Quantum Algorithms
for pattern-matching in genomic sequences

Aritra Sarkar

Delft University of Technology

Challenge the future

Quantum Algorithms

for pattern-matching in genomic sequences

by

Aritra Sarkar

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Engineering

at the Delft University of Technology,
to be defended publicly on Friday June 22, 2018 at 10:00 AM.

Student number:	4597982	
Project duration:	November 6, 2017 – June 18, 2018	
Thesis committee:	Prof. dr. ir. Koen Bertels,	Q&CE, TU Delft (Supervisor)
	Dr. ir. Carmen G. Almudever,	Q&CE, TU Delft (Daily-supervisor)
	Dr. Zaid Al-Arz,	CE, TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Fast sequencing and analysis of (microorganism, plant or human) genomes will open up new vistas in fields like personalised medication, food yield and epigenetic research. Current state-of-the-art DNA pattern matching techniques use heuristic algorithms on computing clusters of CPUs, GPUs and FPGAs. With genomic data set to eclipse social and astronomical big data streams within a decade, the alternate computing paradigm of quantum computation is explored to accelerate genome-sequence reconstruction. The inherent parallelism of quantum superposition of states is harnessed to design a quantum kernel for accelerating the search process. The project explores the merger of these two domains and identifies ways to fit these together to design a genome-sequence analysis pipeline with quantum algorithmic speedup. The design of a genome-sequence analysis pipeline with a quantum kernel is tested with a proof-of-concept demonstration using a quantum simulator.

Acknowledgements

Two years back I started my masters program in Electrical Engineering with specialisation in Computer Engineering. The focus on Quantum Computing was the factor behind my interest in this particular masters program.

I would like to express my great appreciation to my thesis supervisor Prof. Koen Bertels and daily supervisor Carmina Almudever. They gave me not only the opportunity to do my master thesis project in the field of quantum computing but also the possibility to involve myself in the overall dynamics of the entire Quantum and Computer Engineering (Q&CE) department. The multiple discussions on all the related topics were always interesting and motivating to deliver a nice contribution to this research field; I truly felt part of the team.

I would like to thank everybody in the Q&CE department for all the discussions and fun moments in and around the 10th floor of the faculty of EEMCS. Special thanks to Zaid Al-Arz for the great support and the in-depth discussions and guidance to get the maximum out of my interdisciplinary project from the field of genomic research. The various talks at QuTech helped me gain a broader overview of this field of research in Quantum Computing - specifically the courses by Leonardo DiCarlo, Stephanie Wehner and Edoardo Charbon. Also, I would like to separately thank Imran Ashraf, Andreas Papageorgiou, Nauman Ahmed, Xiang Fu and Nader Khammassi for all the spontaneous discussions that paved the way to speed-up the research. The Quantum Hackers meets helped in voicing my opinion from the application developer's perspective about the ongoing development of the quantum programming toolchain.

I would like to take this opportunity to thank my fellow batchmates at TU Delft, Santosh Malagi, Prashanth GL and Charu Agrawal, for those interesting technical (and sometimes philosophical) discussions in the thesis period.

Finally, I would like to show my gratitude to my girlfriend and my parents who have been standing unconditionally by my side for the last years. I have always been able to count on them during the good and the bad times. Thank you all very much!

Aritra Sarkar
Delft, The Netherlands
June 2018

List of Acronyms

ALU	arithmetic logic unit
ANN	artificial neural networks
ASIC	application specific integrated circuits
BAM	bidirectional associative memory
BQP	bounded-error quantum polynomial
BWA-MEM	Burrows-Wheeler aligner using maximal exact matching
BWT	Burrows-Wheeler transform
CAS	content addressable storage
CPU	central processing unit
DNA	dioxyribonucleic acid
FPGA	field programming gate array
GATK	genome analysis toolkit
GPU	graphics processing units
LD	Levenshtein distance
NGS	next generation sequencing
NWA	Needleman-Wunsch algorithm
POVM	positive operator valued measure
QEC	quantum error correction
QFT	quantum Fourier transform
QML	quantum machine learning
QPU	quantum processing unit
QRAM	quantum random access machine
QTM	quantum Turing machines
QuAM	quantum associative memory
RAM	random access memory
RG	reference genome
RNA	ribonucleic acid
SIMD	single instruction multiple data
SMS	single molecule sequencing
SR	short read
SVD	singular-value decomposition
SWA	Smith-Waterman algorithm
UTM	universal Turing machine
WGS	whole genome sequencing

Contents

1 Introduction **1**
1.1 Thesis scope 1
1.2 Research approach 2
1.3 Organisation 2

2 Quantum computing and Genomics **3**
2.1 Computer engineering 3
2.1.1 A historical perspective. 4
2.1.2 The transistor era. 4
2.1.3 Accelerator model 5
2.2 Computer science. 6
2.2.1 Universal computation models 6
2.2.2 Computational complexity 7
2.2.3 Algorithmic complexity 9
2.3 Quantum computation 9
2.3.1 Quantum states 9
2.3.2 Quantum operators. 10
2.3.3 Quantum hardware models 11
2.3.4 Quantum systems view. 14
2.3.5 Quantum complexity 15
2.4 Bioinformatics 16
2.4.1 DNA structure 16
2.4.2 Sequencing technologies 18
2.4.3 Reconstruction technologies 18
2.4.4 Big data perspective 19

3 Algorithms for DNA sequence reconstruction **21**
3.1 Problem definition 21
3.1.1 Sub-sequence alignment 21
3.1.2 Quantum pipeline 22
3.2 Bioinformatics algorithms for sequence reconstruction 22
3.2.1 Naive linear search 23
3.2.2 Improved exact string matching 23
3.2.3 Approximate matching using dynamic programming 24
3.2.4 Heuristic algorithms 25
3.2.5 De novo assembly 26
3.3 Quantum search algorithms 27
3.3.1 Grover's search 28
3.3.2 Generalising quantum search algorithms 30
3.4 Quantum pattern matching 33
3.4.1 Approach 1: Conditional Oracle call 33
3.4.2 Approach 2: Quantum phone directory and Hamming distance 36
3.4.3 Approach 3: Quantum associative memory 37

4 Implementations of quantum pattern matching algorithms **41**
4.1 Development tools 41
4.1.1 OpenQL compiler framework 41
4.1.2 QX quantum computer simulator 43
4.1.3 Integrated development environment 44
4.1.4 Verification and testing 44

4.2 Assumptions . . . 45
4.2.1 Number of qubits . . . 45
4.2.2 Noise models . . . 45
4.2.3 Number and type of gates . . . 45
4.2.4 Qubit topology . . . 46
4.3 Algorithmic utilities . . . 46
4.3.1 Reference patterns . . . 46
4.3.2 Unitary decomposition . . . 47
4.3.3 Multi-controlled NOT . . . 48
4.3.4 Oracle for Boolean functions . . . 50
4.4 Quantum pattern matching . . . 51
4.4.1 Approach 1: Conditional Oracle call . . . 51
4.4.2 Approach 2: Quantum phone directory and Hamming distance . . . 52
4.4.3 Approach 3: Quantum associative memory . . . 54
4.4.4 Quantum associative memory with distributed queries . . . 54
4.4.5 Quantum associative memory with improved distributed queries . . . 55
4.4.6 Quantum associative memory with closed match . . . 56
4.4.7 Quantum non-linear search algorithms . . . 56
4.5 Proposed algorithm for sub-sequence index search . . . 57
4.5.1 Complexity in quantum processors . . . 59
4.6 A summary of considered algorithms . . . 61

5 Conclusions and future work **63**
5.1 Conclusion . . . 63
5.2 Future work . . . 64
5.2.1 Other quantum computing models . . . 64
5.2.2 Other quantum algorithms . . . 65
5.2.3 Other bioinformatics applications . . . 65
5.2.4 Other application domains . . . 65
5.2.5 System design . . . 65

Bibliography **101**

1 Introduction

All fiction that does not violate the laws of physics is fact.
- David Deutsch

While striving to unify the energy forms in quantum gravity, theoretical physicist John Archibald Wheeler, put forth a more fundamental idea of Digital Physics – the Universe expressed by the coherence of toggling bits, a gigantic computer. Is the evolving Universe then just another holographic simulation unfolding itself? – A question, only a machine as powerful as the Universe can answer. At the base of any pyramidal discovery by machine learning lies an enormous pool of *data*, which are processed to extract *information*. Patterns are classified to build a *knowledge* base. Logical decisions on these carry us further up the ladder towards wisdom, or *intelligence*. But, can we transcend to the epitome of programmed *enlightenment* – artificial intelligence in quantum computing. What was once confined to the pages of science fiction, is now being discussed increasingly in research laboratories around the World. This thesis rides the crest of this wave by investigating algorithms to search patterns using fundamental physical theories for computing.

An intriguing notion that has charmed many great minds is the principle of *emergence* - structure from chaos, of complexity from simplicity, of negative entropy and creativity. One such marvellous creation of the Universe is of life - orchestrated with chemical reactions among inanimate molecules. The most interesting of these molecules, pervasive to all living organisms big or small, aquatic or aerial is DNA (and RNA). This *code of life* encodes the spectacle of biochemical processes: metabolism and reproduction. It is a program, which depending on environmental inputs act out these processes - eventually evolving the next generation with a compressed memory of the most eventful of these stimuli by natural selection. Understanding this code better will enable us to harness many wonders from personalised drug discovery to genetically modified crops. These sequences of DNA form the haystack of data for the pattern matching efforts in this thesis.

1.1. Thesis scope

The analysis of DNA sequences poses a big data challenge due to the sheer volume of data generated. Even in state-of-the-art super-computing clusters, executing the whole genomics pipeline can take time in the order of days for a human DNA. This low throughput is a major constraint in the development of precision medicine applications. Sequence reconstruction is one of the most computation-intensive blocks of the analysis pipeline. This thesis explores strategies to accelerate the algorithmic primitive of sub-sequence alignment by harnessing the inherent parallelism in quantum algorithms. The domain of quantum search-based pattern matching is researched and algorithms are developed specifically to suit DNA sequence data.

The primary aim of this thesis is two-fold. On the theoretical side, the thesis explores the possibilities in the merger of the field of bioinformatics algorithms and quantum computation. It discusses the various algorithms in both these fields and highlights the synergistic bridges between them. On the more practical side, the assumptions for implementation of these approaches are considered. An executable scaled down simulation of the quantum algorithm is constructed to test the viability of pattern recognition on genomic data for the purpose of sequence reconstruction.

The thesis is highly interdisciplinary, merging some of the core ideas of computing, biology and engineering. The major highlights of this thesis are:

- Survey of various pattern matching algorithm in genomics
- Implementations of different existing strategies for quantum pattern search in a simulator
- Development of a *new* quantum algorithm for DNA sequence alignment
- Discussions on the software architecture of a full-fledged quantum accelerated genome sequence analysis pipeline

1.2. Research approach

During the research period of the thesis (around 8 months), an iterative amalgamation of breadth and depth exploration is followed. New promising avenues of quantum algorithms are explored to understand its applicability, while in parallel, implementations of more conductive approaches are programmed and tested on a quantum simulator. Some branches of quantum search techniques are trimmed for the scope of the thesis. These decisions are based on the availability of prior research on ways to model the problem of genomic sub-sequence alignment, using these approaches.

Currently available quantum computing hardware (or simulators) precludes testing problem instances of realistic sizes. However, a primary goal of the thesis is to provide a malleable foundation for implementing a full-fledged application pipeline in the future. Thus, the discussions on theoretical validity of the algorithms are extended by supportive proof-of-concept simulations. The challenge of the thesis is indeed discussing practical implementation of a proposition that is still transitioning from science fiction to serious theoretical discussion.

1.3. Organisation

This thesis is structured around chapters.

In Chapter 2, a background in the interdisciplinary fields concerning this thesis is presented. It gives a broad overview of the developments towards the concerned problem statement from the perspective of computer engineering, computer science, quantum computing and bioinformatics.

Chapter 3 discusses the existing classical and quantum algorithms for pattern matching in general. The basic structure of these algorithms is presented and compared.

In Chapter 4, the implementation of these algorithms is presented in the OpenQL and QX simulator platform. These algorithms are modified for genome sequence data. A new strategy is developed that is better suited for the specific application.

Chapter 5 concludes the thesis and gives recommendations for future work.

2

Quantum computing and Genomics

Science is beautiful when it makes simple explanations of phenomena or connections between different observations. Examples include the double helix in biology, and the fundamental equations of physics. - Stephen Hawking

As mentioned in the previous chapter, the focus of this thesis is to study and develop pattern matching algorithms for genomic sequences that can be accelerated when executed on a quantum computer. The thesis is developed at the intersection of multiple fields as shown in Figure 2.1. This chapter provides the necessary background in each of these fields to appreciate the challenges from each perspective.

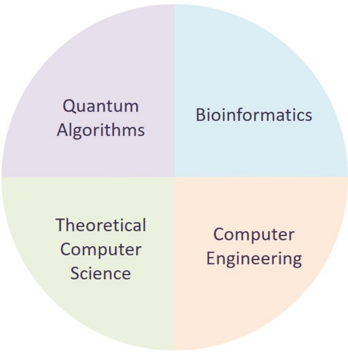


Figure 2.1: Related disciplines of the thesis

Firstly, the outlook of computer engineering are presented, that motivates the research in quantum computers. Then, some fundamental theories and programming models of computer science are introduced, that provide hard bounds on the results. Next, a background of quantum computation is elucidated from both the theoretical and practical stance. Finally, some elementary knowledge of genetics is premised so that the following chapters can be well knitted with these multi-disciplinary viewpoints.

2.1. Computer engineering

Marvelling on the pervasive and magnificent civilisation that human beings have established, much of its incredible engineering genius has been made possible by precise computations. But even before the advent of the space age, before the computer age, or even the industrial age, humanity had firmly seated themselves at the epitome of evolutionary success. Use of tools to aid in various activities is arguably a key aspect of intelligent behaviour, which very few animals exhibit. While tools in general, can mean anything from fire to wheels, we shall in this section, have a brief replay of the evolution of computing methods over the ages - ultimately converging on the current challenges that motivate us for further research in this domain.

2.1.1. A historical perspective

The widely used decimal system of counting is mostly attributed to our ten fingers. While this method of computation is limited to junior schools and might not seem a cognitive spectacle, we shall see in Section 2.4.1, even monkeys sharing $\sim 99\%$ of our genetic codes, cannot do any of those simple multiplication tables. Counting on fingers require some amount of memorisation. An easier method for early humans have been marks on cave walls or counting with stones. The abacus and such a device are not much different and in Section 2.2.1 the fundamental model of computation, the Turing Machine, is shown to rely on a very similar construct. Counting with stones, or abacus has a distinct similarity with quantum computation [1], as we cannot introduce more stones or make them vanish at one's will. The total number of stones is conserved, and we can only put them aside or arrange them into piles that would reflect our calculation result. This leads directly to the intuitive notion of no-cloning and reversible computation where only the erasure of information costs energy [2].

As civilisation progressed, analog and mechanical computing devices like the Antikythera mechanism or the astrolabe were invented as special purpose tools to aid calculation. A more general purpose calculator was invented in the 17th century as the slide rule. Eventually, more autonomy was added as the machinery became more complex, like Pascal's calculator or Jacquard's loom. The loom wove pattern controlled by a paper tape constructed from punched cards. The cards could be changed without altering the mechanical design of the loom - a landmark achievement in programmability.

Charles Babbage conceptualised and invented the first mechanical computer, the difference engine, in the early 19th century, to aid in navigational calculations. Thereafter, he realised a much more general design called the analytical engine. The input programs and data were provided via punched cards, while output devices included a printer, a curve plotter and a bell. It incorporated a decimal arithmetic logic unit (ALU), control flow in the form of conditional branching and loops, and integrated memory; making it the first design for a general-purpose computer that could be described in modern terms as Turing-complete.

The era of modern computing began with a flurry of developments concerning World War II. These devices were electro-mechanical (electric switches driving mechanical relays) having low operating speeds. They were eventually superseded by much faster all-electric computers, originally using vacuum tubes. In 1941, Konrad Zuse built Z3, the World's first working electro-mechanical programmable, fully automatic digital computer. Replacement of the hard-to-implement decimal system by the simpler binary system added to the reliability.

Purely electronic circuit elements gradually replaced their electro-mechanical equivalents in the early computers like Colossus, ENIAC and EDVAC. As further innovation progressed, some of the architectural concepts seen in today's computers became established. These included the pervasive Boolean algebra for the ALU, instruction-sets for the microcode and the concept of firmware (for device drivers) and the operating system.

Another significant milestone is the concept of stored program architecture introduced in 1945 by John von Neumann. The Princeton architecture evolved to be associated with a common bus to the random-access memory (RAM) preventing instruction fetch and a data operation at the same time. The alternate more complex Harvard architecture has a dedicated set of address and data bus for each.

2.1.2. The transistor era

The next generation of devices featured silicon-based junction (later, field effect) transistors in the 1950s. They were smaller and more reliable while requiring less power than vacuum tubes, so dissipating less heat. What followed is an unprecedented boom in the electronics industry, bringing with it new applications and dependence on computation heavy insights.

Gordon Moore observed [3] in 1965 that the number of components per integrated circuit doubles in approximately two years. Moore's law has proved accurate for more than four decades and has been instrumental in guiding the research plans and targets for the semiconductor industry. However, it is an observation or projection based on human ingenuity, not a physical or natural law. The trend has sustained through a number of enabling factors like complementary metal-oxide-semiconductor (CMOS), dynamic random-access memory (DRAM), chemically-amplified photoresist, deep UV excimer laser photolithography, etc.

It is observed in Figure 2.2 that, the single thread performance, operating frequency and power stagnated around 2005 shifting the trend to multi-core processors. The power consumption of a

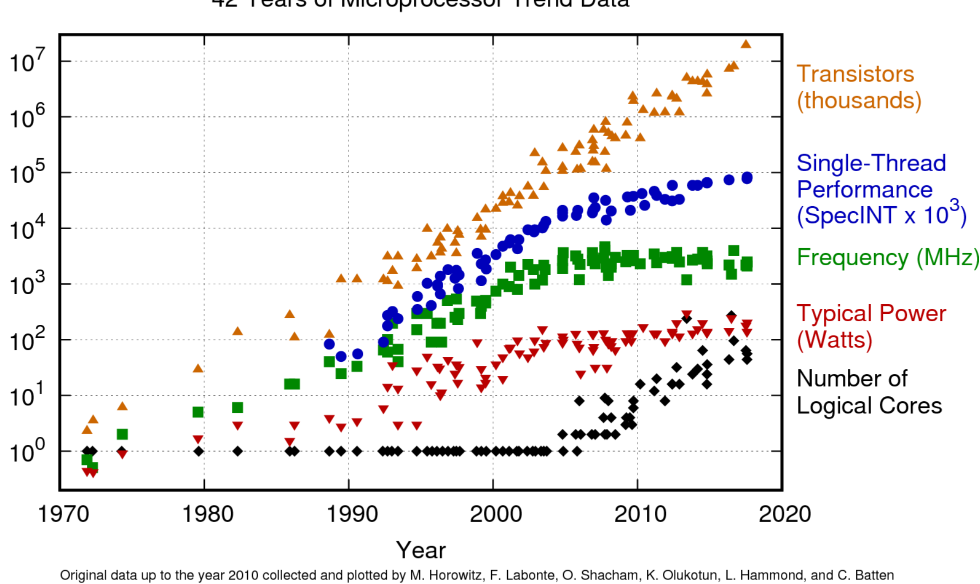


Figure 2.2: 42 Years of Microprocessor Trend Data [4]

transistor is attributed to either the leakage switching current, or the static power, given by the relation:

$$P = P_{dynamic} + P_{static} = aCV^2 f + VI_{leak}$$

where, $f_{max} \propto (V - V_t^2)/V$, and $I_{leak} \propto exp(-qV_t/kT)$. Thus, an exponential increase in frequency (the main drive for performance increase before 2005) also gave an exponential increase in power, limiting the number of transistors that can be kept powered on, per unit chip area. At the architectural level, pipelining by instruction level parallelism and speculative execution reached a saturation, suggesting the adoption of explicit parallelism. However, the speedup a program can harness in a multi-core environment is limited by the parallel fraction of a program, given by Amdahl's Law:

$$\text{n-core speedup} = \frac{1}{(1 - f_{parallel}) + \frac{f_{parallel}}{n}}$$

Not all programs have a structure that can be beneficially partitioned, taking into account the communication and synchronising overheads among the parts. Thus, the transition to specialised architectures for specific algorithmic structures is evident. Finally, the advances in memory and processor-memory interconnect technology did not reach the same level of performance as the processors, increasing the access stall between each level of cached memory hardware.

2.1.3. Accelerator model

The shift to multi-core alleviated the computing industry, but only for a few years. The drawbacks of effectively utilising multiple cores, as discussed, keep diminishing the performance returns for the engineering investment. The next generation of processors included various trade-offs, like dim-silicon (slowing down cores) and dark-silicon (keeping a majority of the transistors off, or utilising them for routing) [5].

It is evident that further performance gains would need to bind the application and hardware closer into application specific integrated circuits (ASIC). A move towards this, preserving the general purpose programmable design, is the growing dependence on specialised on-chip hardware in the form of graphics processing units (GPU), field programmable gate arrays (FPGA) and digital signal processors (DSP). The CPU would off-load tasks to connected accelerators based on the application and structure of the program. This is an active research area with more specialised accelerators like neuromorphic chips (mixed-signal architecture reflecting neural structure), ASIC miners (for cryptocurrency mining using blockchain) and tensor processing units (for neural network based machine learning), among others.

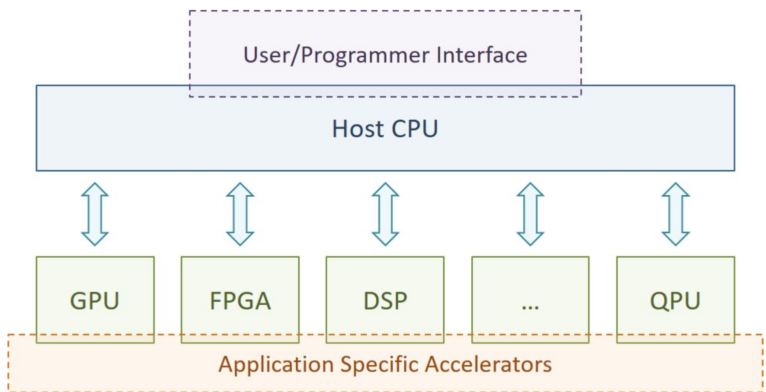


Figure 2.3: Accelerator as Co-Processor: heterogeneous computing model

Alternate forms of computations are being explored as well. Memristor refers to two-terminal non-volatile memory devices based on resistance switching effects [6]. They are useful for nano-electronic memories, computer logic and neuromorphic architectures. Another example is of DNA computer which uses DNA biochemistry instead of silicon-based hardware [7]. It is characterised by slow processing speed but having potentially high parallelism. Finally, quantum computers, the subject of this thesis, are also a promising candidate which uses fundamental physical principles of quantum mechanics for computation. Quantum processing units (QPU) are also being developed in the accelerator model of heterogeneous computation as shown in Figure 2.3.

2.2. Computer science

In this section, the premise of theoretical computer science that motivates the algorithms in the thesis is presented. The spectrum in between mathematical models of computation and practicality of algorithms for real-world applications is often overlooked. An asymptotic speedup might render useless for the concerned problem size, whereas a less lucrative speedup for the right problem size might give an enormous boost in computing resources needed for the calculation. The thesis is built on the fundamentals of algorithm design as discussed here, adding to it the implementation issues to conserve practicality.

2.2.1. Universal computation models

Computation models allow us to describe how a set of outputs are computed given a set of inputs. It relates to the organisation of units of computation, memories, and communications. These models enable us to study the performance of algorithms independently of the variations that are specific to particular implementations and specific technology.

Models of computation can be broadly classified as:

- Sequential models (e.g. Finite state machines, Pushdown automata, Turing Machine, Parallel random-access machine)
- Functional models (e.g. Lambda calculus, Recursive functions, Combinatory logic, Cellular automaton, Abstract rewriting systems)
- Concurrent models (e.g. Kahn process networks, Petri nets, Synchronous Dataflow, Interaction nets, Actor model, Process calculus)

Some of the concepts in the thesis are related to these models, for example, lambda calculus [8] and functional programming languages like Python. The Turing Machine model, however, is very closely related to the further discussions in the thesis. It is presented here.

The abstract idea of Turing Machine was proposed by Alan Turing in 1936 to solve the nonexistence of Entscheidungsproblem (an algorithm that evaluates the universal validity of a first-order logic statement). This mechanical automaton model of the CPU enumerates a subset of strings from a recursively-enumerable set (Turing recognisable language). The logic of any Turing machine can be encoded efficiently (within polynomial factors) as input interpreted by another Turing machine. The idea of Universal Turing Machine (UTM) and thus of stored-program is derived from this formal notion.

A blueprint of a UTM is shown in Figure 2.4. It consists of an infinitely long tape with cells that

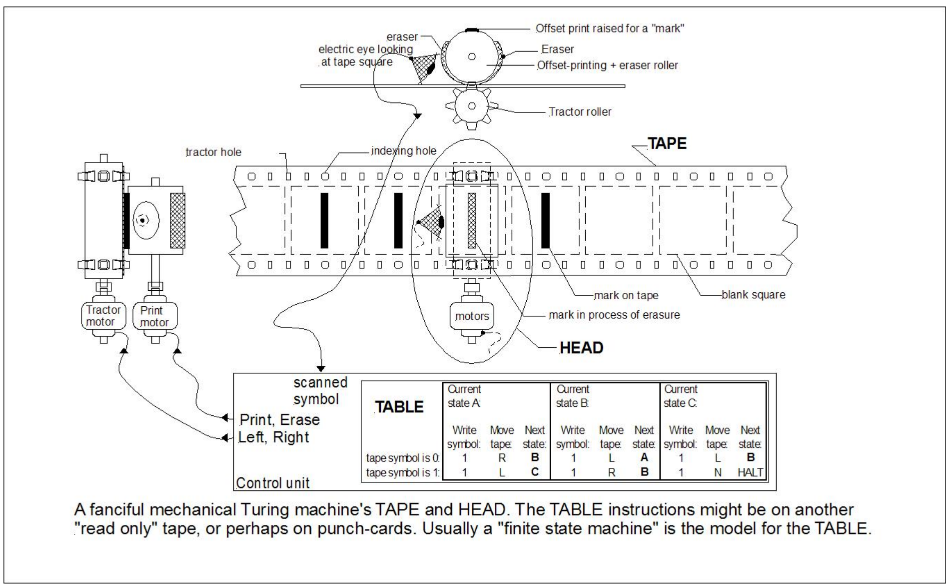


Figure 2.4: Turing Machine mechanics

can be in two states (0/1, on/off, a coin placed up/down). A head has two components, a reader that senses the state of the cell under the head; and based on the internal decision a printer conditionally toggles the state. A set of motors move the tape left or right. Now, based on the state of the machine and the input, the decision is taken by the UTM for next machine state, the printer and motor control. This decision is based on an algorithm encoded as a program (or a simple state machine). In Section 2.4.1, similarities between a Turing Machine and the biological DNA code execution are discussed.

Turing completeness refers to the ability for a system of instructions to simulate a Turing machine. Many equivalent models listed above can be thus shown to be universal computation models, for example, Rule 110 Wolfram automaton exhibiting class 4 behaviour [9]. A Turing complete programming language is theoretically capable of expressing all tasks that can be accomplished by computers. While nearly all programming languages are Turing complete if the limitations of finite memory are ignored, there are esoteric languages called Turing tarpits (e.g. GolfScript) that produces semantically obfuscated but minimal length programs for an algorithm.

2.2.2. Computational complexity

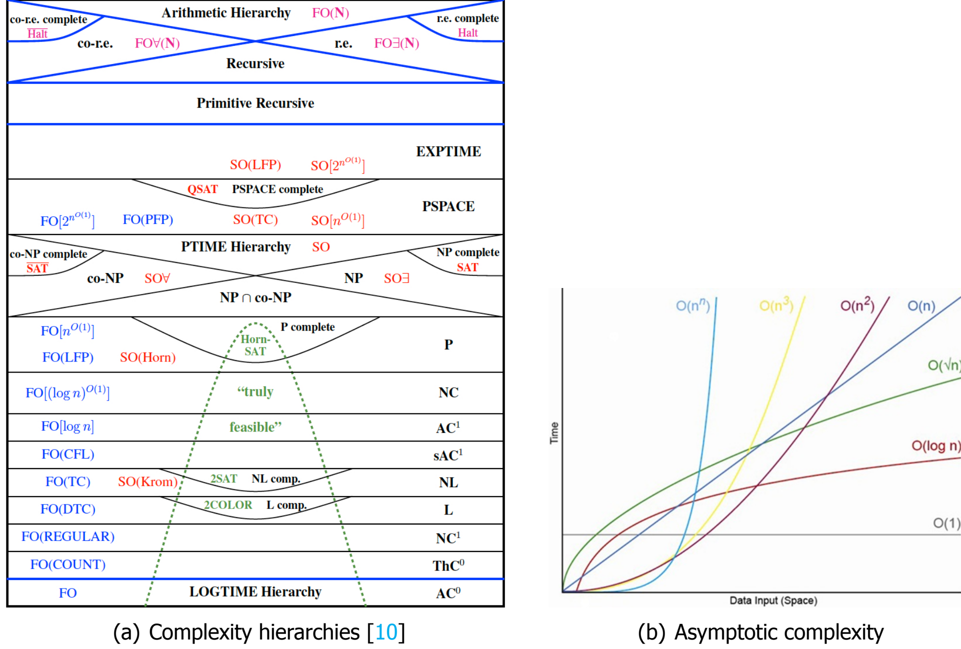
The fundamental question that theoretical computer science strives to answer is: "What are the capabilities and limitations of computing machines?" This question is answered by 3 intertwined approaches:

- Computability theory: What problems are solvable for computers?
- Complexity theory: What makes some problems easy or hard for computers?
- Automata theory: What formal model of computation is required for a problem?

Computability theory works at a larger scale, often answering fundamental questions in mathematics and computation, like Gödel's Incompleteness Theorem, the Entscheidungsproblem or the Halting problem. While these results pose critical limits on the feasibility of any algorithms that are developed, the work in this thesis, or quantum algorithms, in general, are at a much constrained scale.

Complexity theory bridges the gap between practical algorithms running on computing hardware made out of simpler grammar of programming languages, and the hierarchy of recursive languages in computability theory. The complexity of algorithms has been classified into a multitude of classes. The boundaries and relationships between these classes are sometimes not proven but are based on current knowledge and popular belief in the scientific community as shown in Figure 2.5(a).

The complexity classes of P and NP, and their relation to quantum complexity classes are of immediate interest for efficient algorithm development. Polynomial time (PTIME or P for short) refers to the class of algorithms that are efficiently solvable (or tractable) by a deterministic Turing Machine (DTM) in an amount of time that is polynomial in the size of the input problem. Non-deterministic polynomial



(a) Complexity hierarchies [10] (b) Asymptotic complexity

Figure 2.5: Theoretical computer science aspects for algorithm design

time (NTIME or NP for short) refers to the set of problems that are tractable by a non-deterministic Turing Machine (NTM) in polynomial time. In contrast to Turing Machines (examples of DTM), in a NTM, the set of rules may prescribe more than one action to be performed for any given situation. The resolution of action is based on thought experiments. A way to look at it is to say that the machine is the ”luckiest possible guesser”; it always picks a transition that eventually leads to an accepting state, if there is such a transition. Alternatively, it can be imagined as the machine ”branches” into many copies, each of which follows one of the possible transitions, thus, instead of a DTM's single ”computation path”, NTM follows a ”computation tree”. If at least one branch of the tree halts with an ”accept” condition, the NTM is said to accept the input.

An alternate definition of NP class is, the set of all decision problems for which the instances where the answer is ”yes” have efficiently verifiable proofs, as $NP \in MA = IP \in AM$ (Merlin-Arthur Interactive Proof systems). Whether an algorithm that can be efficiently checked for correctness can also be efficiently solved is an open question [11] (one of the Millennium problems). However, P can easily be reasoned to be a subset of NP (might not be a proper subset). Given a certificate for a problem in P, we can ignore the certificate and just solve the problem in polynomial time, alternatively, a DTM is also trivially a NTM that just happens to not use any non-determinism. Another important concept here is of NP-Completeness, which is the set of problems in NP such that every other problem in NP can be transformed (or reduced) to it in polynomial time.

The Bachmann–Landau notation (or asymptotic notations) are used to describe the limiting behaviour of a function with the domain tending towards a particular value (often infinity). The big O notation is used to classify algorithms according to their running time or space requirements growth rate with input size. A description of a function in terms of big O notation provides an upper bound on the growth rate of the function. Formally, for real/complex valued functions f and g defined on some unbounded subset of the real positive numbers,

$$f(x) = O(g(x)) \text{ as } x \to \infty$$

iff ∀ sufficiently large x $\exists\ M \in \Re$ and $x_0 \in \Re$ s.t. $|f(x)| \leq Mg(x)\ \forall\ x > x_0$ The O notation for a function f is derived by the simplification rules:

- If $f(x)$ is a sum of several terms, term with largest growth rate is kept
- If $f(x)$ is a product of several factors, constants are omitted

For example, $f(x) = 4x^4 + 2x^3 + 100$, the order is $O(x^4)$. Infinite asymptotics often looks over the lower order terms and constants, which might be the deciding factor for practical applications. As

shown in Figure 2.5(b), even for exponential problems in $O(n^n)$ versus constant time $O(1)$, there is a cross-over of applicability, where the preference shifts. It is important to estimate where the problem of interest lies on the horizontal axis for stricter comparison among algorithms. Other kinds of bounds on asymptotic growth rates use the symbols o, Ω, ω, and Θ. The Θ notation is used to denote average-case complexity. While it is much more difficult to prove bound without guarantees on input data, it is of more practical significance and will be used at times in the thesis.

2.2.3. Algorithmic complexity

Alternate to the views presented above, complexity can also be assessed using algorithmic information theory. The Kolmogorov complexity of a solution (output) is the length of the shortest computer program that produces it. It represents a measure of the computational resources needed to specify the operation to create the object.

This requires a fixed universal description language (the sensitivity of complexity is relative to the choice of the language). It can be thought of in terms of Turing tarpits discussed in Section 2.2.1. It is upper bounded by the length of the output itself (plus a constant factor). However, the exact value is uncomputable and is intimately connected with the uncomputability of the Halting problem [12].

The notion of Kolmogorov entropy is very useful for compression. Kolmogorov randomness is achieved if there exists no shorter computer program that can produce the output than itself (i.e. it is incompressible). Also, the Kolmogorov complexity of the output of a Markov information source, normalised by the length of the output, converges almost surely (as the length of the output goes to infinity) to the Shannon information entropy of the source [13].

2.3. Quantum computation

The discussion in Section 2.1 and Section 2.2 have given the basic practical and theoretical premise of the thesis. In this and the forthcoming section, the fields more directly related to the thesis is explained.

Nature being inherently quantized, classical computation adheres to the laws of quantum mechanics as well. The quantum-computing-specific phenomena act however in the atomic/sub-atomic realm (a size Moore's transistor scaling law is fast approaching) and harnessing these form the basis of quantum computation. Quantum computer has come up on the Gartner hype cycle [14] for emerging technologies for a few years now. In this section, the role of quantum computation, construction architectures, its basic principles, the inherent advantages as well as known limitations is presented.

2.3.1. Quantum states

Much of quantum computing is based on the abstract linear algebra of vector spaces, however, for an intuitive understanding, a simplistic model of the hydrogen atom is considered as depicted in Figure 2.6. In quantum computers, information is stored in quantum bits or qubits. It is a two-level system, which can be generalised to d-level systems, called qudits, though not needed for this thesis. Quantum numbers describe values of conserved quantities in the dynamics of a quantum system and form the levels for quantum computation. They are discrete sets of integers or half-integers (this distinguishes it from classical mechanics where system variables are continuous, such as mass, charge, or momentum). For electrons, quantum numbers are the sets of numerical values giving acceptable solutions to the Schrödinger wave equation for the hydrogen atom, for example, Principal, Azimuthal, Magnetic are Spin quantum number. A single electron around the nucleus can be thought of as a qubit if its energy states are restricted to only the ground state and the first excited state, or other properties, for example, the spin states of up or down.

The principle of superposition states that the quantum state can be an arbitrary superposition of its basis states. Basic states correspond to measurable parameters (called Hamiltonians), like the energy level, or the spin; and are denoted by the Dirac notation. In the hydrogen example, the energy levels corresponds to the basis $|0\rangle, |1\rangle$, while the spin states are $|+\rangle, |-\rangle$. Mathematically, superposition of qubits can thus be a linear combination of $\alpha|0\rangle + \beta|1\rangle$, or, $\gamma|+\rangle + \delta|-\rangle$. These two sets of basis are orthonormal, thus, a state on one of the basis, is completely unrecognizable in the other basis, as the interconversion of the basis is given by, $|+\rangle = \frac{|0\rangle + |1\rangle}{2}$ and $|-\rangle = \frac{|0\rangle - |1\rangle}{2}$, and can be posed in terms of complementary properties in Heisenberg's uncertainty principle. Here, the complex constants (called

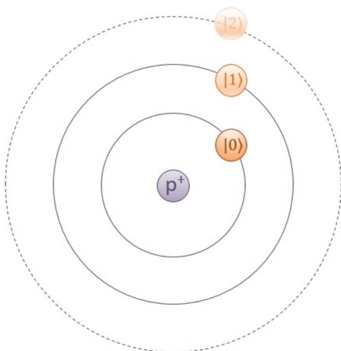


Figure 2.6: Hydrogen atom, a simple quantum system

probability amplitudes) should be normalised as their squared value corresponds to the probability of actually observing the state in the particular basis state, and thus should be preserved as long as the system is closed. Thus, $|\alpha|^2 + |\beta|^2 = 1$ and $|\gamma|^2 + |\delta|^2 = 1$. This is essentially a generalisation of the theory of probability to complex numbers. The state of a n-qubit system is an arbitrary superposition over 2^n basis states with normalised complex amplitudes as coefficients, with an irrelevant *global phase*. Mathematically, it is a vector in n-dimensional Hilbert space (the complex generalisation of Euclidean space).

Another peculiar quantum mechanical phenomenon is entanglement between states, causing two qubits to be maximally correlated (or anti-correlated) to each other. Such states are called Bell states, for example, the singlet, $|\Psi^-\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |1\rangle_B - |1\rangle_A \otimes |0\rangle_B)$. The qubits A, B are called EPR pairs, and a measurement result on any qubit completely determines the result prior to measuring the other qubits. This is due to projective measurement and is discussed subsequently. It is important to note, entanglement can be used for securing quantum communications, as it is monogamous, however, it cannot be used for faster-than-light signalling.

Finally, another fundamental postulate of quantum mechanics, the no-cloning principle, restricts copying of arbitrary quantum states. While this adds to the security of quantum communication, for quantum computation algorithms, it is often a necessary evil, preventing the initialisation circuit's output to create multiple copies by fanout. In Section 4.5.1, the thesis will present ideas to work around this limitation.

2.3.2. Quantum operators

The model of quantum computer is of in-memory computing, where the data (qubits) are stationary, and the operations (gates) are applied on it. Quantum gates are arbitrary unitary operators (and thus reversible) that can be simultaneously applied to the entire superposition similar to classical SIMD. This gives rise to quantum parallelism. Tables 2.1 and 2.2 presents some of the useful gates for this thesis. Except for the parameterised rotation gates, the rest are called discrete gates, as the unitary is fully defined. Just like Boolean logic, quantum gates can also be manipulated using some useful relations as given in the Table. These relationships will be essential in the design of the final algorithm presented in Chapter 4.

