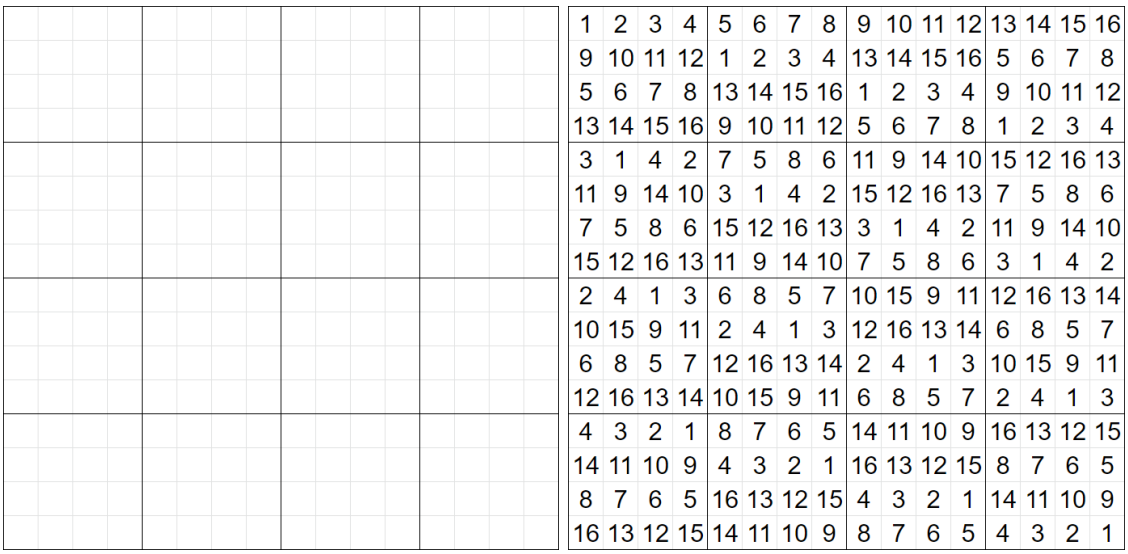


(a) *Empty* **(b)** *Solved*

Figure 6.5: *Sudoku puzzle* ($n = 2$)

$n = 3$ is normal Sudoku.

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



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

Figure 6.6: *Sudoku puzzle* ($n = 4$)

This generalized version of Sudoku is in the time complexity class NP, which means the Grover search algorithm can be used to solve it. Furthermore, this problem has key components in it which are analogous to protein folding, but much simpler than that.

6.3 Designing a quantum solver for the Sudoku puzzle

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

6.3.1 Register definitions

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

$$\text{cell}[i][j] = |0 \dots 010 \dots 0\rangle\,, \quad \forall\, (0 \leq i, j < n^2).$$

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

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

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

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

6.3.2 Oracle operator

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

6.3.2.1 Constraint definitions

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

The verifications are defined as follows.

Cells shall be one-hot encoded

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

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

Numbers in each row shall be unique

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

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

Numbers in each column shall be unique

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

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

Numbers in each square shall be unique

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

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

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

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

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

6.3.2.2 Implementation of the UNIQUE_ONE constraint

In order to implement a UNIQUE_ONE($[x_0, \ldots, x_{n-1}]$) constraint, we use the WeightedAdder component from Qiskit. This takes n qubits and sums them up into

a $\log_2(n)$ sized array. We want the sum to be exactly 1, which means that the output of the sum array should be equal to $|0\ldots01\rangle$. Adding a NOT gate to the least significant qubit, the output should be $|0\ldots0\rangle$. This can be tested using a multi-controlled NOT, or $M-CNOT$ gate.

