Quantum Algorithms for Finding Extrema with Unary Predicates

and related problems

by

Man Hon Chan

B.Eng.(Computer Engineering), The University of Hong Kong, 2005

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University of British Columbia

August,. 2007

Abstract

We study the problem of finding the maximum or the minimum of a given set $S = \{x_0, x_1, \ldots x_{n-1}\}$, each element x_i drawn from some finite universe $\mathscr{U}$ of real numbers. We assume that the inputs are abstracted within an oracle $\mathscr{O}$ where we can only gain information through unary comparisons in the form "Is x_i greater than, equal to, or less than some constant k?" Classically, this problem is solved optimally with a runtime of $\Theta(n + \lg|\mathscr{U}|)$.

In the setting of Quantum Computing, we show that at least $\Omega(\sqrt{n} + \lg|\mathscr{U}|)$ queries are required to solve the problem even with bounded error. Combining variants of the Grover's search [1, 2] algorithm and the optimal classical unary extrema finding algorithm, we have derived a series of new quantum algorithms, some running in time as fast as $O(\sqrt{n}\lg^* n + \lg|\mathscr{U}|)$. This shows that quantum computers can accelerate the speed in the unary comparison model asymptotically. Inspecting our tools, we find convincing arguments that our lower bound is most probably tight, but we may need an entirely new approach to solve the problem optimally.

The technique used in our algorithm can also be extended to solve variations of quantum statistics problems. For instance, our result can be directly extended to approximation of extrema of real numbers, similar to that of [3]. Moreover, we can also solve the quantum k-select problem optimally in time $O(\sqrt{kn})$ with constant success probability. We hope that our ideas and tools will prove to be useful in other areas.

Contents

List of Tables

List of Figures

Acknowledgements

The author would like to express his deepest gratitude his supervisor David G. Kirkpatrick for his unfailing support and encouragement. Not only did he provide motivations and ideas, he has helped the author to find tremendous joy in doing researches and solving challenging problems.

The author would also like to thank his second reader Will Evans for his suggestions and guidance.

The author is also greatly in debt to his family and friends for always being supportive and caring. This research would not have been successful without the help from Bartholomew Furrow, Zephyr Liu, Jack Mai, and many more others. The feedbacks, suggestions and encouragements are what make this work possible.

Chapter 1

Introduction

Quantum Computing has become an exciting field over the previous decades. Although significantly younger than most topics in classical algorithms and complexity theory, it is very interesting in its own right. The very notion of quantum computing was introduced in the beginning of the 1980s. In 1994, Shor presented the celebrated algorithm [4] for discrete logarithm and factoring integers, showing how quantum computers could possibly revolutionarize the age of computation. There were other significant contributions including the Grover's search algorithm [1] and several of its extensions (e.g. BBHT[2], BCWZ[5]), which are more general and far more powerful than any classical counterpart.

In Furrow's thesis [6], he put together and optimized previous known results, and presented a couple of basic quantum algorithmic tools that prove to be useful in many circustances. He tackles real life problems, using quantum algorithms to bypass classical lower bounds. One could imagine how powerful a sublinear extrema finding algorithm could change many existing algorithms. For instance, the naive brute force solution to the maximum submatrix sum problem is $O(n^4)$. With various dynamic programming and optimization, one can solve it in $O(n^3)$. However, using a maximum finding algorithm that runs in square root time, one can achieve a runtime of $O(n^2)$. Not only is this asymptotically faster than the current best classical algorithm, the idea to the solution is much more succinct. With a similar spirit, we wish to tackle problems that are fundamental or frequently reused so that any improvement could be cascaded and impact

many other algorithms.

The problem of locating the maximum among a given set is a well studied problem. With quantum computers, an optimal $O(\sqrt{n})$ algorithm was presented in [7]. It works under the assumption that one can take any two elements from the input set and rank them in constant time. This is usually the case in the *RAM model* or when we have an external comparison oracle. Arguably, this binary comparison operation may not be feasible if inputs are distributed or if inputs are not infinitely precise. One of such cases is manipulation of real numbers, where each number can only be approximated. Directly comparing two real numbers requires us to approximate the two real numbers until they can be clearly distinguished. In this dissertation, we study a more restrictive model, where one can only compare an element with a constant. For instance, suppose we are given n real numbers, each a root of some blackbox monotone function in the range $[0, 1)$. We can easily compare the root of the function f with a constant k simply by evaluating $f(k)$. The sign of $f(k)$ tells us whether the root is to the left or to the right of k. However, there is no way one could directly compare the roots of two blackbox functions directly. As one of the motivating problems of our research, we are given n such blackbox functions and we want to approximate the maximum to a certain absolute error efficiently. The notion of approximation is introduced because we assume that we cannot represent the root of any of our functions perfectly. There are also cases where two roots may be indistinguishable within the specified error, and they could both be the maximum. Under this setting, we cannot directly apply binary comparison algorithms. We need different approaches to solve this new class of problems.

In our thesis, we shall first formalize the problem of unary maximum finding and introduce notations that would be useful throughout our discussion. Maximum finding and minimum finding are basically the two faces of the same

coin, extrema finding. In keeping with the discussion of the optimal, classical counterpart [3], we primarily focus on finding the maximum. We then introduce the basics of quantum computing and the model of computation adapted in this thesis.

Next, we look into specific quantum search algorithms, which are the heart of our speedup over classical algorithms. These tools include `Grover` [1] and `BBHT` [2]. The primary problem they tackle is a variant of the satisfiability problem: Given a blackbox Boolean function F over the domain $\{0, \ldots, n-1\}$, we would like to find some x such that $F(x)$ is `True`. Classically, without gaining further information of F, one can only revert to a linear search for such an x. This implies a lower bound of $O(n)$ in both the average and the worst case. This remains the case even when we allow our algorithm to err with a fixed constant probability. Using quantum parallelism and amplitude amplification, we can achieve an expected time of $O(\sqrt{n})$, which breaks our classical lower bound of $\Omega(n)$. We study these tools and their theories in order to use these algorithms correctly.

After that, we present our new algorithm for finding maximum using only unary predicates. Classically, this problem is optimally solved with an algorithm that runs in $O(n + \log_2 |\mathscr{U}|)$. Hereafter, we shall denote $\lg(x)$ as the logarithm of x with respect to the base 2. The above runtime would then be written as $O(n + \lg |\mathscr{U}|)$. Our new algorithm is presented as a series of improvements over previous known algorithms, with occasionally new tricks. We have achieved the expected runtime of $O(\sqrt{n} \lg^* n + \lg |\mathscr{U}|)$. We hope that the techniques used in our algorithms will prove to be useful in other circumstances.

Finally, we would present ideas for future explorations.

1.1 Unary Maximum Finding

We now formally define the unary maximum finding problem. As input, we are given three items:

- n, denoting the numbers of elements there are.
- $\mathscr{U}$, a finite and totally ordered set denoting the domain of all input elements.
- $\mathscr{O}$, a blackbox oracle which maps $\{0,\ldots,n-1\} \times \mathscr{U} \rightarrow \{`\prec',`\equiv',`\succ'\}$. $\mathscr{O}(i,k) = `\prec'$ (resp. $`\equiv',`\succ'$) means that for the i^{th} element x_i, $x_i <$ (resp. $=,>$) k. With a totally ordered $\mathscr{U}$, this oracle must also satisfy that $\forall k_1 < k_2 \in \mathscr{U}$
  $(\mathscr{O}(i,k_1) = `\prec') \Rightarrow (\mathscr{O}(i,k_2) = `\prec')$, and
  $(\mathscr{O}(i,k_2) = `\succ') \Rightarrow (\mathscr{O}(i,k_1) = `\succ')$.
  For simplicity we write $x_i <$ (resp. $=,>$) k interchangeably as $\mathscr{O}(i,k) =$ $`\prec'$ (resp. $`\equiv',`\succ'$).

We can also define an equivalently powerful oracle $\mathscr{O}'$ as a threshold predicate. $\mathscr{O}'$ would then be a mapping from $\{0,\ldots,n-1\} \times \mathscr{U} \rightarrow \{\texttt{False},\texttt{True}\}$, where $\mathscr{O}'(i,k)$ is Trueif and only if $x_i < k$. The two definitions are equivalently up to a constant factor 2. Given an oracle $\mathscr{O}$ in our first definition, we can create a threshold oracle $\mathscr{O}'$ simply by

$$\mathscr{O}'(i,k) = \begin{cases} \texttt{True} & \text{if } \mathscr{O}(i,k) = `\prec' \\ \texttt{False} & \text{otherwise} \end{cases}$$

On the other hand, given a threshold oracle $\mathscr{O}'$, we can implement $\mathscr{O}(i,k)$ as

```
if O'(i,k) return '≺'
if (k = max(U)) or O'(i, succ(k)) return '≡'
return '≻'
```

Here, we define $succ(k)$ to be the successor of k in the universe $\mathscr{U}$, that is, the minimum of all numbers in $\mathscr{U}$ which are greater than k. For contiguous integral domain, this equals to the value $k+1$. This operation depends only on the $\mathscr{U}$ and does not impact our $\mathscr{O}$. As a result, the two definitions are equivalent. We shall stick with our first ternary function as that is more intuitive.

In our hypothetical maximum root approximation problem, our global interval is $[0,1)$. Since we want to approximate the maximum root within an error of ϵ, it suffices to divide the global interval into buckets of size ϵ. Our universe $\mathscr{U}$ will be a collection of such buckets. A root is ' $\succ$ ' (resp. ' $\prec$ ') than a bucket if it lies to the right (resp. left) of the bucket boundaries. A root is ' $\equiv$ ' a bucket if it lies within the bucket. The size of our universe would then be $\frac{1}{\epsilon}$.

Given the input, we would like to output $\lambda^* \in \mathscr{U}$ as the maximum of the given set $\{x_i\}$. More precisely, we want to ensure that

- For all $i \in \{0,\ldots,n-1\} \mathscr{O}(i,\lambda^*) \neq$ ' $\succ$ ' and
- There exists $i \in \{0,\ldots,n-1\}$ such that $\mathscr{O}(i,\lambda^*) =$ ' $\equiv$ '

In general, it does not matter if we return the index or the exact value itself. We choose to return the exact value as it is generally more useful. Consider our maximum root approximation problem with $n=1$, it is of little value to return the index as it provides no further information. Moreover, simply returning the index may give us an illusion that the error term ϵ is irrelevant, which is unfortunately not the case. Taking aside degenerated cases, the two different types of returning the result do not affect the intrinsic difficulty of the problem. Given a specific index, we can perform a binary search to retrieve its exact value. On the other hand, we can perform a search to find the corresponding index given its value. We shall later see that the costs in binary searching or performing a complete search are below the lower bound for the unary maximum finding problem itself. As a result, an extra conversion step does not incur any

penalty to our complexity.

There are many ways we can calculate the cost or runtime of an algorithm. That heavily depends on the model of computation we are using. In general, we either focus on only oracle calls or the total runtime. The rationale behind charging only oracle calls is that information can only be extracted from calls to the oracle. They record the path our algorithm evolves and gradually approaches the final answer. Furthermore, the oracle call should be much more expensive and dominating in most situations (e.g. evaluating of black box functions). Afterall, the hardness of our problem relies on a hard unary oracle where no binary comparison can be effectively simulated. However, counting only oracle calls is certainly an underestimate of the total runtime any algorithm has to take. It is only fair if we incorporate the cost of all auxiliary work such as memory managements. It is also unrealistic when we assume that we can manipulate real numbers arbitrarily or perform complicated arithmetics in constant time. For instance, one may try to approximate our blackbox functions with polynomials and estimate the roots. While this may be useful in many cases, the cost associated in performing interpolation and an additional root finding is simply too big to be practical in our scenario.

In our following discussion, we restrict ourselves to only simple elementary operations (e.g. addition, subtraction, average of two numbers) on the index domain $\{0, \ldots, n-1\}$ or the universe domain $\mathscr{U}$. Suppose we denote $T(\mathscr{O})$ as the number of oracle calls in an algorithm, and $T(Elementary)$ as the number of elementary arithmetic operations, we maintain that $T(\mathscr{O}) = \Theta(T(Elementary))$. As a result, we can refer both of them interchangeably. The reader will be informed when this claim does not hold. We should take into account the actual runtime of oracle calls and elementary arithmetic operations. In general, the cost of elementary operations is linear to the description size. Making a call to

the oracle require us to supply the arguments to it, which has a cost linear to the description size as well. Therefore, the cost in calling the oracle almost always dominates the cost of elementary operations. With $T(\mathscr{O}) = \Theta(T(Elementary))$, it then suffices to count only oracle calls to derive the complexity of our algorithms.

1.1.1 Lower bound of classical unary maximum finding

The classical lower bound of the unary maximum finding problem is shown to be $O(n+\lg|\mathscr{U}|)$ by Gao et al.[3]. The state of computation of any algorithm can be expressed as an n-tuple of sets, $(S_0, S_1, \ldots, S_{n-1})$ where S_i denotes the possible values of x_i. The lower bound is proved via a collective adversary strategy, where we try to prepare the worst test case for our algorithm. Let $\mathscr{C} = \bigcap_{i=0}^{n-1} S_i$ denote the subset of $\mathscr{U}$ that is feasible for all inputs. Any algorithm cannot output the maximum with certainty if $|\mathscr{C}| > 1$. Moreover, if $|\mathscr{C}| = 2$, the collective adversary could set all but 1 entry to the minimum of $\mathscr{C}$, forcing any correect algorithm to ask $\Omega(n)$ questions before reaching any conclusion. Initially, we have $|\mathscr{C}| = |\mathscr{U}|$ and it takes $\Omega(\lg|\mathscr{U}|)$ queries to reduce $|\mathscr{C}|$ to 2. It follows from the combination that it takes at least $\Omega(n + \lg|\mathscr{U}|)$ queries for any algorithm to correctly solve this problem.

Theorem 1 *(Gao et al.: Theorem 2.1) Determining* $\max\{x_0, x_1, \ldots, x_{n-1}\}$ *requires* $n + \lceil \lg|\mathscr{U}| \rceil - 1$ *unary predicate ealuations, in the worst case, even when it is given that* $|\{x_0, x_1, \ldots, x_{n-1}\}| \leq 2$.

It is also straightforward to extend this idea to the case of unary minimum finding. It takes $\Omega(n+\lg|\mathscr{U}|)$ to compute the extrema given our model. We can also understand this as a combined effort in (i) certifying maximality by looping through all possible entries, taking $\Omega(n)$ time; and (ii) outputting the answer,

taking $\Omega(\lg x_{max})$ time. Making $\lg x_{max} = \Omega(\lg |\mathscr{U}|)$, an adversary could easily force our algorithm to run in $\Omega(n + \lg |\mathscr{U}|)$ time, establishing our lower bound.

1.1.2 Classical unary maximum finding

In a binary model, we can tackle the problem simply by iterating through all entries and keeping track of the largest seen entry. Suppose we are given the exact value of each entry, the same approach would also work in the unary model. However, knowing all the values requires n binary searches, each at cost $O(\lg |\mathscr{U}|)$. This $O(n \lg |\mathscr{U}|)$ approach is certainly overkill. Given a current estimate λ, checking if the next entry is larger is immediate. If the next entry is smaller, determining its value is unnecessary. In a randomized setting, we can loop through all entries in a randomized order $(x_{\pi_0}, x_{\pi_1}, \ldots, x_{\pi_{n-1}})$. During our loop, we only need to perform a binary search on x_{π_i} when it is strictly greater than all x_{π_k}, $k < i$. That happens with probably at most $\frac{1}{i+1}$. Therefore, the expected runtime of this approach is $O(n + \sum_{i=1}^{n} \frac{1}{i} \lg |\mathscr{U}|) = O(n + \lg |\mathscr{U}| \lg n)$.

The optimal classical unary maximum finding is attributed to Gao et al. [3]. This algorithm will be reused heavily in our future discussion and is thus presented here for completeness. The readers are however strongly advised to review the original paper for a thorough and rigorous discussion of the algorithm and its applications.

To begin with, we first assume that our universe $\mathscr{U}$ is the integer range $[0, m)$. Any finite, totally ordered universe can be remapped to an integral range with some appropriate m. As we are only using unary predicates, we can represent our current knowledge on the inputs as a vector of feasible ranges $\{S_i\}$. We further denote the ranges with its extrema, giving $S_i = [\lambda_i, \mu_i)$. The maximum lower bound λ_{max} is defined accordingly as $\max\{\lambda_i\}$ and is an underestimate of our desired output λ^*. Clearly, for any i with $\mu_i \leq \lambda_{max} + 1$, x_i cannot be

greater than our current estimate and the corresponding x_i is deemed impotent. Using the idea in the lower bound proof and discarding impotent elements, we have $\mathscr{C} = [\lambda_{max}, \min\{\mu_i \mid x_i \text{ not impotent}\})$. In the same spirit, an algorithm can only make progress by reducing $|\mathscr{C}|$.