Like $\{NAND\}$, $\{NOR\}$, or $\{AND, OR, NOT\}$ gate sets in classical logic, there exists a universal set of gates on quantum logic, that allows any arbitrary n-qubit unitary to be decomposed with members of the set. Examples of such sets are, $\{H, S, Toffoli\}$, $\{H, T, CNOT\}$, $\{R_y, R_z, CNOT\}$ (not discrete). All of classical logic can be simulated using either the Toffoli or the Fredkin gate. Any classical logic can be constructed of NAND and FanOut circuits, proving quantum logic as a super-set of classical logic. NAND can be made of Toffoli using the inputs $A, B, 1 \mapsto A, B, nand(A, B)$, while FanOut can be constructed with $1, A, 0 \mapsto 1, A, A$. The FanOut is not a violation of no-cloning principle as it is perfectly possible to clone only the non-superposing states $|0\rangle$ and $|1\rangle$. Also it is paramount to note, in this thesis, such reversible transformation of classical algorithm's circuits are not considered as quantum algorithm, as given the current state of the art, running the quantum version of the circuit would result in no benefits in terms of both time efficiency (lower clocking frequency), and energy (though reversible, the quantum hardware requires enormous amount of energy to cool it to near absolute zero Kelvin). Another vital

aspect of the universality of gates requires at least 1 non-Clifford gate (e.g. T, Toffoli) as a part of a discrete set.

Measurement of a state is inherently probabilistic depending on the amplitude of the vector in the measurement basis. This is based on the general theory of Positive Operator Valued Measure (POVM). The quantum state collapses the wave-function to one of its classical eigenvalues. On measuring the state $\psi = \frac{1}{\sqrt{3}}|0\rangle + \frac{\sqrt{2}}{\sqrt{3}}|1\rangle$, the eigenvalue corresponding to the state $|0\rangle$ is obtained with probability $1/3$rd. The quantum state becomes $|0\rangle$ if $|0\rangle$ is observed, and thus becomes a classical bivalued state, represented by double lines in a circuit after the measurement symbol. This is called projective measurement and is a significant difference from classical theories where measurement can be performed without affecting the system under test. Thus to reconstruct the probability distribution of the state asymptotically, multiple measurements are required, called state tomographic trials. The exact complex amplitude cannot be assessed, thus allowing a degree of freedom. This arbitrary phase factor can be ignored with no effect of the solution probability.

$|\psi\rangle$

2.3.3. Quantum hardware models

Apart from using gates to perform quantum computation, there are other models that are being developed for hardware adoption. In this section, a brief discussion of these models is presented. The hardware back-end choices follow from the models.

Circuit model

The quantum circuit model described in Section 2.3.2 is an extension of the classical Boolean circuit model for reversible computation. It was developed alongside the first quantum oracular algorithms. Oracles are constructs that can provide true-false answers to elaborate questions (discussed in more details in Chapter 3). The constructs for answering might be complex lengthy circuitry, finally resulting in a probabilistic yes-no for quantum logic. The crux of the quantum algorithm design involves a clever process to distil the solution such that the measurement probability is as biased as possible towards the solution, to such effect that even single runs can produce the results with sufficient confidence. Popular Oracles are those of Deutsch, Deutsch-Jozsa, Simon, Shor and Grover.

Adiabatic computation

Adiabatic computation was conceived specifically to solve satisfiability problems of the NP-Complete class. This paradigm is based on the adiabatic theorem, i.e. the possibility of encoding a specific instance of a given decision problem in a certain Hamiltonian (by capitalising on the well-known fact that any decision problem can be derived from an optimisation problem by incorporating into it a numerical bound as an additional parameter). The system is initialised in a ground state of an easy to construct Hamiltonian and slowly (adiabatically) the system is deformed to the desired Hamiltonian where measurement of the final ground state reveals the desired optimal value. The speedup of the algorithm depends crucially on the scaling of the energy gap as the number of degrees of freedom in the system increases. It was shown to be polynomially equivalent to the circuit model, which implies, application to an intractable computational problem might not be feasible (intractability many possibilities of getting stuck in eigenvalues of local minima) and remains an open empirical question.

Measurement based quantum computation

Measurement-based quantum computation allow only non-unitary measurements as computational steps. Its variants are teleportation quantum computing and one-way quantum computing (cluster state model). The measurements are performed on a pool of highly entangled states each done in a different basis which is calculated classically given the result of the earlier measurement. It has shown promises in a different kind of more fault-tolerant quantum computer architecture.

Topological quantum computation

Topological quantum field theory (TQFT) model are based on exotic physical systems (topological states of matter). The model was proved to be efficiently simulated on a standard quantum computer and is thus equivalent, however, its merit lies in its high tolerance to errors resulting from any possible realisation of a large-scale quantum computer. This is due to many global topological properties

Table 2.1: Commonly used single qubit quantum gates

Name	Unitary	Circuit	Relations
Identity	$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	I	
Hadamard	$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	H	
Pauli-X	$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	X	$X = iZY = HZH$
Pauli-Y	$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$	Y	$Y = iXZ$
Pauli-Z	$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	Z	$Z = iYX = HXH$
Rotation-X	$R_x(\theta) = \begin{bmatrix} cos\frac{\theta}{2} & -isin\frac{\theta}{2} \\ -isin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix}$	$R_x(\theta)$	
Rotation-Y	$R_y(\theta) = \begin{bmatrix} cos\frac{\theta}{2} & -sin\frac{\theta}{2} \\ sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix}$	$R_y(\theta)$	
Rotation-Z	$R_z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}$	$R_z(\theta)$	
T	$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$	T	
$T^\dagger$	$T^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix}$	$T^\dagger$	$I = TT^\dagger$
Phase	$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$	S	$S = T^2$

Table 2.2: Commonly used multi-qubit quantum gates

Name	Unitary	Circuit	Relations
SWAP	$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$		$SWAP = CX_{01}CX_{10}CX_{01}$
Controlled-NOT	$CX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$		$CX_{01} = H_0H_1CX_{10}H_1H_0$ $X_1CX_{01} = CX_{01}X_1$ $Z_0Z_1CX_{01} = CX_{01}Z_0$
Controlled-Phase	$CZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$		$CZ = H_1CX_{01}H_1$
Toffoli	$CCX = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$		$CCX_{012} = H_2CX_{12}T_2^{\dagger}CX_{02}$ $T_2CX_{12}T_2^{\dagger}CX_{02}T_2T_1^{\dagger}H_2$ $CX_{01}T_1^{\dagger}CX_{01}S_1T_0$
Fredkin	$CSWAP = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$		

are, by definition, invariant under deformation, so since most errors are local, information encoded in topological properties imparts the robustness.

Quantum random access machine

For the numerous classical computing models presented in Section 2.2.1, the corresponding quantum analogues can be derived, like quantum automata, or quantum Turing machines (QTM). The models interesting for the development of quantum hardware and programming languages are presented here. While the QTM model is simplistic, for practical purposes transition descriptions get complex and clumsy compared to other equivalent models like the circuit and the QRAM model. The quantum circuit model does not provide a mechanism for controlling with a classical machine the operations on quantum memory. The model of quantum random access machine (QRAM) is built on the usual trend of a quantum computer based on the circuit model is controlled by a classical device encoding the algorithm. QRAM provides a convenient model for developing quantum programming languages as it is based on quantum pseudocode description of the computation.

Physical implementation

Nowadays, there are various potential candidates for the hardware back-end being pursued. *Superconducting integrated circuits* are Josephson junction based harmonic oscillators coherently controllable and measurable by magnetic flux pulses and microwaves. *Quantum dot architectures* are based on individual electrons confined in quantum dot (quantum well potential), made in silicon or Si^{28} or $GaAs$, controllable using magnetic flux pulses, and inter-dot gate voltage and measurable using tunnelling current measurements. *Ion traps* are based on alkali metal ions confined in an ion trap using electric fields and controllable/measurable by laser pulses. Other candidates like *NMR type spin qubits* (not a viable candidate anymore), *Nitrogen-vacancy centre based*, *photonic qubits* and *Majorana fermion based topological qubits* (not yet built) are actively being researched as well.

2.3.4. Quantum systems view

Moving towards a broader view of constructing a useful quantum computer, the requirements from the models must be quantifiable. This is given by the DiVincenzo's criteria [15] for quantum computation and communication (last 2):

1. A scalable physical system with well characterised qubits.
2. The ability to initialise the state of the qubits to a simple fiducial state.
3. Long relevant decoherence times.
4. A universal set of quantum gates.
5. A qubit-specific measurement capability.
6. The ability to inter-convert stationary and flying qubits.
7. The ability to faithfully transmit flying qubits between specified locations.

Useful quantum computation would require a full stack architecture. The underlying quantum processor needs to be interfaced with the quantum algorithmic descriptions. Such a full stack is on the roadmap of Delft University of Technology's quantum research. The QuTech quantum computer system stack [16] is described bottom-up here:

- Quantum chip refers to the physical hardware housing the qubits as discussed above.
- Quantum-Classical interface comprises of ADC and DAC and their controls for interacting with the physical qubits.
- Micro-architecture takes into account the precise timing controls and the instruction pipelines.
- Quantum Instruction Set Architecture defines the runtime operations of both classical control and quantum parts of the algorithm. It encapsulates the hardware dependence.
- Quantum Runtime Unit is responsible for scheduling the operations required for the compiler code. This includes quantum error correction (QEC) and qubit logical to physical mapping.
- Compiler and Programming language is the interface for the algorithm designer to precisely define the quantum operators and state in abstracted high-level constructs.
- Quantum Algorithm descriptions are in computer science or mathematical state evolution designed to perform the desired task. They need to be decomposed into programming constructs as input to the compiler.

Of course, a full stack quantum computer is an area of active research, but it is not available for immediate use at the time of writing this thesis. However, quantum simulators are available that

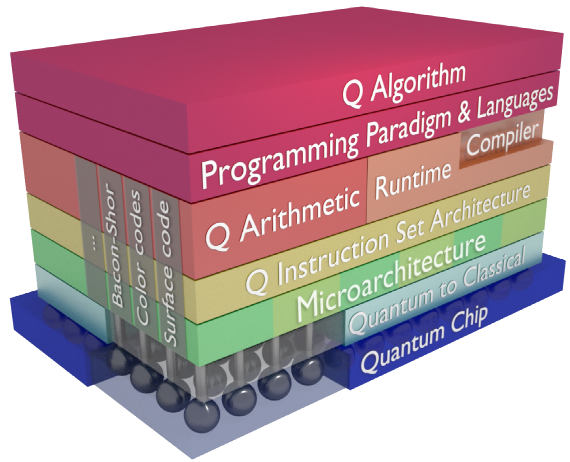


Figure 2.7: Quantum computer system stack [16]

abstract the layers of the stack below the compiler. The output of the compiler is an optimised low-level code (typically, in some form of quantum assembly language), which can be simulated. Simulation is obviously not scalable else we wouldn't need a quantum computer in the first place. However, it is possible to handle this exponential complexity of simulation for a reasonable qubit size of ~ 50, which is enough for proof of concept demonstration of the quantum parts of the algorithm considered in the thesis. Quantum computers are modelled as accelerators as described. Thus, it would also require computation support of the host CPU. A typical hybrid algorithm will have a quantum part, as well as a classical part. The classical part can be executed in the host CPU, while the quantum part is offloaded to the accelerator/simulator.

Notably some of the early quantum computing services will probably be modelled as cloud based Platform as a Service (PaaS), for example, Q Experience [17] (IBM), Alibaba Cloud [18] (Alibaba), Quantum Infinity [19] (QuTech), Forest [20] (Rigetti). This would require delegated computation from a smaller, constricted quantum device at the client end to maintain blind computation. Such models and other privacy issues are not considered in this thesis.

2.3.5. Quantum complexity

In Section 2.2.2, the complexity classes for classical computation is discussed. To compare the efficiency of quantum algorithms, it is crucial to compare them with the current best algorithm on classical computers and their complexity. It is worth noting here that, the other accelerator models considered in Section 2.1.3 are theoretically equivalent to a classical computation and do not provide any new capabilities from the standpoint of computability theory. For example, if the space required for the solution of a problem grows exponentially with the size of the problem on von-Neumann machines, in a DNA computer it still grows exponentially, thus the amount of DNA required will soon be too large to be practical. The relationships of quantum computation complexity classes with the classes hierarchy are not fully understood and are riddled with many open conjectures. While there are many quantum classes as well, two of them are particularly important for the thesis and are presented below.

Bounded-error quantum polynomial (BQP) time is the class of decision problems solvable by a quantum circuit whose length and number of qubits scale polynomially with respect to the instance size. Like the classical complexity class of BPP (bounded-error probabilistic polynomial time), the error probability is bounded by at most 1/3 for all instances; BPP being the class considered to be practical on a classical computer. Using Chernoff bound, the constant 1/3rd can be reduced arbitrarily on repetition. Adding post selection capabilities to BQP gives the class PostBQP (its classical version being BPP_{path}). As P is a subset of BQP, it is interesting to study algorithms that fall outside P, but in BQP. Such algorithms include integer factorisation (Shor's algorithm), discrete logarithm, Jones polynomial approximations for certain roots of unity, etc.

The second quantum complexity class of importance is Quantum Merlin Arthur (QMA), the quantum analogue of Merlin Arthur (MA) probabilistic complexity class. QMA is related to BQP in the same way NP is related to P, or MA is related to BPP. It consists of decision problems for which there is a

polynomial-size quantum proof (Merlin) which convinces a polynomial-time quantum verifier (Arthur) with high probability for every true answer; else it is rejected by the verifier with high probability for false answers. Examples of such problems are QCSAT (quantum circuit satisfiability), non-identity check, quantum clique, etc. Quantum Interactive Polynomial (QIP(k)) time is a generalisation of QMA where Merlin and Arthur can interact for k rounds. Though the P and PSPACE relation is not proved, the general belief for the hierarchy is:

$$P \subseteq BPP \subseteq BQP \subseteq NP \subseteq MA \subseteq QMA \subseteq PP = PostBQP \subseteq PSPACE = QIP$$

Thus, though quantum computers might not be able to make NP problems efficient [21], there are problems of interest in BQP as well. Also, it is interesting to note, for many problems in NP, quantum algorithms offer polynomial speedup, which can boost these problems into the domain of practicality (like the application explored in this thesis). Based on current physical theory, quantum computers are the most general kind of computers physically allowed [22]. More exotic computers would require some refinement of the laws of physics (like non-linearity allowing faster than light travel, violating uncertainty principle, or using closed timelike curves). Till then, it is the most powerful machine at our disposal, one that in principle, is a model of the Universe itself - a quantum extension of John Wheeler's remark - It from Qubit [23].

2.4. Bioinformatics

The field of bioinformatics uses computer programming methodologies for analysis, particularly in the field of genomics. Genomics concerns the application of DNA sequencing methods and bioinformatics algorithms to understand the structure and function of the genome of an organism. This discipline has revealed insights with scientific and clinical significance. These include studies on causes that drive cancer progression, intra-genomic processes influencing evolution, enhancing food quality and quantity from plants and animals. An exciting prospect is personalised medicine, in which accurate diagnosis testing can identify patients who can benefit from precisely targeted therapies [24].

In this section, the basic structure of DNA is presented. Thereafter, the current sequencing and analysis pipeline is explained. Finally, the challenges of modern genomics is elucidated from the data volume perspective.

2.4.1. DNA structure

Deoxyribonucleic acid (DNA) is a thread-like long polymer made of nucleotides carrying the genetic instructions used in the growth, development, functioning and reproduction of all known living organisms (and many viruses). It is the storehouse of biological information.

Within eukaryotic cells (of animals, plants, fungi and protists) DNA is organised into long structures called chromosomes as shown in Figure 2.9. During cell division these chromosomes are duplicated in the process of DNA replication, providing each cell with its own complete set of chromosomes. Eukaryotic organisms store most of their DNA inside the cell nucleus and some of their DNA in organelles, such as mitochondria or chloroplasts, in contrast to prokaryotes (like bacteria and archeas), which store their DNA directly in the cytoplasm.

In eukaryotic chromosomes, chromatin proteins such as histones compact and organize DNA. These compact structures guide the interactions between DNA and other proteins, helping to control which parts of the DNA are transcribed, called methylation. The spatial arrangement of the DNA affects gene activity and expression. The study of heritable gene function that does not directly involve the DNA sequence is called epigenetics.

The four nucleotide bases found in DNA are the purines: adenine (A) and cytosine (C); and the pyrimidines: guanine (G) and thymine (T). A fifth pyrimidine nucleobase, uracil (U), usually takes the place of thymine in Ribonucleic acid (RNA). RNA is another polymeric molecule essential for various biological roles in coding, decoding, regulation, expression of genes and also form the genetic makeup of some viruses. These four bases are attached to the sugar-phosphate to form the complete nucleotide. Adenine pairs with thymine and guanine pairs with cytosine represented by A-T and G-C base pairs (bp). This structure is formalised by Chargaff's Parity rules, which are:

1. A double-stranded DNA molecule globally has percentage base pair equality, i.e. $\%A = \%T$ and $\%G = \%C$
2. Both $\%A = \%T$ and $\%G = \%C$ are valid for each of the two DNA strands, a global feature of the

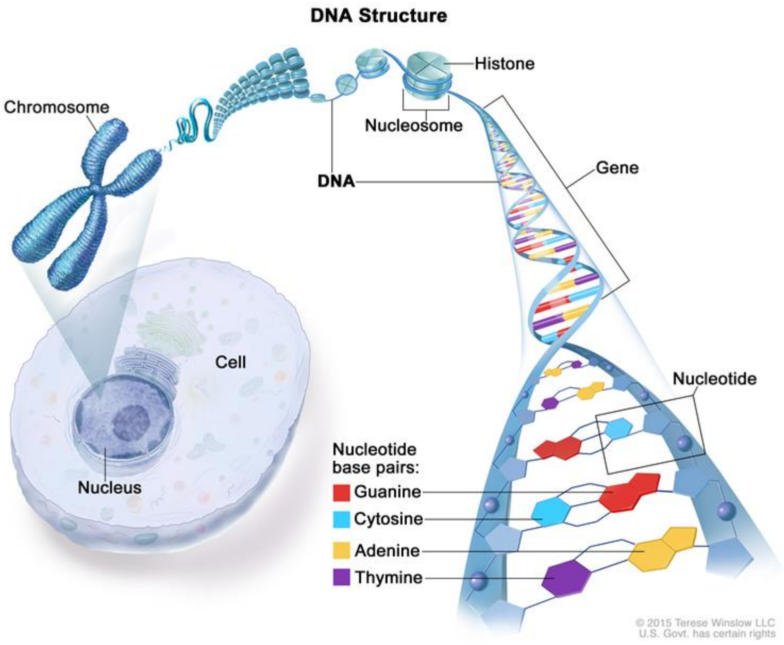


Figure 2.8: DNA structure

base composition in a single DNA strand

Besides DNA sequence, some of the basic relevant terms of bioinformatics are introduced here:

- Gene - sequence of DNA (or RNA) which codes for a molecule that has a function.
- Allele - variation in codes for a gene.
- Genotype - the allele for a gene that the organism possesses.
- Phenotype - observable characteristics or traits in an organism due to the genotype and other environmental factors.
- Genome - the total genetic material of an organism consisting of DNA (or RNA). It includes all genes (the coding regions), the non-coding DNA, as well as the genetic material from the mitochondria and chloroplasts.

The length of genomes varies greatly among organisms. The size correlates with features like cell size, cell division rate, taxon, body size, metabolic rate, developmental rate, organ complexity, geographical distribution, or extinction risk. They vary as much as 1759 bp in viruses (Porcine circovirus type 1) to 13×10^{10} in fishes (Protopterus aethiopicus), 15×10^{10} in plants (Paris japonica) and 67×10^{10} in amoebas (Polychaos dubium). The human genome is approximately 3.289×10^{8} bp long. In general, somatic cells have two identical copies of 23 chromosomes (one from each parent). Comparative genetics study genetic similarities between organisms. Interesting results from various sources suggest similarity of human genes with other organisms like other human (99.9%), chimpanzees (99%), mouse (92%), cats (90%), cows (85%), dogs (84%), zebra-fish (73%), chicken (65%), fruit-fly and banana (60%), honey-bee (44%), grapes (24%), yeast (18%), E. Coli bacteria (7%). The similarities imply some basic metabolic principles that unfold in the cellular realm resulting in same 'sub-routines' and gives the first-cut implication on the error tolerance bound for genomics. However, it is evident that genetic similarity is not a complete picture of biological similarity. The non-coding regions (sometimes, shortsightedly referred as junk DNA) are an active area of research as they form around 98% of the human genome. They consist of meta-codes for the 3D folding of the DNA in the chromosome and other expression control (enhancers, silencers, promoters, insulators, operators, etc) information for the genes.

Looking at the above definitions from an abstracted view of computation, parallels can be drawn in the biological model of information processing for metabolism. The program as the genome, the phenotype as the output, and the environmental chemistry triggering as inputs. More specifically, *evolution and metabolism can be abstracted as an evolving quine (self-replicating code) that follows an ouroboros learning model*. The processes of DNA replication (production of identical DNA helices

from a single double-stranded DNA molecule) and Gene expression (triggering of metabolic activities by chemical factors of the cell environment using protein synthesis) are responsible for choreographing most of the functionality of a cell in an organism.

2.4.2. Sequencing technologies

Sequencing technologies can be broadly classified into three generations. The first generation of sequencing machines relied on Sanger chemistry. These included Applied Biosystems Prism 3730 and the Molecular Dynamics MegaBACE. The read sizes were in the range of 500 bp to 1000 bp long.

The second generation of DNA sequencing platforms are the most popular ones that are in use presently. These machines are referred to as short-read sequencers or next-generation sequencers (NGS). They are characterised by highly parallel operation, higher yield, simpler operation and much lower cost per read [25]. Commercial platforms include Genome Sequencer from Roche 454 Life Sciences, Solexa Genome Analyzer from Illumina, SOLiD System from Applied Biosystems, Heliscope from Helicos among others. NGS reads are shorter (in the 50 bp to 500 bp range) and thus pack less information per read, increasing the complexity of sequence assembly. To satisfy minimum overlap criteria they required higher coverage, which in turn escalates data volume for processing. The characteristic error profiles of the machines change with technology, and the algorithms can be tuned for some of the platform-specific errors.

The third generation sequencers produce even longer reads, with much better quality than the first generations. They are called Single Molecule Sequencers (SMS). Examples of such platforms are Pacific Biosciences machines and Oxford nanopores which uses quantum tunnelling resistance differences to identify base pairs on a graphene sheet.

Sequencers produce observations of the target DNA molecule in the form of reads. Reads are sequences of base calls plus a numeric quality value (QV) for each base call. Useful platforms deliver paired-end reads, i.e., read pairs constrained by relative orientation and separation in the target. These reads span repeats longer than individual reads and is thus beneficial to assembly. Most platforms excluded QV analysis due to overheads on processing and memory.

The development of this thesis is considered keeping in mind input data from NGS platform (particularly Illumina reads). Particularly, the FASTQ data format for input data is considered as it is the de facto standard for storing the output of high-throughput sequencing instruments. It comprises nucleotide sequence reads (FASTA) plus its corresponding quality scores (confidence values of the measurement). The format has four lines per sequence. The first line begins with a '@' character and is followed by a sequence identifier and an optional description. The next line is the raw sequence letters. Line three begins with a '+' character and is optionally followed by the same sequence identifier and description. And the final line encodes the quality values for the sequence in line two in ASCII characters. An example (Source: Wikipedia) is shown below :

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

2.4.3. Reconstruction technologies

Once the sequencing of the biological sample provides a collection of raw reads, the data processing steps begin. These steps are summarised in the Broad Institute's GATK pipeline in Figure 2.9, which includes the major processing stages like alignment, duplicate marking, variant calling, annotation and conclusion. One of the most computation-intensive yet parallisable of this process is the Map-to-Ref step for reconstruction as shown in Figure 2.10 and will be the focus of the thesis.

Ideally, storing 3 billion bp of the human genome encoded in a Radix-4 (2 bit) numbers would be around 700 MB. However, reads are over-sampled for sequencing, resulting in data of around 200 GB (30x sequence data plus quality values). As the cost of whole-genome sequencing (WGS) continues to drop as shown in Figure 2.11(b), more and more data are churned out. This staggering computational demand motivates this thesis.

The reads obtained from sequencing needs to be reconstructed to a complete genome before analysis can be carried out. Sequence *reconstruction* can be done in two ways.

- De-novo *assembly* of reads

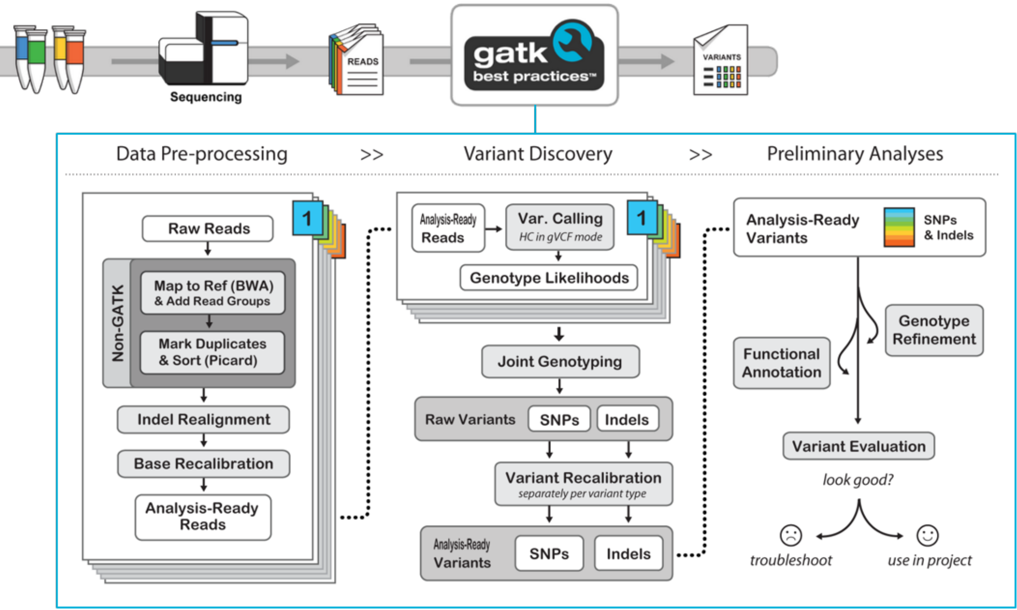


Figure 2.9: Broad Institute GATK Best Practices Pipeline [26]

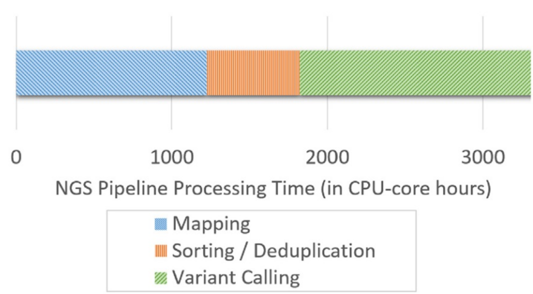


Figure 2.10: Breakdown of processing time per NGS pipeline stage [27]

- Ab-initio reference based *alignment* of reads

Normally multiple copies of the DNA is made before fragmenting it for the sequencer. Thus, a portion of the data will be preserved in multiple copies which are chopped off at different places resulting in data overlaps which facilitate stitching. This method is called De-Novo reconstruction, as no other data than the sequenced read is used for reconstruction. However, it is computationally expensive and done normally for the first time a new species is sequenced.

In Ab-Initio method, the DNA reads are matched to a trusted existing reference of the species. This is similar to a pattern matching problem, of finding the index of the read in the reference. However, this method introduces bias based on the reference since ironically, the next step after reconstruction is to discover variation from the reference for identifying implications. Even ab-initio method is computationally infeasible for exact matches, and thus heuristic methods are employed in state of the art toolchains.

Sequence reconstruction being the primary target of the thesis, detailed algorithms will be discussed in further detail in Chapter 3.

2.4.4. Big data perspective

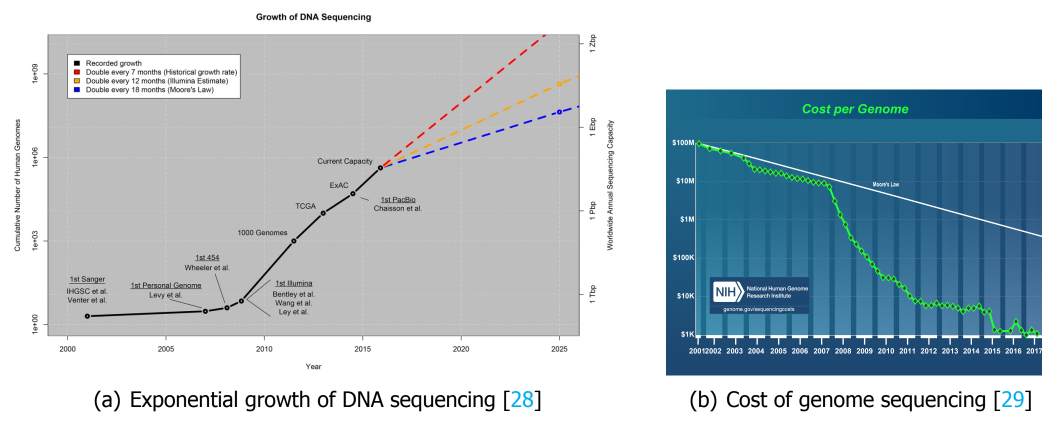
The digital age proliferated in the early 2000s, with an enormous increase in data volume. The ability of commonly used software tools to capture, curate, manage, and process data within a tolerable elapsed time was challenged with the advent of 'big data'. Big Data is characterised with the Vs concept, which commonly includes:

- Volume: the quantity of generated and stored data

- Variety: the type and nature of the data
- Velocity: the speed at which the data is generated and processed
- Veracity: the data quality of captured data can vary greatly
- Value: the potential insights that can be derived after processing

Data Mining on such sources typically involves processes for anomaly detection, association rule learning, clustering, classification, regression or summarization.

Social media (like Twitter feeds) and nuclear reactor measurements (like CERN's particle collider) are common examples of big data, where parallel processing of unstructured data requires an infrastructure of computation clusters. Big data analytics has helped health-care improve by providing personalised medicine and prescriptive analytics, clinical risk intervention and predictive analytics, waste and care variability reduction, automated external and internal reporting of patient data, standardised medical terms and patient registries and fragmented point solutions.



(a) Exponential growth of DNA sequencing [28]

(b) Cost of genome sequencing [29]

Figure 2.11: Trends in genomics data

In this thesis, our focus will be on data generated from genomics research or DNA sequencing service platforms. An analysis comparing data in genomics, astronomy, video (YouTube), and Internet data (Twitter), suggested that the term "genomical" [28] instead of "astronomical" should be used to refer to enormous data sizes. Genomics data is projected to become the largest producer of big data within the decade, eclipsing all other sources of information generation, including astronomical as well as social data as shown in Figure 2.11(a). At the same time, genomics is expected to become an integral part of our daily life, providing insight and control over many of the processes taking place within our bodies and in our environment. Therefore, effective and cost-efficient computational solutions are necessary to allow society to benefit from the potential positive impact of genomic information.

Rapid progress in genomics is based on exponential advances in the capability of sequencing technology. However, to keep up with these advances, which outpace Moore's Law, new computational challenges of efficiently analysing and storing the vast quantities of genomics data needs to be addressed. Despite the continual development of tools to process genomic data, current approaches are still yet to meet the requirements for large-scale clinical genomics. In this case, patient turnaround time, ease-of-use, resilient operation and running costs are critical.

At the moment, algorithms developed for DNA analysis use a considerable amount of approximations and heuristics to manage the computational complexity of DNA analysis, especially for human genomes. This results in introducing various biases in the results, which limit our ability to derive the full potential benefit from the results. Quantum computing promises to become a computational game changer, allowing the calculation of various algorithms much faster. We shall, in this thesis, look for potential ways to accelerate DNA sequence analysis and strive to reduce the heuristic approximations in the quantum algorithms.

3

Algorithms for DNA sequence reconstruction

Science is what we understand well enough to explain to a computer. Art is everything else we do.
- Donald Knuth

In this chapter, the central problem of the thesis is formally defined. The various existing classical and quantum algorithms that relate to the problem of genomic pattern matching is described.

3.1. Problem definition

This thesis explores the approaches towards genomic pattern matching using quantum algorithms. The problem is dissected at two levels of granularity: the quantum kernel that yields a promising speedup for the core module; and the broader design of the entire pipeline that would house the kernel as its integral part.

3.1.1. Sub-sequence alignment

The core module of an alignment based genome sequence assembly is a reference based sub-sequence aligner. This problem is the crux of biomolecular sequence analysis, dictated by the first fact, *high sequence similarity usually implies significant functional or structural similarity* [30].

A distinction between the use of *string* and *sequence* is imperative here. A string refers to an ordered list of contiguous characters, whereas, a sequence allows interspersing of characters of the sequence with other characters outside interest as long as the ordering of the embedded sequence is maintained. To illustrate this, for the reference string, $cryptography$, an example of a sub-string (as well as a sub-sequence) is $graph$, while gap is only a sub-sequence.

Optimal alignment of a sub-sequence is based on the metric of edit distance for approximately matching the sub-string spanning the comparison length. The transformation operations allowed are insertion, deletion and substitution/replacement. Multiple insertions in the pattern are termed as gaps. Each required transformation for an alignment trial can be associated with a weighted penalty as the cost function for the algorithm. It is worth noting that, edit or Levenshtein distance is upper bounded by the Hamming distance between the two string.

The recurrence relation for calculating the Levenshtein distance between strings T and P of length N and M respectively can be established. Here, the most general case is considered, where each operation between each pair of characters can have different associated cost.

$$LD_{T,P}(i,j) = min \begin{cases} LD_{T,P}(i-1,j) + f_{ins}(T_i) \\ LD_{T,P}(i,j-1) + f_{del}(P_i) \\ LD_{T,P}(i-1,j-1) + f_{match}(T_i,P_j) \end{cases}$$

where, $f_{match}(T_i,P_j) = sub(T_i,P_j)$, when $T(i)$ is substituted with $P(j)$, r if otherwise they are equal. Similarly the functions f_{ins} and f_{del} can be defined based on a transformation matrix. $LD_{T,P}(i,j)$ is the distance between the first i characters of T and the first j characters of P.

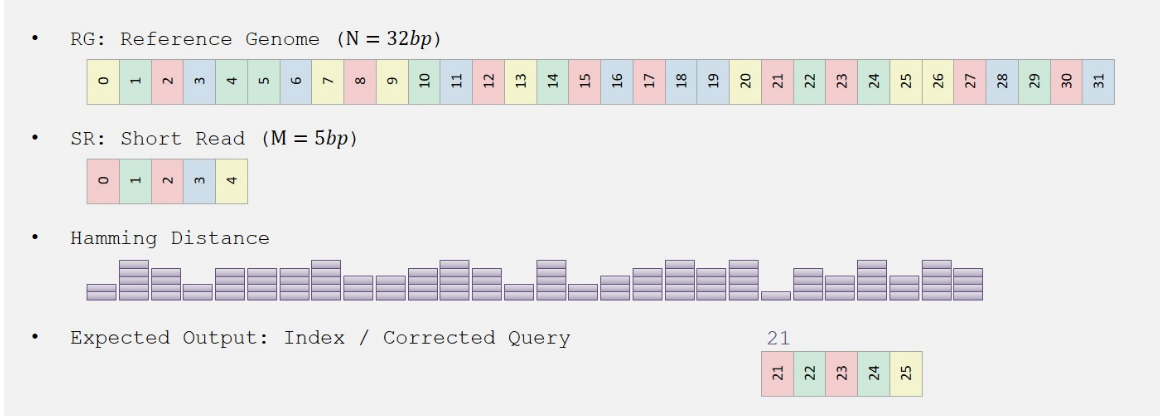


Figure 3.1: DNA sub-sequence alignment problem

The sub-sequence alignment problem is thus defined as the index $i \in N$ of T where the alignment of P starts, which gives the minimum edit distance. The resultant edit transcript forms the other output from the required algorithm if multiple sequences need to be stitched for reconstruction. This is shown in Figure 3.1. The short read is sequentially matched for each of the $N - M + 1$ starting index in the reference genome. The aligner outputs the index of minimum Hamming distance (here, 21) and optionally, the nearest match in the reference.

3.1.2. Quantum pipeline

While the sub-sequence aligner solves the core module, it needs to be embedded in a larger wrapper to achieve practicality for the genomic pattern matching. This design of the full pipeline would be very different between a classical and a quantum computer. In classical genomic analysis, the wrapper tracks the indices for multiple such pattern matches, and eventually uses the data to reconstruct the collection of patterns (of short reads) to a genome, before proceeding with the rest of the GATK pipeline as discussed in Section 2.4.3. Few additional details would be presented in the discussion on BWA-MEM, but this thesis would steer clear of the rest of the classical pipeline and concentrate on implementing the sub-sequence aligner itself. Thus, the index of the alignment is the final result for all quantum algorithms to be discussed.

However, even *obtaining the index of a single search pattern in a larger reference is not a trivial single run in the quantum algorithm*. The approach is very different from some of the best classical algorithms based on dynamic programming. The primary reason is - the answer lies in a probabilistic state and on measuring the quantum state, we obtain a single result based on the Hamiltonian. Multiple trials of the projective measurement converge on an estimate of the result. To add to this, quantum states cannot be cloned, thus, the entire process needs to be repeated to get a single approximate index or the modal state. Though this is not the primary focus of the thesis, it will be dealt with some more depth in Section 4.5.1 while comparing the various schemes in totality.

3.2. Bioinformatics algorithms for sequence reconstruction

The impact of computation on biology is on the rise, from the prediction of symptoms by medical imaging to studies on neuronal or protein structures. A vast majority of computational microbiology algorithm deals with pattern matching of character strings. This is because the most fundamental molecular chemistry structure of the DNA can be abstracted well with just the nucleotide bases as explained in Section 2.4.1. A set of three consecutive bases produce the codon for amino acids which encode the production of enzymes and proteins required for various metabolism. Indeed, plenty of inferences can be derived just from the analysis of the DNA sequence. Before starting to develop corresponding quantum algorithmic constructs, it is vital to assess the development of the classical algorithms, from the most basic idea, to what's being currently used. Since the principles of quantum computation are fundamentally different, it's not the state-of-the-art algorithm but a more basic primitive for which the quantum kernel needs to be constructed.

The following section gives a broad overview of some of the milestone algorithms in the field of

pattern matching and DNA sequencing. It is by no means a comprehensive survey [31] [32], and is selectively focused on the ideas more crucial towards the development of the quantum counterparts.

3.2.1. Naive linear search

The naive approach to sub-string matching involves matching the pattern string P of length M in a reference string T of length $N \geq M$. Both the strings P and T is made up of characters from the alphabet Σ. Thus, each character comparison takes time proportional to the number of binary digits required to represent the characters in the alphabet. Let this be represented by $A = |\Sigma|$. In the naive approach, the pattern is placed at the first index, and each consecutive characters in the pattern and the corresponding character in the reference is compared. If the end of the pattern is reached, the index is returned with a success flag, else, the pattern is shifted by one place and the comparison is restarted from the beginning of the pattern. Shifting P with respect to T is actually done by a constant time increment of the search pointer thus no data transfer is incurred.

In the worst case, the number of comparisons is thus equal to $M * (N - M + 1) * A$. An example of such a case is when the pattern over the DNA alphabet is, say, AAC in a reference $AAAAAAAT$. Here, $A = 2$ as two bits are sufficient to encode the 4 bases of A, C, G, T as $00, 01, 10, 11$. $N = 8$ and $M = 3$. Thus the first comparison proceeds as AAC, AAA, or $000001, 000000$ and thus takes 6 units of time. The worst case as is follows is $3 * (8 - 3 + 1) * 2 = 36$.

3.2.2. Improved exact string matching

The core idea of improving on the naive approach is developing a strategy such that, after a mismatch, more than one place shift is done to the pattern comparison position. This requires some pre-processing of either P or T before the search phase. Two of the most popular exact string matching algorithm of Boyer-Moore and Knuth-Pratt-Morris is sketched here. The pre-processing of the original algorithms are slightly different. The description as given in [30] is followed here.

Fundamental pre-processing

The fundamental pre-processing algorithm calculates $Z_i(S)$ for each position $i \in [0 \cdots (|S| - 1)]$ as the length of the longest substring of S that starts at index i and matches a prefix of S. This step itself can be used as a search algorithm if $S = P\$T$ of length $N + M + 1$ ($ is a character outside the alphabet set for the pattern and reference text strings). In this method, $Z_i(S)$ is always less than M as encountering $ for prefix will always result in a mismatch. Indices j for which $Z_j(S) = M$ are alignment positions where P occurs in $T[(j-M-1) \cdots (j-2)]$. As an example, let's consider $T = gaacgtaactta$ and $P = aact$ ($N = 12$ and $M = 4$). Thus, $Z(aact\$gaacgtaactta) = \{0, 1, 0, 0, 0, 0, 3, 1, 0, 0, 0, 4, 1, 0, 0, 0, 1\}$. Since, $Z_{11} = M$, the position of P in T is at index $11 - 4 - 1 = 6$ of T (by convention for this thesis, 1st character is at index 0). There exist algorithms for calculating Z_i in linear time and independent of the length of the alphabet (as only a comparison for a mismatch is required).