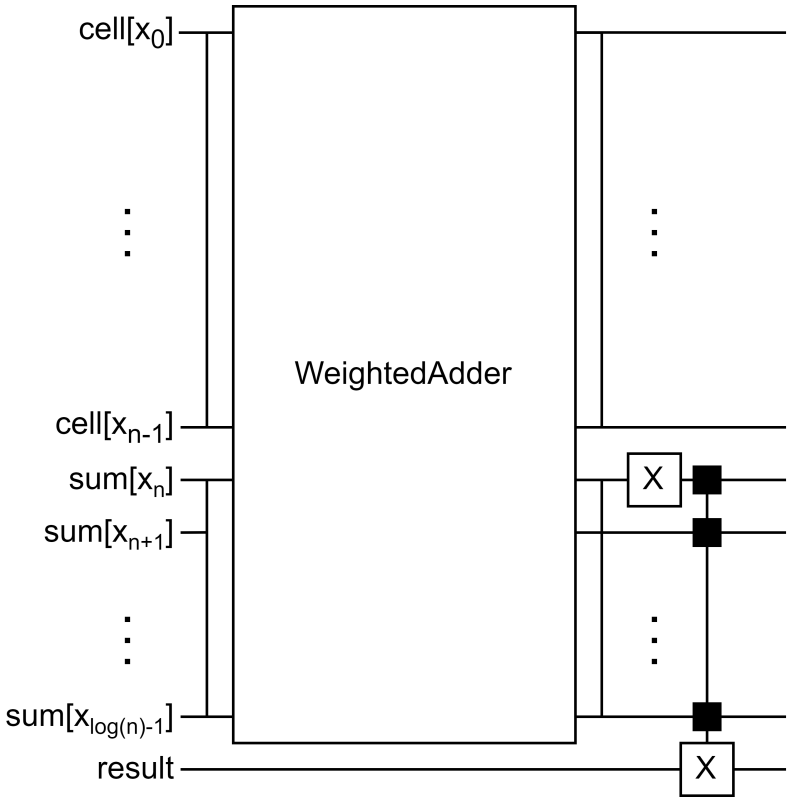


Figure 6.7: *UNIQUE_ONE constraint implementation*

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

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

6.3.3 Grover's framework: The amplitude amplification technique

This chapter is based on the chapter on Grover's algorithm from the Qiskit Texbook[16].

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

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

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

This is done by applying an n-dimensional Hadamard-matrix to the $|0\ldots0\rangle$ vector:

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

We can see this initial state in the following figure.

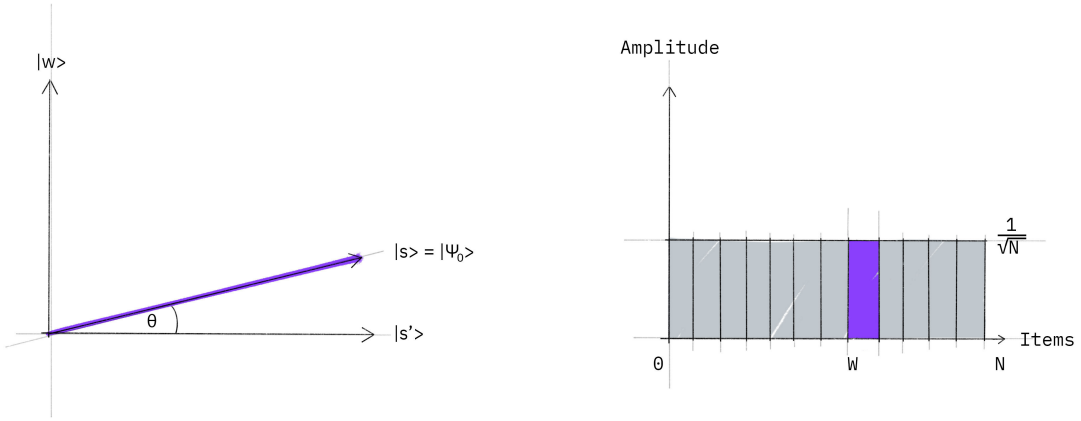


Figure 6.8: *Initialization[16]*

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

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

This effect can be seen on these figures:

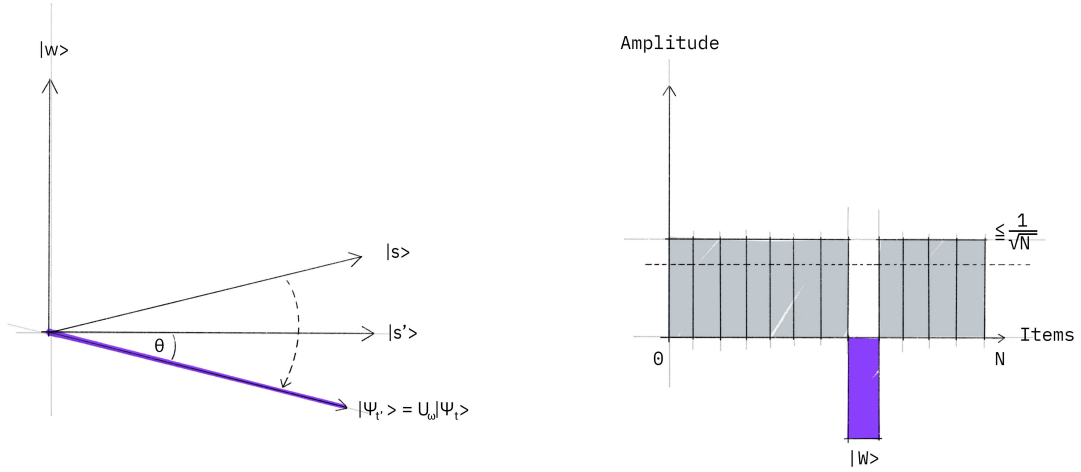


Figure 6.9: *Phase kickback[16]*

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

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

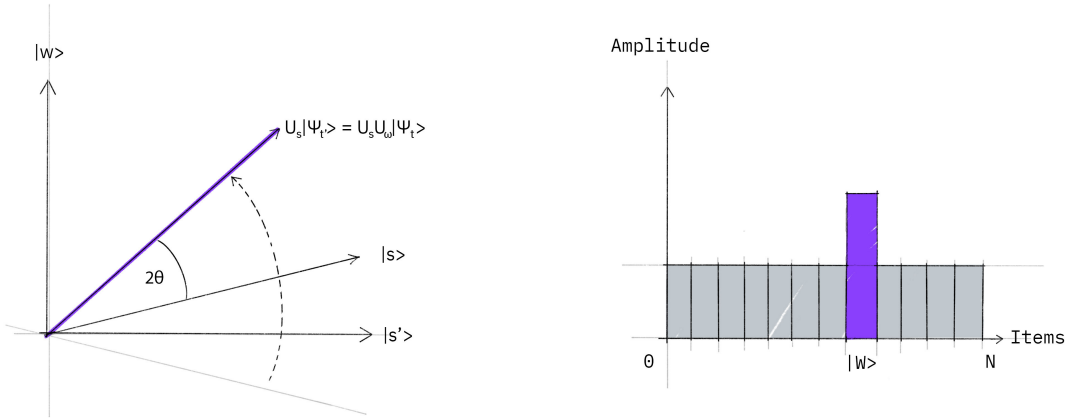


Figure 6.10: *Reflect over the average amplitude[16]*

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

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

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

6.4 Memory problems in Grover search implementations

In this capter I have introduced Grover's algorithm and how to use it to solve a generalized version of the Sudoku puzzle. I have fully implemented the algorithm described above in Qiskit, using the following 4 quantum logic gates: the Hadamard (for the phase-kickback), the Grover (the diffuser operator), the Sum (WeightedAdder) and the Multi-controlled NOT gate (for the oracle bit).

Unfortunately, as I further discuss in the next chapter, Grover search implementations in practice turn out to use such a large amount of memory for even small problem instances, that I was unable to run them on my PC. For example, a (3×3) Sudoku matrix would require 67 TB of memory. As I was formulating parts of the verifier algorithm for quantum protein folding, I have run into a similar problem with memory usage.

This is why I started working on quantum algorithm simulation implementations focusing on memory efficiency.

Chapter 7

Memory efficient quantum algorithm simulation framework

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

7.1 Problems with quantum protein folding

The WeightedAdder component came to my attention, because a natural way to encode protein structures is by creating a 2D or 3D grid and laying the aminoacid chain down on it[12]. From a single vertex in a 3D grid, we can step in 4 or 6 directions: up, down, left, right and inwards and outwards in the 3D case. We can encode these naturally, using one-hot encoding, by introducing 6 bits of information. If we assume that the chain starts in the origin, then we can encode a chain shape by giving the directions of the $(n-1)$ steps it takes.