To efficiently handle all the ranges, the authors in [3] have grouped elements with the same μ_i in the same bucket, called a *block*. The blocks are ordered descendingly according to the associated μ. In the following descriptions, B_i refer to the buckets with β_i as their associated μ. The pseudo code of the classical optimal unary maximum finding algorithm is presented here:

```
B_0 ← {0, ..., n − 1}
s ← t ← 0
β_0 ← max(𝒰)
λ_max ← 0
while β_s > λ_max + 1 do
  select a random i from B_s and remove it
  j ← ½(λ_max + β_t)
  if x_i < j then
    β_{t+1} ← j
    insert i into B_{t+1}
    t ← t + 1
  else
    λ_max ← j
    j ← β_t
    while x_i ≥ j do
      λ_max ← j
      clear B_t
      t ← t − 1
      j ← β_t
    end while
    insert i into B_t
  end if
  if B_s is empty then
    s ← s + 1
  end if
end while
return λ_max
```

By constructions, β_i are descending. At completition, $\beta_s \leq \lambda_{max} + 1$ certifies that all remaining elements are impotent and thus λ_{max} is indeed the desired

Problem		Lower Bound	Upper Bound
Extreama (Binary)	Classical	$\Omega(n)$	$O(n)$
	Quantum	$\Omega(\sqrt{n})$	$O(\sqrt{n})$
Extreama (Unary)	Classical	$\Omega(n + \lg \lvert\mathscr{U}\rvert)$	$O(n + \lg \lvert\mathscr{U}\rvert)$
	Quantum	$\Omega(\sqrt{n} + \lg \lvert\mathscr{U}\rvert)$	$O(\sqrt{n} \lg^* n + \lg \lvert\mathscr{U}\rvert)$

Table 1.1: Bounds on extrema problems

answer. Furthermore, we discard blocks with $\beta_i < \lambda_{max}$. As a result β_t, the upper bound of the last block, reflects the minimum of all upper bounds. This implies $\mathscr{C} = [\lambda_{max}, \beta_t)$. Throughout the algorithm, we either query with the median of the range $\mathscr{C}$ in order to halve $|\mathscr{C}|$ or we compare with some β_t to discard elements. Moreover, we can verify that whenever we discard an element, $|\mathscr{C}|$ will increase by no more than twice of its original size. Eventually, the program would have reduced $|\mathscr{C}|$ to 1 and discarding all impotent elements. This thus runs in $O(n + \lg |\mathscr{U}|)$ time. A complete analysis can be found in [3].

The importance of this algorithm is immediate. It proves that the lower bound is tight and we can solve the unary maximum finding problem in $\Theta(n + \lg |\mathscr{U}|)$ time optimally. It is one of the fundamentals in many of our algorithms.

1.2 Quantum Unary Maximum Finding

The same problem of Unary Maximum Finding problem can be extended to the quantum setting. We would still be working with the three tuple $(n, \mathscr{U}, \mathscr{O})$ together with all our assumptions. However, we add to our oracle the power of *quantum parallelism*. As a result, we can input a superposition of input and get an entangled superposition with the corresponding results. This, in most case, requires no more than a straight forward classical-to-quantum transformation. Some of the details will be covered in the following chapters.

The binary maximum finding problem is well studied in the quantum setting,

Name	Description	Runtime
Sample and Improve	Adaptation of the optimal algorithm from [7]	$O(\sqrt{n} + \lg \lvert\mathscr{U}\rvert \lg n)$
Budgeted Sample and Improve	Make use of `Budgeted_Sampling` to accelerate `Sample_And_Improve`	$O((\sqrt{n} + \lg \lvert\mathscr{U}\rvert) \lg \lg n)$
Geometric Budgetting	With a geometrically decreasing budget, isolate the $\sqrt{n}$ term from the $\lg \lg n$ factor	$O(\sqrt{n} + \lg \lvert\mathscr{U}\rvert \lg \lg n)$
Progressive Approximation	Isolate the $\lg \lvert\mathscr{U}\rvert$ term from the extra factor	$O(\sqrt{n} \lg n + \lg \lvert\mathscr{U}\rvert)$ $O(\sqrt{n} \lg \lg n + \lg \lvert\mathscr{U}\rvert)$
Large Sampling	Make use of large pseudo-random sampling to further accelerate the runtime	$O((\sqrt{n} + \lg \lvert\mathscr{U}\rvert) \lg^* n)$

Table 1.2: Runtime of various algorithms

with an optimal $O(\sqrt{n})$ algorithm derived in the work by Dürr and Høyer [7]. The unary maximum finding problem is, unfortunately, untouched. The first question in general is whether we can outperform classical algorithms once we have access to quantum parallelism, as demonstrated in the binary maximum finding problem. However, one could show that the binary search term $\Theta(\lg \lvert\mathscr{U}\rvert)$ is optimal in both the classical and the quantum setting. There is an intrinsic lower bound of $\Omega(\sqrt{n} + \lg \lvert\mathscr{U}\rvert)$. Table 1.1 lists our current best understanding for the maximum finding problems in various flavours.

We derive new algorithms iteratively in an attempt to narrow the gap between our upperbound and the intrinsic lowerbound. Classicaly, we have already seen that a direct adaptation of the best binary algorithm may not yeild a good unary algorithm. That is also the case in the quantum setting. By adapting the optimal algorithm we could achieve an expected time of $O(\sqrt{n} + \lg \lvert\mathscr{U}\rvert \lg n)$ (c.f. $O(n + \lg \lvert\mathscr{U}\rvert)$ from the randomized classical algorithm). Further studies show that we could in fact solve the problem in $O((\sqrt{n} + \lg \lvert\mathscr{U}\rvert) \lg \lg n)$ or even

$O(\sqrt{n} + \lg|\mathscr{U}|\lg\lg n)$ expected time. Unfortunately, these algorithms perform worse than the classical algorithm when $\lg|\mathscr{U}|$ dominates. We then approach the problem from an approximation perspective, yielding algorithms with expected runtime $O(\sqrt{n}\lg n + \lg|\mathscr{U}|)$ or $O(\sqrt{n}\lg\lg n + \lg|\mathscr{U}|)$. These algorithms provide speedup even when $\lg|\mathscr{U}|$ is considerably large. With large pseudo-random sampling, we further push the limit [1] to $O((\sqrt{n} + \lg|\mathscr{U}|)\lg^* n)$. Table 1.2 lists some of our achieved runtimes for the unary maximum finding problem.

Finally, we show that how one could reuse our previous results to generate algorithms with expected runtime of $O(\sqrt{n}\lg^* n + \lg|\mathscr{U}|)$, $O(\sqrt{n} + \lg|\mathscr{U}|\lg^* n)$ or $O((\sqrt{n} + \lg|\mathscr{U}|)\lg^*(\lg^* n))$.

The details of each algorithm will be covered in Chapter 4.

[1] $\lg^* n$ is defined to be the number of times one has to take lg before n becomes less than 1.

Chapter 2

A brief introduction to Quantum Computing

Computing can be viewed as the manipulation of data in order to get from our initial state to an useful final state. Suppose we want to sum up n integers, we would first create an output register and subsequently add numbers to this register. At the end, we can find the accumulated sum in our output register. What we have done is that we gradually transform our input and auxiliary work bits (temporary variables) to a good final state.

Classically, all information are stored in bits and we have different electronic gates at our disposal. In the quantum world, we have access to quantum information and can utilize various quantum effects such as superposition and entanglement to aid our process of transforming input data to useful target states.

This chapter aims at providing a brief, and necessarily superficial, overview on quantum computing and various notations.

2.1 A brief history of Quantum Computing

It all began with the problem of simulation. In his article [8] in 1982, Nobel Prize winner Richard P. Feynman proposed that a quantum physical system with R particles cannot be simulated efficiently with ordinary computers without an

exponential slowdown. However, a classical physical system can be simulated by ordinary computers efficiently in polynomial time. The major challenge of the quantum simulation problem is the description size of the system. The description of a quantum system with R particles involes expressing a function $\phi(x_1, x_2, \ldots, x_R, t)$ across its full domain which is exponential in R. He went on and suggested that if we can use computers that run according to the law of quantum mechanics, this problem becomes tractable. This suggests that a quantum computer can potentially provide an exponential speedup over classical ones.

The quantum model of computation and the idea of a universal quantum computer is formalized by Deutsch in [9] in 1985. The construction of a universal quantum Turing machine was also improved by Bernstein and Vazirani in [10], in which the authors showed how one can construct a universal quantum Turing machine to simulate any given quantum Turing machine with polynomial efficiency.

The field had not gained much attention until Shor published his well known integer factorization algorithm [4]. The current RSA crytosystem depends on the intractability of factoring arbitrary large composites. Given a quantum algorithm, we now have a polynomial time algorithm not only to test for primality but to actually retrieve the corresponding factors. This shows that quantum computers could potentially be faster than classical computers not only on hypothetical problems of marginal interest.

In 1996, Grover introduced his famous search algorithm [1]. With the assumption that there is a unique solution to our satisfiability problem, it finds the satisfying index in expectedly $O(\sqrt{n})$ time. The technique employed was later generalized [2, 7] and applied in different scenarios (e.g. waiving the uniqueness assumption, extreama finding, etc.), providing quadratic speedup over the

best known classical algorithms. In the following decades, new algorithm based on these primitive techniques were derived, such as various graph algorithms, element distinctness and collision algorithms. In 2006, Bartholomew Furrow showed how to combine these results and apply these tools to various geometry and combinatorial problems [6]. Some of these results will be used in our algorithm.

The theory is, however, far more developed than the practice. By far, only very small scale quantum computers have been built and tested. Quantum effects are only observable in very fine scale and they can easily be corrupted by external interference. The problem of coping with errors in quantum computations has long be considered intractible mainly because of the No-Cloning theorem [11]. According to the theorem, one could not copy quantum information from one storage to another one. Duplication is essential in most classical error correction schemes and this theorem seems to be a huge obstacle. It was not until Shor demonstrated an error correcting scheme in [12] that established the theory of quantum error-correcting codes. Given a modest error probability in low level components, we can now employ different error-correcting techniques to support arbitrarily long quantum computations.

2.2 Qubits

Consider a simple coin flip where the output is unknown to us. This simple coin flip can be formalized as a system with two possible outcomes *head* and *tail*, each with its respective probability p_h and p_t. For simplicity, we can express this as a probabilistic mixture $p_h[head] + p_t[tail]$, where $p_h + p_t = 1$. This system remains in an unknown state until we actually observe or *measure* it. Once measured, we will know the exact outcome of this system and it degenerates to either $1[head] + 0[tail]$ or $0[head] + 1[tail]$. This measurement is destructive

as we can not return this system to the previous mixed state without another flip. On a similar note, it is debatable if we can actually clone the state of an unknown coin. There is no way we can measure the probability p_h or p_t.

Now consider the same experient where we flip a "coin" the size of an electron. Quantum mechanics tells us that a simple probabilistic mixture is not an adequate representation of the system. The correct way to formalize the system takes the form $\alpha_h[head] + \alpha_t[tail]$, where both coefficient are complex and $|\alpha_h|^2 + |\alpha_t|^2 = 1$. As discussed, the system remains in an unknown state until we measure it. The probability of measuring a head (resp. tail) event is now given by $|\alpha_{head}|^2$ (resp. $|\alpha_{tail}|^2$). Since the sum of the probabilities equals to 1, we will always detect some valid outcome whenever we measure. The coefficient α is called the amplitude of observing the corresponding state. The significance of complex amplitude would be clear in the next section.

This notation can be extended to any physical system with 2 orthogonal observables. Without loss of generality, we denote them as $|0\rangle$ and $|1\rangle$. The *ket* notation $|\phi\rangle$ is a mathematical tool to denote a quantum state and is widely used in quantum related context. A state of this system would take the form $|\phi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$. Similar to orthonomal bases of a vector space, the observables are completely arbitrary. One may exchange the role of $|0\rangle$ and $|1\rangle$. We can also choose to operate on the rotated basis $\{\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\}$. The choice of our observables depends only on how we build our measurement device.

We have portrayed a qubit as a generalized binary random variable. However, a qubit cannot be reused. By the No-Cloning theorem [11] one cannot back up a qubit for future uses. We understand that measurements collapse the state of a qubit and subsequent measurements will always end with the same outcome. For instance, suppose we start with a qubit with state $|\phi\rangle = \frac{1}{\sqrt{3}}(\sqrt{2}|0\rangle + i|1\rangle)$. We have with probability two thirds measuring state 0 and probability one third

measuring state 1. However, once we measured 0, we stayed 0 all the time. Future measurements must always agree with the first measurement. If we measure 1 from the qubit for the first time, we would have collapsed the state to contian only $|1\rangle$ for all remaining computation. In some implementation, the measurement operation is so destructive that the qubit itself is destroyed and further measurements would not even be possible. As a result, we have to reprepare the qubit before we can perform the same measurement.

This concept can be extended to system with arbitrary number (> 1) of levels. A qubit is a device capable of encapsulating a two level system. A sequence of qubits makes a quantum register. Concatenating qubits provide a system with exponential number of levels. For instance, an k bit register can represent 2^k integers. With k qubits, we can have 2^k different observables, including $|0\rangle|0\rangle\ldots|0\rangle$, $|0\rangle|0\rangle\ldots|1\rangle$, ..., etc. For simplicity, we write $|0\rangle|0\rangle\ldots|0\rangle$ as $|00\ldots0\rangle$. Similarly, we can write $|00\ldots1\rangle$ or $|00\ldots10\rangle$. Since these are basically binary numbers, we will also use the notation $|0\rangle$, $|1\rangle$, $|2\rangle$, etc. Generally, if we have k bits, we would have $|0\rangle$ up to $|2^k-1\rangle$. These 2^k states provides a basis for the concatenated multi-level system. A quantum computer refers to a collection of qubits / quantum registers and devices to manipulate them.

2.3 Unitary Transform and Entanglement

The state of a probabilistic (quantum) system can be described by a vector of probabilities (amplitudes). Different states are isolated and progress independently. A probabilistic system evolves through a series of redistribution of probabilities. When we perform any operation (state transition) on an n-level

system, we can express that as:

$$\begin{pmatrix} c'_0 \\ c'_1 \\ \vdots \\ c'_{n-1} \end{pmatrix} = \begin{pmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,n-1} \\ p_{1,0} & p_{1,1} & \cdots & p_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n-1,0} & p_{n-1,1} & \cdots & p_{n-1,n-1} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix}$$

This transition matrix means that if my system is in state i, it has probability $p_{i,j}$ to evolve into state j. A valid transition matrix must preserve the probability of each state, giving the formula $\sum_j p_{i,j} = 1$. Such a matrix is called a Markov matrix.

The general idea applies in the quantum setting. We have different basis states and they operate independently and linearly. Instead of using probabilities, the transition matrix is replaced with one that contains complex entries. A valid transition must preserve the amplitude and ensure that $\sum_i |\alpha_i|^2 = 1$ after the transition. It turns out that such a transition must be unitary. Given any unitary transition matrix A, we have $A^{-1} = A^\dagger$ where $A^\dagger$ denotes the adjoint (conjugate tranpose) of A. This also implies that all allowed transitions are inherently invertible. This is clearly not required in the classical case.

Some basic single qubit operators include

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \; X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \; Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \; Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Consider the basic operation X. Given the single-qubit state $|\phi\rangle = |0\rangle = 1|0\rangle + 0|1\rangle$, the result would be

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

Similar, X flips the state $|1\rangle$ into $|0\rangle$. As a result, X can be thought of as the *not* gate in electronics. Operator Z flip the sign of the amplitude of $|1\rangle$ and leave the amplitude of $|0\rangle$ unchanged. One of the most important gates is the Hadamard gate. The corresponding matrix is:

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

It transforms $|0\rangle$ into $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle$ into $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Consider an initial state $|0\rangle$, the probability of getting $|0\rangle$ is 1. After the transformation, the probability of measuring $|0\rangle$ or $|1\rangle$ is exactly $(\frac{1}{\sqrt{2}})^2 = \frac{1}{2}$. The same analysis applies for initial state $|1\rangle$. Although the amplitude for $|1\rangle$ is $-\frac{1}{\sqrt{2}}$, the measurement probability is still $\frac{1}{2}$ after squaring. It appears that Hadamard is similar to a fair coin toss probabilistically.

However, there are two subtleties. Firstly, in a probabilitic model, a coin toss operator is usually depicted as:

$$CoinToss = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

This is a complete random shuffle and it destroyed whatever information stored before. As a result, this transformation is not invertible and is not unitary. Therefore, this is not valid in the quantum computing setting. Secondly, the effect of cascading $CoinToss$ and H are completely different. Consider an initial state [0] (resp. $|0\rangle$ quantumly), performing $CossToss$ twice brings us the state $\frac{1}{2}[0] + \frac{1}{2}[1]$ which is a perfect mixture of both state. In the quantum case, performing H once brings us the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ Preforming H again gives

us

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

Performing H twice surprisingly brings us back to our original state. This is the effect of interference in Physics and is generally absent in any probabilistic computing model. One can also verify that $HH = I$. Such operators are called self-inverse. The effect of interference is one of the main reasons why quantum computers could be more powerful than probabilistic computing, while the major restriction is that we can only perform reversible operations.