Boyer-Moore algorithm

The Boyer-Moore algorithm has a linear worst case search time but what makes it efficient is that it has a sub-linear average case. It employs two main heuristics over the naive approach. Firstly, for every right-shift of P in T on misalignment, the character-by-character comparison starts from the right-end of P. Secondly, it employs the "extended bad character match rule" that takes both $O(M)$ time and space. By this rule, on mismatch at position $(M - i)$ of P with position j of T with characters y and x respectively, P is right-shifted such that, the closest x to the left of $(M - i)$ in P aligns with position j of T. This is an improvement for sparse alphabets (like DNA sequences) over the bad character match rule, which just shifts P right such that the rightmost x in P aligns with T_j. The extended version trades more look-up table space in pre-processing (the simple version needs only $O(A)$ space).

Kruth-Pratt-Morris algorithm

The Kruth-Pratt-Morris algorithm also runs in worst-case linear time like the pre-processing based search and the Boyer-Moore algorithm. However, it has two main advantages. It can be adapted for real-time string matching and its improvement, as suggested by the Aho-Corasick algorithm, extends the linear time for a set of patterns. The pre-processing step involves calculating the failure function, $W_i(P)$ for $i \in [0 \cdots (M - 1)]$ as the length of the longest proper suffix of $P[0 \cdots i]$, that matches a prefix in P. Thus for $P = aact$, the reference text independent pre-processing gives $W(aact) = 0, 1, 0, 0$. For

Table 3.1: Cost matrix

Sub	A	C	G	T	Ins
A	1	-1	-1	-1	-0.4
C	-1	1	-1	-1	-0.4
G	-1	-1	1	-1	-0.4
T	-1	-1	-1	1	-0.4
Del	-0.4	-0.4	-0.4	-0.4	0

the search phase, if a mismatch is encountered at position $(i+1)$ in P and k in T, the alignment is shifted to the right by $(i - W_i)$ places. Additionally for real time search, W_i^c for each character $c \in \Sigma$ is calculated such that, $P(i+1) = c$, and the required W_i^c is chosen such that the mismatch character $T_k = c$.

The two approaches discussed above are fundamental for understanding DNA sequence alignment and easily extendable for the development of more complex algorithms in further sections. Another approach for an exact matching algorithm which gives similar run-time is based on suffix trees, which can also generalise to other problems in computer science but are not of immediate interest.

3.2.3. Approximate matching using dynamic programming

While the time complexity of some of the best exact approaches is attractive, the performance degrades quickly as they are not suitable for an approximate match, something very common for DNA sequences. These are due to read errors and needs to be considered for the algorithms to be developed in the thesis. The only way to extend the exact algorithms is to include wildcard characters in the alphabet which are to be ignored in the match phase. This increases the search space as now each wildcard character can match with any alphabet.

Needleman-Wunsch algorithm

The notion of sub-sequence and Levenshtein distance is already introduced in Section 3.1.1. Transformation matrices like BLOSUM and PAM are popular for protein sequence comparison, whereas for DNA, normally an unweighted simpler matrix is used. For example, the BLAST algorithm uses $+5$ for matches and -4 for mismatch penalty. *No single scheme is right for all application and remains an experimentally tunable metric.* A fully worked out example over the DNA alphabet is considered here. Let the cost matrix be defined as given in Table 3.1 over the three operations and four alphabets. One substitution equals an insertion followed by a deletion, thus, unless they add up to a value less than the substitution cost, these operations are redundant. Now for translating $s_1 = aactgtgcacta$ to $s_2 = aacctggcactt$, the following alignment is considered:

$s_1 : \texttt{aac}\downarrow\texttt{tgtgcact}\underline{\texttt{a}}$
$s_2 : \texttt{aacctg}\uparrow\texttt{gcact}\underline{\texttt{t}}$

This alignment costs $10 - 0.4 - 0.4 - 1 = 8.2$. Alignments for approximate matching are not unique. For the sequence considered, a higher cost but valid edit would be to align the sequence with only substitutions.

$s_1 : \texttt{aac}\underline{\texttt{tgt}}\texttt{gcact}\underline{\texttt{a}}$
$s_2 : \texttt{aac}\underline{\texttt{ctg}}\texttt{gcact}\underline{\texttt{t}}$

This gives a cost of $8 - 4 = 4$ (in fact the edit penalty here equals the Hamming distance).

The process described above is called the Global Sequence Alignment (GSA) between two strings. The minimum alignment cost can be found by the Needleman-Wunsch algorithm. The table is initialised with the two strings aligned along the two axes. The first row and first column are filled by starting from 0 and subtracting 1 for each cell. Thereafter, it is filled gradually by considering cells which already has their top, left, and top-left neighbours already filled. The maximum of these values are taken, and 1 is added/subtracted based on a match/mismatch on the corresponding characters of the row and column of the two strings respectively. The table for the two strings of the previous section is shown in Figure 3.2. The colour gradient represents the value, while the superimposed alignment grid shows which of the three paths were used to calculate the cell's value. It is evident that multiple paths exist with same total cost in the edit graph. As a simple example, both

$s_1 : \texttt{aac}\downarrow\texttt{tgtgcact}\underline{\texttt{a}}$
$s_1 : \texttt{aa}\downarrow\texttt{ctgtgcact}\underline{\texttt{a}}$

aligns to
s_2 : aacctg↑gcactt
The graph is traversed from the two diagonal ends to find the transcript for the minimum edit path.

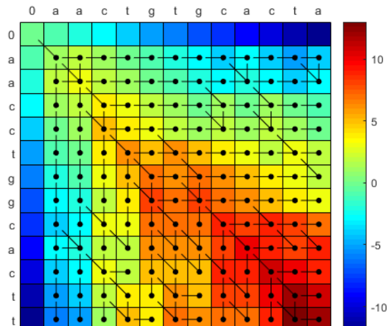


Figure 3.2: Scoring matrix and alignment grid for Needleman-Wunsch example

Smith-Waterman algorithm

In many biological applications, the alignment score between two substrings of s1 and s1 might actually be larger than the alignment score between the entire strings. The Local Sequence Alignment (LSA) problem attempts to maximize it over all possible substrings. It finds the longest paths from the source $(0, 0)$ to every other vertex by adding edges of weight 0 in the edit graph, making the source vertex a predecessor of every vertex in the graph. This is called the Smith-Waterman algorithm. For each node, the scoring matrix elements $s_{i,j}$ is calculated as:

$$s_{i,j} = max \begin{cases} 0 \\ s_{i-1,j} + \delta(s1_i, -) \\ s_{i,j-1} + \delta(-, s2_j) \\ s_{i-1,j-1} + \delta(s1_i, s2_j) \end{cases}$$

where, the function δ represents the weighted insertion, deletion and substitution penalties. The process is similar to the GSA, except now, *for each pair of nodes in the matrix, a path is attempted*, instead of only the end-to-end path. A case is shown here, where LSA proves more advantageous than GSA.
$GSA - s_1$: ..t..cc.c.agt..tatgt.caggggacacg..a.gcatgcaga.gac
$GSA - s_2$: aattgccgcc.gtcgt.t.ttcag....ca.gttatg..t.cagat..c
$LSA - s_1$: ....tcccagttatgtcaggggacacgagcatgcagagac
$LSA - s_2$: aattgccgccgtcgttttcagcagttatgtcagatc..................
Though GSA has 23 matches, it has 26 penalties; whereas, the 12 matches and 6 substitutions of LSA, often reflect a higher degree of structural similarity based on the circumstance. In long-read alignment, it is preferable to find local matches because a long read is more fragile to structural variations and misassemblies in the reference compared to mismatches towards the ends.

3.2.4. Heuristic algorithms

There are many advanced approaches that are built on the basic algorithms because natively, they are costly in terms of memory and processing. Modifications include allowing multiple consecutive insertion/deletion with a lesser sub-sequence penalty, called gap penalty. Various other improvements include adopting it for multiple sequences, block alignment, four-Russian speedup, graph-based approaches, etc. Here, a historic development of the currently used algorithms is presented, bridging the previous section with the state-of-the-art. The quantum approaches are significantly different and tends to avoid heuristics to harness the quantum parallelism for speedup, preserving algorithmic simplicity.

Local alignment software like FASTA and BLAST were first developed in the 1990s. Faster methods to find DNA sequence matches of capillary sequencing reads against a large reference genome was developed since, e.g. MegaBLAST, SSAHA2, BLAT and PatternHunter. With NGS technology even faster

(10-1000 times) algorithms were developed, e.g. SOAP, MAQ, Bowtie and BWA. This generation of sequencers generated millions of short (~ 100bp) reads. Efficiently aligning these reads against a long reference sequence (like the human genome) poses a new challenge for alignment tools. Acceleration techniques like spaced seed templates spanning the entire read, or quickly filter out poor matches, by applying q-gram filtration or by bounding the search process are used.

Hash table indexing is a popular choice for speeding up alignment. Alternatively, a Smith–Waterman-like dynamic programming [33] can be applied between a query pattern and the suffix tree of the reference, aligning the search pattern against each sub-sequence sampled from the suffix tree via a top-down traversal. The suffix tree can also be represented by an FM-index [34]. Instead of using the original string, the Burrows-Wheeler Transform (BWT) can be used for the FM-index, to reduce memory footprint [35]. This algorithm, BWT-SW, is still slower than BLAST on long query sequences as it finds all matches. Heuristics are used to further accelerate it, creating Burrows-Wheeler Aligner (BWA) [36], trading Smith-Waterman algorithm's guarantee to find all local hits with speed for both short and long query sequences. It follows the seed-and-extend paradigm. BWA-SW builds FM-indices for both the reference and the query sequence. It implicitly represents the reference sequence in a prefix trie and represents the query sequence in a prefix directed acyclic word graph (DAWG). Two heuristic rules are applied to accelerate the process over BWT-SW. Traversal on the DAWG is carried in the outer loop pruning low-scoring matches at each node to restrict the dynamic programming. Secondly, only largely non-overlapping alignments on the query sequence are considered. One of the popular algorithms today is an improvement of BWA-SW called, BWA-MEM, and is based on an algorithm finding super-maximal exact matches (SMEMs) [37].

3.2.5. De novo assembly

Returning to the two types of sequence reconstruction as sketched in Section 2.4.2, the algorithms discussed so far fall under alignment-based approaches. It is clear from the discussions that, current techniques trade speed and memory for accuracy. These heuristics make the problem tractable for higher sized genomes like that of a human. However, the approximations and errors introduced are on the way against further progress in critical application domains like personalised medicine. Given enough computing power, de novo sequencing is highly desirable. Thus, since quantum computing presents itself as an ultimate computing machine, it is worth exploring the problem of de novo sequencing. There are two main methods for de novo sequencing: Overlap/Layout/Consensus (OLC) methods (relying on overlap graph) and de Bruijn Graph (DBG) methods (using K-mer graph).

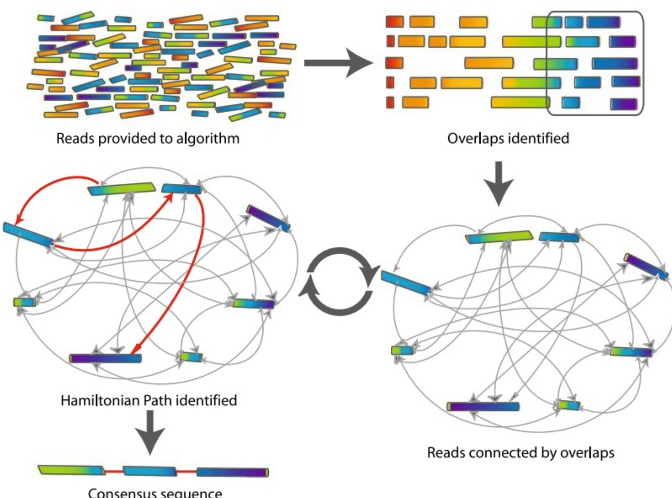


Figure 3.3: Overlap-Layout-Consensus genome assembly algorithm [38]

Overlap-Layout-Consensus method

An overlap graph represents the sequencing reads as nodes and their *overlaps pre-computed by (computationally expensive) pair-wise sequence alignments* as edges. Paths through the graph are the potential contigs and can be converted to sequence. Formally, it is a Hamiltonian cycle, a path that

travels to every node of the graph exactly once and ends at the starting node, including each read once in the assembly. There is no known efficient algorithm for finding a Hamiltonian cycle as it is in the NP-Complete class. Though it was feasible for microbial genome (in 1995) and the human genome (in 2001) NGS projects have abandoned it due to the computational burden.

de Bruijn graph

In de Bruijn graph the nodes represent all possible fixed-length strings of length K (K-mer graph). The edges represent fixed-length suffix-to-prefix perfect overlaps between sub-sequences that were consecutive in the larger sequence. In WGS assembly, the K-mer graph represents the input reads. Each read induces a path and those with perfect overlaps induce a common path. Thus, perfect overlaps are detected implicitly without any pair-wise sequence alignment calculation, the major advantage with respect to OLC methods. However, compared to overlap graphs, K-mer graphs are more sensitive to repeats and sequencing errors as K is much less than read size. By construction, the graph contains a path corresponding to the original sequence. For an ideal scenario, this is the Eulerian path, that traverses each edge exactly once, which is a tractable problem with run-time proportional to the number of edges in the graph.

Though graphs built from real sequencing data are more complicated, assembly by this approach is a by-product of the graph construction. Real-world WGS data induces various problems in both these methods. Examples are spurs (short, dead-end divergences from the main path), bubbles (paths that diverge then converge), frayed rope pattern (paths that converge then diverge) and cycles (paths that converge on themselves) [25]. Common causes of these complexities are attributed to repeats in the target and sequencing error in the reads. Most optimal graph reductions belong to NP-hard class of problems, thus assemblers (like Euler, Velvet, ABySS, AllPaths, SOAPdenovo) rely on heuristics to remove redundancy, repair errors or otherwise simplify the graph.

The choice of algorithms is based on the quality, quantity and structure of the genome data. Current short-read sequencing technologies produce very large numbers of reads favouring DBG methods. SMS for third generation sequencing produces high-quality long reads which could favour OLC methods again. For sure, to solve de novo sequencing, graph algorithms are required, a field not as well developed as search algorithms for quantum computing. While for OLC methods, quantum computing is unlikely to give significant advantages in the pathfinding stage as it is NP-Complete. However, *de novo assembly still needs the pair-wise alignment as a major pre-processing step*. Thus, sub-sequence alignment is a primitive [39] that is pervasive in bioinformatics - the focus of the quantum algorithms developed in this thesis.

3.3. Quantum search algorithms

The field of quantum algorithm development came into focus in the mid-1980s with the works of David Deutsch [40] and others. A decade later, Peter Shor [41] showed an advantage of quantum computing over classical computation in practical disciplines like cryptography leading to widespread research boost in this domain. Shor's algorithm for factorisation is often partnered with Grover's search algorithm [42] [43] as the two most popular quantum algorithms for demonstrating practical computational advantage. Many quantum algorithms has been developed since then. A curated directory can be found in the Quantum Algorithm Zoo [44], which categorically describes the various quantum algorithm. The first category is *Algebraic Number Theoretic*. It includes problems like factoring, discrete-log, Pell's equation, verifying matrix products, constraint satisfaction, etc. The *Approximation and Simulation* category includes problems like quantum simulation, adiabatic algorithms, semi-definite programming, Zeta functions, simulated annealing, etc. However, the category that interests this thesis most are the *Oracular* algorithms. This includes many sub-categories like searching, Abelian Hidden Subgroup, non-Abelian Hidden Subgroup, Bernstein-Vazirani, Deutsch-Jozsa, structured search, pattern matching, welded tree, graph collision, matrix commutativity, counterfeit coins, search with wildcards, network flows, machine learning, and many more. Note, not all these algorithms provide a super-polynomial speedup. New fields of quantum algorithms research employ applying the rules of quantum mechanics to game theory to model the situation of conflict between competing agents. The impact of quantum information processing on classical scenarios can be studied. Quantum games can be also used to analyse typical quantum situations like state estimation and cloning. Quantum walks also provide another promising method for developing new quantum algorithms. It was also shown that quantum walks can be used to perform a universal quantum computation.

The development of quantum algorithms is a very lively area of research. For this thesis, however, the focus is on a small part of this landscape. In this section, Grover's search is reviewed to build the foundation for the other algorithms in the thesis. Thereafter, some preliminary modifications of the original algorithms by others are stated, that increases the applicability of the search procedure.

3.3.1. Grover's search

Section 3.1 models the core of the quantum algorithm as a kernel that would allow indexing a search string in the reference string. Since no pre-processing is done on the reference string, except slicing it to the search string size chunks, the database is essentially unsorted. L. Grover, in his paper [43] describes the quantum approach to solving the search problem in such a database. A closer look at Grover's search is elucidated here. It is the foundation for the more specialised algorithms that are discussed in later sections.

In the original Grover's algorithm, there is exactly one item which matches the search criteria. The artificial mathematical formalism of Grover's search is to reduce the number of queries required to the database to find the answer by a polynomial (more specifically, quadratic) factor. A one-to-one correlation between the classical worst-case time of $O(N)$ queries (N being equal to the number of database entries), and the quantum run-time of $O(\sqrt{N})$ is not fully justified, as the quantum query itself works in a different technique, evolving the entire superposition of the database states. However, this is the inherent parallelism of quantum algorithms that we tend to harness.

Grover search is however provably optimal, thus no other algorithm, classical or quantum, can give a better runtime with the same initial conditions. However, it makes up for the lower (with respect to QFT) speed-up benefit in 2 ways:

1. Grover assumes an unstructured database search, which is rarely the case. We often have some idea of the data which can be exploited.
2. Searching is a very general problem in computer science and thus the impact factor of the time reduction is of great interest to researchers.

An alternate view of Grover's algorithm can be "inverting a function" instead of "searching a database". Given a function $y = f(x)$ that can be evaluated on a quantum computer, Grover's algorithm can calculate x. It can be used to efficiently determine the number of solutions to an N-item search problem, allowing it to perform exhaustive searches on solutions of NP-complete problems, reducing the required computational resource.

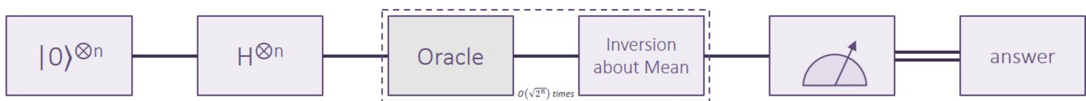


Figure 3.4: Grover search steps

Grover's search starts out with an equal superposition of states, i.e. each database entry has an equal probability of being the answer. The initial state can be described as:

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

It is an oracular algorithm, i.e. it assumes the existence of an Oracle function (a common algorithm construct), which can produce an yes or no answer for a query in constant time. In the search procedure, the Oracle is consulted, which rotates the phase of the answer by π radians. Thus the Unitary matrix is a diagonal matrix with all diagonal elements being 1 except at the row/column where the search entry and can be described as:

$$O_{jk} = \begin{cases} 0, & \text{if } j \neq k \\ -1, & \text{if } j = k \text{ and } j = i_0 \\ 1, & \text{otherwise} \end{cases}$$

$$|\psi_1\rangle = O\,|\psi_0\rangle$$

The next step is an inversion about the mean value of the states. This is known as the Grover gate, or the diffusion operator which is responsible for the amplitude amplification of the result. This operation

can be described as:

$$G = 2\left|\psi_1\right\rangle\left\langle\psi_1\right| - I$$

$$\left|\psi_2\right\rangle = G\left|\psi_1\right\rangle$$

Grover search guarantees the probability of the solution state to reach near unity on iterating the last two steps $\sqrt{N}$ times.

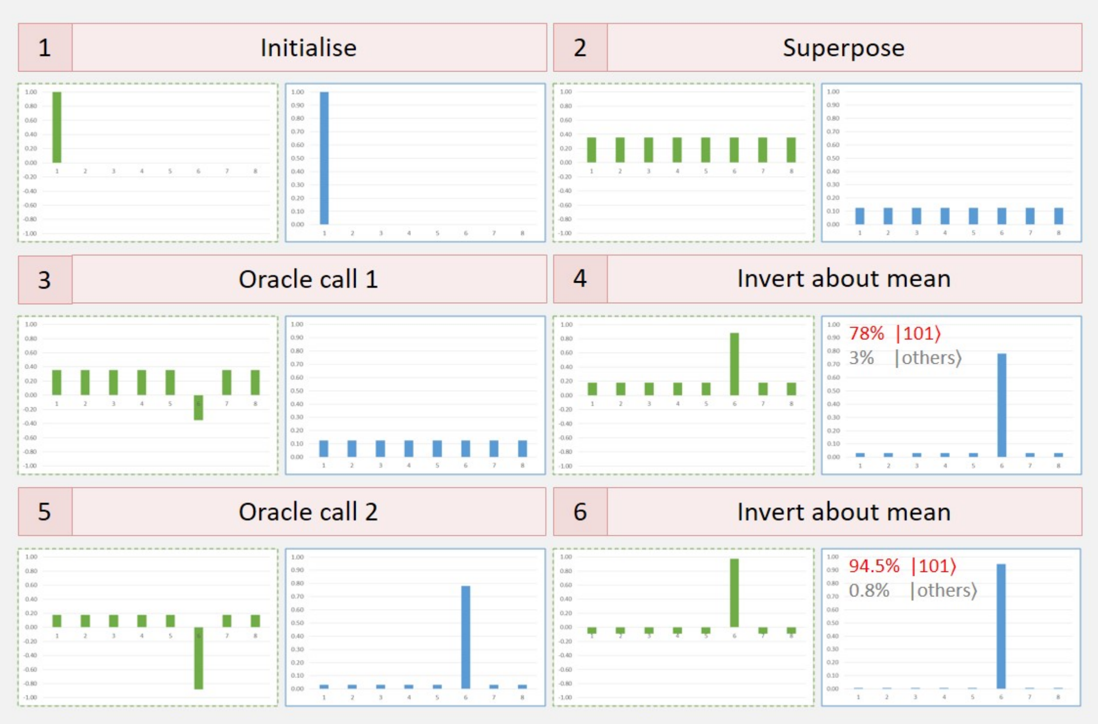


Figure 3.5: Grover search example for 3 qubits

An example run for the search is shown in Figure 3.5. For the 3 qubit case, there are a total of 6 steps as the iteration requirement is $\sqrt{2^3} \approx 2$ for the Oracle and inversion about mean step. At each step, the internal real amplitude of the states is shown in green (left), and its squared value, the measurement probability is shown in blue (right). The initialisation step erases each qubit's state, resetting it to $|0\rangle$. Then, the Hadamard gate on each qubit takes the state to an equal superposition of every possible 3-qubit basis states (3-bit binary strings). The next step is the Oracle call, which is a black box for the algorithm. It marks one of the states by inverting it (rotating it by π). However, this negative sign has no effect on the measurement probability. The amplitude is amplified by the inversion about mean step. The first run gives a measurement probability of 78%. Repeating it for the optimal number of iteration increases it to 94.5%.

For a detailed algebraic analysis, let the state at iteration j of the Grover search be:

$$|\psi(k_j, l_j)\rangle = k_j\left|i_0\right\rangle + \sum_{i \neq i_0} l_j\left|i\right\rangle, \text{ where } k_0 = l_0 = \frac{1}{\sqrt{N}}$$

The first step of the iteration marks i_0, to flip the state to $-k_j\left|i_0\right\rangle$. The mean is thus given by:

$$\mu_j = \frac{(N-1)l_j - k_j}{N}$$

Each state gets transformed by the Grover gate from $\alpha_j\left|i\right\rangle$ to $(2\mu_j - \alpha_j)\left|i\right\rangle$. Thus, the recursive relation for the states can be expressed as:

$$k_{j+1} = 2\frac{(N-1)l_j - k_j}{N} - (-k_j) = \frac{N-2}{N}k_j + \frac{2(N-1)}{N}l_j$$

$$l_{j+1} = 2\frac{(N-1)l_j - k_j}{N} - l_j = \frac{-2}{N}k_j + \frac{N-2}{N}l_j$$

The recurrence can be solved by taking $1/N = sin^2\theta$, to give the closed-form equation :

$$k_j = sin((2j+1)\theta \quad \text{and} \quad l_j = \frac{1}{\sqrt{N-1}}cos((2j+1)\theta$$

Setting $k_t^2 = 1$, where j_{opt} is the optimal number of iterations, we get,

$$j_{opt} = \frac{(2m+1)\pi - 2\theta}{4\theta} = \frac{(2m+1)\pi - 2sin^{-1}(1/\sqrt{N})}{4sin^{-1}(1/\sqrt{N})}, \text{ where } m \in \mathbb{Z}$$

However, the equation is continuous while $j \in \mathbb{Z}^+$. Approximating the equation, if we iterate $\lfloor \pi\sqrt{N}/4 \rfloor$ times, the probability of failure is just $1/N$ when N is large.

3.3.2. Generalising quantum search algorithms

Grover's search was enhanced by two subsequent research [45] [46] that will allow us to apply this search in our context. The improvements discussed in this section are:

- Multiple known number of solutions
- Arbitrary distribution of initial amplitude
- Multiple unknown number of solutions by randomising iterations over multiple runs
- Multiple unknown number of solutions by counting number of solutions

Multiple known solutions

The case for multiple known solutions is considered first. Let t be the number of solutions (known in advance), and S be the set of states considered as solutions. The transformation generalises to:

$$|\psi(k_j, l_j)\rangle = \sum_{i\in S} k_j |i\rangle + \sum_{i \notin S} l_j |i\rangle \mapsto |\psi\left(\frac{N-2t}{N}k_j + \frac{2(N-t)}{N}l_j, \frac{-2t}{N}k_j + \frac{N-2t}{N}l_j\right)\rangle = |\psi(k_{j+1}, l_{j+1})\rangle$$

The modification involves taking $t/N = sin^2\theta$, to give the solutions as:

$$k_j = \frac{1}{\sqrt{t}}sin((2j+1)\theta) \quad \text{and} \quad l_j = \frac{1}{\sqrt{N-t}}cos((2j+1)\theta)$$

The probability of measuring any one of the solution state is maximised when l_j is close to 0, which yields the relation:

$$j_{opt} = \frac{(2m+1)\pi - 2\theta}{4\theta} = \frac{(2m+1)\pi - 2sin^{-1}(t/\sqrt{N})}{4sin^{-1}(t/\sqrt{N})}, \text{ where } m \in \mathbb{Z}$$

which can now be approximated for integer iteration as $\left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{t}} \right\rfloor$. The solution state probability upper-bounded by $1/t$, can now be written wholly in terms of t and N as:

$$k_{opt}^2 = \frac{1}{t}sin^2\left(\left(2\left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{t}} \right\rfloor + 1\right)sin^{-1}\sqrt{\frac{t}{N}}\right)$$

Arbitrary initial amplitude

The second improvement that is needed is to consider an arbitrary initial amplitude for multiple known solutions. Instead of working with the amplitudes directly, the mean and variance of the solution and non-solution states are considered.

$$\overline{k_j} = \frac{1}{t}\sum_{i\in S} k_j \quad \text{and} \quad \sigma_k^2 = \frac{1}{t}\sum_{i\in S} |k - \overline{k}|^2$$

$$\overline{l_j} = \frac{1}{N-t}\sum_{i\notin S} l_j \quad \text{and} \quad \sigma_l^2 = \frac{1}{N-t}\sum_{i\notin S} |l - \overline{l}|^2$$

Note, the variance equations are time-independent. The mean over these states after the solution states are marked (Oracle called) is given by:

$$\mu_j = \frac{(N-t)\overline{l_j} - t\overline{k_j}}{N}$$

The dynamics dictated by Grover's algorithm can be described by the time-dependence of this average, giving the recurrences as:

$$k_{j+1} = 2\mu_j + k_j \quad \text{and} \quad l_{j+1} = 2\mu_j - l_j$$

Since the $2\mu_j$ factors are added to every term in the set, the mean itself evolves as:

$$\overline{k_{j+1}} = 2\mu_j + \overline{k_j} \quad \text{and} \quad \overline{l_{j+1}} = 2\mu_j - \overline{l_j}$$

The solution to this recursion in closed-from is given by:

$$\overline{k_j} = \overline{k_0} cos(\omega j) + \overline{l_0}\sqrt{\frac{N-t}{t}} sin(\omega j)$$

$$\overline{l_j} = -\overline{k_0}\sqrt{\frac{t}{N-t}} sin(\omega j) + \overline{l_0} cos(\omega j)$$

where, $\omega = cos^{-1}\left(1 - \frac{2t}{N}\right)$ The optimal number of iterations and probability of success is given by:

$$j_{opt} = \frac{(2m+1)\frac{\pi}{2} - 2tan^{-1}\left(\frac{\overline{k_0}}{\overline{l_0}}\sqrt{\frac{t}{N-t}}\right)}{2cos^{-1}\left(1 - \frac{2t}{N}\right)}$$

$$P_{max} = 1 - \sum_{i \notin S} |l - \bar{l}|^2$$

These two relations are very useful as j_{opt} is used to calculate the number of iterations that the program needs, and thereby the number of gates that would be executed. The P_{max} value helps in understanding the applicability of the search algorithm on a given set of data.

Multiple unknown solutions (by randomising iterations of multiple runs)

The next modification that is needed is the case for multiple solutions when the number of solutions is not known in advance. There are two ways in which such a problem can be attacked. When the number of solutions is not known, the number of required iterations cannot be predicted in advance. Thus, if a random iteration limit is chosen over all possible values for iterations (from the value for 1 solution to all states being solution states), then with a finite probability, the right number of iteration will be chosen. If this probability is high, the solution state is amplified with a high probability. This is the intuition behind the first method. Using trigonometric formula for compound angles and summation trigonometric series expansions, for real numbers α, β and an arbitrary positive integer δ, we can derive:

$$\sum_{j=0}^{\delta-1} cos(\alpha + 2\beta j) = \frac{sin(\delta\beta)cos(\alpha + (\delta-1)\beta)}{sin\beta}$$

for the case, $\alpha = \beta$,

$$\sum_{j=0}^{\delta-1} cos((2j+1)\alpha) = \frac{sin(2\delta\alpha)}{2sin\alpha}$$

Let t be the number of unknown solutions. The total probability of measuring a solution state after j iteration (using previously derived relations and $|S| = t$) is,

$$P_{soln} = \sum_{i \in S} k_j^2 = sin^2((2j+1)\theta)$$

The average success probability when $0 \leq j \leq \delta$ (and simplified by the relation $2sin^2\gamma = 1 - cos(2\gamma)$), is,

$$P_\delta = \sum_{j=0}^{\delta-1} \frac{1}{\delta} sin^2((2j+1)\theta) = \frac{1}{2\delta} \sum_{j=0}^{\delta-1} \Big(1 - cos(2(2j+1)\theta)\Big) = \frac{1}{2} - \frac{sin(4\delta\theta)}{4\delta sin(2\theta)}$$

To get a P_{soln} more than $1/4$, the second term should be less than $1/4$. Since $sin(4\delta\theta) \leq 1$, δ can be chosen as,

$$\delta \geq \frac{1}{sin(2\theta)} = \frac{1}{sin\left(2sin^{-1}\sqrt{\frac{t}{N}}\right)} = \frac{1}{2\sqrt{\frac{t}{N}}\sqrt{1-\frac{t}{N}}} = \frac{N}{2\sqrt{(N-t)t}}$$

Thus, the value to be chosen for δ, and thus the number of iterations to be performed, depends on the fraction of states that are solution. Now for the algorithm, an arbitrary value of δ is chosen, and another increment factor $1 < \lambda < 4/3$ is chosen. At each iteration, the Grover's search is performed with $0 \leq j < \delta$. If the measurement result after j iteration is not the solution, δ is incremented to $min(\lambda\delta, \sqrt{N})$. The value of δ on the rth such iteration is $\lambda^{r-1}\delta$. Let $\delta_c = 1/sin(2\theta)$. The critical stage is reached when $r_c = \lceil log_\lambda \delta_c \rceil$. This happens with probability,

$$\sum_{r=1}^{r_c} (1 - P_{\delta\lambda^{r-1}}) = \sum_{r=1}^{r_c} \left(1 - \frac{1}{2} + \frac{sin(4\delta\theta)}{4\delta sin(2\theta)}\right)$$

The expected number of iteration when the critical state is reached (if at all it reaches, observed P_{fail} for rounds before it is 1), is thus expanded (using geometric series expansion),

$$E[r_c] = \frac{1}{2}\sum_{r=1}^{r_c} \delta\lambda^{r-1} = \frac{1}{2}\sum_{r=0}^{r_c-1} \delta\lambda^{r} = \frac{\delta(\lambda^{\lceil log_\lambda \delta_c \rceil} - 1)}{2(\lambda-1)} \leq \frac{\delta(\lambda^{1+log_\lambda \delta_c} - 1)}{2(\lambda-1)} \leq \frac{\delta(\lambda\delta_c - 1)}{2(\lambda-1)} < \frac{\delta\lambda\delta_c}{2(\lambda-1)}$$

After the critical stage, further increase in δ always succeeds with probability greater than $1/4$. Thus, the limiting case is reached when $P_{fail} = 3/4$ and is upper bounded by,

$$E[r_{c^+}] = \frac{1}{2}\sum_{u=0}^{\infty} \left(\frac{3}{4}\right)^u \frac{1}{4}\lambda^{u+r_c} = \frac{\lambda^{r_c}}{8}\sum_{u=0}^{\infty} \left(\frac{3\lambda}{4}\right)^u = \frac{\lambda^{r_c}}{8}\left(\frac{1}{1-\frac{3\lambda}{4}}\right) = \frac{\lambda^{\lceil log_\lambda \delta_c \rceil}}{8-6\lambda} \leq \frac{\lambda\delta_c}{8-6\lambda}$$

The total expected number of iteration of the Grover algorithm (each time with different number of Grover iterations in them), when $t \leq 3N/4$, $\delta = 1$ and $\lambda = 6/5$ can be derived as

$$E[r_c] + E[r_{c^+}] < \frac{\delta\lambda\delta_c}{2(\lambda-1)} + \frac{\lambda\delta_c}{8-6\lambda} = \left(3 + \frac{3}{2}\right)\delta_c = \frac{9N}{4\sqrt{(N-t)t}} = O\left(\sqrt{\frac{N}{t}}\right)$$

This is approximately 4 times the number of iterations had t been known in advance. The case for no solution is handled with a time-out, while the case for $t > 3N/4$ can be solved in constant time by classical sampling.

Multiple unknown solutions (by counting)

The second method for multiple solutions is more intuitive. It divides the algorithm into two steps. In the first step, another quantum algorithm counts the number of solutions and then, the algorithm for known multiple solution is used to maximise the solution probability. Formally, counting is the cardinality of an inverse Boolean function B with input 1.

$$t = |B^{-1}(1)|$$

There are different ways to do quantum counting. The first method is by using quantum Fourier transform (QFT) to find the period θ as k_j evolves, to find t. The number of iterations executed is encoded as part of the state,

$$|\psi(k_j, l_j, \gamma)\rangle = \sum_{j=0}^{2^\gamma - 1} \left[\frac{1}{\sqrt{2^\gamma}} |j\rangle \left(\sum_{i \in S} k_j |i\rangle + \sum_{i \notin S} l_j |i\rangle\right)\right]$$

Now, if the original qubits in $|i\rangle$ is observed, the state collapses to (within normalisation factors),

$\sum_{j=0}^{2^\gamma-1} k_j |j\rangle$ or $\sum_{j=0}^{2^\gamma-1} l_j |j\rangle$. Running discreet QFT on this state, on measurement, with high probability the value f can be estimated. The number of solutions can be calculated as, $t = N sin^2(f\pi/2^\gamma)$. The value of γ helps to balance the accuracy with run-time and needs to be typically increased gradually over multiple runs until f becomes large.

Another conceptually simpler way [47] to count is to determine the fraction of state marked by the Oracle. A state is created such that an extra qubit is $|1\rangle$ if it is a solution, $|0\rangle$ otherwise,

$$|\psi(k_j, l_j)\rangle = \sum_{i \in S} k_j |i\rangle |1\rangle + \sum_{i \notin S} l_j |i\rangle |0\rangle$$

Now, measurement by state tomographic trials on this qubit gives the proportion of solution state to the total number of states with increasing degree of accuracy.

An arbitrary amplitude and multiple solutions without counting is based on finding the number of time the original Grover algorithm is to be run. However, quantum counting involves states and procedures that are different from the qubit encoding of Grover. The second method is decomposed to the required Oracle unitary. Let the Oracle is based on the Boolean function B, such that it's elements, $b_i \in \{0, 1\}$ $i \in \{0 \cdots (N-1)\}$ is marked when it is a solution state. The unitary matrix is formed such that the most-significant qubit is the count qubit. In terms of diagonal matrices and element-wise subtraction it can be written as,

$$O_{count} = \begin{bmatrix} diag(1-B) & diag(-B) \\ diag(-B) & diag(1-B) \end{bmatrix}$$

3.4. Quantum pattern matching

There are a handful of quantum pattern matching algorithms already proposed. In this thesis, the focus is on using Grover's quantum search primitive for pattern matching. There are also other approaches to pattern matching that does not depend on Grover's search. For example, a Quantum Walk based searching [48] [49], Quantum Fourier Transform (Abelian group Hidden Structure Problem (HSP)) based template matching [50] [51], Dihedral group HSP based pattern matching [52] [53] and Adiabatic Computation [54]. Also, there are algorithms which propose using the structure of the problem (for e.g., a sorted database) [55] to speedup the search, or using a mix of classical and quantum strategies [56]. It is too wide to explore the entire domain of quantum algorithms for application to pattern matching. The search methods discussed in this section are chosen based on how intuitive it is to adapt for bioinformatics applications and in some sense closer to classical approaches of naive searching introduced in Section 3.2.1. Nevertheless, the other approaches remain promising and it would be interesting to compare and contrast as a future scope keeping this thesis in perspective.

The three approaches discussed here are based on the following articles:

- Quantum pattern matching [57] - it introduces the concept of having a set of Oracles (one for each alphabet) and conditionally invoking one from the set based on the search pattern.
- Fast quantum search algorithms in protein sequence comparisons: Quantum bioinformatics [58] - it describes the concept of a quantum phone directory where the database is encoded as a set of indices and stored patterns and evolves to the Hamming distance from the search pattern.
- Quantum associative memory [59] - introduces the concept of quantum neural network based pattern storage and recall.

In this section, the core idea (algorithm block diagram) and the calculations for the state evolution (like the number of iterations and solution probability) are discussed. The notations are tweaked to maintain consistency with the previous section, on which these algorithms have their foundations.

3.4.1. Approach 1: Conditional Oracle call

The quantum pattern matching algorithm proposed by P. Mateus and Y. Omar [57] is the first algorithm explored here. Grover's algorithm (or the variants discussed in the previous section) cannot be directly used for pattern matching. In Grover's search, the initial input state is an equal superposition of all the possible strings of the size of the search pattern. Thus, the number of states in the superposition is A^M. The Oracle then marks the answer state so that the output is the search pattern. However, if the index of the pattern matching is the requirement, the state needs to be initialised such that it stores a superposition of indices in $N - M + 1$. The Oracle marks the index where the pattern matches. So in a naive Grover's search implementation, the entire pattern matching comparison is off-loaded

to the Oracle, and the algorithm is not useful without a description of the Oracle construction. The Oracle, however, is a unitary matrix with the diagonal elements being -1 for the answer index and 1 otherwise. Thus the *answer needs to be known for the Oracle construction*, the exact problem that needs to be avoided to practically program a pattern matching application.

