Computing Ramsey Numbers

School of Computer Science & Applied Mathematics
University of the Witwatersrand

Brendan Griffiths
2426285

Supervised by Dr Warren Carlson

November 11, 2024

A report submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in partial fulfilment of the requirements for the degree of Bachelor of Science with Honours

Abstract

Ramsey Numbers and Ramsey Theory are very difficult fields of research. Even after nearly a century of research, only a few Ramsey Numbers are known. However, the nature of finding a Ramsey Number is an algorithmic process which can potentially be automated on a computer. This report investigates the capability and difficulty of computing a Ramsey Number on both classical computers and on the potentially faster quantum computers. I show that Ramsey Numbers are in the classical complexity class $\mathbf{\Pi}_2^P$, and the quantum class of *QMA*. Additionally, I design, implement and investigate three classical algorithms and two quantum algorithms for solving Ramsey Numbers and show that these algorithms are only able to compute the next Ramsey Number $R(5,5)$ within the next 10^{135} years.

Declaration

I, Brendan Griffiths, hereby declare the contents of this research report to be my own work. This proposal is submitted for the degree of Bachelor of Science with Honours in Computer Science at the University of the Witwatersrand. This work has not been submitted to any other university, or for any other degree.

Acknowledgements

I would like to thank Dr Maartens for hosting a Graph Theory workshop that helped me learn the foundations of Ramsey Numbers, as well as Dr Nape for organising a workshop with IBM's collaboration to teach the basics of Quantum Computing. Additionally, Prof. Shktov's assistance in the complexity theory proofs was invaluable in putting together the theoretical underpinnings of this report.

Importantly, I would also like to thank my supervisor, Dr Warren Carlson, for suggesting this topic and then assisting me in all aspects of putting it together.

Contents

Glossary

TM: A Turing Machine - the mathematical object representing computations and algorithms.

P (Polynomial Time): The complexity class of problems solvable on a classical Turing Machine in a polynomial amount of time.

NP (Non-Deterministic Polynomial): The complexity class of problems that verify a solution on a classical Turing Machine in a polynomial amount of time.

PSPACE (Polynomial Space): The complexity class of problems that verify a solution on a classical Turing Machine in a polynomial amount of space

BPP (Bounded Probabilistic Polynomial): The complexity class of problems that can be solved correctly with probability $\frac{2}{3}$ on a classical probabilistic Turing Machine in a polynomial amount of time.

QMA (Quantum-Merlin-Arthur): The complexity class of problems that verify a solution on a quantum Turing Machine in a polynomial time. *QMA* is the quantum analogue of *NP*.

BQP (Bounded Quantum Polynomial Time): The complexity class of problems that can be solved correctly with probability $\frac{2}{3}$ on a quantum Turing Machine in polynomial time. *BQP* is the quantum analogue of *BPP*

RAMSEY: The decision problem for computing if $n = R(F, H)$

LOWER-RAMSEY: The decision problem for computing if $n < R(F, H)$

UPPER-RAMSEY The decision problem for computing if $n \geq R(F, H)$

CIS-RAMSEY (*CLIQUE-INDEPENDENT-SET-RAMSEY*): The decision problem for computing if $n = R(k, l)$ where k and l are the size of the clique and independent-set respectively.

LCR (*LOWER-CIS-RAMSEY*):The decision problem for computing if $n < R(k, l)$ where k and l are the size of the clique and independent-set respectively.

UCR (*UPPER-CIS-RAMSEY*): The decision problem for computing if $n \geq R(k, l)$ where k and l are the size of the clique and independent-set respectively.

List of Figures

List of Tables

Chapter 1

Introduction

One of the most interesting aspects of the 21st century is the scale of everything. The internet processes trillions of bytes of data every month, thousands of ships sail across the oceans every day, billions of people interact with each other every hour. Such large numbers are very difficult for a human to reason about individually, but this is often made worse when certain events only occur at these incredibly large scales. These emergent phenomena and behaviour are incredibly difficult to spot when looking at a local scale, and even harder to work with.

Graph Theory has become one of the most prolific tools to understand relationships between objects. It is a natural candidate for trying to understand how the growth of networks, interactions and other complex system behaviour.