In some literatures, the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is shorthanded as $|+\rangle$ and the state $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ as $|-\rangle$. Given k qubits at $|0^n\rangle$, one can apply the Hamadard Transform to all for them, producing the state $|+^k\rangle$. Expanding $|+^k\rangle$ we have $\frac{1}{\sqrt{2^k}}(|00\ldots00\rangle + |00\ldots01\rangle + |00\ldots10\rangle + \ldots + |11\ldots1\rangle) = \frac{1}{\sqrt{2^n}}(|0\rangle + |1\rangle + |2\rangle + \ldots + |2^n - 1\rangle)$, a uniform superposition of all possible bit-patterns of k bits. This is often used as an initialization step. Note that redoing this will revert the state back to $|0^k\rangle$

An important 2-qubit operator is the controlled-not (CX) operator. It is best described by "If the first qubit is 1, apply X to the second qubit. The matrix form is given as:

$$CX = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix}$$

Consider the following scenario where we start with the state $|00\rangle = 1|00\rangle + 0|01\rangle + 0|10\rangle + 0|11\rangle$, represented as $(1,0,0,0)^T$ in the vector form. Firstly, we apply H to the first qubit and reach the state $|+0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$. We then

apply the CX operator to both of the qubits. CX would turn the state $|10\rangle$ to $|11\rangle$ but will leave the state $|00\rangle$ unchanged. The final state thus becomes $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. This is a special state in which the two qubits are entangled. Suppose we now measure the first qubit and get a result of 1. The measurment would collapse the state of the first qubit to only contain $|1\rangle$. However, the only feasible state for this to happen is precisely $|11\rangle$. By measuring the first qubit and getting an 1, we have also collapsed the second qubit to contain only 1. This phenomenon does not hold for all multi-qubit systems. Suppose we are given a state $|\phi\rangle = \frac{1}{2}(|00\rangle + |01\rangle - |10\rangle - |11\rangle)$, we can factorize it to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |-\rangle|+\rangle$. The measurement of the first qubit is completely independent of the second qubit and they both behave like a fair coin. Consider another example with the state

$|\phi\rangle = \frac{1}{\sqrt{18}}(|00\rangle + 2i|01\rangle + 2|10\rangle - 3|11\rangle)$,

which can be rewritten as

$\frac{\sqrt{5}}{\sqrt{18}}|0\rangle(\frac{1}{\sqrt{5}}(|0\rangle + 2i|1\rangle)) + \frac{\sqrt{13}}{\sqrt{18}}|1\rangle(\frac{1}{\sqrt{13}}(2|0\rangle - 3|1\rangle))$

With probability $\frac{5}{18}$, we could get a measurement of 0 in the first qubit, collapsing the state of the second qubit to $\frac{1}{\sqrt{5}}(|0\rangle + 2i|1\rangle)$. With probability $\frac{13}{18}$, we could get a measurement of 1 in the first qubit, collapsing the state of the second qubit to $\frac{1}{\sqrt{13}}(2|0\rangle - 3|1\rangle)$.

Entanglement is an extremely important property in quantum computation. As we have described, quantum states evolves independently and linearly. Suppose we are given a multi-qubit quantum state after a long computation. Without entanglement, each qubit behaves as a separate binary probabilistic system on its own. We cannot tell if the answers we read off from each byte correspond to the same computational path. With entanglement, this guarantee is assured.

2.4 Quantum Circuits and (Approximately) Universal Gate Set

We have discussed qubits and unitary transformation. However, we have yet to establish how this can be more powerful than classical electronic circuits. To begin with, we first tackle the question if all classical electronic circuits can be done with quantum circuits. This may sound trivial, but this is not immediate when we consider all the restrictions of a qubit system.

Firstly, most of the classical circuits are not invertible. Let us consider a circuit that takes in n input bits and produce m output bits. This circuit cannot be invertible if $n > m$ by simple cardinality argument. Some of the useful functions may simply be not invertible even when $n \leq m$. Our input bits will be lost and there is no way we could recover them given the output bits. To overcome this problem, we can simply make our circuits echo the input as part of the output. As a result, our equivalent quantum circuit will take in n bits and producing $(n + m)$ bits. However, this is clearly not unitary as the dimensions simply do not match. To handle this problem, we have to add m dummy bits as our input. A standard way to convert a classical circuit to a quantum circuit involves adding m dummy input qubits, all initialized to $|0\rangle$ to which the m result bits would be exclusively-ORed with it. Since we have initialized the dummy qubits to $|0\rangle$ before hand, the dummy qubits will now contain the result. This quantum circuit is also self-inverse, as a second application will remove all data written in the dummy qubits. Figure 2.1 shows the schematics of the two circuit models.

Now that we have our revised circuit model, we have to establish that for any classical circuit, there is a quantum circuit that computes it. Classically, we know that the gate $NAND$ is universal. Any electronic circuits (e.g. AND, OR,

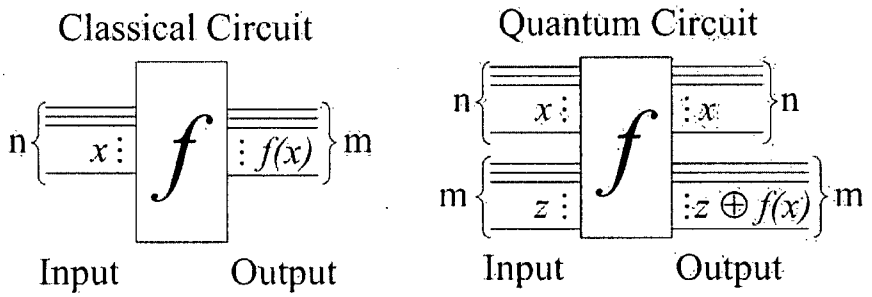

Figure 2.1: Differences between classical circuits and quantum circuits

NOT) can be implemented as some combinations of *NAND*. We have already shown that the X operator is indeed a *NOT* gate. We now present the Toffoli gate, a 3-qubit quantum unitary gate. Similar to the CX gate, it flips the last qubit only if the first two qubits are both 1. Mathematically, it transform the state $|a\rangle|b\rangle|c\rangle$ into $|a\rangle|b\rangle|c \oplus ab\rangle$. Together with an X gate, we have access to the *NAND* equivalent in the quantum setting and this establishes the fact that all classical circuits can be implemented with quantum circuits. More importantly, any classical circuits using only $\{AND, OR, NOT, XOR\}$ can be implemented in the quantum setting with no more than a constant factor overhead. There is, unfortunately, a technical detail about assignments that we have to attend to. Consider the function $y = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$, we can implement it as the circuit as in Figure 2.2 classically. However, since assignments or cloning are not allowed, and we need to do our echoing, the quantum circuit has to take in extra bits as the work spaces. The number of anxillary qubits required depends on how we implement our circuits. Luckily, we never need more anxillary qubits than the original complexity of our circuits. Combining these procedures, our quantum circuits will take as input n input qubits, m output qubits and r anxillary qubits (initialized to $|0\rangle$). The input qubits will be echoed while the result will be exclusively or-ed into the m output qubits. The r qubits will be reset to 0 at the end of the gate so that they will be reused. Figure 2.3 shows one

of the possible implementations of the same expression $y = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$.

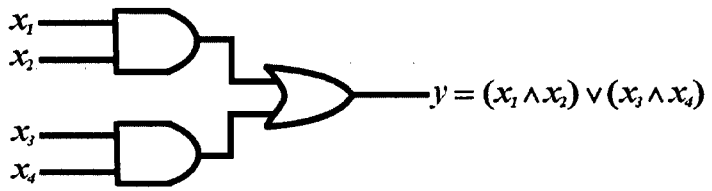

Figure 2.2: Classical Circuit of $y = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$

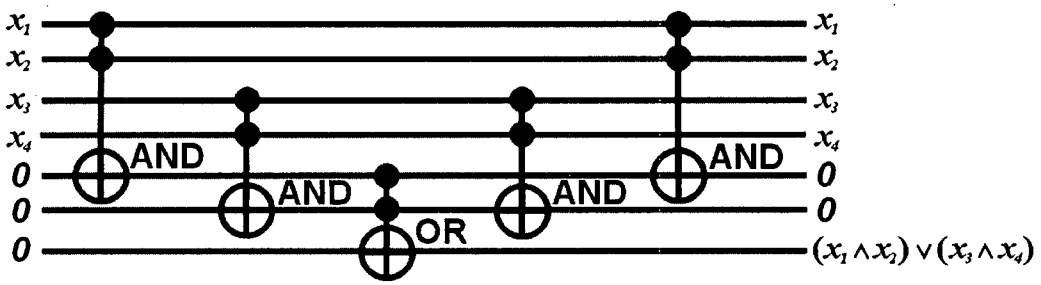

Figure 2.3: Quantum Circuit of $y = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$

Directly from the study of circuit theory, we know that any Turing machine in P can be expressed as a circuit of polynomial size. Together with the argument above, any classical algorithm can be implemented with quantum circuits. The major question is whether we can retain the same efficiency as our algorithm. There are two major questions, namely assignments and branching. Consider the follow code fragment:

```
y ← 0 // Initialization
y ← y ∨ x_0
y ← y ∨ x_1
y ← y ∧ x_2
return y
```

Apart from the initilization, the remaining three lines rewrite the content of *acc* and is not invertible. Using the tricks mentioned above, we can rewrite this as:

```
y0 ← y1 ← y2 ← y3 ← 0 // Initialization
y1 ← y1 ⊕ (y0 ∨ x1)
y2 ← y2 ⊕ (y1 ∨ x2)
y3 ← y3 ⊕ (y2 ∧ x3)
return y3
```

Since we are "writing" those results in different registers, no information is lost and this code fragment is now invertible. By doing so, we have to introduce a new anxillary quantum register for each assignment. Since we only need k anxillary registers if we have k assignments, there is only a constant factor overhead to prepare the state. The second problem concerns the `if` statement and branchings. Classically, we have to evaluate a Boolean expression and select which branch we should be taking. However, measurements collapse superpositions and are not unitary. Luckily, we can postpone measurements. After evaluating the expression, we simply wire that qubit to controlled gates. All our remaining calculations would only be performed if it satisfies all previous conditions. This is similar to the concept of multiplexers commonly used in elctronics. If needed, we then measure the expression qubits at the end of our computation to recover the history of our computation. Entanglement guarantees that the history and the result will always match perfectly. Similar, since we only need extra resources for each branching we have, the added overhead is proportional to the actual work needed in the original algorithm. As a result, all classical algorithms can be implemented quantumly with at most a linear overhead. The potential drawback is that we may need asymptotically more memory than that of the classical algorithm. In general, we would try to free up anxillary qubits where possible for reuse.

We have discussed that any classical computation can be transformed into an equivalent quantum part with no more than a linear overhead. However, there are more unitary transforms than there are circuits. The power of quantum computing could be significantly better than classical algorithms or circuits.

However, there are two potential problems:

- We may not be able to specify an arbitrary unitary transformation. After-all, if we are using an electronic computer to manage its internal quantum module, we only have a limited representative power from classical bits. For instance, we may not be able to specify a transform with irrational coefficients.

- The transform may not be realizable efficiently. An n-qubit unitary transform is actually a $2^n \times 2^n$ matrix. It is unreal to assume that we can get any type of unitary transform off the shelf. At some point, we must construct our own circuits with small pieces. Since the underlying matrix is exponential in size, it is not surprising that some of transforms would take exponential number of small pieces to construct.

In this thesis, we try to stay away from any precision or irrational gates by restricting ourselves to the set of primitive gates we have discussed before [1]. This finite set is approximately universal since combinations of these gates can approximate any unitary transform to within any given error. Moreover, extensive error correcting schemes have been developed over these gates. We believe that our chosen set of unitary transform is less demanding and more practical, while maintaining most of the power of unitary transforms.

2.5 Model of Computation

With the foundations of qubits and unitary transform, we can carry on to construct more powerful computational models. Classically, all algorithms can be expressed as a Turing Machine. It is true that we can model any quantum algorithms as a Quantum Turing Machine. Unfortunately, the details in expressing

[1] We also consider the rotational gate $R(\frac{pi}{4})$ which transforms $|0\rangle$ to $|0\rangle$ and $|1\rangle$ to $e^{i\frac{pi}{4}}|1\rangle$

algorithms in this model are unnecessarily tedious.

To simplify our discussion, we would base our algorithms on a computation model similar to the standard RAM model. We have access to all the standard features including a set of classical registers and the ability to do simple arithmetic operations on them in constant time. To add quantum capability, we add another independent set of quantum registers. We are allowed to perform initialization (to $|0\rangle$) and apply simple gates to any of these registers. For instance, we can apply the Hadamard transform H to each bit in the register. As in the RAM model, we avail ourselves standard arithmetics in constant time. It is worthwhile to mention that although the cost of multi-bit arithmetics depends on the number of bits, we still treat them a constant cost atomic operation. This is similar to computers nowadays, where adding 2 4-bit integers is as costly as adding 2 32-bit integers. The main reason is that the underlying circuits for arithmetics are highly optimized and parallelized, and it generally takes more time to interpret a command instead of executing it. With hardware accelerations, basic arithmetic operations such as additions and substractions on an n-bit registers can be as fast as $O(\lg n)$, using $O(n \lg n)$ gates. Complicated operations such as multipliations, division or Quantum Fourier Transform (QFT) could take as long as $O(n \lg n)$ (or even longer). For uniformality, we avail ourselves an elementary operation if and only if the counterpart is available in the standard RAM model.

Similar to standard quantum computing assumptions, we also assume that all quantum computations are error checked and are decoherence free. We do not capture any error in our pseudocode, but one may want to add error checking code appropriately in practical implementations.

Many quantum algorithms are not exact. They cannot guarantee the correctness of their output, and may even produce incorrect outputs. However,

most of them can, fortunately, guarantee that their output is correct with a probability no less than two thirds. We may rerun the algorithm several times to increase our confidence. Some of the algorithms such as factoring produces positive certificates. As a result, the correctness is guaranteed on positive instances and we only have to deal with those negative instances. The problem of having incorrect outputs may be arguably unavoidable. Firstly, no quantum circuit, including our finally measurement, could be error-free. Secondly, one can prove that some of the speed ups are only available in a bounded error model, where we only guarantee a success probability significantly better than half (generally two thirds). This nondeterminism may also impact the running as well. Some algorithms may run forever if it gets unlucky in every round, but have a very good expected behaviour. In this thesis, we focus on algorithms that succeed with a constant probability significantly larger than half and with a good expected runtime. This expectation is only dependant on probabilistic measurement or coin tosses during the executation of our algorithm and is independent of our input. More precisely, we are interested in the worst expected behaviour among all possible inputs as in [6]. We would also like to point out that quantum computers have the inherit ability to provide pure randomness. It is less likely that an adversary could force our algorithm to take any worst path.

2.6 Summary

We have reviewed the basic notations of quantum information and computation. We have also surveyed a couple of useful gates and reviewed the model of computation generally used in the setting of quantum computations. Our model resembles the RAM model with the addition of quantum registers and quantum gates. We are also more restrictive as we only allow gates that are primitive

(readily implementable), small (with a small unitary transform matrix) and error-recoverable. Since the runtime of some quantum algorithms depend heavily on probabilities and may not even terminate, all our runtime analysis will focus on the worst expected runtime over all possible inputs instead.

Chapter 3

Quantum Search and Amplitude Amplification

Over the pass few decades, people have studied different problems and derived efficient quantum algorithms on them. One of the fundamental problems of interest is the Database Search problem. Given a collection of elements, we would like to find some good entry subject to a given predict. One such example is that we are given a phone book and we would like to search for the company with the telephone (135)792-4680. This task is easy if we are given a phone book sorted according to the phone entry. However, most directories store entries according to the name, and there is no particular ordering on their phone numbers. This is like searching a record from an unordered database, which can be generalized to the satisfiability problem we have mentioned before.

We now formally define the search problem. As input, we are given two items:

- n, denoting the numbers of entries there are.
- F, a predicate function that maps $\{0, \ldots, n-1\} \rightarrow \{\texttt{False}, \texttt{True}\}$. ($F(i) = \texttt{True}$) if and only if the i^{th} entry satisfies the predicate.

The predicate function is implemented as a black box and information can only be extracted through actual invocation of the predicate. As output, we have two possibilities:

- some i such that $F(i) = \texttt{True}$
- $NotFound$

We further denote $m = |F^{-1}(1)|$ as the number of satisfying indices, and $a = \frac{m}{n}$ as the fraction of domain elements satisfying the predicate of satisfying the predicate.

Linear search through the whole domain provides a natural classical solution to this problem. This yields an $O(n)$ linear time algorithm. The runtime is relatively independent of a and m. To understand the difficulty of the problem, we can analyse it with an adversary argument. We take on a role as the oracle and try to answer queries in such a way so as to force the algorithm to run as long as possible. A simple strategy is to provide negative answers until the last m entries. Even in the extreme case with $a = \frac{1}{2}$, any algorithm would still need to scan through half of the set before hitting a good entry. On the other hand, we can also tackle this problem with a randomized algorithm which probes random entires iteratively. The probability that we hit a good entry is a, and it takes expectedly $O(a^{-1})$ time before this algorithm terminates with a good entry. In the extreme case with $a = \frac{1}{2}$, a randomized algorithm can complete the task in expected constant time. There are, however, three drawbacks with the randomized approach:

- This algorithm may never terminate if our random index generator is flawed.
- With low probability this algorithm may take many steps before terminating.
- This algorithm can never terminate if $m = 0$, in which case it should have returned $NotFound$.

The first issue is related to pseudorandomness. We would not discuss the issue in depth as with quantum computers, we have access to full randomness and this issue would be solved. The second issue and the third issue mainly concern about when we should stop trying and how we can avoid testing some entries repeatedly. Theoretically, we can mark entries that we have checked before and avoid them in later iterations. However, the workload in remembering tested entries and avoiding them could impose severe overhead to the algorithm. An alternative approach is that we will keep trying for a fixed number of times (e.g. $8n$). We return a good index immediately if we find one, and return $NotFound$ if we keep failing till the end. We successfully bound our runtime by sacrificing our success probability of outputting a correct answer. We can see that the probability of success with $8n$ trials is at least $\frac{7}{8}$ and the error is also one-sided; we only err when we output $NotFound$ but m is actually positive.

The behaviour of this simple randomized algorithm resembles the scenarios we have discussed in the previous section. As discussed in our model of computation, we are only interested in the worst expected time to achieve the correct answer with error bounded by some constant over all possible cases. This this example, the runtime of the algorithm is given by $O(\min(8n, a^{-1}))$ with a constant error rate. The quantum algorithm that we are going to discuss share many of the features of this randomized algorithm. The same reasoning will still apply.