For matching a sub-string, only a sequential set needs to be considered. The input state as described in the paper is,

$$|\psi_0\rangle = \frac{1}{\sqrt{N-M+1}} \sum_{i=0}^{N-M} |i, i+1, \cdots, i+M-1\rangle$$

Henceforth, an example case of an input alphabet of $\Sigma = \{0,1\}$ and $T = 111000000$ is considered. Let the search pattern for which the index is to be found, be $P = 10$. The initial state for this sample case with $N = 9$ and $M = 2$ thus becomes

$$|\psi_0\rangle = \frac{1}{3}\{|0,1\rangle + |1,2\rangle + \cdots + |7,8\rangle\}$$

The circuit description in the paper however always generates a unique superposition of N states. How are the last states, with the starting index $(N-M+1) > i > (N-1)$ described? It is noted that, in the state of the input circuit, the digits of the input state saturates at N. Thus, the possible matching index states and the corresponding search pattern strings are,

$$\{|0,1\rangle, |1,2\rangle, |2,3\rangle, |3,4\rangle, |4,5\rangle, |5,6\rangle, |6,7\rangle, |7,8\rangle, |8,8\rangle\} \longmapsto \{11, 11, 10, 00, 00, 00, 00, 00, 00\}$$

Note that the last state has repeats, which cannot be described with the recurrence relation given in the paper. To encode $\{0, 1, \cdots, 8\}$ in binary strings would require $q' = \lceil log_2 N \rceil = 4$ qubits. The paper provides the argument that only $\{0, 1, \cdots, 7\}$ as the starting index needs to be encoded, so $q = log_2(N-M+1) = 3$ will suffice. However, since the circuit encodes the remaining pattern index as well for comparison, q qubits are not sufficient. Rendering the initialisation circuit with $q' = 4$ qubits gives an equal superposition of states with $i \in \{0, 1, \cdots, 15\}$ resulting in the initial state (qubit encoding of binary strings) of,

$$|\psi_0\rangle = \frac{1}{4}\{|0000, 0001\rangle, |0001, 0010\rangle, \cdots, |1110, 1111\rangle, |1111, 1111\rangle\}$$

Note that this is different from the initial state till $\frac{1}{3}|7,8\rangle$ that was actually required to be encoded.

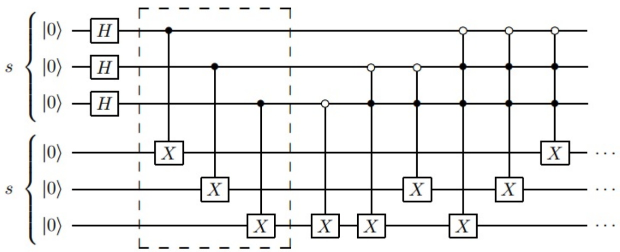


Figure 3.6: Initialisation circuit examples, as given in [57]

The circuit given in the paper shows a gate pattern as an example (Figure 3.6), giving no intuition of why it is the way it is. This is analysed to figure out that, the first set of Hadamard gates on the qubits encoding the first character's of the pattern creates a superposition like,

$$\frac{1}{4}(|0000\rangle + |0001\rangle + \cdots + |1111\rangle)$$

on the first q' set of qubits. The set of CNOTs then copies this to the next q' qubits creating the state

$$\frac{1}{4}(|0000, 0000\rangle + |0001, 0001\rangle + \cdots + |1111, 1111\rangle)$$

The Toffoli gates then implements an increment-by-1 circuit creating the state

$$\frac{1}{4}(|0000, 0001\rangle + |0001, 0010\rangle + \cdots + |1110, 1111\rangle + |1111, 1111\rangle)$$

Note that the increment saturates at all 1 and is easy to verify with a sum-of-product expression of the

Toffoli controls.

In Grover's search, the Oracle function basically stores the relationship between the database and the search string. This relationship thus needs to change for each search string, making it impractical for implementation. The key idea in quantum pattern matching is to define a "compile once, run many" approach for the Oracle. *The algorithm defines multiple Oracles, one for each character of the alphabet.* The Boolean function that the Oracle encodes is -1 for the indices where the reference string matches the Oracle's defining character $\sigma \in \Sigma$.

$$f_\sigma : \{0,1\}^{2^{q'}} \to \{-1,1\}$$

The Boolean function that maps to $\{0,1\}$ is converted to $\{-1,1\}$ by a phase-kickback process of $(-1)^{f_\sigma(i)}$ for implementing the gate level circuit for the Oracle function. *The Oracle construction is independent of the search string*, giving this algorithm its usefulness. The Oracle circuit assembly, however, depends on the search pattern as shown in the algorithm anatomy in Figure 3.7.

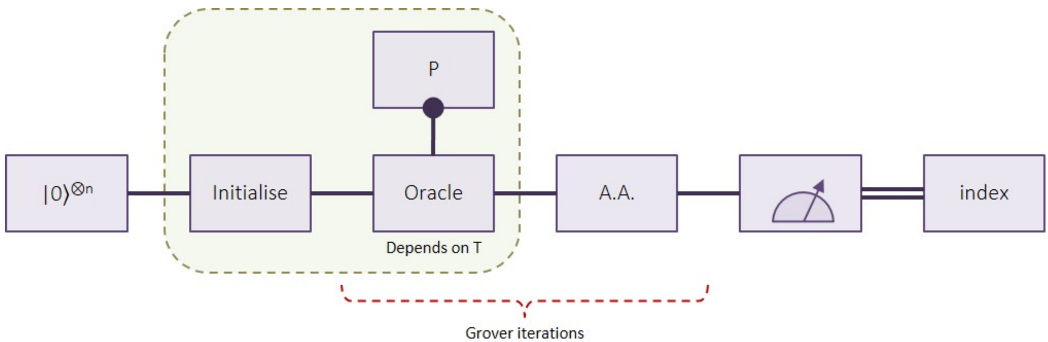


Figure 3.7: Algorithm anatomy

At every iteration step, all the A Oracles exist in the circuit but only one of them is control activated by the step's corresponding character of the search pattern. Since the model of quantum computer is based on in-memory computation, the exact circuit need not be pre-compiled if the underlying micro-architecture and classical control are fast enough to allow real-time circuit interpretation. The circuit for constructing an arbitrary Boolean function is not provided in the paper. The circuit is devised that allows generating an Oracle automatically in a high-level programming language in the kernel. In the implementation, a sequential run through the Boolean function is performed. If the state of a particular index needs to be marked, the Boolean value of the index is taken and a CPhase gate is applied on all the qubits to the Oracle, with inverted control on qubits where the Boolean index encoding is 0. Continuing the example, the Boolean function for σ_1 acting on $q' = 4$ qubits is $f_1 = [1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0]$. Thus, the positions of $0,1,2$, or the qubit states of $|0000\rangle, |0001\rangle, |0010\rangle$ are marked using 3 CPhase Gates over 4 qubits. The first CPhase will have all the controls inverted, for the second CPhase q'_0, q'_1, q'_2 are inverted and for the third CPhase q'_0, q'_1, q'_3 are inverted. The inversion is carried out by wrapping Pauli-X gate on those qubits before and after the CPhase. The multi-qubit CPhase is converted to a multi-qubit CNOT by wrapping a Hadamard on one of the qubits and then decomposing.

The third part of the circuit is the Grover amplification process over the entire non-ancilla qubit set. In the initialisation circuit, the Oracle as well as for the Grover gate, the circuit construction uses n-qubit Controlled-X gates. These need to be decomposed to Toffolis using ancillas for the purpose of simulating for the thesis. Multi-qubit CNOT decomposition is described further in Section 4.3.3.

The research acknowledges the need for approximate matching for DNA as an application for quantum algorithms. The algorithm incorporates that by checking $O(\sqrt{N-M+1})$ uniformly random positions sampled from M (as $N \gg M$, it is over-sampled). For a pattern matching M' positions out of M, the state will be amplified with a ratio of M'/M times, directly correlating with the measurement probability of the position index. There can be multiple solutions whose cardinality might not be known a priori. The algorithm employs the randomising iterations of multiple runs as described in Section 3.3.2. However, it does not consider that the approach converges to the correct solution only on repeating the experiment multiple times (for gradually increasing δ). Thus, for a general pattern matching

case, it might not be suitable. The paper states (without proof) that, the algorithm's efficiency (with $P_{soln} \geq 1/4$) is easy to verify when the symbols in the pattern occur only once and the alphabet is rich. Neither of these is implied in the case of DNA sequences and the implication of this will be discussed in more depth in Chapter 4. The compile time is $O(ANlog^2(N))$ and the runtime of the algorithm is $O(Mlog^3(N) + N^{3/2}log^2(N)log(M))$. These shall be verified in the results for the specific case of the application for the thesis.

3.4.2. Approach 2: Quantum phone directory and Hamming distance

The next method is based on the quantum algorithm for bioinformatics as proposed by L.C.L. Hollenberg [58]. It closely resonates with the scope of this thesis as it considers quantum search algorithms in the context of protein sequence alignment in bioinformatics. The task is to find the location in the reference database of an exact or closest match to a query pattern; by finding the minimum Hamming distance between the sample sequence and the database sub-sequences. A block diagram of the algorithm is shown in Figure 3.8.

The amino-acid alphabet size of $A = 20$ is considered for the encoding into 5-qubits. The initial state is composed of two-qubit registers, the index and the pattern - similar in architecture to a random access machine (RAM) or a phone directory with name and number. Essentially, the set of patterns are sorted into an ordered list due to the second register of the database that tags the data. This is different from the previous method, as the real data is encoded in the states, while in the previous approach, the states represented positions in the reference, while the real data was only encoded as part of the Oracle function. The initial state can be described as,

$$|\psi_0\rangle = \frac{1}{\sqrt{N-M+1}} \sum_{i=0}^{N-M} \Big(|T_M(i)\rangle \otimes |i\rangle \Big)$$

where, $T_M(i)$ represents a sub-sequence of the reference T of length M starting at the position i. Thus the number of qubits in total is given by $q_{data} + q_{tag} = AM + \lceil log(N-M+1) \rceil$.

The next step in the algorithm is the most crucial contribution. *It evolve the data qubits to their Hamming distances with respect to the search pattern*. This operation can be done on the entire superposed state highlighting the power of quantum operators. Simply, a CNOT (with the query pattern as control) is done on the data qubits, which results in the Hamming distances. The black box nature of the Oracle function is now simplified (at the cost of increasing the qubit complexity with position information). For a perfect match, the Oracle just needs to mark the states with the value of 0, thus making it a fixed function with no dependence on either the reference or the search pattern. Once the state is amplified according to the modified Grover's algorithm (for an unknown number of solutions), the location of the sequence in the database can be determined by making a measurement on the entangled second register.

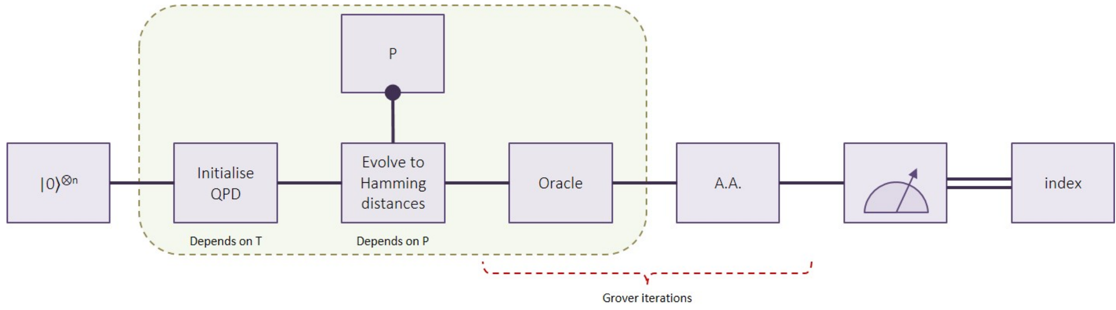


Figure 3.8: Algorithm anatomy

For approximate matching, the Oracle needs to be modified such that it finds the minimum value, instead of an exact 0. The article proposed to incrementally employ Oracles that marks incremental Hamming distances $(0, 1, 2, \cdots, d_{limit})$. For each distance, the modified Grover's algorithm is repeated till either an optimal solution is obtained (and the entire process is aborted), or the algorithm halts with a failure after maximum iterations $O(\sqrt{N})$.

3.4.3. Approach 3: Quantum associative memory

Finally, the idea of quantum associative memory for pattern recognition is explored. This method, proposed by D. Ventura and T. Martinez over a series of (in most part redundant) articles [60] [61] [62] [63] [59], is more well known and improved by further researchers.

Associative memory is a type of memory organisation that works very different from the random access memory (RAM) that are normally present in computing devices. Associative memories are also called content-addressable storage (CAS), where, instead of the index of the element to be retrieved (like in a RAM), a partial description of an element is passed as the input query. The element in the memory with the nearest match to the query is retrieved by the CAS. CAS is used in routers to find the destination address of a packet based on MAC addresses of the machines registered on the routing table. Building a CAS is costly (almost twice that of a DRAM of the same size) in digital electronics as it requires a parallel comparator. Associative memories are useful for modelling human memory storage and retrieval and is thus widely studied in the field of Artificial Neural Networks (ANN).

Similarly, the idea of quantum associative memory was developed under the umbrella of quantum neural networks. Intuitively, the entire parallel search operation is reduced to operations on a superposition of states (memories), resulting in an exponential increase in the capacity of the memory (or in the number of comparisons reduced to a constant time). However, this thesis will study the algorithm as a purely mechanical pattern matching algorithm, without going into the discussion of learning efficiency.

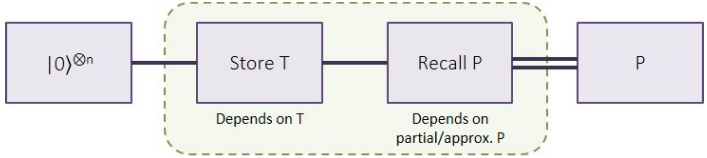


Figure 3.9: Algorithm anatomy

The algorithm consists of two major blocks, a pattern store and a pattern recall as shown in Figure 3.9. The pattern store starts from an all-zero state to encode the reference text string T. A set T_M of M length substring patterns ($T_M(i)$ where $i \in \{0 \cdots (N-M+1)\}$) are made from T, each starting from a consecutive index. The number of operations on 1-3 qubits for the initialisation step is polynomial in the length (M) and number ($N-M+1$) of patterns to be stored. A set of $2M+1$ qubits is required for storing the patterns, the first M qubits ($x_1 \cdots x_n$) is used to actually store the patterns (the neurons in a quantum associative memory) while the remaining (g,c) are ancillary qubits for bookkeeping and are restored to the 0 state after every storage iteration for a pattern in T_M. The detailed algorithm requires custom unitaries, as shown in Figure 3.10. It stores m binary pattern of length n with the function $f : z \rightarrow s$, where $z \in \{0,1\}^n$ and $s \in \{-1,1\}$. $\hat{F}_a$ is the Pauli-X gate on qubit a, $\hat{F}^0_{ab}$ is CNOT with $NOT(a)$ as control and $\hat{F}^1_{ab}$ is CNOT with a as control. $\hat{A}^{ab}$ is a matrix where the elements from row and column $4a+2b$ to $4a+2b+2$ is replaced with $\hat{F}$ in a 8×8 identity matrix. The set of conditional-Hadamard-like transform operators $\hat{S}^{s_p p}$, one for each associated pattern is used to store (memorise) the reference, are given by,

$$\hat{S}^{s_p p} = CR_y(2sin^{-1}(-s/\sqrt{p})) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \sqrt{\frac{p-1}{p}} & \frac{-s}{\sqrt{p}} \\ 0 & 0 & \frac{s}{\sqrt{p}} & \sqrt{\frac{p-1}{p}} \end{bmatrix}$$

An example of the initialisation process can be found in [62]. The paper provides an in-depth analysis of why this unitary evolution works and how it can be extended to quantum network based circuits.

M qubits can store a total of A^M possible patterns, while the number of stored patterns is much smaller, i.e. $N-M+1 \ll A^M$. Suppose, M' is the fraction of M qubits which are known from the query. For the recall phase, two different Oracles are constructed. The first Oracle, O_τ marks the states in the memory for which the M' fraction of qubits matches. The second Oracle, O_P *marks all the stored states in the memory*. This is not equivalent to marking (flipping the entire state by a global phase of π) the entire state vector, as due to the amplitude amplification operation, all A^M state might have non-zero amplitudes, but only the stored states are marked. The Oracles are applied on the initialised

1. Generate $|\tilde{f}\rangle = |x_1 \dots x_n, g_1 \dots g_{n-1}, c_1 c_2\rangle = |\bar{0}\rangle$
2. for $m \geq p \geq 1$
3. for $1 \leq j \leq n$
4. if $z_{pj} \neq z_{p+1j}$ (where $z_{m+1} = \{0\}^n$)
5. $\hat{F}^0_{c_2 x_j}|\tilde{f}\rangle$
6. $\hat{F}^0_{c_2 c_1}|\tilde{f}\rangle$
7. $\hat{S}^{s_p,p}_{c_1 c_2}|\tilde{f}\rangle$
8. $\hat{A}^{z_1 z_2}_{x_1 x_2 g_1}|\tilde{f}\rangle$
9. for $3 \leq k \leq n$
10. $\hat{A}^{z_k 1}_{x_k g_{k-2} g_{k-1}}|\tilde{f}\rangle$
11. $\hat{F}^1_{g_{n-1} c_1}|\tilde{f}\rangle$
12. for $n \geq k \geq 3$
13. $\hat{A}^{z_k 1}_{x_k g_{k-2} g_{k-1}}|\tilde{f}\rangle$
14. $\hat{A}^{z_1 z_2}_{x_1 x_2 g_1}|\tilde{f}\rangle$
15. $\hat{F}_{c_2}|\tilde{f}\rangle$

Figure 3.10: Initialising amplitude distribution in a quantum system [62]

state $|\psi_0\rangle$ in the sequence,

$$|\psi_j\rangle = \left(GO_\tau\right)_{\text{j times}} GO_P GO_\tau |\psi_0\rangle$$

where, G is the Grover gate (inversion about mean) and the optimal iteration, j_{opt} is $O(\sqrt{2^M})$

$N - M + 1$ is the number of stored patterns out of A^M possibilities of the qubit basis states. r_0 and r_1 represents the number of marked states corresponding to stored and non-stored patterns, respectively. Let,

$$a = \frac{2(N - M + 1 - 2r_1)}{A^M} \quad \text{and} \quad b = \frac{4(N - M + 1 + r_0)}{A^M}$$

The initial amplitude of the marked (k) and unmarked (l) states for stored and non-stored patterns (based on the subscripts 1 and 0 respectively) are given by,

$$k_0 = 4a - ab \quad \text{and} \quad k_1 = 4a - ab + 1$$

$$l_0 = 2a - ab \quad \text{and} \quad l_1 = 4a - ab - 1$$

This gives the mean k and l as,

$$\bar{k} = 4a - ab + \frac{r_1}{r_0 + r_1} \quad \text{and} \quad \bar{l} = -ab + \frac{2a(A^M + N + M - 1 - r_0 - 2r_1) - (N - M + 1 - r_1)}{A^M - r_0 - r_1}$$

The optimal number of iterations and maximum probability can now be calculated as,

$$j_{opt} = \frac{(2m+1)\frac{\pi}{2} - 2tan^{-1}\left(\frac{\bar{k}}{\bar{l}}\sqrt{\frac{(r_0 + r_1)}{N - r_0 - r_1}}\right)}{2cos^{-1}\left(1 - \frac{2(r_0 + r_1)}{A^M}\right)}$$

$$P_{max} = 1 - (A^M - N + M - 1 - r_0)|l_0 - \bar{l}|^2 - (N - M + 1 - r_1)|l_1 - \bar{l}|^2$$

Note that, if we set $r_0 = 0$ and $l_0 = \bar{l}$, these equations reduce to the j_{opt} and P_{max} calculated for the arbitrary initial distribution case.

The article also proposes an alternative non-unitary matrix evolution for the recall step. Measurement of a quantum system is not a unitary. Since the pattern recall mechanism requires the decoherence and collapse of the system at the end, an explicit use of non-unitaries might be beneficial for faster retrieval. The introduction of spurious states (non-zero amplitudes for non-memorised patterns) into the superposition during unitary evolutions can be avoided increasing the measurement probability.

The element of this non-unitary recall operator R (for the search query P) is given by,

$$R_{\phi\chi} = \begin{cases} 1, & \text{if } \phi \equiv \chi \text{ and } h(\phi, P_\phi) \geq 1 \\ -1, & \text{if } h(\phi, P_\phi) > h(\chi, P_\chi) \geq 1 \\ 0, & \text{otherwise} \end{cases}$$

where, $h(.,.)$ is the Hamming distance function over an alphabet $\Sigma' = \Sigma \cup \{?\}$, the wild-card character $?$ matching everything. This would probably come at the cost of added computing energy cost. Though the principle can be tested on a classical simulator using matrix evolution (like MATLAB), the translation of the non-unitary operations to a quantum-classical hybrid code is not understood. This approach of pattern recall is thus not pursued in the thesis. However, the idea of using two Oracles, one for marking the solution and another for marking all stored states, is used for the development of the final proposed algorithm in the thesis.

4

Implementations of quantum pattern matching algorithms

What I cannot create, I do not understand.
- Richard Feynman

In this chapter, the details of the implementation of the quantum algorithms are presented. Firstly, the choice of development tools and algorithmic utilities are introduced. Then, the major assumptions and their justifications for the algorithm development is highlighted. The implementations of quantum algorithms for pattern matching are developed. Finally, a new algorithm is designed for the specific requirement of genomic sub-sequence alignment.

4.1. Development tools

An introduction to the various development tools for the thesis is presented in this section. In-house tools developed in the department are preferred for the implementations. For additional support MATLAB is used.

4.1.1. OpenQL compiler framework

To enable quantum programmers to express complex algorithmic constructs, a high-level programming language is needed. This provides an abstraction for the low-level details of the computer architecture. Many different quantum programming languages and compiler frameworks are proposed by various research groups, such as Scaffold [64], LiQui|> [65] (later, Microsoft Quantum Development Kit using Q# [66]), ProjectQ [67], Quipper [68], Rigetti Forest [69]. The OpenQL framework is under development at QuTech, which allows hybrid quantum-classical coding in Python or C++. OpenQL is inspired by OpenCL and is based on heterogeneous programming model with the quantum device modelled as an accelerator to the general purpose computer (host processor).

The compiler bridges the semantic gap between the quantum algorithm and the quantum hardware. The high-level human-readable quantum algorithms are transformed into quantum operations, optimising them to produce executable machine code for the target quantum platform. Usually, a quantum compiler translates the program to some form of Quantum Assembly Language (QASM) that is a description of the quantum circuit for the particular purpose. The gates used for the decomposition are based on the supported gate-set of the underlying quantum computing device (simulator or processor). Optionally, the compiler optimises the circuit to reduce the length (latency) or width (number of qubits), and to use scheduling schemes (like As-Soon-As-Possible or As-Late-As-Possible) to produce a scheduled QASM where multiple consecutive operations can be performed in the same cycle if they are on different qubits. Additionally, the compiler also maps the quantum circuit to the topology of the physical quantum processor chip, satisfying its constraints, for example, nearest-neighbour interactions. The QASM based quantum instructions are then translated to executable-QASM (eQASM) which also includes the timing and physical qubit identifiers. It is then sent to the micro-architecture for execution on the quantum processor chip.

The OpenQL framework provides a compiler for compiling and optimising quantum code to produce the intermediate Common QASM [70] and the compiled eQASM for various target platforms (superconducting qubits, spin-qubits, NV-centers, etc.) The architecture of the compiler is shown in Figure 4.1. While the eQASM and microcode are platform-specific, the QASM is a hardware-agnostic circuit description. OpenQL compiler back-end envisions various features like decomposition (like Toffoli using H, T and CNOT), optimisation (under development), scheduling, mapping and interfacing with the QASM compiler. For this thesis, OpenQL is used mainly as a translator of high-level programming constructs to QASM. The back-end can be a simulator instead of a real platform. In this project, the Python wrappers on OpenQL is used for development. OpenQL is under active development. The programs in this thesis is compatible for the *Release Version 0.4.0* of OpenQL (commit $9f0e266bec6c66d93f95d850337f9dfbb864d379$ dated 19th May 2018).

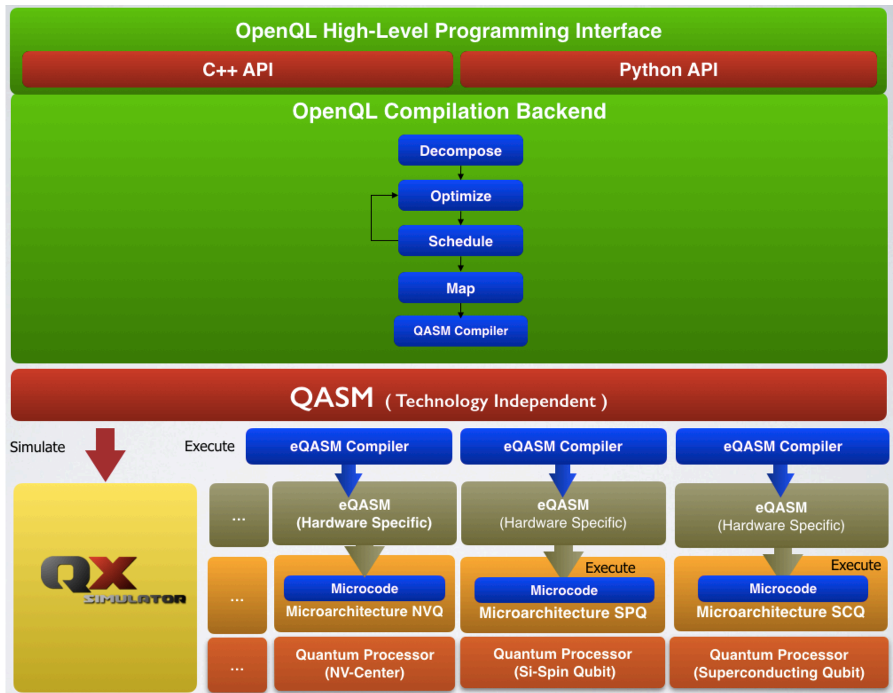


Figure 4.1: OpenQL high-level programming interface

The Qxelarator library allows execution of the compiled QASM on the QX binary and receives the measurement outcomes in the high-level OpenQL code. This encapsulates the quantum architecture (in this case, the simulator) and allows interleaving classical and quantum code blocks in a single program.

There are two main reasons for using OpenQL for the project, instead of directly coding in QASM.

- Firstly, the high-level constructs offer *code compression*. For example, to create the uniform superposition state at the start of Grover's algorithm, the Hadamard gate needs to be applied on all the data qubits. While the QASM lines of code are proportional to the number of gates (and thus the instance size of the problem), the OpenQL code can employ iteration to perform Hadamard operations on subsequent qubits, keeping the code size constant. This greatly increases the readability of the code, as OpenQL allows breaking down the code into logical blocks called kernels.
- Secondly, for an end-user application, beside the quantum accelerated part, there are other mathematical calculations that need to be performed. For example, in the previous chapter it was shown, an estimation of the number of iterations to reach the optimal solution probability for search algorithms can be pre-calculated. Theoretically, quantum computers being universal computing machines, it should be possible to do all computation on the quantum device itself. However, given the current state of technological advancement, it is not feasible to use quantum computers as stand-alone general purpose computing devices. These *auxiliary classical pre/post-processing* of the algorithm can be done in the native code running on the host processor.

4.1.2. QX quantum computer simulator

In order to experiment with algorithms in the absence of a large physical quantum computer, quantum computer simulators are used to verify their feasibility, correctness, scaling and predict their behaviour on a real quantum system. This is different from the term, quantum simulation, where one quantum system is simulated in another more accessible quantum system to study its properties. There are many available quantum simulators [71] - developed either as a teaching tool, for development of algorithms or using as an interface to quantum hardware. QX [72] is a universal quantum computer simulator that takes as input a specially designed quantum assembly language (QASM) and provides through aggressive optimisation, high simulation speeds for qubit state evolution.

The feasibility of quantum simulation is dependent on the number of qubits required for the circuit. The efficiency of simulators reduces exponentially with respect to the number of qubits because the number of states increases as 2^q. There are however other factors that also have an effect. For example, it is quite trivial to initialise a million qubits and only perform Pauli-X/Y/Z operations on them or simulate reversible versions of classical circuits. Basically, it is dependent on how sparse the state vector (or the density matrix) is. If the state is in a highly superposed/entangled state, manipulating qubits in the order of 50 starts to become quite challenging even in super-computing clusters. The currently available *qsim* servers in the department (with 28 HT cores, @ 2.00 GHz and 384GB memory) can simulate ≈35 qubits if the states and operations are non-sparse. There are fields of research that are trying to make a more efficient simulation with tensor networks. Alternatively, it is argued that the quantum speedup advantage can be reasoned in terms of how much of the quantum phenomena of superposition and entanglement is harnessed [73]. Thus, 50 qubits is widely regarded as the supremacy limit for quantum computation i.e. quantum computation will be able to calculate something that is classically intractable.

It needs to be noted that, the 50 qubit limit is not needed to exhibit useful quantum phenomena. A simple example for this is, putting a qubit to a $|+\rangle$ state and measuring it on the $\{|0\rangle, |1\rangle\}$ computational Z-basis which would simulate an exact unbiased coin, a perfect random number generator, which no classical algorithm can. Such strategies are the basis of research in quantum communications, whose application can typically be realised with fewer qubits.

An important distinction is to be made here regarding physical and logical qubits. Many quantum computing ventures are on the verge of reaching the 50 limit, however, 50 physical qubits are highly error-prone and cannot be addressed individually to perform useful computation. Thus, multiple qubits are encoded using error-correcting codes (ECC) to represent a single logical qubit. This process is a bit different from classical redundancy due to the no-cloning principle. Discussion on QEC is outside the scope of this thesis. However, it needs to be stressed that, simulators might still be very useful to study quantum algorithms as long as 50 or more logical qubits with high fidelity do not become a reality.

In simulators, the internal state vector can be obtained in its totality. Thus, for simulating the algorithm, there is no need to repeat the execution multiple times and average the measurement probability. It is impossible to reconstruct the exact state vector from measurements in a real quantum processor. Only the probabilities (squared amplitudes) can be estimated with increasing degrees of resolution with multiple measurements. The exact complex amplitude remains hidden, which is useful in the algorithm design stage to understand how the quantum system evolves. QX allows this feature with the *display* directive, that prints out the state vector in verbose at the point in the circuit where the directive is placed. This is extremely handy for debugging scaled down versions of quantum algorithms and might outlive the supremacy limits.

Using a simulator also implies a few fine-prints. Firstly, for an arbitrary algorithm, it limits the problem size what can be simulated on the available computer resource. This is not a major problem from the algorithm design aspect as it is easy to reason out the extension of the algorithm for larger program size once it is tested for a smaller size. However, it might not be directly emulated for the real World problem size leaving room for speculation about the practical efficiency on a real quantum processor. Many quantum simulators allow introducing noise-models. The QX simulator has an option to set a symmetric depolarising channel with parameterised error probability. These models are constantly being updated to more realistic error models as more data from practical experiments are being available. Thus, an algorithm on the simulator might not execute exactly the same on a real processor if the exact environmental model is not considered.

4.1.3. Integrated development environment

In order to merge OpenQL and QX Simulator into an integrated development environment (IDE) for quantum algorithm research, the *Quantum Innovation Environment (QuInE) is developed as part of the thesis*. It integrates algorithm specification using OpenQL (Python), QASM based coding and Circuit drawing tools for aiding various phases of an algorithm development cycle. These methods are integrated for inter-conversion and can be executed to generate result plots. QuInE is envisioned to be an introductory learning tool for intuition development, at the same time, featuring capabilities to support large-scale quantum programming. It is still in pre-alpha release and this thesis has been instrumental in the understanding of the requirements for a quantum programming tool for high-level algorithms. It is coded in PyQt, and is influenced by Quirk [74], QMechanic [75] and Cadence OrCAD Capture [76]. A screenshot of QuInE is shown in Figure 4.2.

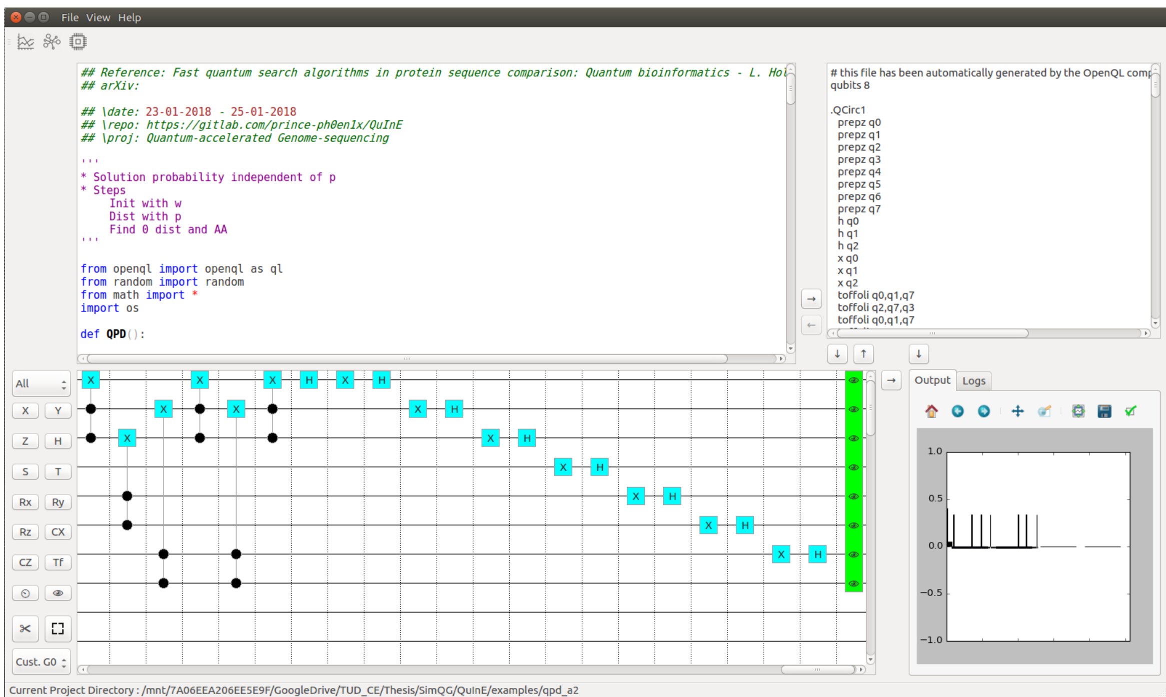


Figure 4.2: Quantum Innovation Environment

4.1.4. Verification and testing

The circuit model of quantum computation is used for the algorithm design as it can directly be translated to executable instructions for the underlying quantum processor. However, some of the work on which the thesis builds further is based on unitary evolution. Unitary evolution is easy to implement as matrix operations in platforms like MATLAB (or its open source alternative, Octave).

The testing philosophy followed is thus based on testing the correctness of the QASM execution of the algorithm in QX, by comparing the results with the MATLAB code's final state vector by unitary evolution. To obtain the result from QX, either the superposed internal state vector can be directly obtained from the execution before measurement (using the *display* directive) or multiple runs are accumulated for a state tomographic estimation.

MATLAB is also used in the thesis to develop additional support libraries as discussed further in Section 4.3 for arbitrary unitary decomposition. Overall, dependence on external tools is reduced over the developmental period so that the entire tool-chain can be coded in Python. External tools are used where specific available packages (e.g. singular value decomposition in MATLAB) endorses easy implementation.

4.2. Assumptions

In this section, the different assumptions are explicitly stated for different parts of the system. It is hard to justify all results of quantum algorithms given so many speculations [77]. Though a hardware agnostic approach is taken for the implementations, the results attempt to quantify the metrics of:

- Scaling of number qubits with respect to problem instance size
- Scaling of number of gate operations with respect to problem instance size

It is useful to compare these algorithms to assess the feasibility and the requirement of the underlying quantum computing stack layers.

4.2.1. Number of qubits

The QX simulator allows declaration of an arbitrarily large number of qubits. The bound is set by the underlying hardware, its processing and memory capabilities for the type of state transitions that need to be performed for the algorithm. Practically, simulating Grover's algorithm with $>\sim 25$ qubits starts to get taxing on a laptop (Core i7 with 8 GB RAM on Ubuntu 16.04). Only one of the algorithms in the thesis required more than 25 qubits for the proof-of-concept simulation. Testing on higher qubit version would not reveal anything more from the algorithm design point of view. In a quantum hardware, the simulation is expected to undergo exponential speedup. However, unless there is some drastic breakthrough in hardware technology (similar to vacuum tube to transistors), qubit number will remain one of the most expensive parameter that needs to be optimised. Keeping this in mind, constructs that use lesser number of (ancilla) qubits are implemented.

4.2.2. Noise models

The other very important parameters that are currently restraining the quantum revolution are environmental noise and gate fidelity. Qubits decohere, losing their pristine state to entangle with the environment, causing errors. The QX supports the depolarising model, where random Pauli errors can occur at any part of the circuit. Errors can be mitigated via various means, e.g. Pauli frames, Quantum Error Correction (QEC), randomised compiling, etc. This is a separate field of research which allows the algorithm design to be agnostic to the errors. QEC is mostly implemented by distributing the information among a set of qubits. Popular codes are Shor code (using 9 qubits), Steane code (using 7 qubits), Surface-7 code, Surface-17 code and Surface-49.

Simulating noise models without QEC is not useful as realistic error rates would not allow more than a handful of gates [78] before the state completely decoheres. On the other hand, using the simplest of the correcting codes (distance 3 or more), will multiply the physical qubit number by a large fraction, preventing simulation of even the proof-of-concept problem size. Thus all algorithms simulated for the thesis assumes no noise scenario.

4.2.3. Number and type of gates

The noise of the system directly influences [79] the number of gates that can be executed. In a no noise scenario, there is no limit on the number of gates (the QASM program size). However, the time complexity of the quantum algorithm is also based on the number of gates executed. Thus to show a speedup of quantum algorithms, the number of elementary quantum operations should be asymptotically less than the number of classical primitives for the algorithm.

Such a comparison, however, assumes equal clock speeds for classical and quantum. Currently, there is a huge gap between the operation throughput of the classical and quantum processors. Also, not all quantum gates take equal time. The time taken by a logical gate also depends on the encoding scheme. For example, some gates are transversal in a particular coding scheme, which makes them easier to parallelise. However, some non-Clifford gates (like T gates) require a non-deterministic procedure in topological codes, called state distillation. Since there are no fixed numbers on the duration of various gates in a system with 1000s of physical qubits, the weights are left as variables.

The primitive gate-set is dictated by the quantum processor. As long as a universal gate-set is used, the compiler can translate an algorithm to the specifically supported gate-set. For the thesis, the allowed gate-set of QX is chosen. This includes the quantum gates for Hadamard, Pauli-X/Y/Z, arbitrary rotation in X/Y/Z, Phase gate, T, T dagger, CNOT, Toffoli, SWAP, controlled rotation, binary controlled Pauli-X/Z and PrepZ. Gate complexity is assessed individually in terms of how each of these gates scales with the problem size.

4.2.4. Qubit topology

In the thesis and in QX simulator, all qubits are connected to each other. This is not always feasible while designing the quantum chip (typically on a plane). This limited connectivity restricts the allowed sets and directivity of multi-qubit gates (3 or more qubit gates are always broken down to 1 and 2 qubit gates) e.g. the IBM Quantum Experience has very limited connectivity, and some qubits allow only CNOTs in one direction (so it is inverted by dressing it with H gates). For the future many-qubit regime, a topological surface of connected ninja-star surface coded logical qubits is being considered. Such 2-D graphs will be restricted to nearest neighbour connectivity. Thus extra operations (like SWAP gates) needs to be added to bring two logical qubits beside each other before an operation can be performed. The initial placement of the program's qubits has an influence on the efficiency of the routing for small circuits. This is the research domain of qubit placement, mapping and routing. In future, these features will be available in the OpenQL compiler but for the most part will be abstracted from the algorithm designer's point of view.

In summary, the assumptions and the justifications for the thesis are tabulated in Figure 4.3.