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

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

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

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

7.2 Design goals

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

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

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

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

7.3 Quantum registers

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

$$\begin{aligned}
|0\ldots000\rangle &\rightarrow c_0 \\
|0\ldots001\rangle &\rightarrow c_1 \\
|0\ldots010\rangle &\rightarrow c_2 \\
|0\ldots011\rangle &\rightarrow c_3 \\
&\ldots \\
|1\ldots111\rangle &\rightarrow c_{2^n-1},
\end{aligned}$$

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

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

7.3.1 General solution

Let there be r registers in the system,

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

and let n_i be number of qubits in register R_i, where $0 \leq i < r$.
The total number of qubits in the system is therefore

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

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

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

The number of qubits U must operate on is therefore

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

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

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

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

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

and its contents are

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

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

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

is a binary indexing sequence on C and it corresponds to

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

where

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

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

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

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

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

and

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

and it corresponds to

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

where

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

and

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

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

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

and

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

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

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

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

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

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

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

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

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

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

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

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

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

$$R'[|b'_{n-1}, \ldots, b'_k|a'_{k-1}, \ldots a'_0\rangle] = \sum_{\substack{0 \leq i < m \\ \forall b_i \in \{0,1\}}} U[|a'_{m-1}, \ldots, a'_0\rangle][|b'_{m-1}, \ldots, b'_0\rangle] \cdot C'[|b'_{m-1}, \ldots, b'_0\rangle]. \tag{7.1}$$

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

In cases where the operation is a mapping or aggregation itself (such as the WeightedAdder from the Grover's search), it can be performed without generating the column itself, the same permutation logic is used, but instead of generating an entire row of U, the

$|a'_{m-1}, \ldots, a'_0\rangle$ "output index" is calculated based on the $|b'_{m-1}, \ldots, b'_0\rangle$ "input index" by a function

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

which then replaces the matrix multiplication as follows:

$$R'[|b'_{n-1}, \ldots, b'_k|a'_{k-1}, \ldots a'_0\rangle] = \sum_{\substack{0 \le i < m \\ \forall b_i \in \{0,1\} \\ u(|b'_{m-1},\ldots,b'_0\rangle) = |a'_{m-1},\ldots,a'_0\rangle}} C'[|b'_{m-1}, \ldots, b'_0\rangle]. \tag{7.2}$$

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

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

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

7.3.2 Presenting the solution on an example

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

The probability amplitudes stored are the following:

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

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

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

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

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

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

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

1. Apply $U^{8\times 8}$ to the probability amplitudes corresponding to $R_{1,0} = |0\rangle$:
   $[c_0, c_1, c_4, c_5, c_8, c_9, c_{12}, c_{13}] = [c'_0, c'_1, c'_2, c'_3, c'_4, c'_5, c'_6, c'_7]$.
   The resulting vector is the first half of the complete result:
   $[r'_0, r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7]$.
2. Apply $U^{8\times 8}$ to the probability amplitudes corresponding to $R_{1,0} = |1\rangle$:
   $[c_2, c_3, c_6, c_7, c_{10}, c_{11}, c_{14}, c_{15}] = [c'_8, c'_9, c'_{10}, c'_{11}, c'_{12}, c'_{13}, c'_{14}, c'_{15}]$.
   The resulting vector is the second half of the complete result:
   $[r'_8, r'_9, r'_{10}, r'_{11}, r'_{12}, r'_{13}, r'_{14}, r'_{15}]$.
3. Iterate over all values for the untouched register R_1 and aggregate the results:
   $[r'_0, r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7, r'_8, r'_9, r'_{10}, r'_{11}, r'_{12}, r'_{13}, r'_{14}, r'_{15}]$..
4. Revert the mapping to the original indexes:
   $[r'_0, r'_1, r'_2, r'_3, r'_4, r'_5, r'_6, r'_7, r'_8, r'_9, r'_{10}, r'_{11}, r'_{12}, r'_{13}, r'_{14}, r'_{15}] =$
   $[r_0, r_1, r_4, r_5, r_8, r_9, r_{12}, r_{13}, r_2, r_3, r_6, r_7, r_{10}, r_{11}, r_{14}, r_{15}]$.

7.4 Quantum operators

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

7.4.1 Hadamard

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

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

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

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

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

This equation is directly implemented and a single column of H is generated on-the-fly when H is applied, using equation (7.1).