3.1 Grover's Algorithm

Grover studied the database search problem and derived a quantum algorithm with quadratic speedup in [1]. We follow the original discussion by assuming that $m = 1$ and $n = 2^k$ for some k. Furthermore, we assume that the predicate F is given as a quantum black box $\mathcal{F}$.

With a quantum implementation of the predicate F, we can build a quantum blackbox $\mathcal{F}$ which transform $|i\rangle$ into $(-1)^{F(i)}|i\rangle$. Recall that any algorithm or circuit can be turned into an equivalent quantum box, taking in an extra output bit to which the result is exclusived or-ed in. In previous sections, we have illustrated how we can feed in a $|0\rangle$ as the output bit and read the result after. To implement $\mathcal{F}$, we first direct our input $|i\rangle|0\rangle$ into F to get $|i\rangle|F(i)\rangle$. We then apply the Z transform to our output qubit, which negates the amplitude of the state if and only if $F(i)$ is 1. We then run F again on our state $(-1)^{F(i)}|i\rangle|F(i)\rangle$ to revert the output bit, giving us $(-1)^{F(i)}|i\rangle|0\rangle$. This requires two invocations of F. There is a more clever way by which we can construct our blackbox $\mathcal{F}$ with only one invocation of F, as noted in the work of Boyer et al. [2].

With $m = 1$, we denote the only good index as q. Our goal is to find q among the n possibilities. The first step of Grover's algorithm is to prepare a superposition of all values in the domain, denoted as

$$|\Psi\rangle = \frac{1}{\sqrt{n}}|0\rangle + \frac{1}{\sqrt{n}}|1\rangle + \ldots + \frac{1}{\sqrt{n}}|q\rangle + \ldots + \frac{1}{\sqrt{n}}|n-1\rangle$$

Remember that $n = 2^k$, we can prepare this state simply by applying Hadamard's gate to each of the k qubits. We denote this operation $S = H^k$. We further define $|A\rangle = |q\rangle$ and $|B\rangle = \frac{1}{\sqrt{n-1}}(|0\rangle + |1\rangle + \ldots + |q-1\rangle + |q+1\rangle + \ldots + |n-1\rangle)$. These two states are orthonormal to each others. The state $|A\rangle$ is a uniform mixture of all good indices while $|B\rangle$ is a uniform mixture of all bad indices. Recall that $a = \frac{m}{n} = \frac{1}{n}$, one can verify that the state $|\Psi\rangle$ is equivalent to $\sqrt{a}|A\rangle + \sqrt{1-a}|B\rangle$. Given the state $|\Psi\rangle$, we can perform a measurement and test the outcome against F. The probability of measuring $|A\rangle$ is a. This is essentially our randomized algorithm with a pure random number generator. The main idea of Grover's algorithm is to boost the amplitude associated with $|A\rangle$ to more than half, thereby increasing our probability of success. In fact, we

can also think of the linear scan algorithm as a probability boost to the success probability. At each failed trial, we increase our chance of success from $\frac{1}{i}$ to $\frac{1}{i-1}$. With only a limited rate of increase, such algorithms take approximately $O(a^{-1})$ time to boost the success probability to at least a half.

The quantum algorithm consists of repeated applications of the Grover iteration, given by $-S\mathcal{F}_0S^{-1}\mathcal{F}$, where $\mathcal{F}_0$ refers to the predicate that is satisfied only at 0. To better understand the algorithm, we first draw our state $|\Psi\rangle$ as a vector in the 2D plane spanned by $|A\rangle$ and $|B\rangle$. Although we may have complex amplitudes in the general case, we shall see that we stay in the real domain before and after each iteration. As a result, our current state (denoted as $|\phi\rangle$) can always be written as a vector in the form $x|B\rangle + y|A\rangle$, where x, y are real coefficients. Applying $\mathcal{F}$ invert the y amplitude associated with $|A\rangle$ and is virtual a reflection along the x-axis. The remaining term $-S\mathcal{F}_0S^{-1}$ is actually a reflection along the vector $|\Psi\rangle = \sqrt{1-a}|B\rangle + \sqrt{a}|A\rangle$. To better understand this, we can do a few mathematical tricks. We define another coordinate system with orthonormal basis $|A'\rangle = |0\rangle$ and $|B'\rangle = \frac{1}{\sqrt{n-1}}(|1\rangle + |2\rangle + \ldots + |n-1\rangle)$. By definition, we have $S|A'\rangle = |\Psi\rangle$. Since $|A'\rangle$ and $|B'\rangle$ are orthonormal, it follows that $S|B'\rangle$ must also be orthonormal to $S|A'\rangle = |\Psi\rangle$. As a result, $|\Psi\rangle = S|A'\rangle$ and its normal $S|B'\rangle$ is also an orthonormal basis for our vector. We have three operations in successions $-S\mathcal{F}_0S^{-1}$. The first operator S^{-1} simply converts the coordinate system from $S|A'\rangle$ and $S|B'\rangle$ to $|A'\rangle$ and $|B'\rangle$. After that, $\mathcal{F}_0$ inverts the amplitude associated with $|A'\rangle$. Since we have a global negation after each iteration, this negation along $|A'\rangle$ is undone and the net effect is that the amplitude associated with $|B'\rangle$ is negated. Finally, the final S transform this negation back to the basis vector $S|B'\rangle$. As we have argued, $S|B'\rangle$ is normal to $|\Psi\rangle$. The net effect is that our state is reflected along $|\Psi\rangle$.

Schematically, we have rotated our state about the origin with an angle 2θ

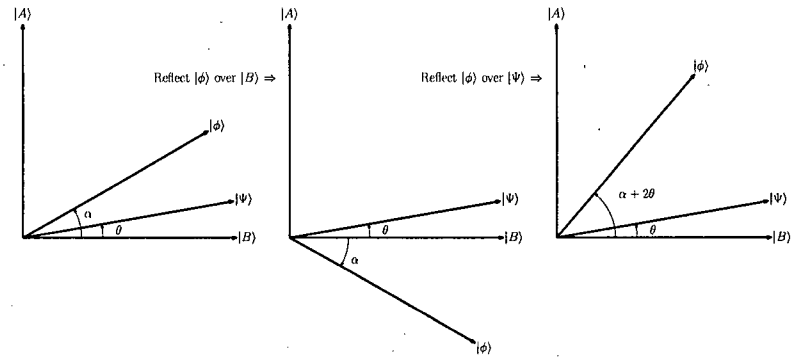

Figure 3.1: One Grover Iteration

(Figure 3.1). Initially, we have $\sin(\theta) = \sqrt{a}$. The probability of getting a good state $|A\rangle$ is $\sin^2(\theta) = a$. After the rotation, the probability of getting a good state is then $\sin^2(\theta + 2\theta)$. After g such rotations, our state would make an angle of $(2g+1)\theta$ with the x-axis. the success probability would then be $\sin^2((2g+1)\theta)$. Ideally, we want $(2g+1)\theta \approx \frac{\pi}{2}$. For small a, we have $\sqrt{a} \approx \sin(\theta) \approx \theta$. We need g to be rougly $\frac{1}{2}(\frac{\pi}{2\theta} - 1)$ for the best result. This translates to an $O(\sqrt{a^{-1}})$ algorithm. Compared to the classical $O(a^{-1})$ algorithm, this provides a quadratic speedup. The pseudo code is presented here for completeness:

```
repeat forever
  |φ⟩ ← |0⟩
  apply H^k on |φ⟩
  θ ← sin^{-1}(1/√n)
  α ← θ
  while α + 2θ ≤ π/2 do
    apply H^k F_0 H^k F on |φ⟩
  end while
  x ← measure |φ⟩
  if F(x) = 1 then
    return x
  end if
end repeat
```

The overall repeat loop is necessary since our state $|\phi\rangle$ may never overlap entirely with $|A\rangle$. We may have to do this a few times before we finally get lucky

with our measurement. In Grover's setting $m = 1$ and this program should never return $NotFound$. The inner Grover iteration make use of S, which is H^k in our case. Note that H^k is self-inverse, and thus S^{-1} is simply H^k. Moreover, we have skipped the global negation after each iteration. That does not affect our measurement probability and can be neglected. The total work within the `repeat` loop takes $O(\sqrt{a^{-1}})$. We can see that we always improve $|\phi\rangle$ until it has a success probability of no less than a half. The expected running time of this algorithm is thus $O(\sqrt{a^{-1}})$. With $m = 1$, it translates to an $O(\sqrt{n})$ algorithm for the database search problem.

Grover's algorithm works by selectively amplifying the amplitude associated with the good term. This is generally referred to as *Amplitude Amplification.* The technique relies on the destructive and constructive interference effect only available in quantum computers. As a result, there is no parallel in classical computers.

Before Grover pubished his algorithm, in 1994, Charles Bennett, Ethan Bernstein, Gilles Brassard and Umesh Vazirani had already proved that an unstructured search for a unique solution over a space of size n requires at least $\Omega(\sqrt{n})$ calls to the query/predicate ($\mathcal{F}$ in our case) [13]. Therefore, the database search problem is optimally solved by Grover's algorithm.

3.2 BBHT

Shortly after Grover's publicaition, Boyer, Brassard, Høyer and Tapp studied the algorithm in greater depth [2] and proposed different ideas to loosen its constraints. In honour with their work, we shall call their algorithm after their initials, `BBHT`.

m is fixed to 1 in Grover's scenario. It is not hard to see that by changing the definition of $|A\rangle$ and $|B\rangle$ accordingly, the algorithm will still proceed in exactly

the same way. The state $|\phi\rangle$ will still rotate about the original with an angle of $2\sin^{-1}(\sqrt{a})$ per iteration. Should we know a, our previous pseudo code would work seamlessly. The same analysis still applies and we will have an algorithm that runs in $O(\sqrt{\frac{n}{m}})$ steps. Note that if we know that $m = 0$, we would have returned $NotFound$ immediately, rather than working through futile iterations.

Another less binding constraint they have tackled is that n need not be a power of 2. As a matter of fact, we only exploited this constraint when we prepare the state $|\Psi\rangle$. Any unitary transform that brings the state $|0\rangle$ to $|\Psi\rangle$ will work. They have suggested the use of a Quantum Fourier Transform for the task. The Quantum Fourier Transform of order n is given by the transition $|i\rangle \rightarrow \sum_{j=0}^{n-1} \omega_n^{ij} |j\rangle$, where ω_n denotes the n^{th} root of unity. On the other hand, we can simply stretch our domain to a power of 2. We wrap our predicate so that any index greater or equal to n will be rejected. The new a will be no less than half of what it used to be, and this translates to no more than a small constant factor ($\leq \sqrt{2}$) on our expected runtime.

In their paper [2], Boyer et al have also tackled the problem when m is unknown. We know that the success probability of the Grover's algorithm depends heavily on how many iterations we are taking. This number also depends heavily on m. The main idea behind the BBHT algorithm is to repeatedly try random number of iterations in a controlled manner. For instance, we can guess that m might be very big and simply measure without doing any amplification. After a few failures, we may believe that m could be smaller and only measure after a few iterations. Furrow [6] further studied the BBHT algorithm and provided a tight error bound for it. We now present the version adopted in Furrow's thesis:

```
procedure BBHT(F)
  g ← 1
  factor ← 1.31
  while g ≤ 2√n do
    |φ⟩ ← |Ψ⟩
    j ← random(m), a random integer in the range [0, g)
    repeat j times
      apply j Grover iterations on |φ⟩
    end repeat
    x ← measure |φ⟩
    if F(x) = 1 then
      return x
    end if
    g ← g × factor
  end while
  return NotFound
end procedure
```

Furrow proves that the probability of failure (returning $NotFound$ even where there is a solution) is less than $.5m^{-.93}$ and the total number of calls to F has an expectation of $\Theta(\sqrt{\frac{n}{m}})$ when $m > 0$. When $m = 0$, all measurements will return a bad index. Since the total amount of work we have to do is bounded by g, which is a geometric sequence bounded by $2\sqrt{n}$, the algorithm will run for $O(\sqrt{n})$ steps before it terminates, returning $NotFound$. As a result, the runtime is bounded by $O(\sqrt{n})$ even when $m = 0$. The actual proof requires a significant amount of math and will not be reproduced in this thesis. Interested readers are advised to read the relevant chapter (Appendix A) in Furrow's thesis [6].

Since the success probability is constant and the error is one sided, we simply assume that we have a BBHT implementation where the success probability is at least $\frac{3}{4}$. This can be achieved by no more than a constant number of invocations of any existing implementation.

BBHT is very powerful since it operates without any prior knowledge about the domain. Boyer et al. have also proved that it is optimal as an blackbox database search algorithm. It is widely used in different algorithms. Its error

is one-sided, and we can boost the success probability simply by rerunning it a few times. It is also a practical algorithm with only a small constant factor in its expectation.

One may also note that all computations in the BBHT pseudo code are rational. The original Grover's algorithm may require computations of angles, including square roots and inverse trigonometric functions which cannot be done in absolute precision. Without the information about m and a, BBHT works entirely in the rational domain and can be coordinated / regulated by ordinary computers.

In addition, we also have the nice uniform distribution on outcome. As we have shown in the Grover's algorithm, the state of our computation can always be represented in the 2D plane spanned by the good vector and the bad vector. The amplitude of all good (and bad) indices are exactly the same in their respective axis. Boyer et al. have also showed that BBHT is unbiased on its output and any satisfying index will be returned with equal probability. As a result, we can treat BBHT as an ideal random sampler.

3.3 Other variants

The technique for Amplitude Amplification can be further fine tuned. For example the BCWZ algorithm invented by Buhrman, Cleve, de Wolf and Zalka [5] tackle the trade-off between running time and the probability of error. Suppose we want to run a database search with error probability bounded by ϵ, it takes $\Omega(\lg \epsilon^{-1})$ rounds of BBHT before we can obtain the desired error bound. Buhrman et al. of [5] showed how one can achieve this in a total expected time $O(\sqrt{n \lg \epsilon^{-1}})$. This introduces a quadratic speed up on the error term but does not take advantage of m even if the latter is large.

Furrow has combined BBHT and BCWZ and created a general error-bounded

search algorithm [6] which take the advantage of both. He has also illustrated how his tools can be used to either improve various known algorithms. However, both of these requires manipulation over irrational number. We will skip over them in this thesis, as some of those techniques require arbitrary manipulation of real numbers. Our ideas only depend ona a search algorithm running in square root time. One can plug in the more elaborated version in place of our `BBHT` where they see fit.

3.4 FindAll

Another fundamental problem people have addressed is the `FindAll` problem. Given a blackbox predicate F, we would like to find all satisfying entries of it. Since we will also make extensive use of it, we would rederive it with the standard `BBHT`.

In the last paragraph of Quantum Counting [14], Brassard et al. have discussed the possibility to repeatedly invoke `BBHT` in order to count how many satisfying indices there are. Although this runs with a larger memory overhead in terms of the counting problem, this approach can be generalized to find the whole satisfying set. A naive iterative `BBHT` approach has two main drawbacks: (i) we may get unnecessary duplicates; (ii) we do not know when to stop. One may try to do complicated analysis to see how we can accomodate with a pure probabilistic point of view. We can also estimate the number of good entries by Quantum Counting [14]. On the other hand, we can simply mark seen entries. For instance, we can keep a Boolean array of size n. Each of the entries marks if we have already collected the corresponding item or not. With this we define our augmented predicate F' such that i is good if and only if $F(i) = 1$ and i is not marked before. With a linear array, the later condition can be checked in constant time and the running time of our augmented oracle is still dominated

by the original predicate. The pseudo code is thus:

```
procedure FindAll(F)
  seen[*] ← False
  Result ← ∅
  repeat forever
    x ← run BBHT with the predicate F'(i) = 1
          if and only if seen[i] = False and F(i) = 1
    if x = NotFound then
      return Result
    end if
    add x to Result
  end repeat
end procedure
```

Since we have marked seen entries, BBHT will always return an unseen good entry or return $NotFound$ when all good entries are seen. The potential error of this code is also one-sided, in that we may terminate before we have depleted all the good entries. At termination, we must have called BBHT while it returns NotFound. This is a certificate that our algorithm has correctly terminated. As a result, the error rate of this FindAll procedure only depends on how good this certificate is. Since BBHT has a bounded error of at most $\frac{1}{4}$, our FindAll procedure will also have a bounded error of at most $\frac{1}{4}$.

The running time of this procedure depends on both n and m. Initially, we have m good indices and it takes no more than $c\sqrt{\frac{n}{m}}$ time to retrieve any one of them, for some appropriately constant c. After marking the first seen entry, we have only $m-1$ good entries and it takes no more than $c\sqrt{\frac{n}{m-1}}$ time to retrieve the next one. Finally, we will have explored all good entries, bring m to 0. BBHT will take at most $c'\sqrt{n}$ time before returning $NotFound$, certifying that all good indices have been explored. Continuing this way, the overall runtime

is no more than:

$$\begin{aligned} T_{findall} &\leq \sum_{k=1}^{m} c\frac{n}{k} + c'\sqrt{n} \\ &\leq c\sqrt{n}\int_0^m \frac{1}{\sqrt{x}}dx + c'\sqrt{n} \\ &= c\sqrt{n}[2\sqrt{x}]_0^m + c'\sqrt{n} \\ &= 2c\sqrt{nm} + c'\sqrt{n} = O(\sqrt{nm}) \end{aligned}$$

This algorithm thus gives us a way to sample all good indices in $O(\sqrt{nm})$ time. When $m = n$, it will take linear time which matches the corresponding classical bound. The drawback of this approach is that we need to use linear memory [1] It seems that initializing the memory may dominate the total runtime. Luckily, we have many ways to bypass this. For instance, we can reuse our allocated memory across runs. We can also perform zero-initialization using 2 arrays and several counters. Interested readers may consult Exercise 11.1-4 from *Introduction to Algorithms* [15].