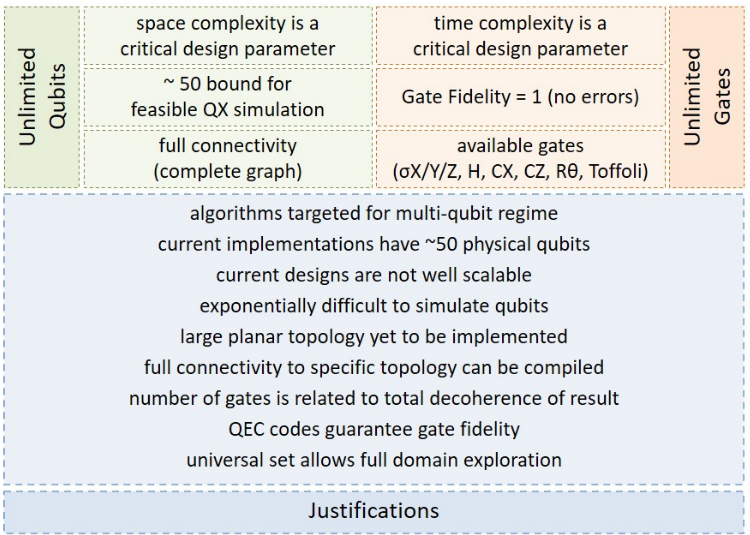


Figure 4.3: Assumptions and justifications for the thesis

4.3. Algorithmic utilities

The quantum pattern matching algorithms require a library of common functionality. These utilities are discussed in this section.

4.3.1. Reference patterns

In order to test the algorithms implemented for the thesis, test patterns need to be generated. The tests are carried out for either of the 3 types of reference patterns: real data segment, random data segment and minimal super-string.

- For real data segment, a segment of the HBB (hemoglobin subunit beta) is chosen from Chromosome 11 (region p15.4) of Homo sapiens from the NCBI database [80] (Annotation Release 109). The start codon of the 147+1 length amino-acid sequence is at location 51 of the 626 base pair sequence Genetic mutations of HBB are responsible for diseases like sickle cell anaemia and beta thalassemia. In sickle cell anaemia, hemoglobin S replaces both beta-globin sub-units in hemoglobin. This can be seen in the range 51 to 71, where the DNA sequence varies at 2 places:
  HBB Gene sub-sequence : $ATG.GTG.CAT.CTG.ACT.CCT.GAG$
  HBS Gene sub-sequence : $ATG.GTG.CA\underline{C}.CTG.ACT.CCT.G\underline{T}G$
  This results in a single amino-acid change between these due to the available redundancy in the codon.
  HBB Amino acid sub-sequence : $Start.Val.His.Leu.Thr.Pro.Glu$
  HBS Amino acid sub-sequence : $Start.Val.His.Leu.Thr.Pro.\underline{Val}$

This sequence emphasises the criticality and the correctness demands on the algorithm, as a 1bp change in 3×10^8bp can be erroneously diagnosed as a fatal disease.

- The second method is generating random data sequence of required size. The distribution of Chargaff's Rules discussed in Section 2.4.1 is followed for generating the random sequence. Since all the 4 nucleotide bases occur in almost equal proportion, a simple random selection preserves this property of real DNA. Only DNA sequence is generated and its implication on amino-acid coding is ignored.
- The final method is for testing exhaustively the algorithm's performance on all combination of the pattern string. *This is an algorithm developed for the thesis.* An ordered set of an alphabet is considered, e.g. $\Sigma = \{A, C, G, T\}$ of size $|\Sigma| = 4$. Then the length of the search pattern is chosen. For the example here, let it be $M = 3$. Thus, the number of possible permutations for all 3-character strings from the alphabet set is 4^3.

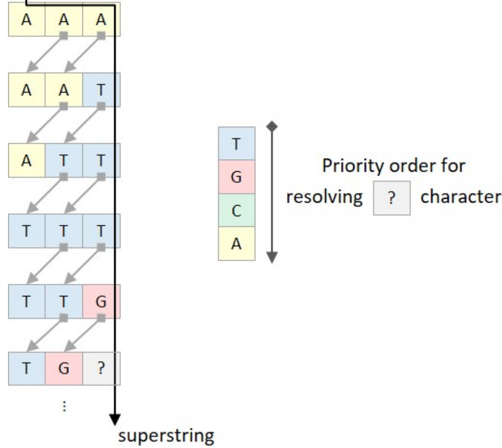


Figure 4.4: Minimal length super-string algorithm example for $|\Sigma| = 4$ and $M = 3$

The algorithm finds the shortest string that contains all these permutations as sub-strings (and occurs only once). The length of this super-string is $|\Sigma|^M + M - 1 = 66$ for our case. The super-string starts with the first character repeated through the entire pattern length, i.e. AAA. For the next consecutive shifted pattern the first $(M - 1)$ characters are fixed, i.e. it has to be of the form, $AA*$. For choosing the $*$, characters are tried in the inverse ordering of the alphabet, i.e. AAT or else AAG or else AAC, as shown in Figure 4.4. Since T has not been considered before as prefix of AA, T is added to the partial super-string; making it $AAAT$. Continuing through this algorithm, the final super-string for the above case becomes,

$AAATTTGTTCTTATGGTGCTGATCGTCCTCATAGTACTAAGGGCGGAGCCGCAGACGAACCCACAA$

Formally, the algorithm can be reasoned as a concatenation of Gray code extended for larger alphabet set.

4.3.2. Unitary decomposition

For the most general algorithms, arbitrary unitary evolution is required. Some of these might have special forms, but it is highly advantageous if a decomposition for an arbitrary unitary is available that translates an input matrix to a sequence of gates which can be directly used as a module in OpenQL. This is in the development roadmap of OpenQL compiler, however, to cater to the immediate need of this thesis, *a MATLAB script is developed with generates the OpenQL kernel code for the decomposition of any arbitrary unitary matrix to a sequence of quantum gate operations.*

The Quantum Shannon Decomposition (QSD) method [81] is chosen for this purpose. The gate set of CNOT and arbitrary Rotations are used in this algorithm, which is possible to simulate in QX. The QSD method is chosen as it offers a decomposition in terms of less gate cost if the underlying architecture allows composition of arbitrary analog wave-forming for controlling microwave pulses to the quantum chip. The QSD algorithm is explained here.

The first step is a Cosine-Sine Decomposition (CSD) which splits an even-dimensional unitary matrix

$U \in \mathbb{C}^{n\times n}$ in smaller $\mathbb{C}^{(n/2)\times(n/2)}$ sized unitaries M_0^0, M_0^1, M_1^0, M_1^1, and real diagonal matrices C, S. M_i sets form quantum multiplexers, while C and S form a multi-controlled R_y gate satisfying, $C^2+S^2 = I^{\otimes l/2}$. The CSD decomposition, in turn, involves Singular Value Decomposition (SVD) routines. Detailed discussion on CSD can be found in [82]. For this thesis, it is used as a MATLAB library from the Qubiter toolset available online at http://www.ar-tiste.com/m-fun/csd_qc.m.

$$U|\psi\rangle = \begin{bmatrix} a & b \\ c & d \end{bmatrix}|\psi\rangle = \begin{bmatrix} M_1^0 & 0 \\ 0 & M_1^1 \end{bmatrix}\begin{bmatrix} C & S \\ -S & C \end{bmatrix}\begin{bmatrix} M_0^0 & 0 \\ 0 & M_0^1 \end{bmatrix}|\psi\rangle$$

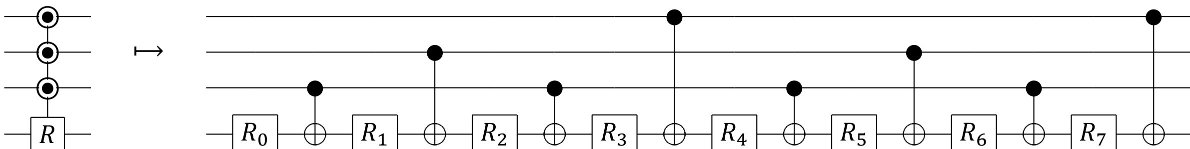
The next step is decomposing the multiplexors M_i, where, when the control bit is 0, M_i^0 is applied, and for 1, M_i^1 is applied. For superposed control qubit states, a proportional mixture of both operations are applied. It is decomposed using the relation $M_i = (I^{\otimes l/2} \otimes V_i)(D_i \otimes D_i^\dagger)(I^{\otimes l/2} \otimes W_i)$, where V and W are unitaries and D is a unitary diagonal matrix (index i is omitted for readability). To obtain D and V, the equation, $M^0 M^{1\dagger} = VDWW^\dagger D^\dagger V^\dagger = VD^2V^\dagger$ is diagonalised using MATLAB's eigenvalue function. Then, W is obtained as $DV^\dagger M^1$.

$$\begin{bmatrix} M_i^0 & 0 \\ 0 & M_i^1 \end{bmatrix} = \begin{bmatrix} V & 0 \\ 0 & V \end{bmatrix}\begin{bmatrix} D & 0 \\ 0 & D^\dagger \end{bmatrix}\begin{bmatrix} W & 0 \\ 0 & W \end{bmatrix}$$

The final step is decomposing the multi-control rotation gates (R_y and R_z) introduced in the previous two steps. Let the set of rotation angles be α_j, where, $j \in [0\cdots(n/2-1)]$. For R_y, these angles can be obtained from the sine and cosine inverses of the diagonal elements in S and C, respectively, multiplied by 2 (due to the $\theta/2$ in the terms in $R_y(\theta)$ operator). For R_z, the angles are obtained by taking natural logarithm of the diagonals in D, multiplied by $-2i$ (inverting the $e^{-i\theta/2}$ in $R_z(\theta)$. The angles are transformed as [83]:

$$\begin{bmatrix} \beta_0 \\ \vdots \\ \beta_{(n/2-1)} \end{bmatrix} = A_{n/2}\begin{bmatrix} \alpha_0 \\ \vdots \\ \alpha_{(n/2-1)} \end{bmatrix}$$

The elements of the inverse of the transformation matrix, i.e. $A_{n/2}^{-1}$ is given by $A_{ij}^{-1} = -1^{b_{k-1}\cdot g_{j-1}}$, where, b_i and g_i are the binary and Gray codes of i respectively. The rotation matrices $R_{y/z}(\beta_i)$ are followed by CNOT gates, with a target on the same qubit, and the control defined by the Gray code transition bit from g_i to g_{i+1}. An example of the 3-bit case is shown below.



With these equivalence relations applied recursively, an arbitrary unitary is decomposed to arbitrary single-qubit rotations and CNOTs.

4.3.3. Multi-controlled NOT

In many parts of the circuit, a multi-qubit controlled NOT is used. It generalises X, CNOT and Toffoli gates to more qubits. Since the QX simulator only supports till Toffoli gates, controls more than 2 qubits

needs to be broken down to simpler NOT gates. Note, for real quantum hardware, the Toffoli gate is normally decomposed with CNOT, H and T gates. Of course, this decomposition itself can be handled by the arbitrary unitary decomposition presented in the previous section without any extra ancilla bits. However, such a decomposition is costly in the number of gates as the number of CNOTs are worst case bounded by $O(4^n)$ where n is the number of control qubits. A *simpler decomposition* is possible for this special case.

Given an ancilla (extra, work qubit), it is possible to break down a multi-controlled CNOT with just lower dimensional NOTs. The reason why such a decomposition is not possible without ancilla is attributed to the controlled-NOT operation having odd parity (just one single state causes flipping target bit) when all the rest of the qubits are in the control qubit set. A lower dimensional NOT (with fewer controls) will be of even parity as states in the order of 2^y will flip the target (where y is the number of qubits in the circuit that are not in the control state). Chaining even operations cannot create an odd operation [84] making it impossible to reduce an all-wires-touched controlled-NOT operation into smaller operations.

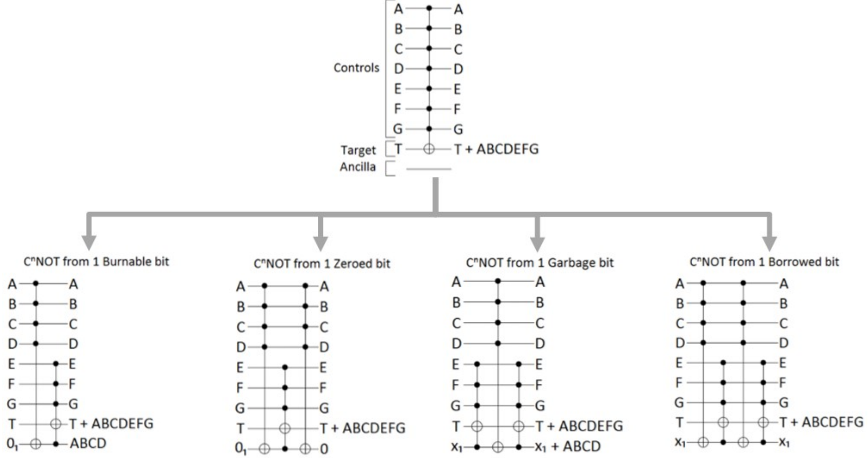


Figure 4.5: Ancilla types [84]

Ancilla qubits can be used in 4 different ways:

- Burnable: Initially in the $|0\rangle$ state but with no restrictions on state after the calculation. They can be thought of as a small amount of negative entropy that can be consumed to perform some irreversible computation.
- Zeroed: Initially in the $|0\rangle$ state and must also be returned to the $|0\rangle$ state after the computation. Circuits are a constant factor larger than those with burnable bits because of the taxing uncompute operation.
- Garbage: Could be in any state initially and more garbage can be added to the state (no need to restore the initial value). The logic is based around toggle detection.
- Borrowed: Could be in any state initially but must be restored to that same state afterwards. They have the downsides of both zeroed and garbage ancilla. The major advantage of borrowed qubits is that they are much easier to find because one part of the circuit can borrow qubit from another part, as long as all the qubits are not in use. Thus for this case, one single ancilla qubit will suffice for breaking down a controlled operation of any dimensionality.

The most common in literature is the zeroed ancilla. It takes $n-2$ number of ancilla qubits for implementing an n-qubit controlled NOT. The ladder structure for computing and uncomputing (returning the ancilla back to the zero state) is shown in Figure 4.6.

The borrowed qubit compresses the qubit complexity of the circuit by a large amount. For most of the algorithms in the thesis, the Grover Gate requires an inversion about the mean of all the qubit state. Thus, an extra ancilla is required. Also, since a no-noise environment is used, the approximately $4\times$ increase in gate complexity is not a big concern (constant factor). This is coded as a function in OpenQL, that takes a kernel object k, a set of control qubit identifiers c, a target qubit identifier t and a borrowed ancilla qubit identifier b. It recursively breaks down the multi-controlled-CNOT into simpler operations (finally to CNOT and Tofolli).

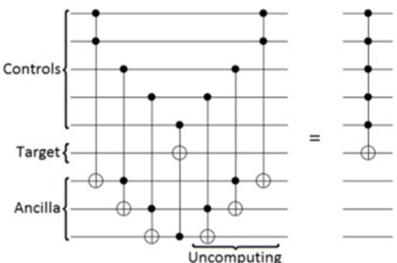


Figure 4.6: Multi-controlled CNOT decomposition using zeroed ancilla

```
# Multi-control CNOT with borrowed qubits (recursive implementation)
def nCXb(k,c,t,b):
    nc = len(c)
    if nc == 1:
        k.gate("cnot",c[0],t)
    elif nc == 2:
        k.toffoli(c[0],c[1],t)
    else:
        nch = ceil(nc/2)
        c1 = c[:nch]
        c2 = c[nch:]
        c2.append(b)
        nCXb(k,c1,b,nch+1)
        nCXb(k,c2,t,nch-1)
        nCXb(k,c1,b,nch+1)
        nCXb(k,c2,t,nch-1)
    return
```

4.3.4. Oracle for Boolean functions

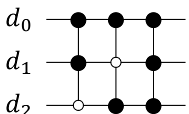
The ancilla-free Oracle encoding uses multi-control Phase gates for each true index of the Boolean function. Let the Boolean function be $B = \{b_0, b_1, \cdots, b_{D-1}\}$, where $D = 2^d$ and $b_j \in \{0,1\}$, d being the number of qubits to the Oracle function (the dimension of the basis state). The elements of the Unitary matrix for the Oracle O is given by,

$$O_{jk} = \begin{cases} 0, & \text{if } j \neq k \\ e^{\pi i j}, & \text{if } j = k \text{ where } i = \sqrt{-1} \end{cases}$$

Thus, it is a diagonal matrix with -1 for marked elements, 1 otherwise. The method to encode this general Oracle is to take the set of indices where $b_j = 1$. Let this set be J. For each $j \in J$, encode j in d binary digits, to form the set J_d with elements j_d. A controlled Phase gate is added for each j_d, over all d qubits, with inverted control at positions where the encoding of j_d equals 0. For example, for the Boolean function $B = \{0,0,0,1,0,1,0,1\}$ over $d = 3$ qubits, the unitary is,

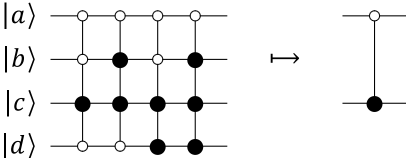
$$O = \begin{bmatrix} +1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & +1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

Thus $J = \{3,5,7\}$, which when encoded in $d = 3$ bits gives $J_{d_2 d_1 d_0} = \{011, 101, 111\}$. The circuit thus becomes,



The gate complexity of this algorithm can be reduced for some cases by constructing the Min-

imal Sum-of-Product (MSOP) expression of the Boolean function. Techniques to minimise Boolean functions like Karnaugh map, Quine–McCluskey algorithm or other heuristics can be used to find the prime implicants. For example, the Boolean function on 4 variables $B(d,c,b,a) = \{4,6,12,14\} = \{0100,0110,1100,1110\} = \overline{d}c\overline{ba} + \overline{d}cb\overline{a} + dc\overline{ba} + dcb\overline{a}$ can be reduced to the MSOP expression, $B(d,c,b,a) = c\overline{a}$. Thus the equivalent circuit can now be used,



The problem of general Boolean function minimisation itself is NP-hard thus the run time of these routines grow exponentially with the number of variables. However, if the Oracle function is a constant this minimisation can be optimised at compile time (or even as a pre-processing step), for the application. For example, if the function depends only on the reference human genome, which is a given constant for the application, the Boolean function can be decomposed once and then hard-coded in the quantum circuit.

4.4. Quantum pattern matching

In this section, the OpenQL Python implementations of the algorithms described in Section 3.4 and their modifications are presented. Every algorithm is first implemented for a binary alphabet and then extended to an alphabet size of 4 for encoding DNA sequence. Here, the DNA pattern matching is explained directly.

OpenQL divides the program into multiple kernels. The kernels can be added multiple times to a program. Finally, the program is executed. The general flow of an OpenQL quantum algorithm for this thesis is given in Figure 4.7.

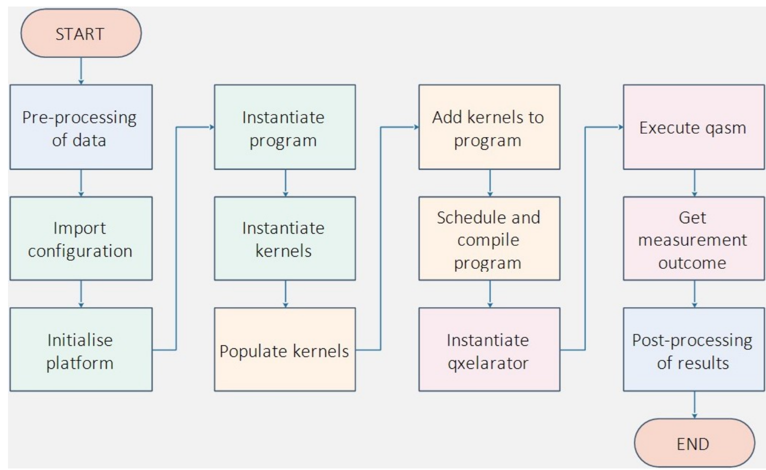


Figure 4.7: OpenQL program flow

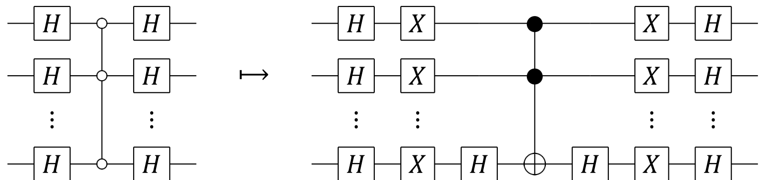
4.4.1. Approach 1: Conditional Oracle call

The code for the quantum pattern matching algorithm is listed in Appendix-1. The OpenQL and Qxelarator libraries import the necessary functionality required for programming and simulating over the Python3 language platform. The 'randStr' function generates a random decimal coded string over a given alphabet size and is used to generate the reference text of a given size. The alphabet, reference string and the search pattern is then defined. For testing the algorithm, the search pattern is chosen to be a sub-string from the random reference string at a predefined index.

The program is initialised on the simulator platform with the estimated number of qubits. It is divided into four kernels - initialisation, Oracle, Grover gate and measurement. If the internal state vector is accessed, the measurement kernel should not be called as it will collapse the state to the

measurement basis. Then, based on the algorithm outlined in Section 3.4.1, the kernels are assembled into the program. A total of $O(\sqrt{N-M+1})$ iterations are performed, each being a random index in the search pattern (N is the length of the reference text, M is the length of the search pattern). The final measurement probabilities are plotted.

Kernel 1 is essentially the initialisation method as described in the original article. Kernel 2 implements ancilla-free Boolean Oracle as described in Section 4.3.4. Kernel 3 is the Grover gate over multiple qubits. It is simplified to a multi-controlled CNOT before being further decomposed.



Kernel 4 measures the first set of indices. The *nCX* function implements a recursive multi-controlled CNOT using borrowed ancilla as described in Section 4.3.3.

The qubit complexity of the implementation is,

$$M\lceil log_2(N)\rceil + 1$$

Thus for the typical numbers of the human genome and Illumina reads, $N = 3 \times 10^8$ and $M = 50$, the number of qubits required is 1451 (which is obviously not a near-term application number).

The gate complexity is non-trivial to calculate exactly as it depends on the universal gate set. Here, the gate set used for this algorithm consists of $\{H, C_cX\}$, where $c = 0$ is the X-gate, $c = 1$ is the CNOT gate, $c = 2$ is the Toffoli gate, and so on.

Let $s = \lceil log_2(N)\rceil$. The initialisation kernel is first decomposed. Firstly, s Hadamard gates are used on the first set of s qubits. Then, the next set of gates is repeated for each set of 2 consecutive qubits, thus $(M-1)$ times. In this set, first, there is a train of s C_1X gates. Then, $s(s+1)/2$ multi-controlled-NOT gates, of which, there are $1C_1X, 2C_2X, 3C_3X \cdots sC_sX$. One control of these is inverted, which requires dressing with X-gates. The total X-gate for each set is thus $2 * s(s+1)/2$. Thus the total initialisation circuit is $sH + (M-1)\{s(s+1)C_0X + sC_1X + 1C_1X + 2C_2X + 3C_3X + \cdots + sC_sX\}$.

The Chargaff's rules are used to estimate the ratio of elements that are marked by the Oracle as $1/A = 0.25$ for the DNA alphabet. The Oracle is applied $\lfloor\sqrt{N-M+1}\rfloor$ times. The Oracle is applied over s qubits, thus over 2^s basis states of which $1/4$ are marked. Thus, there are 2^{s-2} CPhase gates (each over s qubits), with half of their controls as inverted. The total number of inverted controls are thus $s2^{s-2}/2$. The X-gates for dressing on both sides sum to $s2^{s-2}$. Each CPhase can be converted to a CNOT with 2 additional Hadamards. Thus the total gates for the algorithm used in the Oracle is $(\lfloor\sqrt{N-M+1}\rfloor)\{2^{s-1}H + s2^{s-2}C_0X + 2^{s-2}C_sX\}$.

Finally the Grover gate on sM qubits is decomposed to CNOTs as $(\lfloor\sqrt{N-M+1}\rfloor)\{(2sM+2)H + 2sMC_0X + C_{sM}X\}$. The total gate complexity is,

$$sH + (M-1)\{s(s+1)C_0X + sC_1X + 1C_1X + 2C_2X + \cdots + sC_sX\}+$$

$$(\lfloor\sqrt{N-M+1}\rfloor)\{(2^{s-1} + 2sM + 2)H + (2^{s-2}s + 2sM)C_0X + 2^{s-2}C_sX + C_{sM}X\}$$

Note, all C_cX, when $c > 2$ needs to be decomposed further using $\Theta(c)$ Toffoli gates by the construction given in Section 4.3.3.

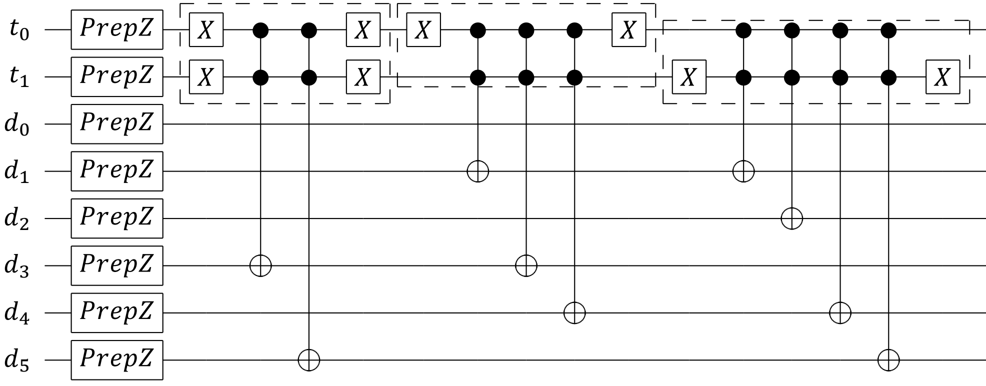
An important observation for this algorithm is that both the number of qubits and the number of gates does not depend on the size of the alphabet. However, since Chargaff's rule is applied, the complexity of the Oracle is inversely proportional to the alphabet size. The intuition behind a rich alphabet is, the Oracle will typically mark only a few states per call, making its circuit construction simpler. Thus, the algorithm implementation is smaller for the DNA alphabet set compared to binary strings.

4.4.2. Approach 2: Quantum phone directory and Hamming distance

The code for the quantum pattern matching algorithm is given in Appendix-2. The OpenQL and Qxelarator libraries and the 'randStr' function are identical to the previous code. The search pattern is chosen to be a sub-string from the reference string at a predefined index. For this algorithm, since a full space of tags (of the form 2^{qt})needs to be initialised, a spurious data for tags not in the range of valid

positions $(N-M+1)$ is required. This dummy data is chosen to be all-zeros. The program is initialised on the simulator platform with the estimated number of qubits. It is divided into five kernels - initialisation, Hamming evolution, Oracle, Grover gate and measurement. The Qxelarator method to access the internal state vector is useD to aggregate the resultant solution state probabilities. The measurement-based technique is also an alternative, which is more realistic for a quantum algorithm simulation, but requires more time and gives less accuracy for assessing the effectiveness of the algorithm. Based on the algorithm outlined in Section 3.4.2, the kernels are assembled into the program. The final stave vector is plotted.

Kernel 1 initialises the quantum phone directory. An equal superposition of tag states is generated. For each tag state, the corresponding substring of the reference at that tag position of length M is encoded in the data qubits using CNOT gates. For example, to encode the reference DNA string of $ACCGT$ where the search pattern is of length $M = 3$, the reference is split into 3 sub-strings with corresponding tags $\{0-ACC, 1-CCG, 2-CGT, 3-AAA\}$. Note, since the total number of tags must be a power of 2, the dummy data is the all-zero string (or in DNA string, all A pattern), but since there is no control qubits, no gate is required. The tag and data is then converted to binary encoding as $\{00-000101, 01-010110, 10-011011, 11-000000\}$. This is encoded using the following gates,



It is observed that this can be optimised if the data encoding follows a gray code, but is not implemented here. Such optimisation will be part of the compiler in the future.

Kernel 2 transforms the initialised data state to their Hamming distance. This is done by inverting the bits of the pattern with 1 as its encoding. E.g. for the pattern $CGT = 011011$, the qubits d_1, d_2, d_4, d_5 are inverted.

Kernel 3 implements ancilla-free Boolean Oracle as described in Section 4.3.4 that marks the all zero Hamming distance in the data qubits.

Kernel 4 is the Grover gate over the entire tag and data qubits space. It is simplified to a multi-controlled-CNOT before being further decomposed. Kernel 5 measures the tag qubits. The *nCX* function implements a recursive multi-controlled CNOT using borrowed ancilla as described in Section 4.3.3.

The qubit complexity of the implementation from data, tag and ancilla qubits is,

$$\lceil log_2(A) \rceil M + \lceil log_2(N-M) \rceil + 1$$

Typical estimates for the DNA alphabet, the human genome and Illumina reads $A = 4$, $N = 3 \times 10^8$ and $M = 50$, yield the number of qubits required is 130 qubits, which is an order of magnitude lesser than the previous approach.

The gate set $\{H, C_cX\}$ is also used for this algorithm. Let $q_d = \lceil log_2(A) \rceil M$ and $q_t = \lceil log_2(N-M) \rceil$. The initialisation kernel is first decomposed. First, q_t Hadamard gates are used on the tag qubits to create a superposition of solution states. Then, conditioned on each tag, the corresponding shifted sub-string of the reference is encoded. The binary encoding of the tag requires half the controls as inverted, requiring X-gate dressing totalling to $q_t 2^{q_t}$. Again using Chargaff's rule, the DNA nucleotides are distributed approximately $1/4$ in each sub-string, requiring $q_d/2$ targets for each tag encoded sub-string. Thus the total initialisation requires $q_t H + q_t 2^{q_t} C_0 X + q_t q_d / 2 C_{q_t} X$.

The search pattern is also a DNA string following Chargaff's rule, requiring the Hamming Oracle to flip half the data bits using $q_d/2 C_0 X$.

The Oracle and the amplification are repeated just once. The Oracle needs to mark the all-zero state and is thus the phase gate on all the data qubits with inverted controls. This requires $2H + 2q_d C_0 X + C_{q_d - 1} X$.

Finally, the amplification is applied to the data and tag qubits together. It requires, $\{2(q_d + q_t) + 2\}H + 2(q_d + q_t)C_0X + C_{q_d+q_t-1}X$. The total number of gates for the algorithm is thus,

$$q_tH + (q_t2^{q_t} + q_d/2)C_0X + q_tq_d/2C_{q_t}X+$$
$$\{2(q_d + q_t) + 4\}H + 2(2q_d + q_t)C_0X + C_{q_d-1}X + C_{q_d+q_t-1}X$$

This is much more benign as the exponentiation term is only in the initialisation part and only on a single qubit X-gate.

4.4.3. Approach 3: Quantum associative memory

Quantum associative memory (QuAM) as described in Section 3.4.3 can be used for pattern completion i.e. when some parts of the character is known, the rest of the pattern can be reconstructed from the stored states. However, the application of this thesis is to find the index of one pattern in another. In this case, the algorithm needs modification. Before the modifications are introduced, the original algorithm is simulated using unitary evolution in MATLAB.

The code for binary strings is given in Appendix-3. A random string over the binary alphabet of a predefined size is initialised. The search pattern is chosen to be a sub-string from the reference string at a known index. However, this is a pattern completion algorithm, so the final result is not an index, but the completed pattern from the quantum database. Since it completes the given pattern, if the pattern has no wildcard characters, the answer is same as the input pattern. Thus, a wildcard character is inserted in the search pattern to test the approximate matching. The classical database is prepared by slicing the reference string in sizes of the search string from each index. The duplicates are removed. These steps form part of the classical pre-processing required before the quantum algorithm is invoked. The quantum database is then initialised as given in Section 3.4.3 Figure 3.10. Note, these steps require an entire quantum gate set simulator since in MATLAB the access to the QX simulator is not present. These are coded in the auxiliary functions for general single qubit gate, controlled NOT, Toffoli and S matrix (special unitary for this algorithm). Finally, the uncomputed ancilla qubits are removed from the state space to reduce the redundant dimensionality. Once the quantum state is initialised, the search function is executed and the results are plotted. The search function creates the non-unitary matrix based on Hamming distances and collapses the stored database to a solution state.

It is not trivial to assess the number of qubits and gates required for this algorithm as it uses a non-unitary matrix which cannot be translated to a sequence of quantum gate evolution. The S_{sp} matrix can be presented as a controlled-Y rotation about an arbitrary angle of $CR_y(-2sin^{-1}(-1/\sqrt{p}))$. The number of qubits required for the initialisation step is $2M + 1$, of which, $M + 1$ are ancilla. Assuming the remainder of the circuit can be implemented with no more than these $M + 1$ uncomputed ancilla, on the M data qubits, the typical numbers for Illumina reads is calculated. The DNA alphabet encoding adds a factor of 2, resulting in required qubits to be 202. This number lies in between the two previous approach. However, this algorithm needs to be modified before it can be used for pattern *index* searching.

4.4.4. Quantum associative memory with distributed queries

A major improvement for the quantum associate memory is the use of distributed queries [85]. In a quantum associative memory, the number of entries is fewer than the total possible combinations (like the initialisation in Grover's search) and they form the memory set. The associative memory network solves the pattern completion problem; that is, it can restore the full pattern when initially presented with a partial pattern such that the known parts exactly coincide with some part of a valid full pattern. The creation of such a selected quantum superposition is non-trivial. The recall phase reconstructs only the remainder of the pattern while the initially presented partial pattern remains intact. The special transformation that allows a higher solution probability is performed by the inversion of the amplitudes around the average of only the amplitudes of memorised patterns. However, it is assumed that Oracle knows a part of one of the memorised patterns and no other pattern has the same part. Then Grover's amplitude amplification process is used to find the pattern having this part and entanglement permits the restoration of the remainder of this pattern.

General associative memory should also retrieve valid memory items when presented with noisy versions of a partial pattern. This improvement solves the problem of associative search for which no part of the input stimulus is guaranteed to be noise free. It is desirable to retrieve the memory state

which is most similar to the given stimulus. In the binary case, this corresponds to finding the minimum Hamming distance between the query and the memory states. This kind of memory is called a pattern correcting associative memory. The introduction of distributed query requires the initialised memory to store every possible basis state, despite most being spurious memory states (not corresponding to sub-strings from the reference pattern). Amplitudes are distributed in the fuzzy distributed query such that the maximal value occurs for some definite state p (the provided search query pattern) and the amplitudes of the other basis states x decrease monotonically with Hamming distance $h(p,x)$. p is called the query centre. The binomial distribution matches the required query model. For all $x \in \{0 \cdots (2^d - 1)\}$, let

$$|b_p^x\rangle = \sqrt{\gamma^{h(p,x)}(1-\gamma)^{d-h(p,x)}}$$

where, γ incorporates a metric into the model which tunes the width of the distribution permitting comparison of the similarity of the stimulus and the retrieved memory at a variable scale. The value γ is chosen as 0.25 for this project. It needs to be determined by the performance requirement for real dataset. The unitary Oracle transformation can be formed as,

$$O = I_{2^d \times 2^d} - 2\,|b_p\rangle\langle b_p|$$

The algorithm is given as in Figure 4.8. D in Step 5 refers to the Grover's gate implementing the inversion about mean. Λ refers to the optimal number of iterations.

1: $|0_1 0_2 \dots 0_n\rangle \equiv |\bar{0}\rangle$; {Initialize the register}
2: $|\Psi\rangle = A|\bar{0}\rangle = \frac{1}{\sqrt{N-m}} \sum_{x \notin M}^{N-1} |x\rangle$; {Learn the patterns using exclusion approach}
3: **repeat**
4: Apply the operator oracle $\mathcal{O}$ to the register;
5: Apply the operator diffusion $\mathcal{D}$ to the register;
6: $i = i + 1$;
7: **until** $i > \Lambda$
8: Observe the system.

Figure 4.8: Quantum associative memory with distributed query [86]

4.4.5. Quantum associative memory with improved distributed queries

Further modification to the model of a distributed query is carried out by merging the concept of the memory state Oracle with the binomial function based Oracle [86]. This algorithm is presented in Figure 4.9. Here, the quantum operators O and D are same as described in the last section. The

1: $|0_1 0_2 \dots 0_n\rangle \equiv |\bar{0}\rangle$; {Initialize the register}
2: $|\Psi\rangle = A|\bar{0}\rangle = \frac{1}{\sqrt{N-m}} \sum_{x \notin M}^{N-1} |x\rangle$; {Learn the patterns using exclusion approach}
3: Apply the operator oracle $\mathcal{O}$ to the register;
4: Apply the operator diffusion $\mathcal{D}$ to the register;
5: Apply operator $\mathcal{I}_M$ to the register;
6: Apply the operator diffusion $\mathcal{D}$ to the register;
7: **repeat**
8: Apply the operator oracle $\mathcal{O}$ to the register;
9: Apply the operator diffusion $\mathcal{D}$ to the register;
10: $i = i + 1$;
11: **until** $i > \Lambda - 2$
12: Observe the system.

Figure 4.9: Quantum associative memory with improved distributed query [86]

second Oracle, I_m, marks all the stored states in the memory, analogous to the idea presented in Section 3.4.3. This method shows considerable improvement over the simple distributed query, though is it more costly in terms of operations.

4.4.6. Quantum associative memory with closed match

One of the latest improvements in this field is to extend the algorithm to closed pattern matching [87]. It considers the case when the wildcard positions are not known in advance. A partial search pattern is known, say s bits, whereas, each entry in the associative memory consists of M bits. The proposed algorithm creates an Oracle such that, all possible combination of matching the partial search pattern as a sub-string of each memory pattern is taken care. This is shown in Figure 4.10. It is important to note that, the partially known characters form a substring, and not a sub-sequence (thus, wildcard characters in between, is not considered). This is a scope of further improvement for this algorithm if it needs to be adopted for genomic sequences (as gaps are quite common in reads).

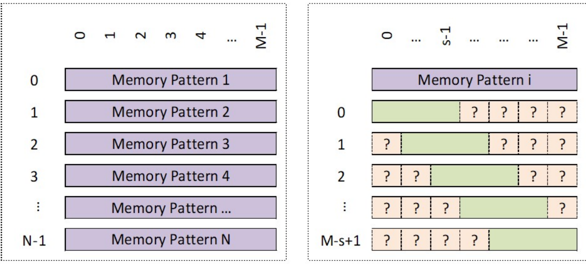


Figure 4.10: Closed pattern search

For the design of the algorithm of this thesis, the closed pattern search is not of much use. This is due to the structure that exists in the memory patterns. The general proposed algorithm considers the memories are independent of each other. In the pattern matching algorithm that is developed here, the memory patterns are consecutive shifted sub-strings of the reference pattern. This means, between two consecutive memory patterns, $M-1$ characters will match. All by one of the closed query of the partial search pattern are thus shared by these memories, leading to redundant association triggers.

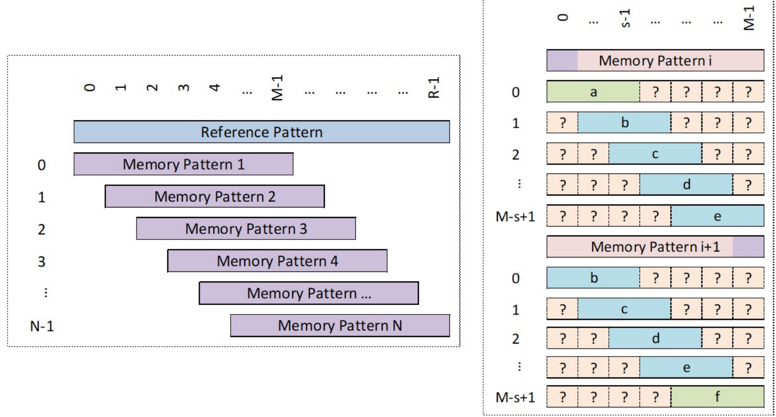


Figure 4.11: Closed pattern match for sequential memory

4.4.7. Quantum non-linear search algorithms

Further improvements to the previous algorithm have been proposed [88]. These, however, use Non-Linear Search Algorithm (NLSA), which is based on the fact that, under some circumstances, the superposition principle of quantum theory might be violated. A nonlinear quantum computer could solve NP-complete and even #P problems in polynomial time using the Weinberg's prescription This nonlinear evolution is the non-conservation of scalar products between non-linearly evolving solutions of a non-linear Schrodinger equation. This effect is called mobility phenomenon.

However, the formalism to encode these NLSA algorithms in the gate model of quantum computation is not yet clear and requires further investigation. These algorithms depend on a noise-free environment where small variations can be zoomed in and exploded to determine the result. Noise in the system would also get amplified thus making the calculation invalid [89].

4.5. Proposed algorithm for sub-sequence index search

The proposed algorithm inherits some of the features from the approaches highlighted in the previous section. It is essentially a novel quantum pattern matching algorithm specifically designed keeping in mind genomic sequences.