7.4.2 Grover

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

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

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

Then G is the following matrix:

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

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

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

It is straightforward to implement G, since the the jth column is $\frac{2}{N}$, except for the jth cell, where it is $\frac{2}{N} - 1$. This matrix is also generated on-the-fly, column-by-column, using equation (7.1).

7.4.3 Sum

The sum operator is one, that can be represented directly using equation (7.2), by defining the

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

function.

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

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

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

Then, u is defined as

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

7.4.4 Multi-controlled NOT

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

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

7.5 Implementation and design patterns

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

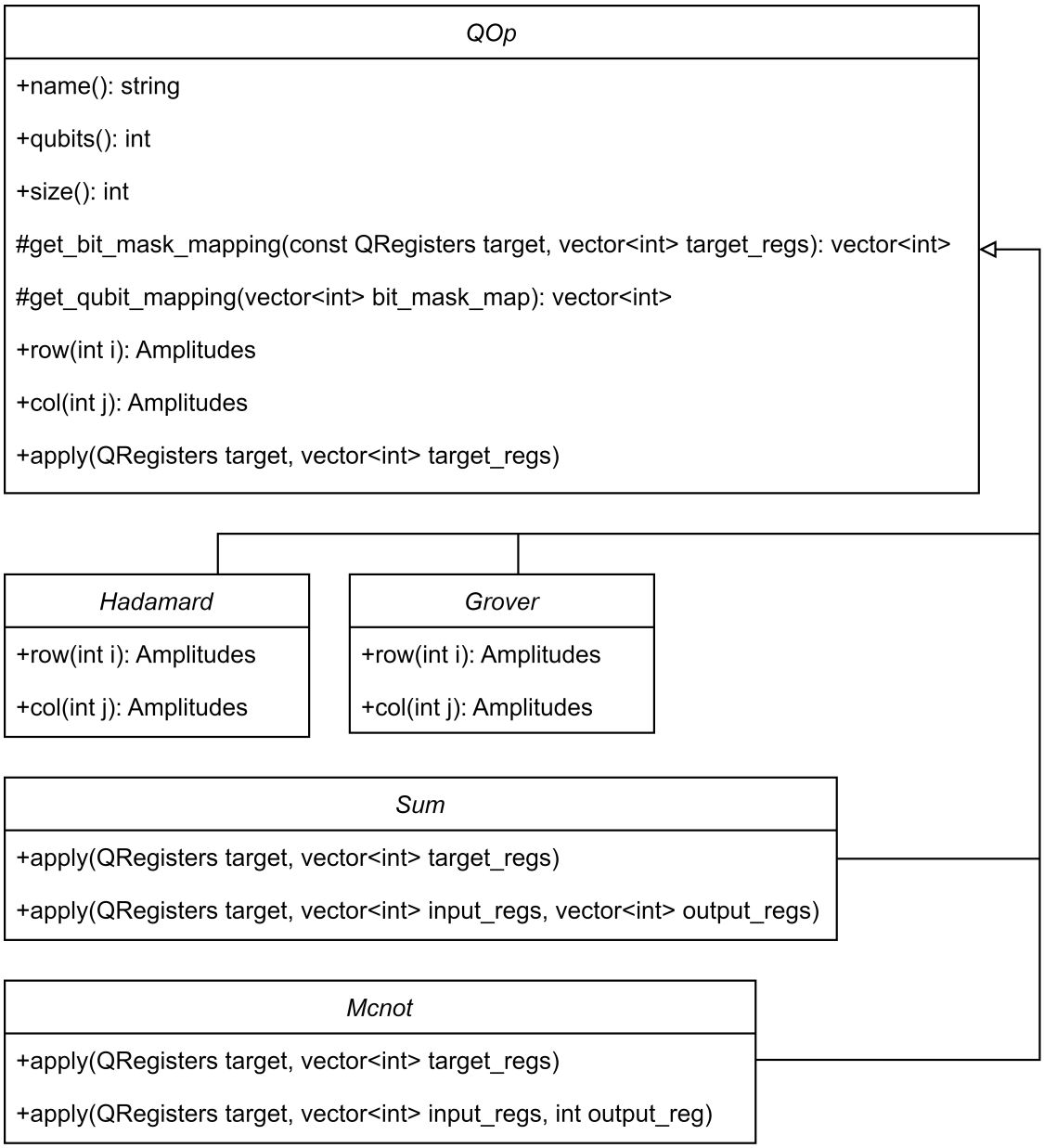


Figure 7.1: *Strategy and Visitor pattern*

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

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

In order to achieve this, I have implemented the Strategy design pattern. The intent of this pattern, according to the Design Patterns book[17], is to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. In this pattern, a common interface is defined for all operators, with the implementation dependent on the specific operator. This common

interface will later allow me to pontentially generate circuit diagrams for an implemented quantum algorithm.