[1] We use a Boolean array of size n to store the marking information. One may use a balanced binary search tree of size $O(m)$ for the same purpose but it takes $O(\lg m)$ to check if something is marked

Chapter 4

Quantum Unary Extrema Finding

We have already discussed the problem of looking for the extremum of a set when we are only given an unary predicate. We have reviewed that the problem takes $\Theta(n+\lg|\mathscr{U}|)$ time to solve classically and have described an optimal algorithm that achieves this bound. We now begin our discussion on how quantum algorithm could provide asymptotic speed up which is not possible classically.

We begin by showing an intrinsic lower bound of this problem. After that, we study related algorithms and see how we can improve them. Our upper bound is derived through a series of optimizations and different techniques applied can be combined to create better results. We will present each technique sequentially with their motivations and effects. We also maintain that our algorithm succeed with a probability at least $\frac{3}{4}$, on par with BBHT.

4.1 Lower bound

We begin by deriving the intrinsic lower bound for this problem. This gives us an idea how difficult this problem is and what we are aiming for. Before we jump right into the extrema problem, we first review two lower bound results from [16] and [17].

Theorem 2 *(Beals et al.[16]: Theorem 4.8 & 4.10) Suppose we are given a blackbox Boolean oracle function F. Any bounded error quantum algorithm must make at least $\Omega(\sqrt{n})$ oracle queries to verify that there is at least one image with value* `True`.

This also implies that the verification of the predicate "$\forall x F(x) =$ `False`" requires $\Omega(\sqrt{n})$ oracle calls. In our problem, suppose we want to verify the maximality of a candidate value λ, it is equivalent to asking if the blackbox oracle function $F(i) = (\mathscr{O}(i, \lambda) = \text{`} \succ \text{'})$ is unsatsifiable or not. On the other hand, given a blackbox Boolean oracle function F, we can construct an equivalent maximum finding problem with

- $\mathscr{U} = \{0, 1\}$
- $\mathscr{O}(i,k) = \begin{cases} \text{`} \prec \text{'} \text{ if } k = 1 \text{ and } F(i) = \texttt{False} \\ \text{`} \equiv \text{'} \text{ if } (k = 0 \text{ and } F(i) = \texttt{False}) \text{ or } (k = 1 \text{ and } F(i) = \texttt{True}) \\ \text{`} \succ \text{'} \text{ if } k = 0 \text{ and } F(i) = \texttt{True} \end{cases}$

We can then run any unary maximum finding algorithm to find the "maximum" of those n entries. By construction, the function F is satifisable when and only when the maximum of these n numbers is 1. Therefore, the lower bound follows and that any quantum unary maximum finding algorithm must make at least $\Omega(\sqrt{n})$ queries to the blackbox function F. Note that each call to our $\mathscr{O}$ defined above can only replace a constant number of invocations of F. As a result, any unary maximum finding algorithm must also make at least $\Omega(\sqrt{n})$ calls to our oracle.

In addition, we have another theorem about ordered searching in a domain:

Theorem 3 *(Høyer et al. Theorem 1) Suppose we are given a totally ordered universe $\mathscr{U}$ and a value u drawn from the universe. Any bounded error quantum*

algorithm determining the value of u with only a blackbox threshold predicate function requires at least $\Omega(\lg|\mathscr{U}|)$ *calls to the oracle.*

When $n = 1$, the maximum find problem naturally reduces to an ordered search problem. As a result, any quantum algorithm solving the quantum unary extrema finding problem requires at least $\Omega(\lg|\mathscr{U}|)$ calls to the oracle predicate.

Combining these two reduction ideas, we have the theorem for the lower bound in the Quantum Unary Extreama finding problem:

Theorem 4 *Any bounded error quantum algorithm solving the quantum unary extreama finding problem requires* $\Omega(\sqrt{n} + \lg|\mathscr{U}|)$ *calls to the predicate oracle.*

A full algorithm may contain extra operations other than oracle calls. This lower bound trivially follows in the more general case.

4.2 Extending previous algorithms

We start our discussion by reviewing the optimal quantum algorithm [7] for maximum finding in the binary comparison model. We denote the original set $S = \{x_i | 0 \leq i < n\}$. The algorithm starts by picking a random element x_i as a candidate λ. It serves as an underestimate of the global maximum, and can be treated as a filter. Elements that are smaller than λ are certainly not maximal and can be neglected. We refer to elements larger than λ as *active elements* and their corresponding set $S' = \{x_i | x_i > \lambda\}$ the *active set.* We iteratively sample an element from S' and update λ to it accordingly. When S' becomes $\emptyset$, the algorithm terminates with λ being the maximum.

The sampling from S' is a typical search problem under a predicate and is best solved by `BBHT`. We denote the size of S' as m. Each selection will expectedly call the oracle $O(\sqrt{\frac{n}{m}})$ number of times. In the binary comparison model, λ can be specified by an index. Given the index of the candidate, we

can verify if another index corresponds to a bigger entry with a single call to the binary oracle. The selection is thus of cost $O(\sqrt{\frac{n}{m}})$. If we restrict ourselves to the unary comparison model, checking if an index corresponds to a larger entry requires a binary search of cost $O(\lg|\mathscr{U}|)$. As a result, each selection will cost expectedly $O(\sqrt{\frac{n}{m}}\lg|\mathscr{U}|)$. Dürr and Høyer proved that their algorithm runs with $O(\sqrt{n})$ calls to the comparison oracle. In the binary model, this algorithms runs in $O(\sqrt{n})$ time and is optimal. In the unary model, this straight replacement of the comparison function will yield an $O(\sqrt{n}\lg|\mathscr{U}|)$ algorithm.

We shall rederive the runtime as it gives us insight how this algorithm progresses. Moreover, we will show an alternative adaptation of this algorithm. Initially, we start with a threshold $\lambda = -\infty$ and $m = n$. The BBHT algorithm will uniformly select a random sample among the m available larger elements. With probability at least half, we would have selected an entry at least as large as the median. Setting λ to that element will halve the active set, reducing m to at most $\frac{m}{2}$. Such halving will occur in expectedly 2 iterations. It takes expectedly $2\lg m+1$ iterations before m is reduced to 0, when the algorithm terminates with the correct answer. We now divide the execution of the algorithm in different stages.

Suppose we start with $m = m_0 = n$, we divide the executions into $\lg m_0$ stages. We call the algorithm being in stage i if $\lfloor\frac{m_0}{2^i}\rfloor \geq m > \lfloor\frac{m_0}{2^{i+1}}\rfloor$. The BBHT selection at stage i requires $O(\sqrt{\frac{n}{m}}) = O(\sqrt{\frac{2^{i+1}n}{m_0}})$ calls to the comparison function. The expected total number of comparison calls in selection is the sum over all stages, giving us $O(\sum_{i=0}^{\lg m}\sqrt{\frac{2^{i+1}n}{m_0}})$, a geometric series dominated by the last term $\sqrt{n}$. In the binary comparison model, each oracle call is $O(1)$ and the expected total selection cost is $O(\sqrt{n})$. In the unary comparison model, where binary comparisons can be simulated with $O(\lg|\mathscr{U}|)$ oracle calls, the expected total selection cost is then $O(\sqrt{n}\lg|\mathscr{U}|)$. Note that this $O(\sqrt{n})$ comparison cost

depends only on the final $\sqrt{n}$ term and is independent on our initial threshold or m_0. At the end of each stage, we will also update λ which takes constant time if we only store the index. This adds an extra $O(\lg m_0)$ term to the total runtime. However, with m_0 upper bounded by n, this does not impact the total run time.

It may already be apparent that we can do more work per update in order to ease the workload of comparison during BBHT. In fact, we can update λ to the *exact* value of our chosen index. With the exact value, each comparison can be done in a single call to the oracle. Updating λ to the exact value requires a full binary search at the end of each stage. The pseudo code is listed as follows:

```
procedure Sample_And_Improve(𝒪, λ₀)
  λ ← λ₀
  do
    t ← BBHT(x_i > λ)
    if t ≠ NotFound then
      λ ← binary_search(t)
    end if
  while t ≠ NotFound
  return λ
end procedure
```

The BBHT selection is done with respect to a predicate, which is a function on the index i, denoted as $f(i) = (x_i > \lambda) \equiv (\mathcal{O}(i, \lambda) = \text{`} \succ \text{'})$ Moreover, we pass in λ_0 as our initial guess. This defines m_0 and could potentially improve our performance if set properly.

Although a single binary search is costly, it only occurs an expected $O(\lg m_0)$ number of times. Therefore, this unary maximum finding algorithm spents $O(\sqrt{n})$ time in running BBHT for sampling, and $O(\lg |\mathcal{U}| \lg m_0)$ time updating λ. We now have the following theorem:

Theorem 5 Sample_And_Improve *solves the problem of unary maximum finding in expected time* $O(\sqrt{n} + \lg |\mathcal{U}| \lg m_0)$ *where* m_0 *refers to the number of active elements we have started with. It succeeds with probability at least half.*

We can start with $\lambda_0 = -\infty$, thereby setting $m_0 = n$ and giving us an $O(\sqrt{n} + \lg|\mathscr{U}| \lg n)$ upperbound for the unary maximum finding problem.

It is important to also analyze the success probability of this algorithm. It is easy to see that the feasibility of λ is guaranteed. The only error we could run into is that the final BBHT call returned NotFound when there were still active element(s) to sample. As a result, our error probability is exactly the same as that of BBHT, which is a constant no more than $\frac{1}{4}$.

The runtime consists of three separate terms:

- $\sqrt{n}$: The optimal cost associated with selecting and verification of maximality. We refer to this term as the BBHT *term.*
- $\lg m_0$: The cost to reduce m to 0, indicating how many iterations one has to do. We call this term as the *m-reduction term.*
- $\lg|\mathscr{U}|$: The optimal cost in retrieving the value given an index. This term is referred as the *binary search term.*

As we shall see, the term $\lg m_0$ is far from optimal. Moreover, we can move this m-reduction term from the binary search term to the BBHT term. We will devote the following sections to techniques that achieve these improvements. We begin by showing how we can reduce m to 0 in $\lg\lg m$ time using larger samples.

4.3 Budgeted Sampling

Our Sample_And_Improve algorithm samples *one* active element at a time and expectedly reduce m to $\frac{m}{2}$ each time. We can also sample $k > 1$ active elements and update λ to the maximum of the k elements. The following lemma shows how effective this reduction is.

Lemma 1 *Given a sample of k randomly chosen active elements with replacement, setting the threshold to the maximum of this sample will expectedly prune*

the active set to size no more than $\frac{m}{k+1}$*, where* m *denotes the original size of the active set.*

The proof is presented in Appendix A.

Recall that the cost of selecting an entry is very sensitive to m. When $m = n$, each selection is essentially free and we can afford a huge sample of size $\sqrt{n}$. When m becomes $\sqrt{n}$, each selection costs $n^{\frac{1}{4}}$. Since our target is bounded by $\sqrt{n}$, we should restrict our k to $O(n^{\frac{1}{4}})$ to get this close to optimal. It seems that we must choose our k according to m, which is unknown throughout the course of our algorithm. Fortunately, we can perform the calculation in reverse. Our primary concern is that the selection step should not take more than $O(\sqrt{n})$ time as that breaks optimality immediately. As a result, we can budget our selection. Within each iteration, we limit ourselves to a fixed budget [1] of $4\sqrt{n}$ and continuously run BBHT to sample elements. The following code samples as much as possible given a budget and a predicate by repeated invocation of BBHT.

```
procedure Budgeted_Sampling(B, F)
  T ← ∅
  while B > 0 do
    t ← BBHT(F) where we limit the number of iterations within BBHT by B
    if t ≠ NotFound then
      Add t into T
    end if
    Decrease B by the number of inner iterations the previous BBHT call has used.
  end while
  return T
end procedure
```

We then figure out the maximum of the sample and update our λ to it. This is repeated until we fail to sample anything with our budget of $2\sqrt{n}$. Note that $2\sqrt{n}$ is generally more than enough for BBHT to sample even if there was only a single element. Failure to sample anything is thus a good certificate that our

[1] The stated cost $\sqrt{n}$ is normalized. Suppose BBHT takes $c\sqrt{\frac{n}{m}}$ to sample an element, we should allow ourselves a budget of $4c\sqrt{n}$.

λ is indeed the maximum. We may redo this a number of times to boost our success probability on demand. Similar to the previous algorithm, we simply keep the success probability equivalent to that of BBHT and would not rerun our sampling too many times. We then return λ. The following pseudo code captures the main ideas:

```
procedure Budgeted_Sample_And_Improve(𝒪, λ0)
  λ ← λ0
  while True do
    T ← Budgeted_Sampling(4√n, x_i > λ)
    if T = ∅ then
      break
    end if
    λ ← max(T)
  end while
  return λ
end procedure
```

The runtime of this algorithm is simply the number of iterations we need, multiplied by the time spent within each iteration. We have already limited our sampling step to be $O(\sqrt{n})$, which also limits the size of the sample to be $O(\sqrt{n})$. We have to find the maximum of our sample T in order to update λ, and this can be done through the classical algorithm in time $O(|T| + \lg|\mathscr{U}|) = O(\sqrt{n} + \lg|\mathscr{U}|)$. The work within each iteration is thus $O(\sqrt{n} + \lg|\mathscr{U}|)$. The number of iterations can be modelled as how fast m approaches 0. As before, we denote the initial size of the active set (induced by λ_0) as m_0 and divide the execution into several stages. We say that our execution is in stage i if $m_0^{\frac{1}{2^{i+1}}} < m \leq m_0^{\frac{1}{2^i}}$. Numerically, we start with stage 0, entering stage 1 only when m drops below $\sqrt{m_0}$ and entering stage 2 only when m drops below $\sqrt{\sqrt{m_0}}$. By solving the equation $m_0^{\frac{1}{2^i}} \leq 2$, we know that m reaches 2 after stage $\lg\lg m_0$ At that point, it takes at most 2 iterations to reduce m to 0. As a result, there are $\Theta(\lg\lg m_0)$ stages. The following lemma tells us how many iterations we have to perform to proceed from stage i to stage $i+1$.

Lemma 2 *Suppose we devote a budget of B and run* `BBHT` *repeatedly to get a sample. We then update our threshold to the maximum of this sample. The expected pruning factor of this process is $O(B\sqrt{\frac{m}{n}})$.*

The proof is presented in Appendix B.

With a budget of $4\sqrt{n}$ per iteration, it takes expectedly constant number of iterations to transit from state i to state $i+1$. We are also spending $O(\sqrt{n} + \lg|\mathscr{U}|)$ per iteration, giving a total runtime of $O((\sqrt{n} + \lg|\mathscr{U}|)\lg\lg m_0)$.

Theorem 6 `Budgeted_Sample_And_Improve` *solves the problem of unary maximum finding in expected time $O((\sqrt{n}+\lg|\mathscr{U}|)\lg\lg m_0)$ with a success probability greater than half.*

Basically, `Budgeted_Sampling` offers us a way to sample without knowing m, and the returned set has a high pruning ratio. At later stages, `Budgeted_Sampling` becomes an ordinary `BBHT`, useful in both sampling and verification. As a result, our algorithm share the same success probability as a `BBHT`. On the other hand, with our huge initial pruning, we are able to reduce m to 0 in $O(\lg\lg m_0)$ iterations. Once again, m_0 is bounded by n and this algorithm has an upper bound of $O((\sqrt{n} + \lg|\mathscr{U}|)\lg\lg n)$. Moreover, the success probability of `Budgeted_Sample_And_Improve` is inherited from `BBHT`, which is a constant.

4.4 Geometric Budgeting

In our first algorithm, the m reduction term is only attached to the binary search term. In our second algorithm our m reduction term is linked with both the `BBHT` and the binary search term. Depending on the relative size of $\sqrt{n}$ and $\lg|\mathscr{U}|$, one algorithm could be better than the other. Luckily, we can apply tricks and optimize the runtime so that the $\lg\lg m_0$ reduction term is only attached to the binary search term. This gives us a definite improvement over

the simple `Sample_And_Improve`.

The main idea is that we can apply the concept of `Budgeted_Sampling` to prune m substantially in time $O(\sqrt{n} + \lg|\mathscr{U}| \lg\lg m_0)$. After that, we can complete the algorithm using the original `Sample_And_Improve`, taking advantage of the much reduced m. The whole procedure is given as follows:

```
procedure Geometric_Budgeting(𝒪, λ₀)
  B ← O(√n)
  λ ← λ₀
  while True do
    B ← B/√2
    T ← Budgeted_Sampling(B, 𝒪(i, λ))
    if T = ∅ then
      break
    end if
    λ ← max(T)
  end while
  return Sample_And_Improve(𝒪, λ)
end procedure
```

This procedure is almost identical to `Budgeted_Sample_And_Improve` except that it decreases B over iterations and perform an additional `Sample_And_Improve` at the end of the procedure. In our previous version, our sampling step serves both as a sampler and a verifier. However, since we are decreasing B every iteration, B could be substantially less than $\sqrt{n}$ when T becomes empty. More precisely, we exit the loop in `Budgeted_Sample_And_Improve` when m drops to 0. In this version, we terminate the loop when m becomes too small and a budget of B is no longer enough to even sample an element. Since λ is unlikely to be the maximum, we follow up by running an extra `Sample_And_Improve` to retrieve the maximum. This trailing call establishes our correctness.