Here the idea of indexed bidirectional associative memory (iBAM) is introduced. Bidirectional associative memory (BAM) is a type of recurrent neural network [90]. It is hetero-associative, meaning, given a pattern, it can return another pattern which is potentially of a different size (auto-associative memories on the other hand, like the Hopfield network, return similar patterns of the same size). From a neural structure view, a BAM contains two layers of neurons, fully connected to each other as shown in Figure 4.12. Input into any of the layer presents the pattern in the other layer. BAM is unconditionally stable, i.e. any set of associations can be learned without risk of instability; and capable of error correction. Associative memories are costly to build in classical logic (2 times large cells as DRAM) as each input query requires a comparison with each single stored memory in parallel. This is the type of parallelism which is inherent in quantum superposition. Thus a quantum bidirectional associative memory (QBAM) is basically two registers entangled with each other. Each of the registers can have a superposition of corresponding memories.

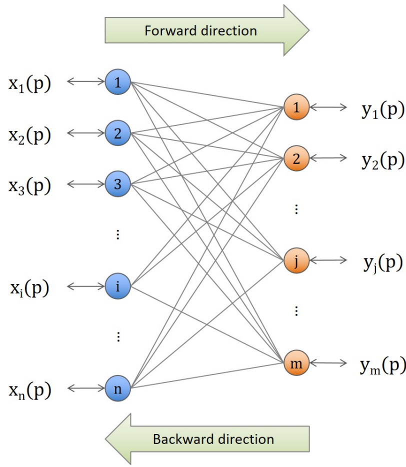


Figure 4.12: Bidirectional associative memory as a recurrent neural network

An interesting case for pattern index search is when one of the registers store the index and the second register stores the pattern, creating a QiBAM (quantum index-bidirectional associative memory). Thus, QiBAM can act both as an associative memory (if the query is a noisy pattern) as well as a RAM (if the query is an index), making it an ideal data-structure for this application.

The block diagram for the proposed algorithm is depicted in Figure 4.13. The initialisation of the QiBAM follows the design as described in Section 3.4.2 and Section 4.4.2. The tag qubits encode the pattern index, while the data qubits form the associative memory. Spurious memories need not be taken into account due to the exclusive nature of the initialisation where the full dimensionality of the tag is encoded. If the tag suffers for round-up approximations, a predefined *junk* memory is encoded for these extra tags. Thus, the arbitrary unitary of *pattern store in associative memory algorithm is modelled as a quantum phone directory encoding - the major algorithmic contribution in this thesis, which would allow the recall of the tagged index instead of pattern completion/correction*. Once the data is encoded, the Hamming distance evolution is carried out. This solves the problem of the black box Grover's marking Oracle. Now a predefined distributed query can be defined over the associative memory with the query centre at a zero Hamming distance. After following the iteration model of improved queries as described in Section 4.4.5, the tag qubits are measured to yield the index of the approximate match for the search pattern in the reference.

The code for the QiBAM algorithm is given in Appendix-6. For this example, the maximal super-string $AATTGTCTAGGCGACCA$ is used over the genomic alphabet of pattern length 2. To test the distributed query, the last memory, CA is not encoded, making the reference genome as $AATTGTCTAGGCGACC$. This is encoded as the database shown in Figure 4.14(a). Now the search query is chosen as CA. The search pattern conditionally toggles the database to evolve it to the Hamming distance. Since CA is not

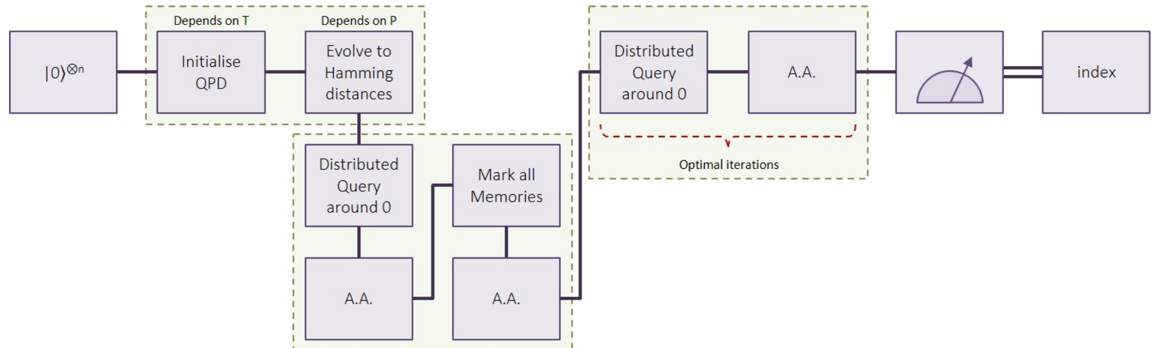
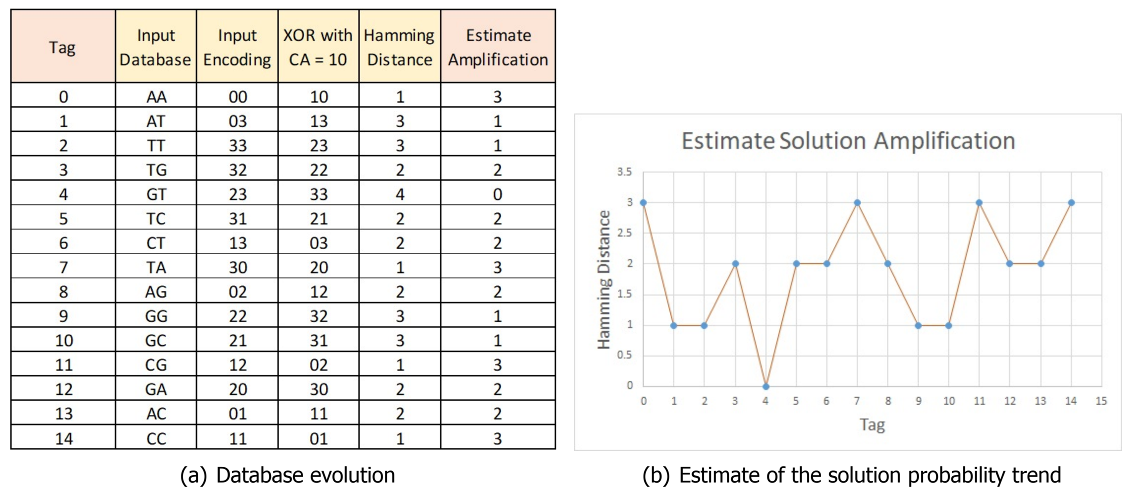


Figure 4.13: Block diagram of proposed QiBAM algorithm

present in memory, we expect the nearest patterns (approximate matches) to have a higher probability of detection, which are $\{AA, TA, CG, CC\}$ (with Hamming distance of 1 in the encoding). The estimated trend for a higher solution probability should be in line with decreasing Hamming distance, as plotted in Figure 4.14(b).



Tag	Input Database	Input Encoding	XOR with CA = 10	Hamming Distance	Estimate Amplification
0	AA	00	10	1	3
1	AT	03	13	3	1
2	TT	33	23	3	1
3	TG	32	22	2	2
4	GT	23	33	4	0
5	TC	31	21	2	2
6	CT	13	03	2	2
7	TA	30	20	1	3
8	AG	02	12	2	2
9	GG	22	32	3	1
10	GC	21	31	3	1
11	CG	12	02	1	3
12	GA	20	30	2	2
13	AC	01	11	2	2
14	CC	11	01	1	3

(a) Database evolution (b) Estimate of the solution probability trend

Figure 4.14: Result estimate of a sample run

The OpenQL algorithm is executed with the Qxelator library returning the internal state vector. This is plotted in Figure 4.15. The left vertical axis shows the staircase state curve for the tag qubits, while the right vertical axis shows the measurement probability of each individual state. There are 4 tag qubits and 4 data qubits (2 Radix-4 numbers for a DNA search pattern of length 2). Thus, the total state space is $2^8 = 256$. It is observed that there are some spurious memories that are recalled (as expected in associative memories). The states with prominent probabilities are the memory states. The envelope of these states (ignoring the spurious memories) gives the same trend as our estimate in Figure 4.14(b).

The qubit complexity of the implementation from data, tag and ancilla qubits is the same as Section 4.4.2,

$$\lceil log_2(A) \rceil M + \lceil log_2(N-M) \rceil + 1$$

yielding a typical estimate for the DNA alphabet, the human genome and Illumina reads $A = 4$, $N = 3 \times 10^8$ and $M = 50$, as 130 qubits. Following the arguments of Chargaff's rules as stated before, the gate complexity for initialisation and Hamming evolution is the same as Section 4.4.2,

$$q_t H + (q_t 2^{q_t} + q_d/2) C_0 X + q_t q_d / 2 C_{q_t} X$$

The Oracle is complex here and depends entirely on the chosen decomposition method for the unitary and the native gate set. The unitary decomposition method using Quantum Shannon Decomposition has a complexity of $3(4^{n-1} - 2^{n-1})$ where n is the dimension of the unitary. The memory Oracle

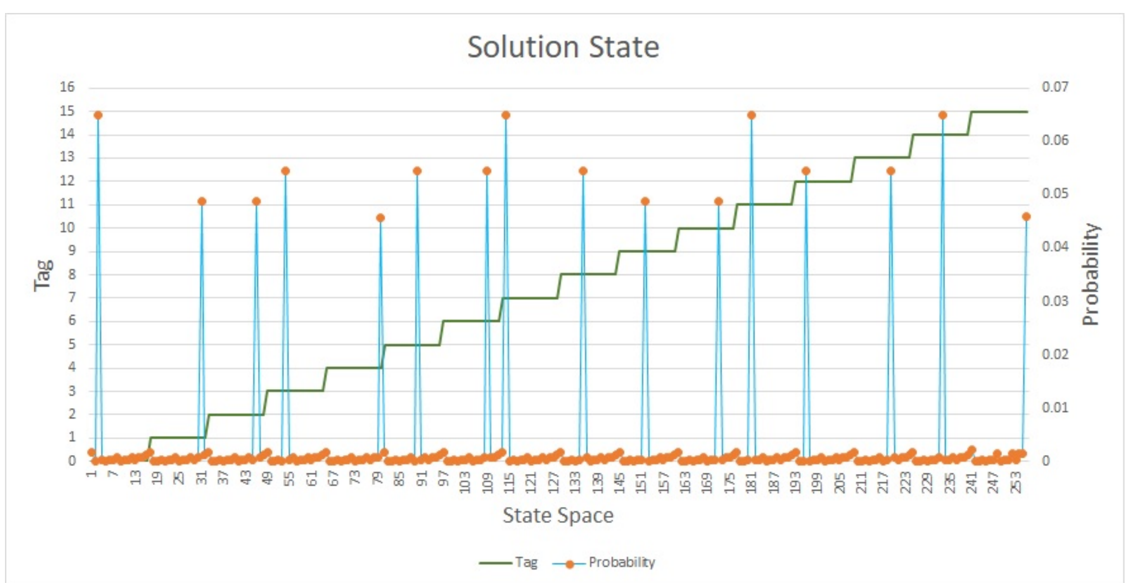


Figure 4.15: Execution example of Quantum Indexed Bidirectional Associative Memory

would mark the states in the initial quantum database. This requires a CPhase over the tag and data qubits for each of the 2^{q_t} memories of which $N-M+1$ are real memories from the reference genome (the rest are dummy). The data qubits, following Chargaff's rule, would have half the bits as 1, thus using a total of M qubits in average for the compute and uncompute. The tag qubits would follow the same behaviour as the initialisation phase, with average X-dressing of $q_t 2^{q_t} C_0X$ gates. Thus, the total for the memory Oracle is $(2^{q_t})\{2H + (M + q_t)C_0X + C_{q_d+q_t-1}X\}$. The Grover gate is decomposed to $\{2(q_d + q_t) + 2\}H + 2(q_d + q_t)C_0X + C_{q_d+q_t-1}X$ gates.

4.5.1. Complexity in quantum processors

It is realised through the course of the thesis that an isolated discussion of a specific quantum algorithm is not sufficient for near-term implementation. The quantum algorithm would have interfaces with other software modules running in parallel as shown in Figure 4.16.

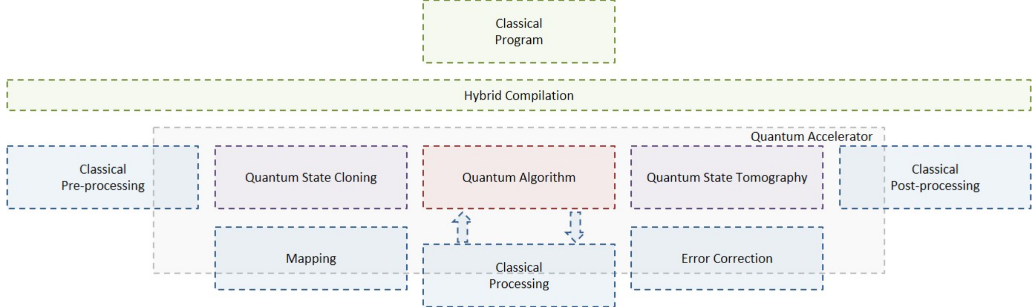


Figure 4.16: Quantum kernel and interfacing software architecture

There are 3 factors that contribute to the overall time complexity of a general quantum algorithm:

- Algorithm: *This pertains to the core algorithm running on a simulator, where the internal state vector can be accessed. It refers to the inherent gate complexity of the algorithm and other classical pre/post-processing involved.*

  In the accelerator model for quantum computing, the computation intensive kernel is offloaded to the quantum processor. This means the more mundane pre/post-processing is still carried out in the host CPU (or a classical processor co-located with the quantum processor in the quantum accelerator). The choice of where to perform these auxiliary calculations would depend on the tolerance of feedback delays between the classical and quantum parts of the code. The complexity

of these auxiliary calculations would add to the complexity of the quantum algorithm if it cannot be performed in parallel. The Qxelarator library allows hybrid programming by allowing the use of Python library functions alongside the quantum kernels in a single code file. The development of the next version of CommonQASM is expected to allow interleaving classical low-level instruction in the kernel, allowing the quantum accelerator to execute classical logic for fast feedback without consulting the host CPU.

- No-cloning: *If the internal state vector cannot be accessed (like in real quantum processors), the experiment needs to be repeated multiple times if the algorithm demands an estimate of the state's probability distribution.*

  Using the internal state vector is a very handy tool for developing quantum algorithms. However, it is also a clever trick which will never be possible on an actual hardware. A basic Grover search with optimal iteration evolves the result with just one run of the algorithm. However, for multiple solutions, this is no longer true. Inferring the result requires reconstructing the probabilistic state vector up to a degree of precision. Thus, to implement the algorithms discussed in the thesis, in an ideal quantum computer, multiple runs are required. With each measurement result, the waveform converges to the actual quantum state - referred to as quantum state tomography. The convergence rate depends entirely on the final state, the measurement basis and the tolerable error bound for the practical application. Quantum state tomography is an area of active research and is beyond the scope of this thesis. Advanced methods based on linear inversion, linear regression, maximum likelihood, Bayesian, compressed-sensing and neural networks [91] exists for estimating the state with fewer tomographic trials.

- Experimental: *It takes into account the complexity overhead for topological mapping and quantum error correction cycles.*

  In real quantum computers, two experimental considerations add to the complexity: error correction and mapping. The topological implications of mapping would increase the total gate count due to physical constraints of the quantum processor chip, for example, nearest neighbour mapping. The field of QEC strives towards a fault-tolerant quantum computer, by error correction cycles. This includes multiple physical operations to perform a single fault-tolerant logical operation. These factors are intertwined. With the increase of gate counts in mapping, circuit duration becomes longer, needing more error correction gate. Some gates are not trivial requiring state distillation, which might add to the decoherence as well as gate complexity.

Thus, every quantum algorithm that depends on a probabilistic result in a noisy environment needs to be repeated, adding a multiplicative factor to the inherent time complexity.

$$O(f(experimental) \times g(no-cloning) \times h(algorithm))$$

4.6. A summary of considered algorithms

In this thesis, search based quantum pattern matching is explored. The studied algorithms are listed in Table 4.1.

Table 4.1: List of studied algorithms

A fast quantum mechanical algorithm for database search [42]
Quantum mechanics helps in searching for a needle in a haystack [43]
Tight bounds on quantum searching [45]
Generalized Grover search algorithm for arbitrary initial amplitude distribution [46]
Quantum pattern matching [57]
Fast quantum search algorithms in protein sequence comparisons: Quantum bioinformatics [58]
Artificial associative memory using quantum processes [61]
Initializing the amplitude distribution of a quantum state [62]
Quantum associative memory with exponential capacity [60]
A quantum associative memory based on Grover's algorithm [63]
Quantum associative memory [59]
Quantum associative memory with distributed queries [85]
Quantum associative memory with improved distributed queries [86]
PROPOSED: Quantum indexed-bidirectional associative memory for DNA sequence reconstruction

In summary, the features of these algorithms are tabulated in Figure 4.17. Most of the features of the basic Grover's quantum search has been modified to construct the proposed QiBAM algorithm. The major contribution of the proposed algorithm is to merge the idea of Hamming distance based search with a distributed query to construct a fixed Oracle capable of approximate matching. Also, considering the requirements of the reconstruction process, the algorithm is designed to produce the index (of a noisy query's nearest match) as the result instead of correcting the noisy query as in quantum associative memory.

Algorithm	Number of solutions	Database type	Oracle	Other oracles	Arbitrary Unitary	Number of Qubits A = size(alphabet) N = size(reference) M = size(pattern)	Index or (Un)Corrected Query	OpenQL Implementation	MATLAB Implementation	Extension to DNA code
[42-43]	1	full superposition	blackbox	-	N	AM	UQ	Y	Y	N
[45]	1	full superposition	blackbox	-	N	AM	UQ	N	Y	N
	multiple known								Y	
	unknown								Y	
[46]	multiple known	arbitrary	blackbox	-	N	AM	UQ	N	Y	N
[57]	unknown	sliding index	alphabet based	-	N	Mlog(N)	UQ	Y	Y	Y
[58]	1	indexed substrings	0 Hamming distance	-	N	AM+log(N-M+1)	I	Y	N	Y
[59-63]	multiple known	arbitrary	blackbox	memory marking	Y	AM	CQ	N	Y	N
[85]	unknown	arbitrary	distributed query	-	Y	AM	CQ	Y	Y	Y
[86]	unknown	arbitrary	distributed query	memory marking	Y	AM	CQ	N	Y	Y
QiBAM	unknown	indexed substrings	distributed query around 0	memory marking	Y	AM+log(N-M+1)	I	Y	Y	Y

Figure 4.17: Summary of algorithms considered for the thesis

5
Conclusions and future work

We can only see a short distance ahead, but we can see plenty there that needs to be done.
- Alan Turing

5.1. Conclusion

This thesis is motivated by the big data problem in genomics. The sequencing pipeline is studied to infer that, DNA *sequence reconstruction* from the multitude of short reads is a computation intensive process - taking in order of days to execute on a super-computing cluster. Different existing classical algorithms for DNA sequence reconstruction are explored - from the naive search to the current state-of-the-art. The pattern matching on genomic sequences required for the reconstruction process is formulated as the *sub-sequence alignment* problem.

The possibilities of harnessing quantum computation to accelerate sub-sequence alignment are explored in this thesis. The basic *quantum search algorithm* of Grover is presented, which allows searching one solution in a full superposition state with quadratic query complexity. However, for construction of the circuit, the solution needs to be known in advance for the Oracle marking step. Modifications of Grover algorithm extend it for multiple solutions and arbitrary amplitude. These improvements are found to be crucial for the development of practical quantum search algorithms.

Three different existing methods to use quantum search for pattern matching are discussed. These approaches are implemented using the QX quantum simulator encapsulated in the OpenQL programming framework. The first approach compiles multiple Oracles, one for each character of the pattern's alphabet. These Oracles are conditionally invoked based on the search pattern. It alleviates the black box nature of the Oracle. However, as a disadvantage, this algorithm requires a large number of qubits and does not produce satisfying results when there are multiple solutions.

The second approach uses a quantum phone directory. Instead of storing the index of the states (as in the previous approach), the starting index and the pattern states itself are initialised as the quantum database. Thereafter, the data patterns are evolved to their Hamming distances with the search query pattern. The data entry with an exact match would have a distance of zero. Thus, the Oracle amplifies this known state and measuring the tag qubits reveal the answer. This method is useful for index searching using exact pattern matching.

The third approach explores the possibility to achieve higher solution probability. This proposition employs quantum associative memory, where two different Oracles are used - one to mark the solution state, the other to mark all stored memory states. By this method, better results are obtained by using the memory Oracle. However, it is a general approach and needs modification for solving the problem of pattern index searching.

The developmental tools used for the implementation include the OpenQL compiler framework, the QX simulator and the developed *Quantum Innovation Environment*. Few assumptions were required for the thesis to bridge the gap between currently available hardware and the high-level algorithm development approach in the thesis. These include bounds on the number of qubits, number and type of quantum operations, environmental noise and qubit topology. In order to generate test patterns for the quantum algorithms, three different methods are used - a realistic dataset, randomly generated data

and a *minimal super-string* approach. In addition, this thesis requires some special gate decomposition techniques that are not currently available in OpenQL. This includes *unitary decomposition*, *multi-controlled NOT gates* and *Boolean function Oracles*. These are developed as utility modules for the pattern matching algorithms.

The three existing approaches for pattern matching are *modified for genomic patterns with an alphabet size of 4*. The *detailed design of each kernel* is discussed to bridge the gap between the algorithmic descriptions given in the original research with the quantum circuit implementation for the simulator. The qubit and gate complexity are derived whenever possible.

A *new algorithm* is designed inspired by the best features of existing strategies. This algorithm allows approximate matching (minimal Hamming distance) of a sub-sequence to a reference pattern and *returns the index of the approximate match location in the reference*. It is based on the concept of Bidirectional Associative Memory. Two quantum registers are entangled as a quantum phone directory, with one storing the index (tag), while the other storing the pattern (data) in the reference at the corresponding index. Then, the data register is evolved to the Hamming distance between each pattern and the search query. A distributed query is constructed using Hamming distance and a binomial function, that amplifies the minimum Hamming distance available among the states in the memory. The mode value of the matching index can be assessed by measuring the aggregate of the collapsed state over multiple runs of the algorithm. Sample *execution results in OpenQL is shown to match the design predictions*. Finally, the complexity of a generic quantum algorithm in realistic hardware is discussed taking into account environmental and no-cloning factors.

Summarising, the core contributions of the thesis are:

- Sub-sequence alignment is justified as a candidate for acceleration using quantum search-based approaches.
- Implementation and simulation of 3 existing quantum pattern matching strategies in OpenQL and QX simulator.
- Development of an algorithm to generate a minimal length super-string to extensively test the algorithms.
- Implementation of a unitary gate decomposer in MATLAB to allow implementation of more advanced algorithms in a circuit model simulator.
- Development of a PyQt based Integrated Development Environment for a more intuitive understanding of the circuit based quantum state evolution using OpenQL, QX, quantum circuits and state vector plots.
- Development of a new approach to address the problem of approximate matching in genomic sequences, combining the ideas of associative memory, indexed directory and distributed query. This algorithm is called *Quantum indexed-Bidirectional Associative Memory* (QiBAM).

5.2. Future work

This thesis forms a first exploration towards the feasibility of quantum acceleration for genomic data. It is hard to do justice to the vast field of quantum algorithms in its entirety. Likewise, some application domains are closely related to DNA sequence alignment and can benefit from the developed algorithms. Some promising yet nascent avenues are suggested for future explorations.

5.2.1. Other quantum computing models

In this thesis, the circuit model of computation is explored. This is driven by the ongoing development of superconductor and semiconductor based quantum processors at QuTech. The central problem in the thesis: finding maximum similarity between patterns, can be modelled as a minimisation of Hamming distance problem. *Adiabatic quantum computers* (like D-WAVE 2000) has shown promises in such global optimisation problems. Since larger scale adiabatic quantum computers are already available, it might be worth implementing genomic applications in such devices before universal gate based quantum computers reach similar readiness levels.

Measurement-based quantum computing and *Topological quantum computing* are two other models which are under development in various research groups. These different models will require different paradigms of quantum programming which might not be entirely translatable by a universal compiler.

5.2.2. Other quantum algorithms

The algorithms explored and extended in this thesis are based on unstructured Grover's search. Due to the wide applicability and intuitive nature of searching, this has been the focus of most quantum search algorithms developed so far. Also, the proven optimality of Grover search is promising from a computational complexity perspective. However, the problem of pattern matching can also be solved by other techniques. *Quantum walk based algorithms* are a promising alternative as bioinformatics local string matching can be modelled as a graph based algorithm. Quantum algorithms for *Abelian and Dihedral hidden structure* can also be used for template matching problems. Existing structural information of genome sequences can also be harnessed to design faster search algorithms.

The various algorithms explored have increasingly transitioned from simple search to applicability in pattern matching and finally to more sophisticated techniques based on learning theory. The wider class of quantum neural network and *quantum machine learning* needs to be studied for suitability in the application of pattern recognition.

5.2.3. Other bioinformatics applications

From the bioinformatics perspective, this thesis solves the problem of finding the index of nearest match of a search pattern in a database of patterns. This is a primitive for both reference-based alignment as well as, the pairwise alignment pre-processing step for de novo assembly. A useful extension of this algorithm is to solve the problem of *many-to-many pattern matching*, where multiple search patterns can be evaluated in parallel. The *quality value of the reads* can also be used to improve the Boolean Oracle with the Hamming distance weighted by the belief factor of each of the matches.

Other problems in bioinformatics requiring similar primitives can be explored. An immediate augmentation is to extend the developed algorithm for an alphabet size of 20 (the amino-acid codon). This will enable research in *protein sequence analysis*. The identification of DNA binding sites and the specificity of target proteins in binding to these regions are two important steps in understanding the mechanisms of biological activities like transcriptional regulation, repair, recombination, splicing, and DNA modification. This is achieved by *motif finding* in the DNA sequence. Since performing state tomography on the associative memory would reconstruct the most probable pattern, it is very similar to finding a motif. *DNA fingerprinting* and *comparative genomics* are other promising avenues where similar large datasets are involved and can be efficiently parallelised by quantum algorithms.

5.2.4. Other application domains

Other areas where similar algorithms are applicable can be explored. The problem of pattern matching can be easily mapped to application like *image recognition*, *pattern-based stock trading* and *speech recognition*. In fact, since associative memory can be generalised to the *Hebbian learning model*, a wide range of pattern matching that are carried out by humans and classical machine learning algorithms can be mapped to quantum associative memory. Many quantum research groups (e.g. Google, Microsoft) consider quantum machine learning of images as a big data problem for exascale computing using quantum technologies. In this project, a preliminary exploration for image recognition is carried out with promising results.

5.2.5. System design

Finally, this thesis forms the initial study towards a hardware implementation for a genome sequence analysis pipeline. A system design of quantum accelerator for this application will be in the timeline of this project. It requires careful design of not only the quantum algorithm but *other layers of the computing stack* tailored for this purpose. Auxiliary algorithms for quantum state tomography, state cloning, mapping, error correction and aggressive compiler optimisation needs to be considered. Architectural considerations of pipelining algorithm stages, security and micro-architecture need to be considered for the specifically targeted hardware. The algorithm developed needs to be encapsulated in a *domain-specific hardware platform* (instead of a universal general-purpose quantum computer) for near-term implementation.

Studying the developed algorithm from *quantum information theory* perspective would help to develop edge test cases and provide insights into the complexity bounds. Rigorous theoretical proofs based on the properties of the input DNA data would help establish the applicability of quantum computing in search based pattern matching in bioinformatics. Such inferences would lead towards research into the prospect of inherent *quantum-biological computation* processes.

Appendix-1

```python
## Reference: Quantum Pattern Matching - P. Mateus, Y. Omar (arXiv preprint quant-ph/0508237)
## \author: Aritra Sarkar (prince-ph0en1x)
## \project: Quantum-accelerated Genome-sequencing
## \repo: https://gitlab.com/prince-ph0en1x/QaGs

#####################################################################################

from openql import openql as ql
import qxelarator

import random
from math import *
import os
import re
import math
import matplotlib.pyplot as plt
import numpy as np

#####################################################################################

def randStr(szA,sz):
    # Generates a random string of length 'sz' over the alphabet of size 'szA' in decimal
    bias = 1/szA                        # IMPROVE: add bias here
    rbs = ""
    for i in range(0,sz):
        rn = random.random()
        for j in range(0,szA):
            if rn < (j+1)*bias:
                rbs = rbs + str(j)  # IMPROVE: BCD version
                break
    return rbs

#####################################################################################

AS = {'00','01','10','11'}  # Alphabet set {0,1,2,3} := {A,C,G,T} for DNA Nucleotide bases
A = len(AS)                 # Alphabet size
N = 11                      # Reference Genome size
w = randStr(4,N)            # Reference Genome  (e.g. w = "22013213")
M = 3                       # Short Read size
ans = 2                     # Known answer for testing
p = w[ans:ans+M]            # Short Read

s = ceil(log2(N-M))
Q_D = s * M                 # Number of data qubits
Q_anc = 1                   # Number of ancilla qubits
anc = Q_D                   # Ancilla qubit id
total_qubits = Q_D + Q_anc

#####################################################################################

def QPM():
    print(w,p)
    config_fn = os.path.join('gateConfig.json')
    platform = ql.Platform('platform_none', config_fn)
    prog = ql.Program('qpm_a4', total_qubits, platform)

    # Kernel 1: Initialization
    qk1 = ql.Kernel('QCirc1',platform)
    Circ1(qk1)

    # Kernel 2: Oracles to mark specific character
    qk2 = ql.Kernel('QCirc2',platform)
```

```
63      bfa = ''.join('1' if w[i] == '0' else '0' for i in range(len(w)))
64      bfc = ''.join('1' if w[i] == '1' else '0' for i in range(len(w)))
65      bfg = ''.join('1' if w[i] == '2' else '0' for i in range(len(w)))
66      bft = ''.join('1' if w[i] == '3' else '0' for i in range(len(w)))
67  
68      # Kernel 3: Grover Amplitude Amplification
69      qk3 = ql.Kernel('QCirc3',platform)
70      Circ3(qk3,s,M)
71  
72      # Kernel 4: Measurement
73      qk4 = ql.Kernel('QCirc4',platform)
74      Circ4(qk4)
75  
76      # Construct Program from Kernels
77      prog.add_kernel(qk1)            # Initialise
78      #for pi in range(0,M):          # Alternate iteration method
79      for r in range(0,int(sqrt(N-M+1))):
80          pi = random.randint(0,M-1)
81          if p[pi] == '0':
82              Circ2(qk2,bfa,pi)
83          elif p[pi] == '1':
84              Circ2(qk2,bfc,pi)
85          elif p[pi] == '2':
86              Circ2(qk2,bfg,pi)
87          else:
88              Circ2(qk2,bft,pi)
89          prog.add_kernel(qk2)        # Conditional kernel call
90          del qk2                     # IMPROVE: Kernel to qubit loose binding being discussed
91          qk2 = ql.Kernel('QCirc2',platform)
92          prog.add_kernel(qk3)        # Inversion about mean
93      # prog.add_kernel(qk4)          # Uncomment if using measurement based analytics
94      prog.compile()
95      # showQasm()
96      qx = qxelarator.QX()
97      qx.set('test_output/qpm_a4.qasm')
98  
99      # Result analytics using Internal State Vector
100     qx.execute()
101     qxopt = qx.get_state()
102     isv = [0]*(2**total_qubits)
103     ptrn = re.compile('\(([+-]\d+.\d*),([+-]\d+[.\d*]?)\)\s[|]([0-1]*)>')
104     for line in qxopt.splitlines():
105         mtch = ptrn.search(line)
106         if mtch != None:
107             ar = float(mtch.group(1))
108             ac = float(mtch.group(2))
109             state = int(mtch.group(3),2)
110             isv[state] = ar**2 + ac**2
111     ploty = [0]*(2**s)
112     for i in range(0,len(isv)):
113         stot = format(i,'0'+str(total_qubits)+'b')[::-1]
114         sopt = int(stot[0:s],2)
115         ploty[sopt] = ploty[sopt] + isv[i]
116     print("PMax:", np.amax(ploty))
117     print("Index:", np.argmax(ploty))
118     plt.plot(ploty)
119     plt.ylabel('Probability')
120     plt.xlabel('Solution space')
121     plt.ylim([0,1])
122     plt.show()
123 
124     # Result analytics using Measurement
125     '''
126     res = [0]*s
127     STT = 1000                      # Number of quantum state tomography trials
128     true_counter = 0
129     for i in range(STT):
130         qx.execute()
131         res[0] = res[0] + qx.get_measurement_outcome(0)
132         res[1] = res[1] + qx.get_measurement_outcome(1)
133         res[2] = res[2] + qx.get_measurement_outcome(2)
```

```
    index = ''.join('1' if res[i] > STT/2 else '0' for i in range(s))
    print("Index:",int(index,2))
    '''

    return

######################################################################

def Circ1(k):
    for Qi in range(0,total_qubits):# Initialise all qubits to |0> state
        k.prepz(Qi)
    for si in range(0,s):           # Uniform superposition of possible starting positions (
    answers)
        k.gate("h",[si])
    for Mi in range(0,M-1):
        for si in range(0,s):       # Copy position encoding to next set of s
            k.gate("cnot",[Mi*s+si, (Mi+1)*s+si])
        for si in range(0,s):       # Increment position encoding
            ic = (Mi+1)*s-(si+1)
            k.gate("x",[ic])        # Inverted control
            nc = []
            for sj in range(ic,(Mi+1)*s):
                nc.append(sj)
            for sj in range((Mi+2)*s-1,s+ic-1,-1):
                nCX(k,nc,sj,anc)    # Decompose multi-controlled CNOT
            k.gate("x",[ic])        # Uncompute inverted control
    return

######################################################################

def Circ2(k,bf,q):
    for fi in range(0,len(bf)):     # Encode oracle function
        if bf[fi] == '1':
            fis = format(fi,'0'+str(s)+'b')
            for fisi in range(0,s): # Encode binary value of function input
                if fis[fisi] == '0':
                    k.gate("x",[q*s+fisi])
            k.gate("h",[(q+1)*s-1]) # CPhase to CNOT conversion
            nc = []
            for qsi in range(q*s,(q+1)*s-1):
                nc.append(qsi)
            nCX(k,nc,(q+1)*s-1,anc) # Decompose multi-controlled CNOT
            k.gate("h",[(q+1)*s-1]) # Uncompute CPhase to CNOT conversion
            for fisi in range(0,s): # Uncompute binary value of function input
                if fis[fisi] == '0':
                    k.gate("x",[q*s+fisi])
    return

######################################################################

def Circ3(k,s,M):
    for si in range(0,s*M):
        k.gate("h",[si])
        k.gate("x",[si])
    k.gate("h",[s*M-1])             # CPhase to CNOT conversion
    nc = []
    for sj in range(0,s*M-1):
        nc.append(sj)
    nCX(k,nc,s*M-1,s*M)             # Decompose multi-controlled CNOT
    k.gate("h",[s*M-1])             # Uncompute CPhase to CNOT conversion
    for si in range(0,s*M):
        k.gate("x",[si])
        k.gate("h",[si])
    return

######################################################################

def Circ4(k):
    for si in range(0,s):           # Measure first set of positions
        k.gate("measure",[si])
    return
```

```
204
205 #############################################################################################
206
207 def nCX(k,c,t,b):
208     nc = len(c)
209     if nc == 1:
210         k.gate("cnot",[c[0], t])
211     elif nc == 2:
212         k.toffoli(c[0],c[1],t)
213     else:
214         nch = ceil(nc/2)
215         c1 = c[:nch]
216         c2 = c[nch:]
217         c2.append(b)
218         nCX(k,c1,b,nch+1)
219         nCX(k,c2,t,nch-1)
220         nCX(k,c1,b,nch+1)
221         nCX(k,c2,t,nch-1)
222     return
223
224 #############################################################################################
225
226 def showQasm():
227     file = open("test_output/qpm_a4.qasm","r")
228     print("\n~~~~~~ CODE FILE ~~~~~~\n")
229     for line in file:
230         print (line,end='')
231     print ()
232     file.close()
233
234 #############################################################################################
235
236 if __name__ == '__main__':
237     QPM()
```