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

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

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

The knowledge of how an operator should be applied to the registers should be given to the operator itself, since the framework relies on clever, operator-specific memory-efficient implementations to function. This is exactly what the Visitor pattern is used for. According to the Design Patterns book[17] we can *"Use the Visitor pattern when many distinct and unrelated operations need to be performed on objects in an object structure, and [we] want to avoid "polluting" their classes with these operations. Visitor lets [us] keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them."*

7.6 Source code availability

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

https://github.com/nemkin/qmem

Chapter 8

Conclusion

In this dissertation I have covered three main topics: bioinformatics and protein folding, quantum walks, and Grover's search algorithm.

8.1 Achievements

My research has started with understanding quantum computing in general, focused specifically on quantum walks. I have read several excellent sources [1, 22, 29, 31, 38] and lecture notes, and found [29] to be a comprehensive general introduction. I reformulated its descriptions to the language of matrices in an explicit manner, which matches both classical random walk descriptions and universal quantum computing hardware requirements. I believe these are easier to understand for someone with a college-level software engineering background who is new to the subject.

Besides this, I gave a generalized requirement for constructing quantum random walks on d-regular graphs employing the position-coin notation and improved the memory requirement of n-dimensional lattice walks. I presented my proofs for both of these advancements.

I have written a quantum walk simulator software, which helped me both understand quantum algorithms in general, as well as this particular algorithm. I have started looking into usecases for quantum walks and found their connection to Grover's search algorithm and solving NP-hard problems in bioinformatics.

I have looked into three bioinformatical problems: DNA sequencing, protein folding and molecular docking and explained their connection to computational problems. I am interested in computer-aided drug design, so I have chosen to further investigate protein folding and have explored Dill's simplified model for a protein structure: the HP model, which is still NP-hard, but can be implemented on a gated general-purpose quantum computer.

I have devised a way to encode a protein in this model in a quantum register, however I have ran into a problem with formulating a verifier algorithm. The simulator frameworks used too much memory, even on the much simpler, but analogous problem of Sudoku. I estimated, that the largest computable protein chain in Qiskit would contain at most four amino acids, on which the problem is trivial.

After further looking into why my incomplete solution in Qiskit uses such large amounts of memory, I have found many inefficiencies in the software, in terms of memory handling. This is due to the fact, that quantum frameworks, such as Qiskit (IBM) and Cirq (Google) focus on a different goal: to allow the programming of quantum computers their respective

vendors sell, which means that simulation, especially memory-efficient simulation, is not their primary concern. Their implementation uses sparse-matrix representation of each unitary matrix and allocates the resources for each individual instance of them. This made it very difficult and expensive to use them for my usecase, which is testing my quantum algorithms for protein folding even on relatively small inputs.

This is why I have decided to write my own quantum simulator framework, that focuses on memory efficient simulation.

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

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

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

8.2 Plans for the future

I would like to further develop my framework and use it to implement quantum protein folding in it using Grover's search algorithm framework and quantum walking.

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

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

Acknowledgements

Firstly, I would like to thank my advisor, dr. Katalin Friedl, for her commitment to spending a significant amount of time with me, discussing my research throughout my master's degree program. Without her supervision and mathematical expertise, this effort would not have been achievable.

Secondly, I would like to thank the Quantum Information National Laboratory for supporting quantum informatics research in our university.

Bibliography

[1] Yakir Aharonov, Luiz Davidovich, and Nicim Zagury. Quantum random walks. *Physical Review A*, 48:1687–1690, 1993.

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

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

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

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

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

[7] Hans-Joachim Böckenhauer and Dirk Bongartz. *Algorithmic aspects of bioinformatics*. Natural computing series. Springer, Berlin; New York, 2007. ISBN 9783540719120.