In a 1928 paper, Frank P. Ramsey proved a lemma about a party problem that would form the foundation of a field of combinatorics, Ramsey Theory [Ramsey 1930]. In this paper, Ramsey showed that at least six people must attend a party before it can be guaranteed that three of them had never met, or all knew each other. This lemma, later called Ramsey's Theorem, is the first instance of modelling how properties and ordering can occur in systems as they grow in size. A proof for the party problem can be found in Appendix A

Ramsey's Theorem demonstrates that as the number of people in attendance at a party grows, it is possible to build out information about the attendees without even knowing the exact relationships. This is incredibly useful as it means that the change of a system can be used to study the system without directly having to interact with it. It also established a framework to analyse the change in the system: What is the point at which a certain substructure of relationships can be guaranteed to exist?

This point became known as a Ramsey Number, and can be generalized to any kind of relationships. How many people must be at a party before four of them all know each other and two don't, before seven and three, two and five etc? The next step, away from people and parties, is can it be guaranteed that there is always a path between logistical hubs, or network switches [Roberts 1984]? It can still go further, can it be guaranteed that in any large structure, that can be modelled as a graph, has any kind of substructures?

Ramsey Numbers are a representative tool in understanding how the world works at scale, but ironically enough, in the century since Ramsey's paper, researchers have still only found a few of them [Radziszowski 2021]. Ramsey Numbers are typically studied by mathematicians, but a question that should be asked, is could a computer tackle the task of what mathematicians have struggled with? They have done so before, and quite successfully, with theorems like the 4-color map problem [Gonthier and others 2008].

Ramsey Numbers are a static value, while they change depending on what particular substructure is being studied, this number will remain the same once its known. The difficulty comes from first finding this value rather than its usage. The computation is then not strictly limited by the amount of time it takes to find the value, as long as it can be done. However, keeping the computation to a timescale of around 50 years is reasonable, as this would allow a researcher who started the computation to learn the answer and then prove it with a mathematical technique.

At a cursory glance, this seems unlikely. To find a Ramsey Number, a computer would need to check all the possible graphs of a certain number of vertices, and this number grows very quickly. At four vertices, there are sixty-four graphs, five vertices has a thousand-and-twenty-four. At ten vertices there are over a 10^{12}, different graphs. Yet, computers are able to exploit clever tricks and algorithms for even seemingly insurmountable problems can be solved by them. Even if a classical computer cannot solve Ramsey Numbers, a quantum computer might be able to.

In the last decade, quantum computers have seen a recent boom. Companies like IBM and D-Wave are leading the way in advancements in practical quantum usage, dubbing their recent success as an era of 'Quantum Utility' [Willsch *et al.* 2022], [qis b]. Quantum computers have been able to perform a computation quickly, that an equivalent classical computer would need several days to compute [Zhu *et al.* 2022].

There are two potential approaches to computing a new Ramsey Number, and in this report I will investigate both. The aim is to identify if Ramsey Numbers can be computed in a reasonable amount of time, around 50 years to a century at most, on classical and quantum computers, both theoretically and empirically. Lastly, I will compare the classical and quantum algorithms and identify the practicality of using a quantum computer in research.

This report is organized as follows. The graph theory foundation of Ramsey Numbers (Chapter 2), their classical computation and complexity (Chapter 3), the quantum computation and complexity (Chapter 4), and lastly a comparative analysis of the quantum and classical performance on modern hardware (Chapter 5).

Chapter 2

Ramsey Theory

Ramsey Theory is a rich field of combinatorics and graph theory. The field is interested in identifying the appearance and disappearance of properties in a structure as it grows.

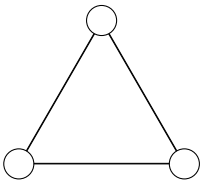
Ramsey Theory is useful in other mathematical fields. It has been used in computational geometry to identify the minimum number of points that can always form a polygon without interior points, or in evaluating decision trees as boolean functions [Rosta 2004].

Ramsey Theory is useful to understand how certain properties can be guaranteed in large graph structures such as computer networks and power grids. For example, if a network engineer wants to build a packet switching network, the minimum number of nodes in the network to guarantee that a failure at any facility will not affect the connectivity of another facility is 7 [Roberts 1984]. It is therefore useful to establish a few results and definitions in Ramsey Theory that can be utilized in this report for algorithm analysis and correctness testing.