Appendix-2

```
## Reference:Fast quantum search algorithms in protein sequence comparison: Quantum
    bioinformatics - L. Hollenberg (arXiv preprint quant-ph/0002076)
## \author: Aritra Sarkar (prince-ph0en1x)
## \project: Quantum-accelerated Genome-sequencing
## \repo: https://gitlab.com/prince-ph0en1x/QaGs

##########################################################################################

from openql import openql as ql
import qxelarator

import random
from math import *
import os
import re
import math
import matplotlib.pyplot as plt
import numpy as np

##########################################################################################

def randStr(szA,sz):
    # Generates a random string of length 'sz' over the alphabet of size 'szA' in decimal
    bias = 1/szA                       # IMPROVE: add bias here
    rbs = ""
    for i in range(0,sz):
        rn = random.random()
        for j in range(0,szA):
            if rn < (j+1)*bias:
                rbs = rbs + str(j)  # IMPROVE: BCD version
                break
    return rbs

##########################################################################################

AS = {'00','01','10','11'}  # Alphabet set {0,1,2,3} := {A,C,G,T} for DNA Nucleotide bases
A = len(AS)                 # Alphabet size
N = 10                      # Reference Genome size
#w = randStr(4,N)           # Reference Genome  (e.g. w = "2302031020")
w = "2302031020"
M = 3                       # Short Read size
ans = 1                     # Known answer for testing
dummyp = "000"
p = w[ans:ans+M]            # Short Read

Q_A = ceil(log2(A))         # Number of qubits to encode one character
Q_D = Q_A * M               # Number of data qubits
Q_T = ceil(log2(N-M))       # Tag Qubits
Q_anc = 1                   # Number of ancilla qubits
anc = Q_D + Q_T             # Ancilla qubit id
total_qubits = Q_D + Q_T + Q_anc

##########################################################################################

def QPD():
    print(w,p)
    config_fn = os.path.join('gateConfig.json')
    platform = ql.Platform('platform_none', config_fn)
    prog = ql.Program('qpd_a4', total_qubits, platform)

    # Kernel 1: Initialization of Quantum Phone Directory
    qk1 = ql.Kernel('QCirc1',platform)
```

```
    Circ1(qk1)

    # Kernel 2: Calculate Hamming Distance
    qk2 = ql.Kernel('QCirc2',platform)
    Circ2(qk2)

    # Kernel 3: Oracle to Mark Hamming Distance of 0
    qk3 = ql.Kernel('QCirc3',platform)
    Circ3(qk3)

    # Kernel 4: Grover Amplitude Amplification
    qk4 = ql.Kernel('QCirc4',platform)
    Circ4(qk4)

    # Kernel 5: Measurement
    qk5 = ql.Kernel('QCirc5',platform)
    Circ5(qk5)

    # Finding optimal iterations for known arbitrary initial amplitude distribution
    t = 1               # Expected number of solutions
    iMx = 2**Q_T
    sMx = 2**(Q_D+Q_T)
    kavg0 = 1/sqrt(iMx)
    lavg0 = (iMx - t)/((sMx - iMx)*sqrt(iMx))
    Pmax = 1 - (sMx-iMx)*lavg0**2 - (iMx-t)*(1/sqrt(iMx) - lavg0)**2
    print("Theoretical PMax:", Pmax)
    T = [0]*5
    for j in range(0,5):
        T[j] = ((j/2+0.5)*pi - atan(kavg0*sqrt(t/(sMx-t))/lavg0)) / acos(1-2*t/sMx)
    print("Suggested Iterations:",T)
    # IMPROVE: Use suggested iterations

    # Construct Program from Kernels
    prog.add_kernel(qk1)            # Initialise
    prog.add_kernel(qk2)            # Transform to Hamming distance
    for r in range(0,1):
        prog.add_kernel(qk3)        # Oracle call
        prog.add_kernel(qk4)        # Inversion about mean
    # prog.add_kernel(qk5)          # Uncomment if using measurement based analytics
    prog.compile()
    # showQasm()
    qx = qxelarator.QX()
    qx.set('test_output/qpd_a4.qasm')

    # Result analytics using Internal State Vector
    qx.execute()
    qxopt = qx.get_state()
    isv = [0]*(2**total_qubits)
    ptrn = re.compile('\(([+-]\d+.\d*),([+-]\d+[.\d*]?)\)\s[|]([0-1]*)>')
    for line in qxopt.splitlines():
        mtch = ptrn.search(line)
        if mtch != None:
            ar = float(mtch.group(1))
            ac = float(mtch.group(2))
            state = int(mtch.group(3),2)
            isv[state] = ar**2 + ac**2
    ploty = isv
    print("PMax:", np.amax(ploty))
    tag = format(np.argmax(ploty),'0'+str(total_qubits-1)+'b')[::-1]
    print("Index:", int(tag[0:3],2))
    plt.plot(ploty)
    plt.ylabel('Probability')
    plt.xlabel('State space')
    plt.ylim([0,1])
    plt.show()
    return

###############################################################################################

def Circ1(k):
    for Qi in range(0,total_qubits):# Initialise all qubits to |0> state
```

```
133         k.prepz(Qi)
134     for Qi in range(0,Q_T):          # Uniform superposition of possible starting positions (
    answers)
135         k.gate("h",[Qi])
136     nc = []
137     for ci in range(0,Q_T):
138         nc.append(ci)
139     for Qi in range(0,N-M+1):
140         Qis = format(Qi,'0'+str(Q_T)+'b')
141         for Qisi in range(0,Q_T):
142             if Qis[Qisi] == '0':
143                 k.gate("x",[Qisi])
144         wMi = w[Qi:Qi+M]
145         print([Qis,wMi])
146         for wisi in range(0,M):
147             wisia = format(int(wMi[wisi]),'0'+str(Q_A)+'b')
148             for wisiai in range(0,Q_A):
149                 if wisia[wisiai] == '1':
150                     nCX(k,nc,Q_T+wisi*Q_A+wisiai,anc)
151         for Qisi in range(0,Q_T):
152             if Qis[Qisi] == '0':
153                 k.gate("x",[Qisi])
154     return
155 
156 ##############################################################################################
157 
158 def Circ2(k):
159     for pi in range(0,M):
160         ppi = format(int(p[pi]),'0'+str(Q_A)+'b')
161         for ppii in range(0,Q_A):
162             if ppi[ppii] == '1':
163                 k.gate("x",[Q_T+pi*Q_A+ppii])
164     return
165 
166 ##############################################################################################
167 
168 def Circ3(k):
169     for Qi in range(0,Q_D):          # Encode binary value 0 of function input
170         k.gate("x",[Q_T+Qi])
171     k.gate("h",[Q_D+Q_T-1])          # CPhase to CNOT conversion
172     nc = []
173     for qsi in range(Q_T,Q_T+Q_D-1):
174         nc.append(qsi)
175     nCX(k,nc,Q_D+Q_T-1,anc)      # Decompose multi-controlled CNOT
176     k.gate("h",[Q_D+Q_T-1])          # Uncompute CPhase to CNOT conversion
177     for Qi in range(0,Q_D):          # Uncompute binary value of function input
178         k.gate("x",[Q_T+Qi])
179 
180 ##############################################################################################
181 
182 def Circ4(k):
183     for si in range(0,Q_D+Q_T):
184         k.gate("h",[si])
185         k.gate("x",[si])
186     k.gate("h",[Q_D+Q_T-1])          # CPhase to CNOT conversion
187     nc = []
188     for sj in range(0,Q_D+Q_T-1):
189         nc.append(sj)
190     print(nc)
191     nCX(k,nc,Q_D+Q_T-1,anc)      # Decompose multi-controlled CNOT
192     k.gate("h",[Q_D+Q_T-1])          # Uncompute CPhase to CNOT conversion
193     for si in range(0,Q_D+Q_T):
194         k.gate("x",[si])
195         k.gate("h",[si])
196     return
197 
198 ##############################################################################################
199 
200 def Circ5(k):
201     #k.display()
202     '''
```

```
203     for si in range(0,s):           # Measure first set of positions
204         k.gate("measure",[si])
205     '''
206     return
207 
208 ##########################################################################################
209 
210 def nCX(k,c,t,b):
211     nc = len(c)
212     if nc == 1:
213         k.gate("cnot",[c[0], t])
214     elif nc == 2:
215         k.toffoli(c[0],c[1],t)
216     else:
217         nch = ceil(nc/2)
218         c1 = c[:nch]
219         c2 = c[nch:]
220         c2.append(b)
221         nCX(k,c1,b,nch+1)
222         nCX(k,c2,t,nch-1)
223         nCX(k,c1,b,nch+1)
224         nCX(k,c2,t,nch-1)
225     return
226 
227 ##########################################################################################
228 
229 def showQasm():
230     file = open("test_output/qpd_a4.qasm","r")
231     print("\n~~~~~ CODE FILE ~~~~~\n")
232     for line in file:
233         print (line,end='')
234     print ()
235     file.close()
236 
237 ##########################################################################################
238 
239 if __name__ == '__main__':
240     QPD()
```

Appendix-3

```
% Reference: Quantum Associative Memory - D. Ventura, T. Martinez (arXiv preprint quant-ph/9807053)
% \author: Aritra Sarkar (prince-ph0en1x)
% \project: Quantum-accelerated Genome-sequencing
% \repo: https://gitlab.com/prince-ph0en1x/QaGs

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function mtlb_qam_a4()
    close all
    clear all
    clc

    AS = {'0','1'};             % Binary alphabet set
    A = size(AS,2);             % Alphabet size
    N = 10;                     % Reference Pattern size
    [w,~] = randStr(A,N);       % Reference Pattern     (e.g. w = "0001110100")
    w = '0001110100'
    M = 3;                      % Search Pattern size
    tst_idx = 2;                % Known answer for testing (solution index starts at 0)
    p = w(tst_idx+1:tst_idx+M); % Search Pattern
%   p = strrep(p,'2','?')       % Insert wildcard for approximate matching

    % Initialise the database
    cdb = prepCdb(w,M);         % Prepare classical database
    ucdb = unique(cdb,'rows');  % Remove duplicate memories
    v = ones(1,size(ucdb,1));   % values to encode (optional)
    qdb = prepQdb(ucdb,v);      % Prepare quantum database
    state = rednDimn(qdb,M);    % Remove uncomputed ancillas
%   dispState(state,0)

    plot(state,'-g')
    hold on
    state = runSrch(p,state);   % Run Quantum search operation
%   dispState(state,0)
    plot(state,'-b')
    axis([1 2^M -1 1])
    psr = remWC(p);
    plot([psr'+1; psr'+1], [-1; 1]','-r')
    [maxV,maxP] = max(state);
    plot([1; 2^M], [maxV; maxV]','-m')
    set(gca,'XTickLabels',0:2^M-1)
end

%% Generates a random string of length 'sz' over the alphabet of size 'szA' in decimal and binary

function [rs,rb] = randStr(szA,sz)
    ranges = linspace(0,1,szA+1);       % assumes equal probablility (add bias here)
    rs = '';
    rb = '';
    for i = 1:sz
        rn = rand();
        for j = 2:szA+1
            if rn < ranges(j)
                rb = strcat(rb,dec2bin(j-2,log2(szA)));
                rs = strcat(rs,num2str(j-2));
                break
            end
        end
    end
end
```

```
61
62 %% Generates a classical database from the reference genome rg with entries of size szss
63
64 function cdb = prepCdb(rg,szss)
65     cdb = [];
66     for i = 1:size(rg,2)-szss+1
67         cdb = [num2str(cdb); num2str(rg(i:i+szss-1))];
68     end
69 end
70
71 %% Initialises the quantum database
72
73 function state = prepQdb(db,v)
74     n = size(db,2);                         % Size of search patterns
75     m = size(db,1);                         % Number of database entries
76     numq = n*2+1;                           % Number of qubits required for state preparation
77     state = [1; zeros(2^numq-1,1)];
78     X = [0 1; 1 0];
79     for p = m:-1:1                          % For each pattern to be stored
80         zp = db(m-p+1,:);
81         % FLIP ==========================================
82         for j = 1:n
83             zp1 = '';
84             if p == m
85                 zp1 = num2str(dec2bin(0,n));
86             else
87                 zp1 = db(m-p,:);
88             end
89             if zp(j) ~= zp1(j)
90                 state = U1(X,0,state);
91                 state = U_CX(0,numq-j,state);
92                 state = U1(X,0,state);
93             end
94         end
95         state = U1(X,0,state);
96         state = U_CX(0,1,state);
97         state = U1(X,0,state);
98         % end FLIP ======================================
99         state = U_Ssp(v(m-p+1),p,1,0,state);
100        % AND ===========================================
101        if(zp(2) == '0' && zp(1) == '0')
102            state = U1(X,numq-1,state);
103            state = U1(X,numq-2,state);
104            state = U_Tf(numq-1,numq-2,numq-n-1,state);
105            state = U1(X,numq-2,state);
106            state = U1(X,numq-1,state);
107        elseif(zp(2) == '0' && zp(1) == '1')
108            state = U1(X,numq-2,state);
109            state = U_Tf(numq-1,numq-2,numq-n-1,state);
110            state = U1(X,numq-2,state);
111        elseif(zp(2) == '1' && zp(1) == '0')
112            state = U1(X,numq-1,state);
113            state = U_Tf(numq-1,numq-2,numq-n-1,state);
114            state = U1(X,numq-1,state);
115        else
116            state = U_Tf(numq-1,numq-2,numq-n-1,state);
117        end
118        for k = 3:n
119            if(zp(k) == '0')
120                state = U1(X,numq-k,state);
121                state = U_Tf(numq-k,numq-1-n-(k-3),numq-1-n-(k-3)-1,state);
122                state = U1(X,numq-k,state);
123            else
124                state = U_Tf(numq-k,numq-1-n-(k-3),numq-1-n-(k-3)-1,state);
125            end
126        end
127        % end AND =======================================
128        state = U_CX(2,1,state);
129        % AND+ ==========================================
130        for k = n:-1:3
131            if(zp(k) == '0')
```

```
132                 state = U1(X,numq−k,state);
133                 state = U_Tf(numq−k,numq−1−n−(k−3),numq−1−n−(k−3)−1,state);
134                 state = U1(X,numq−k,state);
135             else
136                 state = U_Tf(numq−k,numq−1−n−(k−3),numq−1−n−(k−3)−1,state);
137             end
138         end
139         if(zp(2) == '0' && zp(1) == '0')
140             state = U1(X,numq−1,state);
141             state = U1(X,numq−2,state);
142             state = U_Tf(numq−1,numq−2,numq−n−1,state);
143             state = U1(X,numq−2,state);
144             state = U1(X,numq−1,state);
145         elseif(zp(2) == '0' && zp(1) == '1')
146             state = U1(X,numq−2,state);
147             state = U_Tf(numq−1,numq−2,numq−n−1,state);
148             state = U1(X,numq−2,state);
149         elseif(zp(2) == '1' && zp(1) == '0')
150             state = U1(X,numq−1,state);
151             state = U_Tf(numq−1,numq−2,numq−n−1,state);
152             state = U1(X,numq−1,state);
153         else
154             state = U_Tf(numq−1,numq−2,numq−n−1,state);
155         end
156         % end AND+ =====================================
157     end
158     state = U1(X,0,state);              % Uncompute ancilla states
159 end
160
161 %% Single qubit arbitrary unitary U applied on qubit pos of state vector
162
163 function state = U1(U,pos,state)
164     spc = ceil(log2(size(state,1)));
165     U = kron(eye(2^(spc−1−pos)),kron(U,eye(2^(pos))));
166     state = U∗state;
167 end
168
169 %% CNOT gate with posc as the control qubit and post as the target qubit of state vector
170
171 function state = U_CX(posc,post,state)
172     spc = ceil(log2(size(state,1)));
173     prec = 10^5;
174     X = [0 1; 1 0];
175     H = 1/sqrt(2)∗[1 1;1 −1];
176     if (posc > post)
177         I = eye(2^(posc−post));
178         O = zeros(2^(posc−post));
179         br = kron(eye(2^(posc−post−1)),X);
180         CX = [I O;O br];
181         U = kron(eye(2^(spc−posc−1)),kron(CX,eye(2^post)));
182         state = U∗state;
183     else
184         state = U1(H,posc,state);       % Dress with H to flip control
185         state = U1(H,post,state);
186         I = eye(2^(post−posc));
187         O = zeros(2^(post−posc));
188         br = kron(eye(2^(post−posc−1)),X);
189         CX = [I O;O br];
190         U = kron(eye(2^(spc−post−1)),kron(CX,eye(2^posc)));
191         state = U∗state;
192         state = U1(H,post,state);
193         state = U1(H,posc,state);
194     end
195     state = round(prec∗state)/prec;
196 end
197
198 %% Toffoli gate with posc1, posc2 as the control qubits and post as the target qubit of state
        vector
199
200 function state = U_Tf(posc1,posc2,post,state)
201     if(posc1 > posc2)
```

```
202         poscm = posc1;
203         poscl = posc2;
204     else
205         poscm = posc2;
206         poscl = posc1;
207     end
208     spc = ceil(log2(size(state,1)));
209     prec = 10^5;
210     X = [0 1; 1 0];
211     Tf = [eye(6) zeros(6,2); zeros(2,6) X];
212     sv = [0 0 0 0];                          % Make poscl, poscm, post consequetive by SWAP
213     if post > poscm
214         state = U_CX(poscm,post,state);
215         state = U_CX(post,poscm,state);
216         state = U_CX(poscm,post,state);
217         tmp = post;
218         post = poscm;
219         poscm = tmp;
220         sv(1) = 1;
221     end
222     if post > poscl
223         state = U_CX(poscl,post,state);
224         state = U_CX(post,poscl,state);
225         state = U_CX(poscl,post,state);
226         tmp = post;
227         post = poscl;
228         poscl = tmp;
229         sv(2) = 1;
230     end
231     if poscl-post > 1
232         state = U_CX(poscl-1,post,state);
233         state = U_CX(post,poscl-1,state);
234         state = U_CX(poscl-1,post,state);
235         swpidx3 = post;
236         post = poscl-1;
237         sv(3) = 1;
238     end
239     if poscm-poscl > 1
240         state = U_CX(poscl+1,poscm,state);
241         state = U_CX(poscm,poscl+1,state);
242         state = U_CX(poscl+1,poscm,state);
243         swpidx4 = poscm;
244         poscm = poscl+1;
245         sv(4) = 1;
246     end
247     U = kron(eye(2^(spc-(poscl+1)-1)),kron(Tf,eye(2^(poscl-1))));
248     state = U*state;
249     if sv(4) == 1
250         state = U_CX(poscl+1,swpidx4,state);
251         state = U_CX(swpidx4,poscl+1,state);
252         state = U_CX(poscl+1,swpidx4,state);
253         poscm = swpidx4;
254     end
255     if sv(3) == 1
256         state = U_CX(poscl-1,swpidx3,state);
257         state = U_CX(swpidx3,poscl-1,state);
258         state = U_CX(poscl-1,swpidx3,state);
259         post = swpidx3;
260     end
261     if sv(2) == 1
262         state = U_CX(poscl,post,state);
263         state = U_CX(post,poscl,state);
264         state = U_CX(poscl,post,state);
265         tmp = post;
266         post = poscl;
267         poscl = tmp;
268     end
269     if sv(1) == 1
270         state = U_CX(poscm,post,state);
271         state = U_CX(post,poscm,state);
272         state = U_CX(poscm,post,state);
```

```
273         tmp = post;
274         post = poscm;
275         poscm = tmp;
276     end
277     state = round(prec*state)/prec;
278 end
279 
280 %% Custom Unitary for storage of pattern
281 
282 function state = U_Ssp(s,p,posc,post,state)
283     spc = ceil(log2(size(state,1)));
284     prec = 10^5;
285     c = 1/sqrt(p);
286     p1 = sqrt(p-1);
287     p2 = -s;
288     Ryt = c*[p1 p2; -p2 p1];
289     % Ryt = [cos(asin(-1/sqrt(p))) sin(asin(-1/sqrt(p))); -sin(asin(-1/sqrt(p))) cos(asin(-1/
        sqrt(p)))]
290     CRyt = [eye(2) zeros(2); zeros(2) Ryt];
291     if (posc > post)
292         U = kron(eye(2^(spc-posc-1)),kron(CRyt,eye(2^post)));
293         state = U*state;
294     else
295         U = kron(eye(2^(spc-post-1)),kron(CRyt,eye(2^posc)));
296         state = U_CX(posc,post,state);  % Swap control and target
297         state = U_CX(post,posc,state);
298         state = U_CX(posc,post,state);
299         state = U*state;
300         state = U_CX(posc,post,state);
301         state = U_CX(post,posc,state);
302         state = U_CX(posc,post,state);
303     end
304     state = round(prec*state)/prec;
305 end
306 
307 %% Remove uncomputed ancilla from state space to reduce dimension
308 
309 function newstate = rednDimn(state,nd)
310     newstate = zeros(2^nd,1);
311     j = size(state,1)/(2^nd);
312     for i = 1:2^nd
313         newstate(i) = sum(state((i-1)*j+1:i*j));
314     end
315 end
316 
317 %% Use non-unitary search operator for searching the pattern
318 
319 function state = runSrch(ss,state)
320     state = fliplr(state')';
321     prec = 10^5;
322     dim = 2^size(ss,2);
323     Rs = zeros(dim);                    % Create Non-Unitary Search Operator
324     for i = 1:dim
325         for j = 1:dim
326             if i == j & distHmngWC(j-1,ss) >= 1
327                 Rs(dim-i+1,dim-j+1) = 1;
328             elseif distHmngWC(j-1,ss) > distHmngWC(i-1,ss) & distHmngWC(i-1,ss) >= 1
329                 Rs(dim-i+1,dim-j+1) = -1;
330             end
331         end
332     end
333     state = Rs*state;
334     state = state/sqrt(sum(state.^2));  % Re-normalize after non-unitary opration
335     state = round(prec*state)/prec;
336 end
337 
338 %% Calculate Hamming distance (with wildcards)
339 
340 function hdw = distHmngWC(s1,y)
341     s = '';
342     hdw = 0;
```

```
343     x = dec2bin(s1,size(y,2));
344     for i = 1:size(x,2)
345         if x(i) == '?' | y(i) == '?'    % Wildcard character
346             s = strcat(s,'0');          % Don't care matches everything
347         else
348             n1 = uint64(x(i));
349             n2 = uint64(y(i));
350             z = bitxor(n1, n2);
351             hd = 0;
352             for j = 1:64
353                 hd = hd + bitget(z,j);
354             end
355             hdw = hdw+hd;
356             s = strcat(s,num2str(hd));
357         end
358     end
359 
360 end
361 
362 %% Create superposition of solution considering wildcards
363 
364 function srn = remWC(srWC)
365     locWC = strfind(srWC,'?');
366     nWC = size(locWC,2);
367     if nWC == 0
368         srn = bin2dec(srWC);
369         return
370     end
371     sr = [];
372     for i = 1:2^nWC
373         sr = [num2str(sr); srWC];
374         WC = dec2bin(i-1,nWC);
375         for j = 1:nWC
376             sr(i,locWC(j)) = WC(j);
377         end
378     end
379     srn = bin2dec(sr);
380 end
381 
382 %% Displays the quantum state
383 
384 function dispState(state,format)
385     i = size(state,1);
386     spc = ceil(log2(i));
387     for a = 0:i-1
388         if format == 0
389             fprintf(strcat(num2str(state(a+1),'%+f'),'\t|',dec2bin(a,spc),'>\n'))
390         elseif format == 1
391             fprintf(strcat(num2str(state(a+1)),'_'))
392         end
393     end
394     fprintf('\n')
395 end
```

Appendix-4

```
% Reference: Quantum algorithms for pattern matching in genomic sequences - A. Sarkar
% \author: Aritra Sarkar (prince-ph0en1x)
% \project: Quantum-accelerated Genome-sequencing
% \repo: https://gitlab.com/prince-ph0en1x/QaGs

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Quantum Shannon Decomposition to OpenQL code

function Ud = QSD_opql(U,prnt,qbsp)

    %% Decompose U to AB1, CS, AB2

    dim = log2(size(U,1));
    splitpt = 2^(dim-1);
    OU = zeros(splitpt,splitpt);
    I = eye(2);
    % X = AP(-pi/2)*Rx(pi);            % Decomposes to [0 1; 1 0]
    % H = AP(-pi/2)*Ry(pi/2)*Rx(pi);   % Decomposes to 1/sqrt(2)*[1 1;1 -1]
    [L0,L1,cc,ss,R0,R1] = fatCSD(U);

    %% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ STEP 1 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    % Decompose AB2 to V, D, W (lower dimension)
    AB2 = [R0 zeros(size(R0,1),size(R1,2)); zeros(size(R1,1),size(R0,2)) R1];

    U1 = AB2(1:splitpt,1:splitpt);
    U2 = AB2(splitpt+1:end,splitpt+1:end);

    if (max(max(abs(U1+U2))) < 1e-10 || max(max(abs(U1-U2))) < 1e-10)
        if (dim > 2)
            'Not supported'            % TBD: How this section behaves for dim > 2
        end
        [delta,alpha,theta,beta] = zyz(U1);
        decomposedU1 = Rz(alpha)*Ry(theta)*Rz(beta);
        if (prnt > 1)
            fprintf('k.rz(%d,%f)\n',qbsp(1),-beta);
            fprintf('k.ry(%d,%f)\n',qbsp(1),-theta);
            fprintf('k.rz(%d,%f)\n',qbsp(1),-alpha);
        end
        if (prnt == 2)
            decomposedU1 = AP(delta)*decomposedU1;
            fprintf('ap q%d,%f\n',qbsp(1),-delta);
        end
        decomposedAB2 = kron(I,decomposedU1);
        if (max(max(U1+U2)) < 1e-10)
            decomposedAB2 = kron(Rz(pi),I)*decomposedAB2;
            if (prnt > 1)
                fprintf('k.rz(%d,%f)\n',qbsp(2),-pi);
            end
            if (prnt == 2)
                decomposedAB2 = kron(AP(-pi/2),I)*decomposedAB2;
                fprintf('ap q%d,%f\n',qbsp(2),pi/2);
            end
        end
    else
        [v,d,~] = eig(U1*U2');
        V = v;
        if dim == 4
            % 1 & 2 Eigenvalues in d are repeated, thus V*V' is not I. Adjustment needed
            % https://nl.mathworks.com/matlabcentral/answers/214557-eigenvectors-are-not-
    orthogonal-for-some-skew-symmetric-matrices-why
            V(:,[1,2]) = orth(V(:,[1,2]));  % TBD: Automate this
        end
```

```
62        D = sqrtm(d);
63        W = D*V'*U2;
64        decomposedAB2 = [V OU; OU V]*[D OU; OU D']*[W OU; OU W];
65
66        if (size(W,1) == 2)
67            decomposedW = QSD_opql2(W,prnt,qbsp);
68        else
69            decomposedW = QSD_opql(W,prnt,qbsp(1:end-1));
70        end
71
72        ar = genMk(dim-1)\(2*log(diag(D))/1i);
73        dd = eye(size(AB2,1));
74        for i = 1:size(D,1)
75            if (i == size(D,1))
76                posc = dim-2;
77            else
78                [~,idx] = find(sprintf(dec2bin(bin2gray(i-1),dim-1)) ~= sprintf(dec2bin(
    bin2gray(i),dim-1)),1);
79                posc = dim-2 - (idx - 1);
80            end
81            dd = U_CX(posc,dim-1,dim) * kron(Rz(ar(i)),eye(2^(dim-1))) * dd;
82            if (prnt > 1)
83                fprintf('k.rz(%d,%f)\n',qbsp(end),-ar(i));
84                fprintf('k.gate("cnot",[%d,%d])\n',posc,dim-1);
85            end
86        end
87        decomposedD = dd;
88
89        if (size(V,1) == 2)
90            decomposedV = QSD_opql2(V,prnt,qbsp);
91        else
92            decomposedV = QSD_opql(V,prnt,qbsp(1:end-1));
93        end
94
95        decomposedAB2 = kron(I,decomposedV)*decomposedD*kron(I,decomposedW);
96    end
97
98    %% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ STEP 2 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
99    % Decompose CS to Ry, CX
100   CS = [cc ss; -ss cc];              % Property Test: cc^2 + ss^2 = eye(size(cc,1))
101
102   tr = genMk(dim-1)\(2*asin(diag(ss)));
103   decomposedCS = eye(size(CS,1));
104   for i = 1:size(ss,1)
105       if (i == size(ss,1))
106           posc = dim-2;
107       else
108           [~,idx] = find(sprintf(dec2bin(bin2gray(i-1),dim-1)) ~= sprintf(dec2bin(bin2gray(
    i),dim-1)),1);
109           posc = dim-2 - (idx - 1);
110       end
111       decomposedCS = kron(Ry(tr(i)),eye(2^(dim-1))) * decomposedCS;
112       decomposedCS = U_CX(posc,dim-1,dim) * decomposedCS;
113       if (prnt > 1)
114           fprintf('k.ry(%d,%f)\n',qbsp(end),-tr(i));
115           fprintf('k.gate("cnot",[%d,%d])\n',posc,dim-1);
116       end
117   end
118
119   %% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ STEP 3 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
120   % Decompose AB1 to V, D, W (lower dimension)
121   AB1 = [L0 zeros(size(L0,1),size(L1,2)); zeros(size(L1,1),size(L0,2)) L1];
122
123   U1 = AB1(1:splitpt,1:splitpt);
124   U2 = AB1(splitpt+1:end,splitpt+1:end);
125
126   if (max(max(abs(U1+U2))) < 1e-10 || max(max(abs(U1-U2))) < 1e-10)
127       if (dim > 2)
128           'Not supported'            % TBD: How this section behaves for dim > 2
129       end
130       [delta,alpha,theta,beta] = zyz(U1);
```

```
        decomposedU1 = Rz(alpha)*Ry(theta)*Rz(beta);
        if (prnt > 1)
            fprintf('k.rz(%d,%f)\n',qbsp(1),-beta);
            fprintf('k.ry(%d,%f)\n',qbsp(1),-theta);
            fprintf('k.rz(%d,%f)\n',qbsp(1),-alpha);
        end
        if (prnt == 2)
            decomposedU1 = AP(delta)*decomposedU1;
            fprintf('ap q%d,%f\n',qbsp(1),-delta);
        end
        decomposedAB1 = kron(I,decomposedU1);
        if (max(max(U1+U2)) < 1e-10)
            decomposedAB1 = kron(Rz(pi),I)*decomposedAB1;
            if (prnt > 1)
                fprintf('k.rz(%d,%f)\n',qbsp(2),-pi);
            end
            if (prnt == 2)
                decomposedAB1 = kron(AP(-pi/2),I)*decomposedAB1;
                fprintf('ap q%d,%f\n',qbsp(2),pi/2);
            end
        end
    else
        [v,d,~] = eig(U1*U2');
        V = v;
        if dim == 4
            % 7 & 8 Eigenvalues in d are repeated, thus V*V' is not I. Adjustment needed
            % https://nl.mathworks.com/matlabcentral/answers/214557-eigenvectors-are-not-orthogonal-for-some-skew-symmetric-matrices-why
            V(:,[7,8]) = orth(V(:,[7,8]));  % TBD: Automate this
        end
        D = sqrtm(d);
        W = D*V'*U2;
        decomposedAB1 = [V OU; OU V]*[D OU; OU D']*[W OU; OU W];

        if (size(W,1) == 2)
            decomposedW = QSD_opql2(W,prnt,qbsp);
        else
            decomposedW = QSD_opql(W,prnt,qbsp(1:end-1));
        end

        ar = genMk(dim-1)\(2*log(diag(D))/1i);
        dd = eye(size(AB1,1));
        for i = 1:size(D,1)
            if (i == size(D,1))
                posc = dim-2;
            else
                [~,idx] = find(sprintf(dec2bin(bin2gray(i-1),dim-1)) ~= sprintf(dec2bin(bin2gray(i),dim-1)),1);
                posc = dim-2 - (idx - 1);
            end
            dd = U_CX(posc,dim-1,dim) * kron(Rz(ar(i)),eye(2^(dim-1))) * dd;
            if (prnt > 1)
                fprintf('k.rz(%d,%f)\n',qbsp(end),-ar(i));
                fprintf('k.gate("cnot",[%d,%d])\n',posc,dim-1);
            end
        end
        decomposedD = dd;

        if (size(V,1) == 2)
            decomposedV = QSD_opql2(V,prnt,qbsp);
        else
            decomposedV = QSD_opql(V,prnt,qbsp(1:end-1));
        end

        decomposedAB1 = kron(I,decomposedV)*decomposedD*kron(I,decomposedW);
    end

    %% Final Decomposition Testing
    % Ud = AB1*CS*AB2;
    % Ud = decomposedAB1 * decomposedCS * decomposedAB2;
end
```

```
200
201 %% Decompose 2x2 unitaries using ZYZ
202
203 function decomposedU2 = QSD_opql2(U2,prnt,qbsp)
204     if (isequal(U2,eye(2)))
205         decomposedU2 = eye(2);
206     else
207         [delta,alpha,theta,beta] = zyz(U2);
208         decomposedU2 = Rz(alpha)*Ry(theta)*Rz(beta);
209         if (prnt > 1)
210             fprintf('k.rz(%d,%f)\n',qbsp(1),-beta);
211             fprintf('k.ry(%d,%f)\n',qbsp(1),-theta);
212             fprintf('k.rz(%d,%f)\n',qbsp(1),-alpha);
213         end
214         if (prnt == 2)
215             decomposedU2 = AP(delta)*decomposedU2;
216             fprintf('ap q%d,%f\n',qbsp(1),-delta);
217         end
218     end
219 end
220
221 %% Arbitrary phase unitary operation for comparing exact matrix values
222
223 function U = AP(t)
224     U = [exp(1i*t) 0; 0 exp(1i*t)];
225 end
226
227 %% Rotation about Y-axis
228
229 function U = Ry(t)
230     U = [cos(t/2) sin(t/2); -sin(t/2) cos(t/2)];
231 end
232
233 %% Rotation about Z-axis
234
235 function U = Rz(t)
236     U = [exp(1i*t/2) 0; 0 exp(-1i*t/2)];    % TBD: Check sign of matrix for convention
237 end
238
239 %% Convert binary number to gray code
240
241 function num = bin2gray(num)
242     num = bitxor(num,bitshift(num,-1));
243 end
244
245 %% ZYZ decomposition of a unitary matrix
246
247 function [delta,alpha,theta,beta] = zyz(U)
248     delta = atan2(imag(det(U)),real(det(U)))/size(U,1);
249     SU = U/exp(1i*delta);
250     A = SU(1,1);
251     B = SU(1,2);
252     cw = real(A);
253     wx = imag(B);
254     wy = real(B);
255     wz = imag(A);
256     sw = sqrt(wx^2 + wy^2 + wz^2);
257     wx = wx/sw;
258     wy = wy/sw;
259     wz = wz/sw;
260     t1 = atan2(wz*sw,cw);
261     t2 = atan2(wx,wy);
262     alpha = t1 + t2;
263     beta = t1 - t2;
264     theta = 2*atan2(sw*sqrt(wx^2 + wy^2),sqrt(cw^2 + (wz*sw)^2));
265 end
266
267 %% Cosine-Sine decomposition
268 % Source: http://www.ar-tiste.com/m-fun/csd_qc.m
269
270 function [L0,L1,cc,ss,R0,R1] = fatCSD(U)
```

```
271
272 % This function performs a special case of fat CSD;
273 % namely, the case that has been found useful in quantum computing,
274 % wherein the matrix U being decomposed
275 % is a 2^n dimensional unitary matrix,
276 % and we partition U into four square matrices of the same size.
277 % This function calls csd() and is a trivial extension of it.
278 % csd() performs thin CSD.
279
280 % U = [U00, U01] = [L0    ][ cc  ss][R0    ]
281 %     [U10, U11]   [    L1][-ss  cc][    R1]
282 %
283 % Thin version of CSD (performed by csd()) gives
284 % cc,ss, LO, L1 and R0, but
285 % it doesn't give R1.
286 % This subroutine calls csd() and then calculates R1
287
288     %ns = number of states
289     %nb = number of bits
290     ns = size(U,1);
291     nb = 0;
292     k = 1;
293     while (k<ns)
294         nb = nb+1;
295         k = k*2;
296     end
297     if (k~=ns)
298         error('dimension of input matrix for csd_qc is not power of 2');
299     end
300     if (k==1)
301         error('dimension of input matrix for csd_qc is 1');
302     end
303
304     nsh = ns/2; %ns half
305     U00 = U(1:nsh, 1:nsh);
306     U10 = U(nsh+1:ns, 1:nsh);
307
308     [L0,L1,R0,cc,ss] = thinCSD(U00,U10);
309     R0 = R0';
310     ss = -ss;
311
312     R1 = zeros(nsh, nsh);
313     for j=1:nsh
314         if abs(ss(j,j))>abs(cc(j,j))
315             U01 = U(1:nsh, nsh+1:ns);
316             tmp = (L0'*U01);
317             R1(j,:) = tmp(j,:)/ss(j,j);
318         else
319             U11 = U(nsh+1:ns, nsh+1:ns);
320             tmp = (L1'*U11);
321             R1(j,:) = tmp(j,:)/cc(j,j);
322         end
323     end
324 end
325
326 function [u1,u2,v,c,s]=thinCSD(q1,q2)
327
328 % Given Q1 and Q2 such that Q1'* Q1 + Q2'* Q2 = I, the
329 % C-S Decomposition is a joint factorization of the form
330 %       Q1 = U1*C*V' and Q2=U2*S*V'
331 % where U1,U2,V are orthogonal matrices and C and S are diagonal
332 % matrices (not necessarily square) satisfying
333 %           C'* C + S'* S = I
334 % The diagonal entries of C and S are nonnegative and the
335 % diagonal elements of C are in nondecreasing order.
336 % The matrix Q1 cannot have more columns than rows.
337 % ( Submitted by S. J. Leon )
338
339     [m,n]=size(q1);
340     [p,n]=size(q2);
341     [u1,c,v]=svd(q1);
```

```
342     z=eye(n);z=hankel(z(:,n));
343     c(1:n,:)=z*c(1:n,:)*z;u1(:,1:n)=u1(:,1:n)*z;v=v*z;
344     q2=q2*v;
345     k=1;
346     for j=2:n
347        if c(j,j)<=1/sqrt(2)
348           k=j;
349        end
350     end
351     b=q2(:,1:k);
352     [u2,r]=qr(b);
353     s=u2'*q2;
354     t=min(p,n);tt=min(m,p);
355     if k<t
356        r2=s(k+1:p,k+1:t);
357        [ut,ss,vt]=svd(r2);
358        s(k+1:p,k+1:t)=ss;
359        c(:,k+1:t)=c(:,k+1:t)*vt;
360        u2(:,k+1:p)=u2(:,k+1:p)*ut;
361        v(:,k+1:t)=v(:,k+1:t)*vt;
362        w=c(k+1:tt,k+1:t);
363        [z,r]=qr(w);
364        c(k+1:tt,k+1:t)=r;
365        u1(:,k+1:tt)=u1(:,k+1:tt)*z;
366     end
367     for j=1:n
368        if c(j,j)<0
369           c(j,j)=-c(j,j);
370           u1(:,j)=-u1(:,j);
371        end
372     end
373     for j=1:t
374        if s(j,j)<0
375           s(j,j)=-s(j,j);
376           u2(:,j)=-u2(:,j);
377        end
378     end
379  end
```

Appendix-5

```
1  % Reference: Quantum associative memory with improved distributed queries - J.P.T. Njafa, S.G
     .N. Engo, P. Woafo
2  % Reference: Quantum algorithms for pattern matching in genomic sequences - A. Sarkar
3  % \author: Aritra Sarkar (prince-ph0en1x)
4  % \project: Quantum-accelerated Genome-sequencing
5  % \repo: https://gitlab.com/prince-ph0en1x/QaGs
6
7  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
8
9  function mtlb_qam_a4_idq()
10     close all
11     clear all
12     clc
13
14     AS = {'A','C','G','T'};      % Alphabet set {0,1,2,3} := {A,C,G,T} for DNA Nucleotide
     bases
15     A = size(AS,2);              % Alphabet size
16     Qa = ceil(log2(A));          % Number of qubits to encode a character of the alphabet
17
18     R = 'TTGTCTAGGCGACCA';
19     N = size(R,2);               % Reference genome size
20     M = 2;                       % Short read size
21     P = 'AA';                    % Search pattern (always a series of A, due to minimal
     Hamming distance as the query center)
22     Pb = '0000';                 % Binary encoding for P
23     Qd = Qa*M;                   % Number of qubits to encode the quantum genomic database
24     SS = 2^Qd;                   % State space
25
26     qbodq = 0.25;                % q for the Binomial distribution for distributed query
27     bp = ones(1,SS);
28     for i = 1:SS
29         hd = sum(sprintf('%s',dec2bin(i-1,Qd)) ~= Pb);
30         bp(i) = sqrt((qbodq^(hd))*((1-qbodq)^(Qd-hd)));
31     end
32     plot([0:SS-1],bp,'v-.b')
33     hold on
34     BO = eye(SS) - 2*bp'*bp;
35     %maxerrabs = max(max(abs(BO*BO')-abs(eye(SS)))) % Check if Unitary
36
37     BOD = QSD_opql(BO,3,[0:Qd-1]);  % Arg2 : 1 - no qasm, no AP; 2 - qasm, AP; 3+ - qasm, no
     AP
38     maxerrabs = max(max(abs(BOD)-abs(BO)))          % Check decomposition error
39
40     %% ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ TEST ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
41
42     s = ones(1,16);
43     s(1) = 0;   % AA
44     s(4) = 0;   % AT
45     s(16) = 3;  % TT
46     s = sqrt(s/SS);                  % Prepare initial state
47     plot([0:SS-1],abs(s).^2,'^-.r')
48
49     s = (BOD*s')';               % Distributed Query
50     s = -s + 2*mean(s);          % Diffuse
51     s = -s;                      % Memorised Oracle
52     s(1) = -s(1);
53     s(4) = -s(4);
54     s = -s + 2*mean(s);          % Diffuse
55     plot([0:SS-1],abs(s).^2,'s-m')
56     axis([0 SS-1 0 1])
57     legend('Distributed Query','Quantum Memory','Final State')
58     xlabel('State')
```

```
59     ylabel('Probability')
60 end
61
62 %% Generates a classical database from the reference genome rg with entries of size szss
63
64 function cdb = prepCdb(rg,szss)
65     cdb = [];
66     for i = 1:size(rg,2)-szss+1
67         cdb = [num2str(cdb); num2str(rg(i:i+szss-1))];
68     end
69 end
```

Appendix-6

```
## Reference: Quantum associative memory with improved distributed queries - J.P.T. Njafa, S.
    G.N. Engo, P. Woafo
## Reference: Quantum algorithms for pattern matching in genomic sequences - A. Sarkar
## \author: Aritra Sarkar (prince-ph0en1x)
## \project: Quantum-accelerated Genome-sequencing
## \repo: https://gitlab.com/prince-ph0en1x/QaGs

##############################################################################################

from openql import openql as ql
import qxelarator

import random
from math import *
import os
import re
import math
import matplotlib.pyplot as plt
import numpy as np

##############################################################################################

def randStr(szA,sz):
    # Generates a random string of length 'sz' over the alphabet of size 'szA' in decimal
    bias = 1/szA                    # IMPROVE: add bias here
    rbs = ""
    for i in range(0,sz):
        rn = random.random()
        for j in range(0,szA):
            if rn < (j+1)*bias:
                rbs = rbs + str(j)  # IMPROVE: BCD version
                break
    return rbs

##############################################################################################

AS = {'00','01','10','11'}  # Alphabet set {0,1,2,3} := {A,C,G,T} for DNA Nucleotide bases
A = len(AS)                 # Alphabet size
N = 16                      # Reference Genome size
w = "0033231302212011"      # Reference Genome: "ATTGTCTAGGCGACCA"
M = 2                       # Short Read size
p = "10"                    # Short Read: "AA"

Q_A = ceil(log2(A))         # Number of qubits to encode one character
Q_D = Q_A * M               # Number of data qubits
Q_T = ceil(log2(N-M))       # Tag Qubits
Q_anc = 1                   # Number of ancilla qubits
anc = Q_D + Q_T             # Ancilla qubit id
total_qubits = Q_D + Q_T + Q_anc

##############################################################################################

def QAM():
    print(w,p)
    config_fn = os.path.join('gateConfig.json')
    platform = ql.Platform('platform_none', config_fn)
    prog = ql.Program('qam_a4', total_qubits, platform)

    # Kernel 1: Initialization of Quantum Phone Directory
    qk1 = ql.Kernel('QCirc1',platform)
    Circ1(qk1)