[8] Chaos. Chemical Compound docked to Protein structure (acetyltransferase) using Gold , molecular surface generated by Chimera software., July 2008. URL https://commons.wikimedia.org/wiki/File:Docking.jpg.

[9] Francis S. Collins and Leslie Fink. The Human Genome Project. *Alcohol Health and Research World*, 19(3):190–195, 1995. ISSN 0090-838X.

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

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

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

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

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

[15] Valeria Fionda. Networks in Biology. In *Encyclopedia of Bioinformatics and Computational Biology*, pages 915–921. Elsevier, 2019. ISBN 9780128114322. DOI: 10.1016/B978-0-12-809633-8.20420-2. URL https://linkinghub.elsevier.com/retrieve/pii/B9780128096338204202.

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

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

[18] Anthony J.F. Griffiths. DNA sequencing. *Encyclopedia Britannica*, September 2022. URL https://www.britannica.com/science/DNA-sequencing. Accessed 14 December 2022.

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

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

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

[22] Julia Kempe. Quantum random walks: An introductory overview. *Contemporary Physics*, 44(4):307–327, 2003.

[23] Frédéric Magniez, Ashwin Nayak, Jérémie Roland, and Miklos Santha. Search via quantum walk. *SIAM Journal on Computing*, 40(1):142–164, 2011.

[24] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, 2005. ISBN 9780521835404.

[25] Garrett M. Morris and Marguerita Lim-Wilby. Molecular Docking. In John M. Walker and Andreas Kukol, editors, *Molecular Modeling of Proteins*, volume 443, pages 365–382. Humana Press, Totowa, NJ, 2008. ISBN 9781588298645 9781597451772. DOI: 10.1007/978-1-59745-177-2_19. URL http://link.springer.com/10.1007/978-1-59745-177-2_19.

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

[27] Viktória Nemkin. Optimizing memory usage in quantum algorithm simulation. TDK, 2022. URL https://tdk.bme.hu/VIK/ViewPaper/Memoriafelhasznalas-optimalizalasa.

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

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

[30] Daniel P. Ryan and Jacqueline M. Matthews. Protein–protein interactions in human disease. *Current Opinion in Structural Biology*, 15(4):441–446, August 2005. ISSN 0959440X. DOI: 10.1016/j.sbi.2005.06.001. URL https://linkinghub.elsevier.com/retrieve/pii/S0959440X0500117X.

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

[32] Andrew W. Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander W. R. Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve Crossan, Pushmeet Kohli, David T. Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, January 2020. ISSN 0028-0836, 1476-4687. DOI: 10.1038/s41586-019-1923-7. URL http://www.nature.com/articles/s41586-019-1923-7.

[33] Thomas Shafee. Protein structure (full).png. URL https://commons.wikimedia.org/wiki/File:Protein_structure_(full).png. Accessed 14 December 2022.

[34] Mario Szegedy. Quantum speed-up of markov chain based algorithms. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 32–41, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[35] The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, October 2015. ISSN 0028-0836, 1476-4687. DOI: 10.1038/nature15393. URL http://www.nature.com/articles/nature15393.

[36] Xiao Wang, Sean T. Flannery, and Daisuke Kihara. Protein Docking Model Evaluation by Graph Neural Networks. *Frontiers in Molecular Biosciences*, 8:647915, May 2021. ISSN 2296-889X. DOI: 10.3389/fmolb.2021.647915. URL https://www.frontiersin.org/articles/10.3389/fmolb.2021.647915/full.

[37] Richard Wheeler. 1GZX Haemoglobin.png. URL https://commons.wikimedia.org/wiki/File:1GZX_Haemoglobin.png. Accessed 17 December 2022.

[38] Feng Xia, Jiaying Liu, Hansong Nie, Yonghao Fu, Liangtian Wan, and Xiangjie Kong. Random walks: A review of algorithms and applications. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 4(2):95–107, 2020.

[39] Laura M. Zahn. Filling the gaps. *Science*, 376(6588):42–43, April 2022. ISSN 0036-8075, 1095-9203. DOI: 10.1126/science.abp8653. URL https://www.science.org/doi/10.1126/science.abp8653.