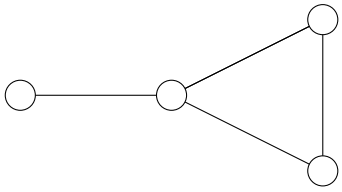
2.1 Graph Theory

Graphs are an abstract representation of many different systems that may be incredibly difficult to explore with traditional mechanisms like Calculus. The simplest example of a graph is modelling the relationships between different people. A person is represented by a node, and if they know someone, there is a connection drawn between their own node, and their acquaintance's. However, graphs can also represent more complex systems like lattices or even computational models of the universe [Wolfram 2020]. Some example graphs can be seen in figure 2.1.

A graph G is formally defined as the pair $(V(G), E(G))$. The set $V(G)$ contains all the vertices of a graph, typically represented as v_i where $i \in \mathbb{N}$ with labels. The set $E(G)$ describes the connections between two vertices, called an edge. These edges are defined as a pair (v_i, v_j) but use a short notation, e_{ij}. The size of the vertex set $|V(G)|$ and edge set $|E(G)|$ are known as the order and size of the graph, respectively.



(a) A K_3 graph



(b) A graph that has a 3-clique

Figure 2.1: An example of two simple graphs: Graph (b) is an extension of graph (b) by adding a fourth vertex that connects to only one other vertex

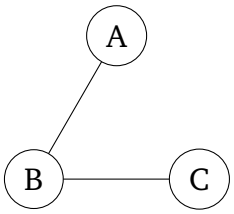
The edges of a graph can have a sense of direction from the ordering of the pair (v_i, v_j). This is extended to the graphs, classifying a graph as either directed or undirected. An undirected graph G, is any graph where (v_i, v_j) and (v_j, v_i) are equal. A graph is simple, when it is undirected, a vertex does not have an edge with itself, and has only one edge between vertices.

It is possible to determine the exact number of simple graphs with order n. Every vertex may have an edge with every other vertex in the graph, which gives $n(n-1)$ edges. However, this double counts the edges because they are undirected, so the maximum number of edges is $\frac{1}{2}n(n-1)$. Any edge can be in one of two states, either it exists ($e_{ij} \in E(G)$) or it does not. That means, there are

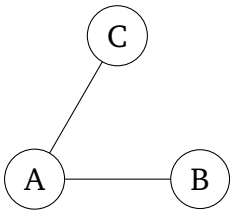
$$g(n) = 2^{n(n-1)/2} \tag{2.1}$$

possible graphs with n vertices.

It is possible that two graphs may have the same edges and vertices if the labels get shuffled. For example, in Figure 2.2, permuting the vertex labels on graph 2.2b counter-clockwise creates the same graph as 2.2a. The number of unlabelled graphs has no known formula, but does have an upperbound of $(2^{n(n-1)/2})/n!$ [Harary and Palmer 1973].



(a) A labelled graph of order 3



(b) A differently labelled graph order 3

Figure 2.2: Two labellings of order 3 graphs that are the same graph under a counter-clockwise permutation.

A graph G will have some structure in itself. This structure can resemble a graph H by removing the vertices and edges from G that are not found in H. This graph H is a subgraph of G, written as $H \subseteq G$. In Figure 2.1, Graph (a) is a subgraph of graph (b)

In all $2^{n(n-1)/2}$ graphs of order n, there are two specific graphs that will always exist. These graphs are known as the complete K_n and empty E_n graphs, where every edge

exists in K_n and no edge exists in E_n. The left graph in Figure 2.1 is an example of the K_3 graph. When K_n is a subgraph, it may also be called an n-clique. The E_n subgraph is called an n-independent set (n-iset). Instead of using a line, and no line to represent the edges, it is also possible to describe the graphs using a color for an edge.

An important part of Graph Theory is the idea of graph colorings [1]. The ability of a graph to represent more complex structures can be extended by specifying that a graph component, either edge or vertex, has a number attached to it. To define this, a k-coloring c_k is mapping from either the graph vertices or graph edges to a set of k numbers, $c : V(G) \mapsto \{1, 2, .., k\}$ or $c : E(G) \mapsto \{1, 2, .., k\}$. These numbers, 1 to k, are called colors and may be represented by an actual colour such as Red or Green.