We begin our runtime analysis by pointing out that the work need for each iteration is $O(B + \lg|\mathscr{U}|)$. We pay B to sample a set of size bounded by B and retrieve its maximum with the classical algorithm, resulting in a total of $O(B + \lg|\mathscr{U}|)$. There are again two contributors to this runtime, the sampling

(B) and the binary search ($\lg|\mathscr{U}|$). We already know that B is decreasing geometrically (hence the name). Therefore, the total amount of work spent in sampling is the sum of a geometric sequence dominated by the first $O(\sqrt{n})$ term. This essentially decouples the m-reduction term from the sampling term. However, we still pay the full price of $\lg|\mathscr{U}|$ in each iteration. We now trace through the reduction of m over the iterations. We will extensively use the result Lemma 2. To simplify our notations, we will scale our budget and all the runtime to the same constant. Given a budget of B, we should have expectedly reduced m to $\frac{B}{\sqrt{n}}\sqrt{m}$. We now define m_i to be the expected value of m after i iterations. We again start with m_0. We apply a budget of $\sqrt{\frac{n}{2}}$ in the first iteration, which expectedly reduces m to $\sqrt{2m}$. As a result, $m_1 = \sqrt{2m_0}$. In the next iteration, we would apply a budget of $\sqrt{\frac{n}{4}}$, giving us $m_2 = \sqrt{4m_1}$. At the $(i+1)^{th}$ iteration, we apply a budget of $\sqrt{\frac{n}{2^i}}$ and reduce m from m_i to $m_{i+1} = \sqrt{2^i m_i}$. Note that it is not mathematically rigorous to directly use the expectation in our formulation. There is a hidden distribution over what m could be, and our reduction is simply a constant times the square root function. Luckily, we only need an upper bound for m and since the square root function is concave, by Jensen's inequality we have $2^i\sqrt{E[X]} \geq E[2^i\sqrt{X}]$. Our m_i serve as good upperbounds.

Expanding $m_{i+1} = \sqrt{2^i m_i}$, we have

$$m_i = m_0^{\frac{1}{2^i}} 2^{\frac{1}{2^i}(1+2\cdot 2+3\cdot 2^2+\ldots+i\cdot 2^{i-1})} \leq 2^i m_0^{\frac{1}{2^i}}$$

We terminate our loop when we failed to sample anything. For that particular iteration, the value of m_i does not change. Intuitively, that brings us $2^{i+1}m_0^{\frac{1}{2^{i+1}}} = 2^i m_0^{\frac{1}{2^i}}$. As a result, our pruning would halt at around the $(\lg\lg m_0)^{th}$ iteration. More formally, after $\lg\lg m_0 + 2$ iterations, our m_i becomes $2^{\frac{1}{4}}\lg m_0$, while our budget becomes $\sqrt{\frac{n}{4\lg m_0}}$. By the runtime analysis

of BBHT, the next BBHT will fail to sample anything with probability at least half. As a result, the while loop will continue expectedly a constant number of times before it stops. The total number of iterations the while loop has done is therefore $O(\lg \lg m_0)$. The total time of this pruning algorithm is thus $O(\sqrt{n} + \lg |\mathscr{U}| \lg \lg m_0)$.

We can then analyse the behaviour of the final `Sample_And_Improve`, which relies heavily on the final m after the pruning process. It is also highly dependent on how many pruning iterations we have done, and how much budget we have before exiting the loop. In order to bound the remaining runtime, we make use of the following lemma:

Lemma 3 *Suppose we have failed to sample anything with* BBHT *running against a budget B, the expected number of good elements is bounded by $O(\frac{n^2}{B^4})$*

The proof is presented in Appendix B.

Suppose we have done t pruning iterations, our budget before exiting the loop would be $\sqrt{\frac{n}{2^t}}$. By Lemma 3, the expected number of remaining larger entries is bounded by $O(2^{2t})$. The runtime of `Sample_And_Improve` depends on the logarithm of the number of remaining elements. Again, as the lg function is concave, we have $E[\sqrt{n} + \lg |\mathscr{U}| \lg m] \leq \sqrt{n} + \lg |\mathscr{U}| \lg E[m] \leq \sqrt{n} + \lg |\mathscr{U}| (2t + const)$. Intuitively, since a budget of $\sqrt{\frac{B}{2^t}}$ does not guarantee maximality, we have to fix the potential error incurred by insufficient budget. `Sample_And_Improve` works by repeatedly selecting elements until we hit a budget of $\sqrt{n}$ and certify maximality with higher confidence. Each iteration within `Sample_And_Improve` improves our confidence in getting the maximum entry. As a result, we will likely spend only twice as many iterations as we have spent on our pruning step. Since the expected number of pruning iteration is bounded by $O(\lg \lg m_0)$, the expected runtime of the final `Sample_And_Improve` step will also be $O(\sqrt{n} + \lg |\mathscr{U}| \lg \lg m_0)$. The total runtime is thus $O(\sqrt{n} + \lg |\mathscr{U}| \lg \lg m_0)$.

The success probability of `Geometric_Budgeting` clearly shares with that of `Sample_And_Improve`. As such, we have the following theorem:

Theorem 7 `Geometric_Budgeting` *solves the problem of unary maximum finding in expectedly* $O(\sqrt{n}+\lg|\mathscr{U}|\lg\lg m_0)$ *time, and with a constant success probability shared with* `Sample_And_Improve`.

As we can see, blending in the idea of geometric budget can isolate the BBHT term from the m-reduction term. This also suggest that if we have a faster m-reduction procedure, we can probably apply the same budgeting idea to isolate the BBHT search term from the m-reduction factor.

4.5 Progressive Approximation

Our previous algorithms perform well with small $\lg|\mathscr{U}|$. When $\lg|\mathscr{U}| = O(\frac{\sqrt{n}}{\lg\lg n})$, `Geometric_Budgeting` solves the problem optimally. Unfortunately, it performs worse than the classical optimal algorithm when $\lg|\mathscr{U}| = \omega(n)$. Looking into our algorithms, we see that some of the binary searches are wasted. For instance, we may not need to perform full binary searches in earlier iterations if their results will be overwritten by later candidates. However, it is also necessary to do some of the binary searches and increase our threshold, since it boosts our probability of finding large elements. To balance the two factors, we seek insight from maximum root approximation problem.

In the maximum root approximation problem, our universe spans the interval $[0,1)$ and we work with buckets of size ϵ. There are $\frac{1}{\epsilon}$ buckets and our problem corresponds to finding which in which bucket the maximum root lies. Given any universe, we can also normalize it so that it spans the inverval $[0,1)$ and each value in the universe corresponding to a bucket of size $\frac{1}{|\mathscr{U}|}$ in the interval. A solution to any of the two problems can easily be translated to fit the setting

of the other problem.

Similar to most approximation algorithm, we can tackle the problem gradually with decreasing error each step. For instance, we can first approximate the maximum within an error of $\frac{1}{2\sqrt{n}}$. This translate to a unary maximum finding with $\lg|\mathscr{U}| = \sqrt{n}$. We can apply the `Budgeted_Sample_And_Improve` to solve this problem in $O(\sqrt{n}\lg\lg n)$ time. With this approximated candidate, we can carry on and approximate the maximum within an error of $\frac{1}{2^2\sqrt{n}}$. Although the error is much smaller than that of the previous iteration, the size of the restricted universe remains the same. We start with an interval of size $\frac{1}{2\sqrt{n}}$ and our final bucket size is $\frac{1}{2^2\sqrt{n}}$. It suffices to divide our starting interval into $2\sqrt{n}$ buckets, translating to an universe with $\lg|\mathscr{U}| = \sqrt{n}$. Solving this approximation problem takes another $O(\sqrt{n}\lg\lg n)$ time. In a total run, we have $\lfloor\frac{\lg|\mathscr{U}|}{\sqrt{n}}\rfloor + 1$ approximation problems to solve and each takes $O(\sqrt{n}\lg\lg n)$ time. It seems that the algorithm runs in $O(\sqrt{n}\lg\lg n + \lg|\mathscr{U}|\lg\lg n)$ time. Even if we start with some estimate with a better m_0, our runtime is no better than that of `Budgeted_Sample_And_Improve`. Moreover, the correctness of this algorithm depends on all subproblems being correct. The success probability of such procedure would not be constant.

Before we deal with the runtime, we should first remedy the error incurred by successive invocations of probabilistic algorithms. The error of this algorithm is huge because once our approximation deviates from the actual maximum, we have no correction mechanism to fix that. Each of our previous algorithms is a self contained box that ends with a failed `BBHT`, with a low probability of error. However, this algorithm has numerous failed `BBHT` before it terminates. Should any of them err, our approximate will be wrong. Fortunately, we can also detect such errors. Suppose we start with an interval $[l, u)$ and we subdivide the interval into $2\sqrt{n}$ buckets. `Budgeted_Sample_And_Improve` would first call

BBHT repeatedly to get samples above l. It is easy to also check if any of the sampled elements are larger than u. If that is the case, it means that at least one of our previous approximations is wrong and we should backtrack. This guarantees that errors are checked and corrected over the execution. Indeed, we keep our promise that whenever we return a value, its maximality is backed by a failed BBHT to sample anything bigger and this certificate is at least half correct. The success probability is ensured.

We now present the pseudocode of our algorithm and analyse its runtime afterward. We first describe it from the prespective of the maximum root approximation problem.

```
procedure Progressive_Approximation(𝒪, ϵ)
  λ ← 0; δ ← 1; u ← δ
  while δ > ϵ do
    while True do // loop (*)
      T ← Budgeted_Sampling(4√n, 𝒪(i, λ))
      if T = ∅ then
        δ ← δ/(2√n)
        u ← λ + δ
        break
      end if
      while ∃x ∈ T s.t. x ≥ u do
        δ ← 2√n δ // backtrack to a larger delta
        u ← λ + 2√n δ // Reset the upper bound
      end while
      λ ← max(T) with error δ
    end while
  end while
  return λ
end procedure
```

Equivalently, the pseudocode for the general unary maximum finding problem is:

```
procedure Progressive_Approximation(𝒪, λ₀)
  λ ← λ₀; Δ ← 2^⌊lg|𝒰|⌋; u ← Δ
  while Δ > 1 do
    while True do // loop (*)
      T ← Budgeted_Sampling(4√n, 𝒪(i, λ))
      if T = ∅ then
        Δ ← Δ/2^√n
        u ← λ + Δ
        break
      end if
      while ∃x ∈ T s.t. x ≥ u do
        Δ ← 2^√n Δ // backtrack to a larger delta
        u ← λ + 2^√n Δ // Reset the upper bound
      end while
      λ ← max(T) with error Δ
    end while
  end while
  return λ
end procedure
```

We now derive the runtime of this algorithm. We first remark that the extra checks for error do not incur any major penalty in the overall complexity. Our algorithm will try to solve the problem when $\Delta = |\mathscr{U}|, \frac{|\mathscr{U}|}{2^{\sqrt{n}}}$, etc. Our error checking may cause each problem with a specific Δ be run more than once. Moreover, the backtracking may be cascading, restarting more than one previous iterations. On the other hand, an instance would only be restarted in two situations. Firstly, it will be restarted if it did not find the actual maximum (there were larger elements but they went undetected). It would also be restarted in a cascading backtracking if, instead of it detecting the error, some later iteration detects the error. Furthermore, these two causes will only occur if the final BBHT maximality certificate was wrong, which happen at most half the time. As a result, the expected number we have to restart some iteration with a specific Δ is no more than a constant. As a result, we only have to focus on the no-error scenario and multiply the expected runtime by 2.

We have already derived a loose bound of $O((\sqrt{n} + \lg|\mathscr{U}|)\lg\lg m_0)$. Fortunately, we can prove a much tighter bound. The total work done within each

iteration of loop (*) is clearly $O(\sqrt{n})$. In the errorless case, each iteration will progress by either (i) continuing with a smaller Δ or (ii) getting a larger estimate for λ. We call the former an instance of *Type I* work and later an instance of *Type II* work. The algorithm can only perform Type I work $O(\frac{\lg|\mathscr{U}|}{\sqrt{n}})$ number of times. Moreover, each instance of Type II work would contribute to a reduction on m. As we have discussed before, it takes expectedly $O(\lg\lg m_0)$ instances of Type II work to reduce m to 1, after which all work would be of type I. This more careful charging and counting scheme shows that the total runtime is $O(\sqrt{n}\lg\lg m_0 + \lg|\mathscr{U}|)$. We have finally isolated the m-reduction term from the binary search term. As the penalty from correcting errors is no more than a constant, we have the following:

Theorem 8 `Progressive_Approximation` *solves the problem of unary maximum finding in expected time* $O(\sqrt{n}\lg\lg m_0 + \lg|\mathscr{U}|)$ *with constant success probability.*

A major observation is that the reduction on m is shared across all iterations. As a result, we only pay once for each reduction on m. The same approximation and charging technique can be applied on `Sample_And_Improve` to yield an algorithm that runs in expected time $O(\sqrt{n}\lg m_0 + \lg|\mathscr{U}|)$. These algorithms are also asymptotically at least as fast as the classical optimal algorithm in all cases.

In addition, we can use the same idea to prove that the problem of unary maximum finding with $\lg|\mathscr{U}| = \sqrt{n}$ is almost as hard as the general problem. If we can optimally solve the problem in the general setting, having an algorithm that runs in $O(\sqrt{n}+\lg|\mathscr{U}|)$ time, we can solve the restricted problem in $O(\sqrt{n})$ time. On the other hand, if we can solve the restricted problem in optimal $O(\sqrt{n})$ time, applying progressive approximation will yield an algorithm that runs in $O(\sqrt{n}+\lg|\mathscr{U}|)$ time.

Theorem 9 *The optimality condition of the general unary maximum finding problem is the same as the one with* $\lg|\mathscr{U}| = \sqrt{n}$. *An optimal algorithm in one of the cases can be modified to solve the other case optimally.*

4.6 Pseudo Large Sampling

`Geometric_Budgeting` and `Progressive_Approximation` show us how we can shift the m-reduction term between the `BBHT` and the binary search term. As much as we want the m-reduction term to be constant, it is also unintuitive as why it could be so. We will devote this section to our current best result in reducing m quickly. We have extensively used the idea of large samples to accelerate the speed we reduce m. We have already shown in Theorem 1 that a sample of size k has a pruning factor of $\frac{1}{k+1}$. Literally, it is always better to use huge samples. Yet, there are two obstacles. Firstly, finding the maximum of the sample may take too long. After all, it is too expensive to sample the whole set and reduce m in "one" step. Secondly, the sampling step may take too long. If our sample consists of more than $\sqrt{n} + \lg|\mathscr{U}|$ elements, writing down the sample would take longer time than we are willing to invest.

In `Budgeted_Sample_And_Improve`, the maximum of the sample is retrieved via the classical optimal algorithm. As a result, the size of the sample is bounded by $\sqrt{n} + \lg|\mathscr{U}|$. Luckily, with the `Progressive_Approximation` procedure, we can afford a sample of size $\frac{n}{(\lg\lg n)^2}$. Finding the maximum of such sample takes no more time than $O(\sqrt{\frac{n}{(\lg\lg n)^2}}\lg\lg n + \lg|\mathscr{U}|) = O(\sqrt{n} + \lg|\mathscr{U}|)$ expectedly.

In general, suppose we have a unary maximum finding algorithm with running time $O(f(n) + \lg|\mathscr{U}|)$ and we have a target of $O(\sqrt{n} + \lg|\mathscr{U}|)$, we can reversely calculate how big our sample T should be with the relation $f(|T|) = \sqrt{n}$. With a much larger size limit, it seems that the second problem is our only obstacle. Surprisingly, we can perform large sampling in constant time.

We start by assuming that n is a prime number. If n is not a prime number, we can pad 0 to our input set until the size is a prime number. By Bertrand's postulate (which is also known as the Chebyshev's theorem), there is always a prime number between n and $2n$. Thus, the padding will not enlarge the input size by more than a factor of 2. This padding can also be done via a wrapper on our original oracle. The wrapper filters out all larger indices and treat their values as 0. To find out how much we should pad until n becomes a prime number, it suffices to search for a prime number in the range $[n, 2n)$. Verifying primality takes only polynomial time on the logarithm of the input. Classically, we can do it in around $O((\lg n)^{12})$ time with a deterministic algorithm [18] or $O((\lg n)^6)$ with a randomized algorithm [19]. Quantumly, it takes around $O((\lg n)^3)$ [4] time. The density of prime is approximately $\Theta(\lg n)$. Finding one prime with `BBHT` in the specfied range takes at most polynomial of $\lg n$ time. Thus, snapping to a prime number boundary can be done within our desired $\sqrt{n}$ time limit.

As discussed, writing down a huge sample may take too long even if each sample can be done in constant amount of time. Sampling against a threshold would only make the situation worse. Indeed, `BBHT` is already optimal if we want to sample anything against a predicate, but it could not run arbitrarily fast. As a result, we must first discard the concept of a threshold and focus on simply sampling any element (even when it is less than our threshold λ) from our original set S. Moreover, we cannot afford sampling items one by one. The idea of solving this sampling problem comes directly from how we can represent our sample. For instance, we can take the first k elements and make them a sample. However, this sample is not at all random, and we cannot apply Lemma 1. If we are working on a sorted set, the maximum of this sample would correspond to the k^{th} smallest element in our set and does not prune m by much. Extending

this, we may try to pick an offset α and mark the next k elements as our sample. Generating such samples requires a single random integer generation and is thus can be done in constant time. Unfortunately, there is no guarantee how often we can get a good sample of this sort.