```

```
    # Kernel 2: Calculate Hamming Distance
    qk2 = ql.Kernel('QCirc2',platform)
    Circ2(qk2)

    # Kernel 3: Oracle to Mark Hamming Distance of 0
    qk3 = ql.Kernel('QCirc3',platform)
    Circ3(qk3)

    # Kernel 4: Grover Amplitude Amplification
    qk4 = ql.Kernel('QCirc4',platform)
    Circ4(qk4)

    # Kernel 5: Oracle to Mark Memory States
    qk5 = ql.Kernel('QCirc5',platform)
    Circ5(qk5)

    # Kernel 6: Measurement
    qk6 = ql.Kernel('QCirc6',platform)
    Circ6(qk6)

    # Construct Program from Kernels
    prog.add_kernel(qk1)            # Initialise
    prog.add_kernel(qk2)            # Transform to Hamming distance
    prog.add_kernel(qk3)            # Oracle call
    prog.add_kernel(qk4)            # Inversion about mean
    prog.add_kernel(qk5)            # Memory Oracle
    prog.add_kernel(qk4)            # Inversion about mean
    for r in range(0,3):
        prog.add_kernel(qk3)        # Oracle call
        prog.add_kernel(qk4)        # Inversion about mean
    # prog.add_kernel(qk6)          # Uncomment if using measurement based analytics
    prog.compile()
    qx = qxelarator.QX()
    qx.set('test_output/qam_a4.qasm')

    # Result analytics using Internal State Vector
    qx.execute()
    qxopt = qx.get_state()
    isv = [0]*(2**total_qubits)
    ptrn = re.compile('\(([+-]\d+.?\d*)e?(-\d*)?,([+-]\d+.?\d*)e?(-\d*)?\)\s[|]([0-1]*)>')
    for line in qxopt.splitlines():
        mtch = ptrn.search(line)
        if mtch != None:
            ar = float(mtch.group(1))
            if mtch.group(2) != None:
                are = float(mtch.group(2))
                ar = ar * 10**are
            ac = float(mtch.group(3))
            if mtch.group(4) != None:
                ace = float(mtch.group(4))
                ac = ac * 10**ace
            state = int(mtch.group(5),2)
            isv[state] = ar**2 + ac**2
    ploty = isv
    isvt = [0]*(2**4)
    for tagi in range(0,2**total_qubits):
        if isv[tagi] > 0.03:
            ti = format(tagi,'0'+str(total_qubits)+'b')
            isvt[int(ti[:4:-1],2)] = isv[tagi]
    for tagi in range(0,16):
        print("Tag : ", tagi, "\t probability ",round(isvt[tagi],5))
    plt.plot(isv)
    plt.ylabel('Probability')
    plt.xlabel('Index')
    plt.ylim([0,1])
    plt.show()
    return

#############################################################################################

def Circ1(k):
```

```
    for Qi in range(0,total_qubits):# Initialise all qubits to |0> state
        k.prepz(Qi)
    # MSQ : ~~~~ m1a0 m1a1 m0a0 m0a1 t0 t1 t2 t3 ~~~~ : LSQ
    for Qi in range(0,Q_T):         # Uniform superposition of possible starting positions (
    answers)
        k.gate("h",[Qi])
    nc = []
    for ci in range(0,Q_T):
        nc.append(ci)
    for Qi in range(0,N-M+1):
        Qis = format(Qi,'0'+str(Q_T)+'b')
        for Qisi in range(0,Q_T):
            if Qis[Qisi] == '0':
                k.gate("x",[Qisi])
        wMi = w[Qi:Qi+M]
        print([Qis,wMi])
        for wisi in range(0,M):
            wisia = format(int(wMi[wisi]),'0'+str(Q_A)+'b')
            for wisiai in range(0,Q_A):
                if wisia[wisiai] == '1':
                    nCX(k,nc,Q_T+wisi*Q_A+wisiai,anc)
        for Qisi in range(0,Q_T):
            if Qis[Qisi] == '0':
                k.gate("x",[Qisi])
    wMi = p
    for Qi in range(N-M+1,2**Q_T):
        Qis = format(Qi,'0'+str(Q_T)+'b')
        for Qisi in range(0,Q_T):
            if Qis[Qisi] == '0':
                k.gate("x",[Qisi])
        for wisi in range(0,M):
            wisia = format(int(wMi[wisi]),'0'+str(Q_A)+'b')
            for wisiai in range(0,Q_A):
                if wisia[wisiai] == '0':
                    nCX(k,nc,Q_T+wisi*Q_A+wisiai,anc)
        for Qisi in range(0,Q_T):
            if Qis[Qisi] == '0':
                k.gate("x",[Qisi])
    return

##########################################################################################

def Circ2(k):
    for pi in range(0,M):
        ppi = format(int(p[pi]),'0'+str(Q_A)+'b')
        for ppii in range(0,Q_A):
            if ppi[ppii] == '1':
                k.gate("x",[Q_T+pi*Q_A+ppii])
    return

##########################################################################################

def Circ3(k):
    k.rz(4,-2.495565)
    k.ry(4,-1.046381)
    k.rz(4,-0.634842)
    k.rz(5,-1.095612)
    k.gate("cnot",[4,5])
    k.rz(5,-1.574113)
    k.gate("cnot",[4,5])
    k.rz(4,-0.663604)
    k.ry(4,-1.337541)
    k.rz(4,0.663604)
    k.ry(5,2.180982)
    k.gate("cnot",[4,5])
    k.ry(5,0.270123)
    k.gate("cnot",[4,5])
    k.rz(4,-4.560313)
    k.ry(4,-2.345810)
    k.rz(4,0.217847)
    k.rz(5,2.141438)
```

```
203     k.gate("cnot",[4,5])
204     k.rz(5,-0.835336)
205     k.gate("cnot",[4,5])
206     k.rz(4,0.573939)
207     k.ry(4,-1.161760)
208     k.rz(4,-0.573939)
209     k.rz(6,0.265150)
210     k.gate("cnot",[4,6])
211     k.rz(6,0.792581)
212     k.gate("cnot",[5,6])
213     k.rz(6,-1.778400)
214     k.gate("cnot",[4,6])
215     k.rz(6,0.309606)
216     k.gate("cnot",[5,6])
217     k.rz(4,0.867198)
218     k.ry(4,-1.020460)
219     k.rz(4,-0.892549)
220     k.rz(5,-0.978970)
221     k.gate("cnot",[4,5])
222     k.rz(5,-2.042658)
223     k.gate("cnot",[4,5])
224     k.rz(4,-0.771181)
225     k.ry(4,-0.386977)
226     k.rz(4,0.771181)
227     k.ry(5,1.644840)
228     k.gate("cnot",[4,5])
229     k.ry(5,0.504899)
230     k.gate("cnot",[4,5])
231     k.rz(4,0.662837)
232     k.ry(4,-1.694500)
233     k.rz(4,-3.616169)
234     k.rz(5,0.160840)
235     k.gate("cnot",[4,5])
236     k.rz(5,-0.807003)
237     k.gate("cnot",[4,5])
238     k.rz(4,-0.523538)
239     k.ry(4,-1.195233)
240     k.rz(4,0.523538)
241     k.ry(6,1.486933)
242     k.gate("cnot",[4,6])
243     k.ry(6,0.346126)
244     k.gate("cnot",[5,6])
245     k.ry(6,-0.150052)
246     k.gate("cnot",[4,6])
247     k.ry(6,0.589344)
248     k.gate("cnot",[5,6])
249     k.rz(4,-4.033427)
250     k.ry(4,-2.661051)
251     k.rz(4,-0.217231)
252     k.rz(5,-0.679055)
253     k.gate("cnot",[4,5])
254     k.rz(5,1.623385)
255     k.gate("cnot",[4,5])
256     k.rz(4,-1.573731)
257     k.ry(4,-0.475009)
258     k.rz(4,1.573731)
259     k.ry(5,2.071674)
260     k.gate("cnot",[4,5])
261     k.ry(5,0.764902)
262     k.gate("cnot",[4,5])
263     k.rz(4,1.904202)
264     k.ry(4,-1.576334)
265     k.rz(4,3.300418)
266     k.rz(5,0.246450)
267     k.gate("cnot",[4,5])
268     k.rz(5,1.446510)
269     k.gate("cnot",[4,5])
270     k.rz(4,-3.121304)
271     k.ry(4,-1.150659)
272     k.rz(4,3.121304)
273     k.rz(6,-0.568080)
```

```
274     k.gate("cnot",[4,6])
275     k.rz(6,-0.678844)
276     k.gate("cnot",[5,6])
277     k.rz(6,-0.184890)
278     k.gate("cnot",[4,6])
279     k.rz(6,1.600579)
280     k.gate("cnot",[5,6])
281     k.rz(4,2.234130)
282     k.ry(4,-2.090960)
283     k.rz(4,1.062383)
284     k.rz(5,0.164323)
285     k.gate("cnot",[4,5])
286     k.rz(5,1.743454)
287     k.gate("cnot",[4,5])
288     k.rz(4,-1.629366)
289     k.ry(4,-0.512242)
290     k.rz(4,1.629366)
291     k.ry(5,1.152369)
292     k.gate("cnot",[4,5])
293     k.ry(5,0.797232)
294     k.gate("cnot",[4,5])
295     k.rz(4,-0.651534)
296     k.ry(4,-2.240981)
297     k.rz(4,2.205052)
298     k.rz(5,1.091160)
299     k.gate("cnot",[4,5])
300     k.rz(5,0.777808)
301     k.gate("cnot",[4,5])
302     k.rz(4,2.741933)
303     k.ry(4,-1.561898)
304     k.rz(4,-2.741933)
305     k.rz(7,-0.392699)
306     k.gate("cnot",[4,7])
307     k.rz(7,0.857984)
308     k.gate("cnot",[5,7])
309     k.rz(7,-0.392699)
310     k.gate("cnot",[4,7])
311     k.rz(7,-0.730732)
312     k.gate("cnot",[6,7])
313     k.rz(7,-1.070810)
314     k.gate("cnot",[4,7])
315     k.rz(7,-0.849024)
316     k.gate("cnot",[5,7])
317     k.rz(7,0.285412)
318     k.gate("cnot",[4,7])
319     k.rz(7,-0.849024)
320     k.gate("cnot",[6,7])
321     k.rz(4,2.422630)
322     k.ry(4,-1.054241)
323     k.rz(4,-0.670996)
324     k.rz(5,-0.346171)
325     k.gate("cnot",[4,5])
326     k.rz(5,-2.395018)
327     k.gate("cnot",[4,5])
328     k.rz(4,-2.330937)
329     k.ry(4,-1.254475)
330     k.rz(4,2.330937)
331     k.ry(5,1.739765)
332     k.gate("cnot",[4,5])
333     k.ry(5,0.618025)
334     k.gate("cnot",[4,5])
335     k.rz(4,-2.169652)
336     k.ry(4,-2.868867)
337     k.rz(4,2.683293)
338     k.rz(5,0.126237)
339     k.gate("cnot",[4,5])
340     k.rz(5,-1.920065)
341     k.gate("cnot",[4,5])
342     k.rz(4,-1.083853)
343     k.ry(4,-1.399634)
344     k.rz(4,1.083853)
```

```
    k.rz(6,0.040519)
    k.gate("cnot",[4,6])
    k.rz(6,0.810583)
    k.gate("cnot",[5,6])
    k.rz(6,1.374535)
    k.gate("cnot",[4,6])
    k.rz(6,0.527623)
    k.gate("cnot",[5,6])
    k.rz(4,-5.305886)
    k.ry(4,-0.764960)
    k.rz(4,0.785715)
    k.rz(5,-0.264883)
    k.gate("cnot",[4,5])
    k.rz(5,1.987388)
    k.gate("cnot",[4,5])
    k.rz(4,1.990355)
    k.ry(4,-0.392324)
    k.rz(4,-1.990355)
    k.ry(5,1.862818)
    k.gate("cnot",[4,5])
    k.ry(5,0.827526)
    k.gate("cnot",[4,5])
    k.rz(4,-1.779818)
    k.ry(4,-1.301020)
    k.rz(4,-1.688079)
    k.rz(5,0.417135)
    k.gate("cnot",[4,5])
    k.rz(5,-1.817636)
    k.gate("cnot",[4,5])
    k.rz(4,-0.537353)
    k.ry(4,-0.105530)
    k.rz(4,0.537353)
    k.ry(6,1.025571)
    k.gate("cnot",[4,6])
    k.ry(6,0.386772)
    k.gate("cnot",[5,6])
    k.ry(6,0.325953)
    k.gate("cnot",[4,6])
    k.ry(6,0.964752)
    k.gate("cnot",[5,6])
    k.rz(4,0.589048)
    k.ry(4,-2.364848)
    k.rz(4,4.476448)
    k.rz(5,-0.862498)
    k.gate("cnot",[4,5])
    k.rz(5,-1.949294)
    k.gate("cnot",[4,5])
    k.rz(4,-0.830300)
    k.ry(4,-0.962399)
    k.rz(4,0.830300)
    k.ry(5,1.652079)
    k.gate("cnot",[4,5])
    k.ry(5,0.567936)
    k.gate("cnot",[4,5])
    k.rz(4,-3.400688)
    k.ry(4,-0.392400)
    k.rz(4,2.556896)
    k.rz(5,-1.608724)
    k.gate("cnot",[4,5])
    k.rz(5,0.535112)
    k.gate("cnot",[4,5])
    k.rz(4,-2.631783)
    k.ry(4,-1.010089)
    k.rz(4,2.631783)
    k.rz(6,-0.749385)
    k.gate("cnot",[4,6])
    k.rz(6,-0.995045)
    k.gate("cnot",[5,6])
    k.rz(6,-0.425068)
    k.gate("cnot",[4,6])
    k.rz(6,1.656585)
```

```
416     k.gate("cnot",[5,6])
417     k.rz(4,-1.259555)
418     k.ry(4,-1.413129)
419     k.rz(4,-0.376948)
420     k.rz(5,-0.023320)
421     k.gate("cnot",[4,5])
422     k.rz(5,0.375551)
423     k.gate("cnot",[4,5])
424     k.rz(4,-0.124097)
425     k.ry(4,-0.211290)
426     k.rz(4,0.124097)
427     k.ry(5,1.630461)
428     k.gate("cnot",[4,5])
429     k.ry(5,0.680158)
430     k.gate("cnot",[4,5])
431     k.rz(4,-5.829541)
432     k.ry(4,-2.443439)
433     k.rz(4,0.018165)
434     k.rz(5,0.242793)
435     k.gate("cnot",[4,5])
436     k.rz(5,-1.876220)
437     k.gate("cnot",[4,5])
438     k.rz(4,1.413556)
439     k.ry(4,-0.974064)
440     k.rz(4,-1.413556)
441     k.ry(7,0.261799)
442     k.gate("cnot",[4,7])
443     k.ry(7,0.261799)
444     k.gate("cnot",[5,7])
445     k.ry(7,0.261799)
446     k.gate("cnot",[4,7])
447     k.ry(7,0.261799)
448     k.gate("cnot",[6,7])
449     k.ry(7,0.261799)
450     k.gate("cnot",[4,7])
451     k.ry(7,0.261799)
452     k.gate("cnot",[5,7])
453     k.ry(7,0.261799)
454     k.gate("cnot",[4,7])
455     k.ry(7,0.261799)
456     k.gate("cnot",[6,7])
457     k.rz(4,3.477399)
458     k.ry(4,-1.335230)
459     k.rz(4,1.300499)
460     k.rz(5,1.438258)
461     k.gate("cnot",[4,5])
462     k.rz(5,-0.801232)
463     k.gate("cnot",[4,5])
464     k.rz(4,-1.104663)
465     k.ry(4,-1.513998)
466     k.rz(4,1.104663)
467     k.ry(5,1.300200)
468     k.gate("cnot",[4,5])
469     k.ry(5,0.817306)
470     k.gate("cnot",[4,5])
471     k.rz(4,-0.537000)
472     k.ry(4,-2.102438)
473     k.rz(4,-1.622373)
474     k.rz(5,-0.525194)
475     k.gate("cnot",[4,5])
476     k.rz(5,0.334422)
477     k.gate("cnot",[4,5])
478     k.rz(4,0.438354)
479     k.ry(4,-1.454119)
480     k.rz(4,-0.438354)
481     k.rz(6,-0.785398)
482     k.gate("cnot",[4,6])
483     k.rz(6,-0.085948)
484     k.gate("cnot",[5,6])
485     k.rz(6,-0.785398)
486     k.gate("cnot",[4,6])
```

```
487     k.rz(6,-1.484849)
488     k.gate("cnot",[5,6])
489     k.rz(4,3.350364)
490     k.ry(4,-2.676777)
491     k.rz(4,2.930095)
492     k.rz(5,0.954045)
493     k.gate("cnot",[4,5])
494     k.rz(5,0.745842)
495     k.gate("cnot",[4,5])
496     k.rz(4,2.944517)
497     k.ry(4,-1.291884)
498     k.rz(4,-2.944517)
499     k.ry(5,1.438536)
500     k.gate("cnot",[4,5])
501     k.ry(5,0.852927)
502     k.gate("cnot",[4,5])
503     k.rz(4,1.249836)
504     k.ry(4,-2.002135)
505     k.rz(4,-2.140359)
506     k.rz(5,-0.086264)
507     k.gate("cnot",[4,5])
508     k.rz(5,-0.840127)
509     k.gate("cnot",[4,5])
510     k.rz(4,-2.964859)
511     k.ry(4,-1.544692)
512     k.rz(4,2.964859)
513     k.ry(6,1.614267)
514     k.gate("cnot",[4,6])
515     k.ry(6,0.309438)
516     k.gate("cnot",[5,6])
517     k.ry(6,-0.036398)
518     k.gate("cnot",[4,6])
519     k.ry(6,1.254285)
520     k.gate("cnot",[5,6])
521     k.rz(4,4.735940)
522     k.ry(4,-1.080950)
523     k.rz(4,-1.398253)
524     k.rz(5,0.141644)
525     k.gate("cnot",[4,5])
526     k.rz(5,1.581510)
527     k.gate("cnot",[4,5])
528     k.rz(4,3.000507)
529     k.ry(4,-1.345787)
530     k.rz(4,-3.000507)
531     k.ry(5,2.303325)
532     k.gate("cnot",[4,5])
533     k.ry(5,0.768309)
534     k.gate("cnot",[4,5])
535     k.rz(4,-4.022805)
536     k.ry(4,-2.559101)
537     k.rz(4,1.595668)
538     k.rz(5,-1.365775)
539     k.gate("cnot",[4,5])
540     k.rz(5,-1.372495)
541     k.gate("cnot",[4,5])
542     k.rz(4,-1.209858)
543     k.ry(4,-0.713527)
544     k.rz(4,1.209858)
545     k.rz(6,-0.392699)
546     k.gate("cnot",[4,6])
547     k.rz(6,1.718203)
548     k.gate("cnot",[5,6])
549     k.rz(6,-0.159657)
550     k.gate("cnot",[4,6])
551     k.rz(6,-1.151992)
552     k.gate("cnot",[5,6])
553     k.rz(4,-2.103083)
554     k.ry(4,-1.642351)
555     k.rz(4,-1.494416)
556     k.rz(5,-0.071267)
557     k.gate("cnot",[4,5])
```

```
    k.rz(5,1.653073)
    k.gate("cnot",[4,5])
    k.rz(4,-0.387850)
    k.ry(4,-0.891440)
    k.rz(4,0.387850)
    k.ry(5,0.437715)
    k.gate("cnot",[4,5])
    k.ry(5,0.234582)
    k.gate("cnot",[4,5])
    k.rz(4,-4.048179)
    k.ry(4,-2.212950)
    k.rz(4,-0.379861)
    k.rz(5,1.259357)
    k.gate("cnot",[4,5])
    k.rz(5,-0.726894)
    k.gate("cnot",[4,5])
    k.rz(4,-0.204942)
    k.ry(4,-1.154988)
    k.rz(4,0.204942)
    k.rz(7,-0.000000)
    k.gate("cnot",[4,7])
    k.rz(7,-1.250684)
    k.gate("cnot",[5,7])
    k.rz(7,-0.338033)
    k.gate("cnot",[4,7])
    k.rz(7,0.000000)
    k.gate("cnot",[6,7])
    k.rz(7,0.000000)
    k.gate("cnot",[4,7])
    k.rz(7,-0.221786)
    k.gate("cnot",[5,7])
    k.rz(7,-1.134436)
    k.gate("cnot",[4,7])
    k.rz(7,0.000000)
    k.gate("cnot",[6,7])
    k.rz(4,0.311756)
    k.ry(4,-0.913364)
    k.rz(4,5.950218)
    k.rz(5,1.175158)
    k.gate("cnot",[4,5])
    k.rz(5,0.505238)
    k.gate("cnot",[4,5])
    k.rz(4,0.562177)
    k.ry(4,-0.920386)
    k.rz(4,-0.562177)
    k.ry(5,1.229257)
    k.gate("cnot",[4,5])
    k.ry(5,0.351218)
    k.gate("cnot",[4,5])
    k.rz(4,-0.714317)
    k.ry(4,-1.273729)
    k.rz(4,2.687247)
    k.rz(5,-0.815124)
    k.gate("cnot",[4,5])
    k.rz(5,1.756678)
    k.gate("cnot",[4,5])
    k.rz(4,-1.191286)
    k.ry(4,-1.322072)
    k.rz(4,1.191286)
    k.rz(6,-0.392699)
    k.gate("cnot",[4,6])
    k.rz(6,0.560325)
    k.gate("cnot",[5,6])
    k.rz(6,0.367070)
    k.gate("cnot",[4,6])
    k.rz(6,-1.604025)
    k.gate("cnot",[5,6])
    k.rz(4,3.443813)
    k.ry(4,-0.528964)
    k.rz(4,-2.730648)
    k.rz(5,0.333271)
```

```
629     k.gate("cnot",[4,5])
630     k.rz(5,2.086082)
631     k.gate("cnot",[4,5])
632     k.rz(4,-0.945221)
633     k.ry(4,-0.446899)
634     k.rz(4,0.945221)
635     k.ry(5,1.423641)
636     k.gate("cnot",[4,5])
637     k.ry(5,0.452590)
638     k.gate("cnot",[4,5])
639     k.rz(4,-3.227600)
640     k.ry(4,-0.800360)
641     k.rz(4,0.895195)
642     k.rz(5,0.879420)
643     k.gate("cnot",[4,5])
644     k.rz(5,-1.175390)
645     k.gate("cnot",[4,5])
646     k.rz(4,-2.704350)
647     k.ry(4,-1.291475)
648     k.rz(4,2.704350)
649     k.ry(6,1.783616)
650     k.gate("cnot",[4,6])
651     k.ry(6,0.493806)
652     k.gate("cnot",[5,6])
653     k.ry(6,0.142692)
654     k.gate("cnot",[4,6])
655     k.ry(6,0.538500)
656     k.gate("cnot",[5,6])
657     k.rz(4,0.074716)
658     k.ry(4,-1.215538)
659     k.rz(4,2.366195)
660     k.rz(5,-0.634980)
661     k.gate("cnot",[4,5])
662     k.rz(5,-2.304112)
663     k.gate("cnot",[4,5])
664     k.rz(4,-0.827265)
665     k.ry(4,-1.142645)
666     k.rz(4,0.827265)
667     k.ry(5,1.674571)
668     k.gate("cnot",[4,5])
669     k.ry(5,0.564013)
670     k.gate("cnot",[4,5])
671     k.rz(4,0.557684)
672     k.ry(4,-0.316529)
673     k.rz(4,4.037339)
674     k.rz(5,-1.658384)
675     k.gate("cnot",[4,5])
676     k.rz(5,0.946438)
677     k.gate("cnot",[4,5])
678     k.rz(4,-1.615404)
679     k.ry(4,-1.423146)
680     k.rz(4,1.615404)
681     k.rz(6,0.785398)
682     k.gate("cnot",[4,6])
683     k.rz(6,-1.895989)
684     k.gate("cnot",[5,6])
685     k.rz(6,0.785398)
686     k.gate("cnot",[4,6])
687     k.rz(6,0.325193)
688     k.gate("cnot",[5,6])
689     k.rz(4,3.212053)
690     k.ry(4,-1.388574)
691     k.rz(4,-1.391053)
692     k.rz(5,1.520843)
693     k.gate("cnot",[4,5])
694     k.rz(5,0.942892)
695     k.gate("cnot",[4,5])
696     k.rz(4,1.229895)
697     k.ry(4,-1.476998)
698     k.rz(4,-1.229895)
699     k.ry(5,1.477189)
```

```
700     k.gate("cnot",[4,5])
701     k.ry(5,0.636033)
702     k.gate("cnot",[4,5])
703     k.rz(4,-1.659093)
704     k.ry(4,-2.163702)
705     k.rz(4,-2.151962)
706     k.rz(5,-0.872758)
707     k.gate("cnot",[4,5])
708     k.rz(5,-1.456869)
709     k.gate("cnot",[4,5])
710     k.rz(4,-0.779867)
711     k.ry(4,-1.548794)
712     k.rz(4,0.779867)
713     return
714 
715 ###############################################################################################
716 
717 def Circ4(k):
718     for si in range(0,Q_D+Q_T):
719         k.gate("h",[si])
720         k.gate("x",[si])
721     k.gate("h",[Q_D+Q_T-1])         # CPhase to CNOT conversion
722     nc = []
723     for sj in range(0,Q_D+Q_T-1):
724         nc.append(sj)
725     nCX(k,nc,Q_D+Q_T-1,anc)         # Decompose multi-controlled CNOT
726     k.gate("h",[Q_D+Q_T-1])         # Uncompute CPhase to CNOT conversion
727     for si in range(0,Q_D+Q_T):
728         k.gate("x",[si])
729         k.gate("h",[si])
730     return
731 
732 ###############################################################################################
733 
734 def Circ5(k):
735     nc = []
736     for qsi in range(0,Q_T+Q_D-1):
737         nc.append(qsi)
738     for Qi in range(0,N-M+1):
739         Qis = format(Qi,'0'+str(Q_T)+'b')
740         wMi = w[Qi:Qi+M]
741         wt = Qis
742         for wisi in range(0,M):
743             hd = int(format(int(wMi[wisi]),'0'+str(Q_A)+'b'),2) ^ int(format(int(p[wisi]),'0'
    +str(Q_A)+'b'),2)
744             wisia = format(hd,'0'+str(Q_A)+'b')
745             wt = wt+wisia
746         for Qisi in range(0,Q_T+Q_D):
747             if wt[Qisi] == '0':
748                 k.gate("x",[Qisi])
749             k.gate("h",[Q_D+Q_T-1])         # CPhase to CNOT conversion
750             nCX(k,nc,Q_D+Q_T-1,anc)         # Decompose multi-controlled CNOT
751             k.gate("h",[Q_D+Q_T-1])         # Uncompute CPhase to CNOT conversion
752             if wt[Qisi] == '0':
753                 k.gate("x",[Qisi])
754 
755 ###############################################################################################
756 
757 def Circ6(k):
758     #k.display()
759     for si in range(0,Q_T):         # Measure tag positions
760         k.gate("measure",[si])
761     return
762 
763 ###############################################################################################
764 
765 def nCX(k,c,t,b):
766     nc = len(c)
767     if nc == 1:
768         k.gate("cnot",[c[0], t])
769     elif nc == 2:
```

```
        k.toffoli(c[0],c[1],t)
    else:
        nch = ceil(nc/2)
        c1 = c[:nch]
        c2 = c[nch:]
        c2.append(b)
        nCX(k,c1,b,nch+1)
        nCX(k,c2,t,nch-1)
        nCX(k,c1,b,nch+1)
        nCX(k,c2,t,nch-1)
    return

############################################################################################

if __name__ == '__main__':
    QAM()
```

Bibliography

[1] R. P. Feynman, A. J. Hey, and R. W. Allen, *Feynman lectures on computation* (Perseus Books, 2000).

[2] R. Landauer, *Irreversibility and heat generation in the computing process,* IBM journal of research and development **5**, 183 (1961).

[3] G. E. Moore, *Cramming more components onto integrated circuits,* Proceedings of the IEEE **86**, 82 (1998).

[4] K. Rupp, *42 years of microprocessor trend data | karl rupp,* (2018).

[5] I. L. Markov, *Limits on fundamental limits to computation,* Nature **512**, 147 (2014).

[6] L. Chua, *Resistance switching memories are memristors,* in *Memristor Networks* (Springer, 2014) pp. 21–51.

[7] L. M. Adleman, *Molecular computation of solutions to combinatorial problems,* Science **266**, 1021 (1994).

[8] J. Alama and J. Korbmacher, *The lambda calculus,* in *The Stanford Encyclopedia of Philosophy*, edited by E. N. Zalta (Metaphysics Research Lab, Stanford University, 2018) summer 2018 ed.

[9] M. Cook, *Universality in elementary cellular automata,* Complex systems **15**, 1 (2004).

[10] N. Immerman, *Descriptive complexity* (Springer Science & Business Media, 2012).

[11] S. Aaronson, $P \stackrel{?}{=} NP$, in *Open problems in mathematics* (Springer, 2016) pp. 1–122.

[12] G. J. Chaitin, *Information-theoretic limitations of formal systems,* Journal of the ACM (JACM) **21**, 403 (1974).

[13] A. Kaitchenko, *Algorithms for estimating information distance with application to bioinformatics and linguistics,* in *Electrical and Computer Engineering, 2004. Canadian Conference on*, Vol. 4 (IEEE, 2004) pp. 2255–2258.

[14] K. Panetta, *Top trends in the gartner hype cycle for emerging technologies, 2017,* (2018).

[15] D. P. DiVincenzo *et al.*, *The physical implementation of quantum computation,* arXiv preprint quant-ph/0002077 (2000).

[16] X. Fu, M. Rol, C. Bultink, J. van Someren, N. Khammassi, I. Ashraf, R. Vermeulen, J. De Sterke, W. Vlothuizen, R. Schouten, *et al.*, *An experimental microarchitecture for a superconducting quantum processor,* in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (ACM, 2017) pp. 813–825.

[17] I. Research, *Ibm quantum experience,* (2018).

[18] *Chinese academy of sciences - alibaba quantum computing laboratory,* (2018).

[19] *Quantum infinity,* (2018).

[20] R. C. Inc., *Rigetti,* (2018).

[21] C. H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani, *Strengths and weaknesses of quantum computing,* SIAM journal on Computing **26**, 1510 (1997).

[22] S. Aaronson, *The limits of quantum computers,* Scientific American **298**, 62 (2008).

[23] S. Lloyd, *The universe as quantum computer,* in *A Computable Universe: Understanding and Exploring Nature as Computation* (World Scientific, 2013) pp. 567–581.

[24] M. A. Hamburg and F. S. Collins, *The path to personalized medicine,* New England Journal of Medicine **363**, 301 (2010).

[25] J. R. Miller, S. Koren, and G. Sutton, *Assembly algorithms for next-generation sequencing data,* Genomics **95**, 315 (2010).

[26] *Broad institute gatk best practices pipeline,* (2018).

[27] E. J. Houtgast, V.-M. Sima, K. Bertels, and Z. Al-Ars, *Hardware acceleration of bwa-mem genomic short read mapping for longer read lengths,* Computational biology and chemistry **75**, 54 (2018).

[28] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, *Big data: astronomical or genomical?* PLoS biology **13**, e1002195 (2015).

[29] *The cost of sequencing a human genome,* (2018).

[30] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology* (Cambridge university press, 1997).

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (The MIT Press, 2009).

[32] N. C. Jones and P. Pevzner, *An introduction to bioinformatics algorithms* (MIT press, 2004).

[33] C. Meek, J. M. Patel, and S. Kasetty, *-oasis: An online and accurate technique for local-alignment searches on biological sequences,* in *Proceedings 2003 VLDB Conference* (Elsevier, 2003) pp. 910–921.

[34] P. Ferragina and G. Manzini, *Opportunistic data structures with applications,* in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on* (IEEE, 2000) pp. 390–398.

[35] T. W. Lam, W.-K. Sung, S.-L. Tam, C.-K. Wong, and S.-M. Yiu, *Compressed indexing and local alignment of dna,* Bioinformatics **24**, 791 (2008).

[36] H. Li and R. Durbin, *Fast and accurate long-read alignment with burrows–wheeler transform,* Bioinformatics **26**, 589 (2010).

[37] H. Li, *Exploring single-sample snp and indel calling with whole-genome de novo assembly,* Bioinformatics **28**, 1838 (2012).

[38] J. Commins, C. Toft, and M. A. Fares, *Computational biology methods and their application to the comparative genomics of endocellular symbiotic bacteria of insects,* Biological procedures online **11**, 52 (2009).

[39] P. E. Compeau, P. A. Pevzner, and G. Tesler, *How to apply de bruijn graphs to genome assembly,* Nature biotechnology **29**, 987 (2011).

[40] D. Deutsch, *Quantum theory, the church-turing principle and the universal quantum computer,* in *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, Vol. 400 (The Royal Society, 1985) pp. 97–117.

[41] P. W. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,* SIAM review **41**, 303 (1999).

[42] L. K. Grover, *A fast quantum mechanical algorithm for database search,* in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (ACM, 1996) pp. 212–219.

[43] L. K. Grover, *Quantum mechanics helps in searching for a needle in a haystack,* Physical review letters **79**, 325 (1997).

[44] S. Jordan, *Quantum algorithm zoo,* Available at http://math.nist.gov/quantum/zoo/ (2018).

[45] M. Boyer, G. Brassard, P. Høyer, and A. Tapp, *Tight bounds on quantum searching,* arXiv preprint quant-ph/9605034 (1996).

[46] E. Biham, O. Biham, D. Biron, M. Grassl, and D. A. Lidar, *Grover's quantum search algorithm for an arbitrary initial amplitude distribution,* Physical Review A **60**, 2742 (1999).

[47] M. John, *Sampling with quantum mechanics,* arXiv preprint quant-ph/0306181 (2003).

[48] Y. Aharonov, L. Davidovich, and N. Zagury, *Quantum random walks,* Physical Review A **48**, 1687 (1993).

[49] S. E. Venegas-Andraca, *Quantum walks: a comprehensive review,* Quantum Information Processing **11**, 1015 (2012).

[50] R. Schützhold, *Pattern recognition on a quantum computer,* Physical Review A **67**, 062311 (2003).

[51] D. Curtis and D. A. Meyer, *Towards quantum template matching,* in *Quantum Communications and Quantum Imaging*, Vol. 5161 (International Society for Optics and Photonics, 2004) pp. 134–142.

[52] G. Kuperberg, *A subexponential-time quantum algorithm for the dihedral hidden subgroup problem,* SIAM Journal on Computing **35**, 170 (2005).

[53] A. Montanaro, *Quantum pattern matching fast on average,* Algorithmica **77**, 16 (2017).

[54] J. Roland and N. J. Cerf, *Quantum search by local adiabatic evolution,* Physical Review A **65**, 042308 (2002).

[55] C. Williams, N. Cerf, and L. Grover, *Nested quantum search and np-complete problem,* (1998).

[56] P. Sousa, F. Mendes, and R. Ramos, *Quantum search followed by classical search versus quantum search alone,* arXiv preprint arXiv:1507.08947 (2015).

[57] P. Mateus and Y. Omar, *Quantum pattern matching,* arXiv preprint quant-ph/0508237 (2005).

[58] L. C. Hollenberg, *Fast quantum search algorithms in protein sequence comparisons: Quantum bioinformatics,* Physical Review E **62**, 7532 (2000).

[59] D. Ventura and T. Martinez, *Quantum associative memory,* Information Sciences **124**, 273 (2000).

[60] D. Ventura and T. Martinez, *Quantum associative memory with exponential capacity,* in *Neural Networks Proceedings, 1998. IEEE World Congress on Computational Intelligence. The 1998 IEEE International Joint Conference on*, Vol. 1 (IEEE, 1998) pp. 509–513.

[61] D. Ventura, *Artificial associative memory using quantum processes,* in *Proceedings of the International Conference on Computational Intelligence and Neuroscience*, Vol. 2 (1998) pp. 218–221.

[62] D. Ventura and T. Martinez, *Initializing the amplitude distribution of a quantum state,* Foundations of Physics Letters **12**, 547 (1999).

[63] D. Ventura and T. Martinez, *A quantum associative memory based on grover's algorithm,* in *Artificial Neural Nets and Genetic Algorithms* (Springer, 1999) pp. 22–27.

[64] A. J. Abhari, A. Faruque, M. J. Dousti, L. Svec, O. Catu, A. Chakrabati, C.-F. Chiang, S. Vanderwilt, J. Black, and F. Chong, *Scaffold: Quantum programming language*, Tech. Rep. (PRINCETON UNIV NJ DEPT OF COMPUTER SCIENCE, 2012).

[65] D. Wecker and K. M. Svore, *Liqui|>: A software design architecture and domain-specific language for quantum computing,* arXiv preprint arXiv:1402.4467 (2014).

[66] *Setting up the q# development environment,* (2018).

[67] D. S. Steiger, T. Häner, and M. Troyer, *Projectq: an open source software framework for quantum computing,* Quantum **2**, 49 (2018).

[68] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, *Quipper: a scalable quantum programming language,* in *ACM SIGPLAN Notices*, Vol. 48 (ACM, 2013) pp. 333–342.

[69] I. Rigetti amp; Co, *Rigetti,* (2018).

[70] *cqasm v1.0 towards a common quantum assembly language,* (2018).

[71] *Quantiki: List of qc simulators,* (2018).

[72] N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, and K. Bertels, *Qx: A high-performance quantum computer simulation platform,* in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (IEEE, 2017) pp. 464–469.

[73] R. Jozsa and N. Linden, *On the role of entanglement in quantum-computational speed-up,* in *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, Vol. 459 (The Royal Society, 2003) pp. 2011–2032.

[74] *Quirk: A drag-and-drop quantum circuit simulator,* (2018).

[75] *Meqanic | the amazing quantum computer puzzle game,* (2018).

[76] *Overview page - orcad capture,* (2018).

[77] C. Gidney, *Constructing large controlled nots,* Available at http://algassert.com/circuits/2015/06/05/Constructing-Large-Controlled-Nots.html (2018).

[78] P. J. Coles, S. Eidenbenz, S. Pakin, A. Adedoyin, J. Ambrosiano, P. Anisimov, W. Casper, G. Chennupati, C. Coffrin, H. Djidjev, *et al.*, *Quantum algorithm implementations for beginners,* arXiv preprint arXiv:1804.03719 (2018).

[79] G. Kalai and G. Kindler, *Gaussian noise sensitivity and bosonsampling,* arXiv preprint arXiv:1409.3093 (2014).

[80] *Ncbi genome data viewer,* .

[81] V. V. Shende, S. S. Bullock, and I. L. Markov, *Synthesis of quantum-logic circuits,* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **25**, 1000 (2006).

[82] R. R. Tucci, *A rudimentary quantum compiler (2cnd ed.),* arXiv preprint quant-ph/9902062 (1999).

[83] M. Möttönen, J. J. Vartiainen, V. Bergholm, and M. M. Salomaa, *Quantum circuits for general multiqubit gates,* Physical Review Letters **93**, 130502 (2004).

[84] C. Gidney, *Constructing large controlled nots,* (2018).

[85] A. Ezhov, A. Nifanova, and D. Ventura, *Quantum associative memory with distributed queries,* Information Sciences **128**, 271 (2000).

[86] J.-P. T. Njafa, S. N. Engo, and P. Woafo, *Quantum associative memory with improved distributed queries,* International Journal of Theoretical Physics **52**, 1787 (2013).

[87] R.-G. Zhou, C.-Y. Shen, T.-r. Xiao, and Y.-c. Li, *Quantum pattern search with closed match,* International Journal of Theoretical Physics **52**, 3970 (2013).

[88] J.-P. Tchapet Njafa and S. Nana Engo, *Concise quantum associative memories with nonlinear search algorithm,* Fortschritte der Physik **64**, 250 (2016).

[89] S. Aaronson, *Guest column: Np-complete problems and physical reality,* ACM Sigact News **36**, 30 (2005).

[90] B. Kosko, *Bidirectional associative memories,* IEEE Transactions on Systems, man, and Cybernetics **18**, 49 (1988).

[91] G. Torlai, G. Mazzola, J. Carrasquilla, M. Troyer, R. Melko, and G. Carleo, *Neural-network quantum state tomography,* Nature Physics , 1 (2018).