A 2-coloring of a complete graph is also representative of an uncolored graph of the same order. This can be done by letting a blue edge represent the existence of an edge, and a red edge as the non-existence of an edge. A visual example is provided in Figure 2.3.

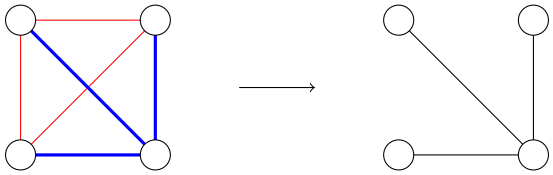


Figure 2.3: A demonstration of how a 2-colored complete graph is also an uncolored graph of the same order

2.2 Ramsey's Theorem

Ramsey's Theorem [Ramsey 1930] states that there is a positive integer $R(k, l)$ that for any graph of order greater than $R(k, l)$ there is either a k-clique or l-independent set in the graph. This integer $R(k, l)$ has since become known as the Ramsey Number of k and l. Simplifying the theorem, it means that all graphs of a large enough order will have either a clique or independent set as part of its structure. The process of adding new vertices to a graph will therefore, predictably create the clique or independent set as at some point the graph must have them.

More recently, the theorem has been generalized to refer to an integer $R(F, H)$ where every graph G will have either F or H as a subgraph [Hattingh *et al.* 2023]. The Ramsey Number $R(F, H)$ generalizes Ramsey Theory from studying the specific substructures of cliques and independent sets, to studying any kind of substructure as the subgraphs F and H. The original formulation of a Ramsey Number $R(k, l)$ can now be restated as $R(K_k, E_l)$. The Ramsey Number $R(F, H)$ is formally defined below, following [Hattingh *et al.* 2023]:

[1] Graph Theory uses the American English spelling of color instead of colour.

Definition 1 Ramsey Number:
There exists a graph G with fewer vertices than the Ramsey Number $R(F, H)$, that does not have F nor H as a subgraph of G, and all graphs with equal or more vertices than $R(F, H)$ will have either F or H as a subgraph.

1. $R(F, H) > n \iff \exists G$, a graph with order n, where $F \nsubseteq G$ and $H \nsubseteq G$
2. $R(F, H) \leq n \iff \forall G$, a graph with order n, where $F \subseteq G$ or $H \subseteq G$

$R(F, H) = n \iff R(F, H) \leq n$ and $R(F, H) > n - 1$

Ramsey Numbers are incredibly difficult to find. As of 2024, there are only 9 known Ramsey Numbers for $R(K_k, E_l)$ with $k, l > 2$ [Radziszowski 2021]. In appendix A, I reprove the Ramsey Number's $R(K_3, E_3) = 6$ and $R(K_2, E_m)$ as an example. To understand the difficulty in proving the upperbound of a Ramsey Number, for a graph with only 10 vertices, a mathematician would need to manually inspect $2^{45} \approx 3 \times 10^{13}$ graphs, or they would need to use the proof technique shown in appendix A to find a contradiction for 10 cases. This seems manageable, but the next closest-to-known Ramsey Number $R(K_4, E_6)$ might be at 36 vertices, or more graphs at $g(36) = 2^{630} \approx 10^{189}$, than there are atoms in the universe, 10^{82} [Planck Collaboration *et al.* 2016]. Ramsey Numbers are a natural candidate for computation, as even the proof technique for $R(K_3, E_3)$ has an algorithmic approach.

Chapter 3

Classical Ramsey Numbers

Ramsey Numbers have an incredibly large search space of $2^{n(n-1)/2}$ possible graphs (Equation 2.1). Even if this number is reduced to the set of unlabelled graphs, there is still an exponential number of graphs that need to be checked. Similar search problems like exact-cover have large search spaces but have achieved success with algorithms such as Knuth's Dancing Links [Knuth 2000].

The computation of $R(k,l)$ can be redefined as deciding if the natural number n is equal to $R(k,l)$. This is the same as deciding if there is a graph G with an order $n-1$ that does not have a k-clique nor l-iset, and that all graphs G with an order of n do have either the k-clique or l-independent set. Therefore, the decision problem *CLIQUE-INDEPENDENT-SET-RAMSEY* (*CIS-RAMSEY*), of deciding if $n = R(k,l)$ is composed of the decision problems *LOWER-CIS-RAMSEY* (*LCR*) and *UPPER-CIS-RAMSEY* (*UCR*) deciding if $n < R(k,l)$ and $n \geq R(k,l)$ respectively.