Taking this idea further, we can generate two integers $\alpha \in [0, n)$ and $\beta \in [1, n)$. Our sample of size k would be

$$T(\alpha, \beta) = \{x_\alpha, x_{\alpha+\beta(mod\ n)}, x_{\alpha+2\beta(mod\ n)}, \cdots, x_{\alpha+(k-1)\beta(mod\ n)}\}$$

where x_i denotes the i^{th} entry in our input set. With this method, we have $n(n-1)$ distinct samples. They are only a small fraction of the all n^k possible random samples of size k. The randomness of our samples is arguably much smaller. Fortunately, the following lemma shows that this pseudo-random sample generator is efficient on all k.

Lemma 4 *The probability that $T(\alpha, \beta)$ is good is at least $\frac{1}{2}$. By definition, a good pseudo-random sample contains some of the largest $\frac{n}{k}$ elements.*

With probability at least half, setting our threshold λ to the maximum element of some pseudo-random sample $T(\alpha, \beta)$ would reduce m_0 to no more than $\frac{n}{k}$. In other words, we can start with a sample of size $\frac{n}{(\lg \lg n)^2}$ and set the λ to the maximum of this. This takes $O(\sqrt{n} + \lg |\mathscr{U}|)$ and would reduce m to $(\lg \lg n)^2$ immediately. We may then apply any of our previous algorithms to take advantage of the reduced m.

Fortunately, we can push this bar even further. Notice that the runtime of `Progressive_Approximation` depends [2] on m. When m is reduced, we can afford a larger sample. For instance, a sample of size $\frac{n}{(\lg \lg \lg \lg n)^2}$ would still run in our desired time $O(\sqrt{n} + \lg |\mathscr{U}|)$ given that m is at most $(\lg \lg n)^2$. Formally,

[2] `Progressive_Approximation` actually depends on the number of active elements in the sample, which is upper bounded by the number of active elements in the whole input set.

we first define:

$$\lg^{(i)} n = \begin{cases} n & \text{if } i = 0 \\ \lg(\lg^{(i-1)} n) & \text{if } i > 0 \end{cases}$$

We denote $\lg^* n$ to be the minimum of i such that $\lg^{(i)} n \leq 1$. Note that $\lg^*$ is an extremely slowly growing function and is often treated as a constant in most practical setting. We further define the sequence:

$$d_i = (\lg^{(2i)} n)^2$$

This sequence d has at most $\frac{\lg^* n}{2}+1$ terms before later entries become undefined. Taking in all our previous results, our new algorithm is as follows:

```
procedure Large_Sampling(𝒪, λ0)
  i ← 0; λ ← λ0
  while d_{i+1} is defined do
    Generate α and β
    𝒪' ← wrapped oracle with respect to our sample of size n/d_{i+1}
    with budget O(√n + lg|𝒳|) run λ ← Progressive_Approximation(𝒪', λ)
    if timed out then
      i ← max(i − 1, 0)
    else
      i ← i + 1
    end if
    end while
end procedure
```

The basic idea is that we will keep getting large samples and find their maxima. The first iteration would reduce m to $(\lg^{(2)} n)^2$. Another iteration would bring m down to $(\lg^{(4)} n)^2$. After the i^{th} iteration, m would become $(\lg^{(2i)} n)^2$. Therefore, m will gradually approach 0. On the other hand, $d_i \leq 1$ in the last iteration. We would have sampled the whole input set and find the global maximum. Therefore, the correctness and success probability are inherited. We aim at running inner iterations within $O(\sqrt{n}+\lg|\mathscr{X}|)$ time. This could fail for a couple of reasons. For instance, we may have hit a bad sample

and could not prune m fast enough. We may also be unlucky in one of the iterations, in that our `Progressive_Approximation` procedure returns a wrong estimate. However, the probability of getting a bad run is still bounded by a constant probability. As a result, each specific iteration would not restart more than a constant number of times expectedly. This is similar to our discussion in `Progressive_Approximation`. It suffices to analyse the perfect scenario. The complexity with error will follow with a larger constant.

The perfect scenario is now easy to calculate. We will loop until d_i becomes undefined, which takes $O(\lg^* n)$ iterations. Moreover, each iteration takes $O(\sqrt{n} + \lg|\mathscr{U}|)$ time. The whole algorithm takes $O((\sqrt{n} + \lg|\mathscr{U}|)\lg^* n)$ time.

Theorem 10 `Large_Sampling` *solves the unary maximum finding problem in* $O((\sqrt{n} + \lg|\mathscr{U}|)\lg^* n)$ *expected time with constant success probability.*

Note that this runtime is not sensitive to m_0 because we have no good way to estimate it. Should we be able to estimate m_0, we would adjust our sample size to take full advantage of a reduced initial m.

4.7 Variants

`Large_Sampling` shows us how we can reduce m to 0 in $O(\lg^* n)$ time. As in `Budgeted_Sample_And_Improve`, the next natural question is whether we can isolate the m-reduction term from either then `BBHT` search term ($\sqrt{n}$) or the binary search term ($\lg|\mathscr{U}|$). Fortunately, the answer is affirmative. We will briefly go over the ideas and leave the details for those who are interested.

Using the idea of `Geometric_Budgeting`, we can replace the sequence d_i with $d'_i = 2^{2i}d_i$. The time required to find the maximum of the sample corresponding to d_i would become expectedly $O(\frac{\sqrt{n}}{2^i} + \lg|\mathscr{U}|)$. The total work done before d_i becomes undefined is thus $O(\sqrt{n} + \lg|\mathscr{U}|\lg^* n)$. The last sample is of size

$\frac{n}{2^{2\lg^* n}}$. The number of remaining active elements would be expectedly $2^{2\lg^* n}$. We can follow this by a simple `Sample_And_Improve` which runs in expected time $O(\sqrt{n} + \lg|\mathscr{U}|\lg^* n)$. The total time required for this strategy is thus $O(\sqrt{n} + \lg|\mathscr{U}|\lg^* n)$.

Similarly, we can apply `Large_Sampling` in `Progressive_Approximation`. We overlay the loops from both algorithm. We continue doing our budgeted sampling in `Progressive_Approximation` while maintaining our d_i separately. After each budgeted sampling, we perform a large sampling and find the maximum according to the current d_i. If this step runs in time, we advance our pointer on d_i and revert our d_i should we time out. This extra pruning step is successful if we get a good large sample and find the maximum in time. Moreover, this step suceeds with constant probability. Clearly, this pruning step is more effective than the original budgeted sampling. We could picture the execution of the algorithm as the original `Progressive_Approximation` algorithm. However, the pruning from `Large_Sampling` may kick in with some constant probability, reseting m to the corresponding d_i. It takes $O(\lg^* n)$ extra pruning before $m = 1$. After that, the whole procedure would boil down to a simple binary search. The correctness of this approach is guaranteed with the underlying `Progressive_Approximation` algorithm, and the extra speedup is provided by the occasionally successful `Large_Sampling` algorithm. The total runtime of this approach is thus $O(\sqrt{n}\lg^* n + \lg|\mathscr{U}|)$.

The most intriguing result is probably that we can cascade the idea of `Large_Sampling`. We have already developed a way to find the maximum in $O(\sqrt{n}\lg^* n + \lg|\mathscr{U}|)$ time. That means that we can start with a sample of size $\frac{n}{(\lg^* n)^2}$ to run in expected time $O(\sqrt{n} + \lg|\mathscr{U}|)$. This set the number of active elements to $(\lg^* n)^2$ instantly. We can repeat and generate a sample of size $\frac{n}{(\lg^{(2)}(\lg^* n))^2}$ and run `Progressive_Approximation` on it. This should also

run in expected time $O(\sqrt{n} + \lg|\mathscr{U}|)$. This process can be repeated until we reduce m to 1 in $O(\lg^*(\lg^* n))$ steps. As a result, this algorithm would run in $O((\sqrt{n} + \lg|\mathscr{U}|)\lg^*(\lg^* n))$. As in our previous discussion, one could probably work on tricks to isolate the m-reduction term from either the `BBHT` term or the binary search term.

Although it seems that these ideas can be cascaded further, we choose to stop the mechanical process because:

- The constant attached to the runtime is probably too big for it to become practical. On the other hand, $\lg^* n$ is already an extremely slowly growing function that many regard as a constant in any practical setting.
- Each `Large_Sampling` requires a wrapper for the oracle so that the internal algorithm can work on the restricted sample. If we cascade the idea of `Large_Sampling` r times, we must at least wrap our oracle r times. As a result, a single call to the oracle would no longer be a constant operation even when each wrapping is just a simple arithmetic transformation.

Theorectically, the ability to cascade `Large_Sampling` hints that our lower bound could be optimal. Although we have not succeeded in getting the desired bound of $O(\sqrt{n} + \lg|\mathscr{U}|)$, it appears that we can get close to the optimum up to any fixed degree with a fixed number of cascading. Moreover, it appears that our techniques can potentially speed up any unary maximum finding algorithm which focuses on m-reduction. As a result, there may not be any tight upper bound for any algorithm working with m-reduction, but we have not formally studied this claim.

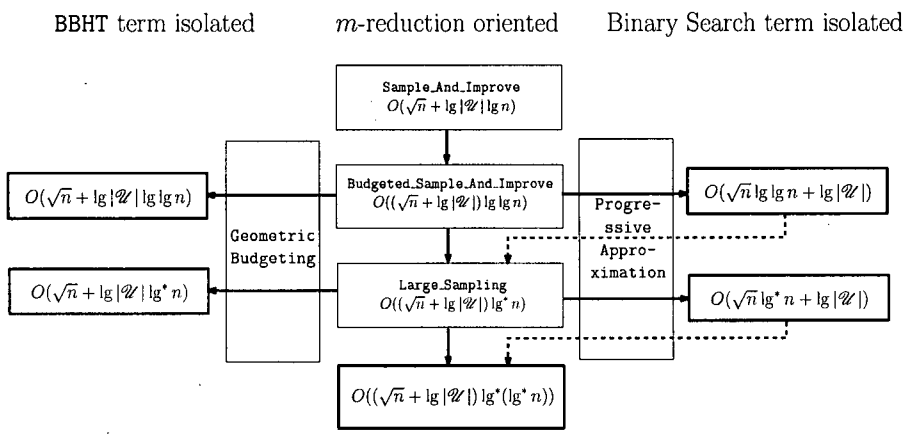

Figure 4.1: Summary on different Quantum Unary Maximum Finding results

4.8 Other applications

4.8.1 Maximum Root Approximation

Maximum root approximation is one of the motivating problems of our research. Our `Progressive_Approximation` procedure is also modelled after an approximation problem. As illustrated in the work by Gao et al. [3], the unary maximum finding problem can still be applied to real numbers outside of the range $[0, 1)$. We only have to find an appropriate universe with the right bucket, and the results follow. For simplicity, we consider the case when all real numbers are non-negative. We cannot directly apply any unary maximum finding algorithm because the universe is infinite. Fortunately, we can probe a good range. The algorithm presented by Gao et al. is as follows:

```
procedure Upper_Bound(n, 𝒪, ϵ)
  u ← ϵ
  for i ∈ {0, ..., n − 1} do
    while x_i > u do
      u ← 2u
    end while
  end for
  return u
```

This procedure simply loops through all elements and find the right upper bound that is applicable to each element. We start with the minimum error margin ϵ and work our way up in an exponential fashion. Note that such an upper bound always exist and is no more than twice the value of the maximum entry. The runtime of this procedure is thus $O(n+\lg \frac{x_{max}}{\epsilon})$. We can then apply the classical unary maximum finding algorithm to approximate the value of the largest real number in time $O(n+\lg \frac{x_{max}}{\epsilon})$.

We can speed up the process considerably in the quantum setting. Suppose we have already gone through several elements and found a non-trivial upper bound u, we would want to skip all later entries whose value are less than u. In other words, given an upper bound u, we are only interested in elements larger than it. As discussed, `BBHT` is an ideal tool for this purpose. Combining these ideas gives the following:

```
procedure Upper_Bound(n, 𝒪, ϵ)
  u ← ϵ
  while True do
    t ← (x_i > u)
    if t = NotFound then
      break
    end if
    while t > u do
      u ← 2u
    end while
  end for
  return u
```

This is yet another bounded error algorithm where the success probability is shared with that of `BBHT`. We basically sample larger entries and update our upperbound to cover it. Similar to `Sample_And_Improve`, with probability no less than half, we would have selected the medium of all entries greater than u. Updating u to cover this entry would also take care of half of the remaining meaningful entries. Similar to the discussion of `Sample_And_Improve`, the expected total selection cost is bounded by $O(\sqrt{n})$, while the total work done

in increasing u is bounded by $O(\lg \frac{x_{max}}{\epsilon})$. As a result, we can find a good upper bound in expectedly $O(\sqrt{n} + \lg \frac{x_{max}}{\epsilon})$. We can then apply any of our quantum unary maximum finding algorithm to retrieve the maximum.

Suppose we apply the `Large_Sampling` algorithm, we will approximate the maximum of n real numbers given by a comparison oracle in expected $O(\sqrt{n} \lg^* n + \lg \frac{x_{max}}{\epsilon})$ time with constant success probability. Note that the probability of error is applified because we need at least two failed `BBHT`, one for the `Upper_Bound` procedure and one for our unary maximum finding algorithm. As discussed before, we can detect failures because `BBHT` is one-sided. We can simply restart some steps when there were errors, and this translates to no more than a constant multiplier in front of our overall runtime (similar to `Progressive_Approximation` and `Large_Sampling`).

The same idea can also be applied to real approximation with relative errors. Interested readers are referred to the discussion in the work by Gao et al. [3].

4.8.2 Quantum K-select

We would also like to point out that some of our ideas are also useful in a completely different scenario. The problem of finding the k^{th} largest entry in a given set was discussed by Nayak and Wu [20]. They have also derived lower bounds for their problem in the binary model. The time required to find the k^{th} largest element is proven to be at least $\Omega(\sqrt{kn})$, but they only gave an $O(\sqrt{kn} \lg(kn) \lg \lg(kn))$ time algorithm. We now present an optimal approach to the quantum k-select problem:

```
while True do
  T ← a large pseudo-random sample of size n/(3k).
  μ ← max(T) // using the optimal algorithm in the binary comparison model [7]
  with budget O(√(kn)) run S' ← FindAll(x_i > μ)
  if we did not time out and |S'| ∈ [k, 3k] then
    break
  end if
end while
return classical-k-select(k, S')
```

The correctness of the algorithm is guaranteed by the final call through the classical optimal algorithm. The time for generating the large sample and finding its maximum is $O(\sqrt{n})$. The budget constraint ensures that each iteration takes no more than $O(\sqrt{kn})$ time. We break if we sample everything above the sample maximum and the size of the reduced set is close to k. Note if there were $3k$ elements above our threshold, selecting all of them with `FindAll` takes expectedly $O(\sqrt{kn})$ time. The following lemma tells us that this occurs with probability at least $\frac{1}{6}$.

Lemma 5 *Suppose we generate a pseudo-random sample of size $\frac{k}{3}$ and find its maximum. We then count the number of elements above this maximum. With probability at least $\frac{1}{6}$, this count falls in the range $[\frac{n}{k}, 3\frac{n}{k}]$.*

The proof is given in Appendix C.

As a result, we will expectedly go through a constant number of iterations before we invoke the classical-k-select on a reduced set of size $O(k)$. The total run time is thus $O(1 + \sqrt{n} + \sqrt{kn}) + O(k) = O(\sqrt{kn})$.

This shows that the technique we have developed for the unary maximum finding problem can also be applied to other areas. We hope that our ideas could bring insight to other researchers as well.

Chapter 5

Conclusions

We have reviewed the problem of extrema finding with only unary predicates. The problem is optimally solved for the classical setting [3], but has not been previously addressed in the quantum setting. Although not as immediate as the extrema finding with a binary oracle, quantum computers do provide a speedup over classical ones.

We have proved that the lower bound of this problem is $\Omega(\sqrt{n} + \lg|\mathscr{U}|)$ and showed a sequence of upperbounds ranging from $O(\sqrt{n} + \lg|\mathscr{U}|\lg n)$ to $O(\sqrt{n}\lg^* n + \lg|\mathscr{U}|)$. There are quantum unary maximum finding algorithms that perform asymptotically faster than any classical counterpart unless $\lg|\mathscr{U}|$ dominates, in which case some of our quantum algorithms are as fast as the classical optimal one. Furthermore, we have generalized our results in three fundamental techniques that can be reapplied and cascaded to improve existing bounds.

5.1 Future Directions

We believe that our lower bound is in effect tight but our approach may not yield any optimal algorithm. All our algorithms are off by a factor induced by the speed we prune our active set. One may need an entire new approach to get a tight upper bound.

In addition, we are interested in how we may apply our tools to solve other

problems. In particular, the large pseudo-random sample generation is efficient and fits well in many scenarios. We hope that some of our techniques could provide insights to other researchers.

Bibliography

[1] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC)*, pages 212–219, 1996.

[2] Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. Tight bounds on quantum searching. *Fortschritte Der Physik*, 46:493–505, 1998.

[3] Feng Gao, Leonidas J. Guibas, David G. Kirkpatrick, William T. Laaser, and James Saxe. Finding extrema with unary predicates. *Algorithmica*, 9:591–600, 1993.

[4] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, 1994.