A decision problem is the simplest form of computational problem. They are formulated as deciding if a string $\langle x \rangle$ is a member of a language L, but languages are defined as sets, so their members are described by some condition. For example the language L may be defined as the set of all strings with that have a number symbol, so deciding membership is done by checking if any of the symbols in $\langle x \rangle$ are the numbers 0,1,2,3,4,5,6,7,8,9. If $\langle x \rangle = 1b0$ then $\langle x \rangle$ is in A, but if $\langle x \rangle = abc$, then $\langle x \rangle$ is not in A.

Using the decision problem notation, the decision problems *LCR*, *UCR* and, *CIS-RAMSEY* for deciding Ramsey Numbers are

Definition 2 :

$$\textit{LCR} = \{\langle n,k,l\rangle : \exists G \text{ of order } n \text{ where } K_k \nsubseteq G \text{ and } E_l \nsubseteq G\}$$
$$\textit{UCR} = \{\langle n,k,l\rangle : \forall G \text{ of order } n \text{ where } K_k \subseteq G \text{ or } E_l \subseteq G\}$$

$$\textit{CIS-RAMSEY} = \{\langle n,k,l\rangle : \langle n,k,l\rangle \in \textit{UCR} \text{ and } \langle n-1,k,l\rangle \in \textit{LCR}\}$$

A reader may notice that the definition of *CIS-RAMSEY* is the decision problem for a k-clique and l-iset Ramsey Number. This is an intentional specification of the problem,

but the general Ramsey Numbers have similar definitions for the decision problems *LOWER-RAMSEY*, *UPPER-RAMSEY* and *RAMSEY*. All the classical algorithms can be easily generalized from *CIS-RAMSEY* towards *RAMSEY*, but the quantum algorithms do not have this same property. As a result, to simplify the report, I shall focus on cliques and independent sets in both cases, and will discuss how to generalize it.

The aim of this section is to explore a classical approach to computing Ramsey Numbers and identify if classical computers are sufficient for this computation. I will do this by performing a complexity class analysis, design three algorithms for Ramsey Numbers, and lastly perform a comparative analysis of the three algorithms to identify the best performing algorithm.

3.1 The Polynomial Hierarchy

The decision problem for Ramsey Numbers, *CIS-RAMSEY*, requires checking for subgraphs, which is itself a decision problem. The complexity class of computing a Ramsey Number is therefore dependent on the complexity of a checking for a subgraph. An understanding of how different complexity classes relate to each other is important to correctly identify the difficulty of Ramsey Numbers.

Decision problems are categorized by the amount of resources required for an algorithm to complete on an abstract computer, called a Turing Machine. The resources are the amount of time it takes for the computation, and the amount of space needed for the computation. They are measured as a function of how long the input is. If the function is in the form of a polynomial like $\text{poly}(n) = \sum_{i=0}^{\infty} a_i n^i$ where a_i is a real number coefficient and n is the size of the input, then it has a polynomial resource usage (runtime / space-usage). If it is an exponential function such as $2^{\text{poly}(n)}$, then it has an exponential resource usage.

The common categories are the complexity classes of *P*, problems that can be decided in a polynomial runtime, *NP*, problems that can verify a possible solution in polynomial runtime, and *EXPTIME*, problems decidable in an exponential runtime. These categories are quite broad however, and there exists a series of classes that fall between them.

The in-between classes form what is known as the Polynomial Hierarchy *PH* [Sipser 2012, p414]. The *PH* is constructed by how difficult a problem is, when considered relatively to another problem. As an example, let's say a problem *A* requires a sorted array of numbers before it can begin solving the problem, but there is no guarantee that it's input array is sorted. Problem *A*'s difficulty is now relative to the difficulty of sorting an array.

If A can be solved in polynomial time once it has a sorted array, then it is said that A is in *P* relative to *SORT*, notationally as P^{SORT}. If A can be verified in polynomial time, then it is in NP^{SORT}.

To show the relative complexity of a problem A relative to B, an algorithm for A makes use of a specific type of Turing Machine called an oracle to solve B [Sipser 2012, Chapter 9.2]. An oracle is a Turing Machine that is assumed to be able to decide if a string

w is in B in constant time. Using oracles, the runtime of a Turing Machine becomes a measure of how many times it needs to query the oracle before it can reach a solution. If it can solve A with a polynomial number of queries, then $A \in P^B$, if it can verify A in a polynomial number of queries, then $A \in NP^B$.

The oracle of a complete problem has some additional power when compared to a standard oracle. This complete oracle can be used to solve any problem in the class, and thus it becomes an oracle for the entire class as well. For example, the problem of boolean satisfiability *SAT* is *NP*-complete [Cook 1971] which means it can solve all other *NP* problems, therefore a *SAT* oracle O may be used to solve any *NP* problem by querying O. This makes an oracle to a complete problem, representative of the whole class. Continuing the example of *SAT* any problem that is P^{SAT} is P^{NP}.

The layers of the *PH* are built using a complete oracle from the previous layer. A problem in *NP* is in the first layer of the *PH*, a problem in NP^{NP} is in the second layer, the third layer is $NP^{NP^{NP}}$ and so on. This notation can become quite cumbersome, however, and is instead replaced with $\mathbf{\Sigma}_i^P$ where i is the current layer of the hierarchy and $\mathbf{\Sigma}_i^P = NP^{\mathbf{\Sigma}_{i-1}^P}$.

The recursive definition of each layer means that are a theoretically infinite number of levels to the *PH*. The number of levels in the *PH* is an open question. However, the class *PSPACE* i.e. all problems that are decidable in a polynomial space-usage, does contain the entire *PH*, because any algorithm that runs in a polynomial runtime can use at most one new piece of memory every step, so it must also have a polynomial space-usage.

So far, the complement of *NP* problems has not really been discussed. This class of problems, known as co*NP*, are harder than *NP* problems, as they require more computation to check an answer. A 'yes' answer to a decision problem, can be thought of as *there is at least one possible solution to this problem*, but a 'no' answer requires checking that *there are no solutions for this problem*. The complementary notation of $\mathbf{\Sigma}_i^P$ is $\mathbf{\Pi}_i^P$ That is the complement of an NP^{NP} problem is in co$NP^{NP} = \mathbf{\Pi}_2^P$.

3.2 Classical Complexity of Ramsey Numbers

The complexity class of *CIS-RAMSEY* is dependent on the classes of *UCR* and *LCR* from definition 2. The first observation of *LCR* and *UCR* is that they are complimentary problems, i.e. if $\langle n, k, l\rangle \in$ *UCR* then $\langle n, k, l\rangle \notin$ *LCR*. The proof of this observation makes use of De Morgan's Laws and is the following:

$$
\begin{aligned}
&\text{let } \langle n, k, l\rangle \notin \textit{UCR} \\
&\quad \implies \neg(\forall G \text{ with } n \text{ vertices where } K_k \subseteq G \text{ or } E_l \subseteq G) \\
&\quad \implies \neg\forall G \text{ with } n \text{ vertices where } \neg(K_k \subseteq G \text{ or } E_l \subseteq G) \\
&\quad \implies \exists G \text{ with } n \text{ vertices where } K_k \nsubseteq G \text{ and } E_l \nsubseteq G \\
&\quad \therefore \langle n, k, l\rangle \in \textit{LCR}
\end{aligned}
$$

The use of a psuedocode style description for a Turing Machine, used in the Lemma 1, is not intended to be implemented as an algorithm on a computer. Instead, it is in-

tended to act as a high level description of the computation, and how that computation has some identifiable and mathematically analysable behaviour. More information on Turing Machines and their use in proofs can be found in [Sipser 2012, Chapter 3].

Using this observation, it can be shown that the complexity class of *CIS-RAMSEY* is at least in the complementary classes of the PH, $\mathbf{\Pi}_i^P$ or in the class of *LCR* if it is closed under its complement, such as *P* or *EXPTIME* (see appendix B.1).