[5] H. Buhrman, R. Cleve, R. de Wolf, and Ch. Zalka. Bounds for small-error and zero-error quantum algorithms. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 358–368, 1999.

[6] Bartholomew Furrow. A panoply of quantum algorithms. Master's thesis, The University of British Columbia, September 2006.

[7] Christoph Dürr and Peter Høyer. A quantum algorithm for finding the minimum. quant-ph/9607014, 1996.

[8] R. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6/7):467–488, 1982.

[9] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London Ser. A*, A400:97–117, 1985.

[10] E. Bernstein and U. Vazirani. Quantum complexity theory. *SIAM Journal of Computing*, 26:5:1411–1473, 1997.

[11] W. H. Zurek W. K. Wootters. A single quantum cannot be cloned. *Nature*, 299:802–803, 1982.

[12] P. W. Shor. Scheme for reducing decoherence in quantum computer memory. *Physical Review*, A 52:4:2493–2496, 1995.

[13] C.H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani. Strengths and weaknesses of quantum computing. *SIAM Journal on Computing*, 26(5):1510–1523, 1997.

[14] Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum counting. *Lecture Notes in Computer Science*, 1443:820, 1998.

[15] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[16] Robert Beals, Harry Buhrman, Richard Cleve, Michele Mosca, and Ronald de Wolf. Quantum lower bounds by polynomials. *Journal of the ACM*, 48(4):778–797, July 2001.

[17] Peter Høyer, Jan Neerbek, and Yaoyun Shi. Quantum complexities of ordered searching, sorting, and element distinctness. *Algorithmica*, 34(4):429–448, 2002.

[18] Neeraj Kayal Manindra Agrawal and Nitin Saxena. Primes is in p. *Annals of Mathematics*, 160(2):781–793, 2004.

[19] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[20] Ashwin Nayak and Felix Wu. The quantum query complexity of approximating the median and related statistics. *Conference Proceeding Annual Symposium of Theoretical Computing*, pages 384–393, 1999.

Appendix A

Expected Pruning with Large Samples

Recall that our active sets are defined by their corresponding threshold λ and contain all elements greater than λ. The only way we can reduce our active set is by raising our threshold, while keeping the feasibility of the threshold. One simple way to achieve this is to sample an active element from the active set and update our threshold to it. By definition, we must have increased our threshold and reduced the active set. We have aruged that choosing a random element from the active set as the new threshold value expectedly halves the size of the active set.

We have also considered choosing a sample with more than 1 element. To maximize the pruning, we should always update our threshold to the maximum of the sample. Given a sample of size k, we would like to know how much this process can prune our active set. Equivalently, we want to know how big the maximum of the sample is. If the sample maximum is the $(i+1)^{th}$ largest element in our original set, updating the threshold to the maximum will prune the size of the active set to i. We are particularly interested when k is large and non-constant. For instance, k is about $\sqrt{m}$ in our `Budgeted_Sampling` algorithm, where m denotes the current size of the active set.

We assume that the original size of the active set is m, and we are given

a sample consisting of k active elements chosen randomly with replacement. This sample can be generated by repeated BBHT as described in previous sections. We define X to be the random variable denoting the size of the active set after the pruning. Note that unless $k = 0$, $X < m$ as any active element would have increased the threshold and remove itself from the active set. As discussed, $X > i$ means that our sample does not contain any of the largest $i+1$ elements. The probability of this happening is thus $(\frac{m-i-1}{m})^k$. Similarly, $Pr[X = i] = P[X > i-1] - Pr[X > i] = \frac{1}{m^k}[(m-i)^k - (m-i-1)^k]$. As such, we have

$$\begin{aligned} E[X] &= \sum_{i=0}^{m-1} iPr[X = i] \\ &= \frac{1}{m^k}[\sum_{i=0}^{m-1} i((m-i)^k - (m-i-1)^k)] \\ &= \frac{1}{m^k}[\sum_{i=0}^{m-1} i(m-i)^k - \sum_{i=1}^{m}(i-1)(m-i)^k] \\ &= \frac{1}{m^k}[\sum_{i=1}^{m-1}(m-i)^k] = \frac{1}{m^k}\sum_{i=1}^{m-1} i^k \end{aligned}$$

Furthermore, we know that

$$\sum_{i=1}^{m-1} i^k \leq \int_1^m x^k dx = \frac{1}{k+1}[x^{k+1}]_1^m \leq \frac{1}{k+1}m^{k+1}$$

Combining, we have

$$E[X] \leq \frac{1}{m^k}\frac{1}{k+1}m^{k+1} = \frac{m}{k+1}$$

We have thus proved Lemma 1 (page 48). When $k = 1$, this lemma says that we will expectedly select the median and half our set. When $k = \sqrt{m}$, this theorem says that we can expectedly reduce m to $\sqrt{m}$.

Appendix B

Expected Pruning with Budgeted_Sampling or Failed BBHT

`Budgeted_Sampling` is used extensively to generate large samples and prune our active set. In Appendix A, we show that with a sample of size k, we can reduce m to expectedly $\frac{m}{k+1}$. This gives us an idea how we should tune k. However, we have also discussed that the optimal choices for k depends on m which is usually unknown. In `Budgeted_Sampling`, we tackle this by setting a fixed budget B and repeatedly running `BBHT` until we run out of our budget. Intuitively, each selection is about $\Theta(\sqrt{\frac{n}{m}})$ and the size of the sample k would be about $\Theta(B\frac{m}{n})$. However, the size of the sample has become a random variable (even when m is known). We have already proved that the pruning factor of a sample of size k is expectedly $\frac{1}{k+1}$. Define K to be the random variable of the returned sample size, we want to bound the expectation of $\frac{1}{K+1}$.

Before we begin, we would like to note that $E[\frac{1}{K+1}] \geq \frac{1}{E[K]+1}$ but there is no nontrivial upperbound for the first term. Furthermore, finding $E[K]$ is also a very hard problem and would require a lot of mathematical work. We now present a very loose upper bound for $E[\frac{1}{K+1}]$ which works with most of our algorithms. This should take less derivation than directly dealing with all

different kinds of distributions.

The idea of the proof is to find another variable K', $K' \leq K$, which follows some well studied distribution. The inequality $K' \leq K$ can be interpreted in several ways. Suppose we are given the log of the previous `Budgeted_Sampling`, denoted as an outcome o, $K(o)$ counts the number of successful `BBHT` calls. With $K' \leq K$, we have $K'(o) \leq K(o)$ for any given outcome o. Equivalently, the probability of getting lower values from K' is higher than K. Therefore, the cumulative probability function of K' is greater or equal to that of K. As a result, we have both $E[K'] \leq E[K]$ and $E[\frac{1}{K'+1}] \geq E[\frac{1}{K+1}]$. Thus, to properly bound $E[\frac{1}{K+1}]$, it suffices to find an upper bound for $E[\frac{1}{K'+1}]$. This is easier than dealing with K directly because we take control over the distribution of K'.

First of all, we should investigate how the sampling is done. `BBHT` works by doing serveral Grover iterations and sampling in a controlled manner. With a budget of B, the execution is similar to running B Grover iteration, each followed by an optional measurement and a reset step. Some of the measurements would succeed and return a valid entry. Our random variable K is simply a count of successful measurements over B similar iterations.

Suppose the expected runtime of `BBHT` is $c\sqrt{\frac{n}{m}}$ for some appropriate constant c, we partition the B steps into groups of size $2c\sqrt{\frac{n}{m}}$ each. We simply ignore the last few iterations if $2c\sqrt{\frac{n}{m}}$ does not divide B. As a result, we have $\lfloor \frac{B}{2c}\sqrt{\frac{m}{n}} \rfloor$ groups. We call a group successful if there is any good measurement within the period bounded by that group. We know define K^* to be the random variable counting the number of successful groups. Clearly, for any execution sequence, $K^* \leq K$. However, K^* is still fairly complicated to analyse. Fortunately, we can think of it as several Bernoulli experiments and use a binomial distribution to approximate it. The major problem of this is that adjacent groups are not

independent experiments and the success probability of each experiment is not the same. However, the success probability is still well bounded by Markov's inequality. The probability of an instance of `BBHT` spanning more than a single group (of size $2c\sqrt{\frac{n}{m}}$) is no more than $\frac{1}{2}$. We now define K' to be a random variable following a binomial distribution with $\lfloor \frac{B}{2c}\sqrt{\frac{m}{n}} \rfloor$ experiments and success probability $\frac{1}{2}$. Since we deliberately decrease the success probabitlity and decouple groups, we have $K' \leq K^* \leq K$.

Next, we are going to find an upper bound for $E[\frac{1}{K'+1}]$. To simplify the notation, we denote N to be $\lfloor \frac{B}{2c}\sqrt{\frac{m}{n}} \rfloor$. With this setting, we have:

$$Pr[K' = i] = \left(\begin{array}{c} N \\ i \end{array} \right) \frac{1}{2^N}$$

The expectation of $\frac{1}{K'+1}$ is thus

$$\begin{aligned} E[\frac{1}{K'+1}] &= \sum_{i=0}^{N} (Pr[K' = i]\frac{1}{i+1}) \\ &= \frac{1}{2^N} \sum_{i=0}^{N} \frac{N!}{i!(N-i)!} \frac{1}{i+1} \\ &= \frac{1}{2^N} \sum_{i=0}^{N} \frac{N!}{(i+1)![(N+1)-(i+1)]!} \\ &= \frac{1}{2^N} \frac{1}{N+1} \sum_{i=1}^{N+1} \frac{(N+1)!}{(i+1)![N-(i+1)]!} \\ &= \frac{1}{N+1} \frac{1}{2^N} (2^{N+1} - 1) \leq \frac{2}{N+1} \end{aligned}$$

By transitivity, we conclude that $E[\frac{1}{K+1}] \leq \frac{2}{N+1} = O(B\sqrt{\frac{m}{n}})$. This completes the proof of Lemma 2 (page 50).

In most of our context, we normalize all runtimes so that the constant is virtually 1. When B is $\sqrt{n}$, our expected pruning is approximately $\frac{1}{\sqrt{m}}$. When

B is $\sqrt{\frac{n}{2^i}}$, the expected pruning is $\frac{\sqrt{2^i}}{m}$.

It is also interesting to know that the same idea can be applied to understand how much information a failed BBHT provides. Suppose we run a BBHT with budget B and it failed to return anything, we would like to bound the expected number of good indices that remain. In the standard BBHT, running with a budget of $\sqrt{n}$ is a certificate if there is 1 or more elements in our set. Suppose we have a budget of $\sqrt{\frac{n}{t}}$ to run our BBHT. We expect that it is effective in returning some good index if m, the number of good indices, is more than t. Unfortunately, this is not mathematically rigorous. We shall prove that the expectation of m given that a BBHT with budget $\sqrt{\frac{n}{t}}$ failed to return is bounded by $O(t^2)$. This bound is probably not tight but is sufficiently powerful in many of our scenarios.

To simplify the notation, we assume that B is normalized and $T = \frac{2\sqrt{n}}{B}$ is an integer. The case where $B > 2\sqrt{n}$ boils down to the error rate of BBHT, and would not be reiterated. Moreover we let M be the random variable of the number of good indices given that a BBHT of budget $B = \frac{2\sqrt{n}}{T}$ failed. For any given m, we can partition and go through the same transformation to get an approximated binomial distribution for the number of successful BBHT. Formally, we construct a random variable K' following a binomial distribution with $N = \frac{B\sqrt{m}}{2\sqrt{n}} = \frac{\sqrt{m}}{T}$ and success probability one half. Since K' is an underestimate, the probability of getting no return from BBHT with K' is larger than that with the original K. Having a failed BBHT is equivalent to $K = 0$, which gives us:

$$Pr[M = m|\text{BBHT with budget } \frac{2\sqrt{n}}{T} \text{ failed}] \leq \frac{1}{2^{\lfloor \frac{\sqrt{m}}{T} \rfloor}}$$

As a result, we have:

$$\begin{aligned}
E[M] &= \sum_i iPr[M=i] \leq \sum_i i\frac{1}{2^{\lfloor\frac{\sqrt{m}}{T}\rfloor}} \\
&= \frac{1+2+\ldots+(T^2-1)}{2^0} + \frac{T^2+(T^2+1)+\ldots+(4T^2-1)}{2^1} + \ldots \\
&\leq \frac{T^2(T^2-1)}{2^1} + \frac{4T^2(4T^2-1)}{2^2} + \ldots \\
&\leq \sum_i \frac{i^2T^4}{2^i} \leq T^4 \sum_i \frac{i^2}{2^i}
\end{aligned}$$

As the fractions $\frac{i^2}{2^i}$ are decreasing geometrically (by at least a factor of $\frac{2}{3}$ in later i), the sum converges and is well bounded by a constant around 150. Together with our defintion of T, we have proved Lemma 3 (page 54).

Specifically, suppose we have run a `BBHT` of budget $\sqrt{\frac{n}{2^i}}$ and failed to sample anything, the expected number of good elements left would be bounded by $O(2^{2i})$.

Appendix C

Properties of Large Pseudo-random Samples

We have shown that a pure random sample of size k has an expected pruning factor of $\frac{1}{k+1}$. Unforturnately, we cannot reuse the theorem when we are working with large pseudo-random samples. For one thing, the elements within our samples are not independent and they are not random. Suppose we start with a set of size n (n is assumed to be a prime) and we want to generate a sample of size k. We generate our pseudo random sample by randomly picking an offset α and a gap β. Denoting x_i as the i^{th} element in our set, our sample T consists of the elements $x_{\alpha+j\beta(mod\ n)}$, where $j \in [0, k)$. The size of T is clearly k, and it contains no duplicates. There are also a total of $n(n-1)$ such pseudo-random samples.

Ideally, we want our pseudo-random samples to be as efficient as purely random samples. For instance, a pseudo-random sample of size k is *good* if it contains at least one of the largest $\frac{n}{k}$ elements. Setting our threshold to the maximum of a good pseudo-random sample will prune m directly to $\frac{n}{k}$. Note that since we do not sample against a threshold, m_0 can only be trivially n. This correspond to a pruning factor of around $\frac{1}{k}$. In addition, we want our sample to be efficient, that the success probability of getting a good sample is at least a constant and the time required to generate one is short.

We will now look at some of the properties of our random sample generator. Given a sample $T(\alpha, \beta)$, we denote t_i to be the i^{th} element in our sample, namely $x_{\alpha+i\beta(mod n)}$.

Lemma 6 *For any given position $p \in [0, n)$ and any index $i \in [0, k)$, $x_p = t_i$ with probability $\frac{1}{n}$.*

We observe that given any gap β, we can set the offset α to be $p - i\beta(mod\ n)$ in order to make $t_i = x_p$. We have $n-1$ possible values of β, and the corresponding choice of α is unique. As a result, we have exactly $n-1$ satisfying samples over the entire space of $n(n-1)$ samples. The probability of this event happening is thus $\frac{1}{n}$.

Lemma 7 *For any two given positions $p, q \in [0, n), p \neq q$ and any two indices $i, j \in [0, k), i \neq j$, the probability that $x_p = t_i$ and $x_q = t_j$ is $\frac{1}{n(n-1)}$.*

The condition can be translated to the following system:

$$\begin{pmatrix} 1 & i \\ 1 & j \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} p \\ q \end{pmatrix} (mod\ n)$$

With $i \neq j$, the coefficient matrix is of full rank and there is only one solution. As a result, this occurs only with probability $\frac{1}{n(n-1)}$ should we choose our sample randomly from our space.

The combination of these two lemmas gives rise to Lemma 4 (page 62). We first mark the largest $\frac{n}{k}$ largest entries, and call their respective positions $p_0, p_1, \ldots, p_{\frac{n}{k}-1}$. A sample T is good if

$$E = \bigvee_{i \in [0,k), j \in [0, \frac{n}{k})} t_i = x_{p_j}$$

is true. For some random T, the probability of E happening is:

$$\begin{aligned} Pr[E] &\geq \sum_{i\in[0,k),j\in[0,\frac{n}{k})} P[t_i = p_j] - \sum_{i,i'\in[0,k)j,j'\in[0,\frac{n}{k})j<j'} Pr[t_i = p_j \wedge t'_i = p'_j] \\ &= k\frac{n}{k}\frac{1}{n} - k^2\frac{1}{2}\frac{n}{k}(\frac{n}{k}-1)\frac{1}{n(n-1)} \\ &= 1 - \frac{1}{2}\frac{n-k}{n-1} \geq \frac{1}{2} \end{aligned}$$

The probability of getting a good pseudo-random sample from our generator is no less than half. As a result, it takes expectedly no more than a constant number of repetition before we actually hit a good pseudo-random sample.

We can also extend this theorem and prove Lemma 5 (page 70). What we want to show is that with good probability, our sample is also representative. It would neither contain only large elements nor completely miss them. There is a significant probability that the largest element is within the largest $\frac{n}{k}^{th}$ to $(3\frac{n}{k})^{th}$ elements.

We have already shown that with probability at least $\frac{1}{2}$, the maximum entry in our sample is among the largest $3\frac{n}{k}$ elements. It suffices to prove that the probablity of the maximum being the largest $\frac{n}{k}$ elements is low. Again, we mark the positions of the largest $\frac{n}{k}$ elements and see how often we hit them with our sample. For a random sample of size $3k$, the probability of hiting one of the positions is $\frac{k}{3n}$. The probablity of hitting any one of the positions is at most $\frac{k}{3n}\frac{n}{k} = \frac{1}{3}$. The probability of the maximum entry behaving well is thus no less than $\frac{1}{2} - \frac{1}{3} = \frac{1}{6}$.