Lemma 1 :

By letting the Turing Machines L and U decide *LCR* and *UCR* in time $T_U(x)$ and $T_l(x)$, respectively, the Turing Machine R that decides *CIS-RAMSEY* works as follows:

define TM R on input $\langle n, k, l \rangle$

1: Simulate $L(\langle n, k, l \rangle)$:
if $L(\langle n, k, l \rangle) = 1$, then reject else continue to step 2
2: Simulate $U(\langle n-1, k, l \rangle)$:
if $U(\langle n-1, k, l \rangle) = 1$, then accept else reject

The runtime of R is the time it takes to execute L and, then execute U. This gives the runtime function for R as $T_R(|\langle n, k, l \rangle|) = T_L(|\langle n, k, l \rangle|) + T_U(|\langle n, k, l \rangle|)$.

The class of *CIS-RAMSEY* is the class of the longer runtime between *LCR* or *UCR* from $T_R(x)$ AS *LCR* and *UCR* are complimentary, *CIS-RAMSEY* is at least in the same class as the *LCR*.

Finding the class of *CIS-RAMSEY* is now just a matter of finding the class of *LCR*. *LCR* is defined by deciding if any graph G of n vertices has either the subgraph K_k or E_l, so solving *LCR* is dependent on finding a subgraph. The *SUBGRAPH-ISOMORPHISM-PROBLEM* (*SIP*) is a well researched problem known to be *NP*-complete [Cook 1971]. It is defined as the following decision problem

$$\textit{SIP} = \{\langle G, H \rangle : H \subseteq G\}$$

By using an oracle for *SIP*, the runtime of *LCR* can be identified from the following lemma.

Lemma 2 :

Let Oracle S decide *SIP* with cost $O(1)$. Now I can define TM L that decides *LCR*:

define TM L on input $\langle n, k, l \rangle$

1: Non-deterministically generate a string $\langle G \rangle$ that represents a graph of n vertices
2: Build the string $\langle G, K_k \rangle$
3: Simulate $S(\langle G, K_k \rangle)$:
if $S(\langle G, K_k \rangle) = 1$ then accept else continue to step 4
4: Build the string $\langle G, E_l \rangle$
5: Simulate $S(\langle G, E_l \rangle)$:
if $S(\langle G, E_l \rangle) = 1$ then accept else reject

Step 1 takes $O(n^2)$, step 2 $O(n^2 + k^2)$ to merge the two strings of length $O(n^2)$ and $O(k^2)$, similarly step 4 is $O(n^2 + l^2)$, and 3 and 5 are constant $O(1)$ by the oracle.

$$\begin{aligned} T_L(|\langle n, k, l\rangle|) &= O(n^2) + O(n^2 + k^2) + O(1) + O(n^2 + l^2) + O(1) \\ &= O(n^2 + \max(k^2, l^2)) \end{aligned} \tag{3.1}$$

Equation (3.2) is a polynomial, therefore L decides *LCR* in non-deterministic polynomial time. Importantly, this is relative to the *NP*-complete oracle of *SIP* which gives the algorithm a non-deterministic polynomial runtime relative to *NP*. Hence,

$$\textit{LCR} \in \textit{NP}^{\textit{NP}} = \mathbf{\Sigma}_2^P$$

and

$$\textit{UCR} \in \text{co}\textit{NP}^{\textit{NP}} = \mathbf{\Pi}_2^P$$

From lemmas 1 and 2, it follows that

Thereom 1: *CIS-RAMSEY* $\in \mathbf{\Pi}_2^P$

The computation, on a classical computer, to decide if a number n is *NOT* a Ramsey Number $R(k, l)$ can be verified in polynomial time relative to verifying that a graph H is a subgraph a graph G in polynomial time.

The next step of *CIS-RAMSEY*'s complexity is to show that it is $\mathbf{\Pi}_2^P$-complete. This is a more involved process than what the scope of this report allows, but the general case *RAMSEY* is at least *NP*-hard [Burr 1987]. *RAMSEY* being *NP*-hard does suggest that *CIS-RAMSEY* is in a similar ballpark, and the *RAMSEY* reduction could be used for *CIS-RAMSEY*. This remains as an avenue of future work.

3.3 Classical Algorithms

The actual algorithms to compute a Ramsey Number that can be implemented on a classical computer are composed of the two algorithms that encode a graph into a computer-usable datastructure, and then performing the subgraph search. This section therefore details these dependent algorithms of a graphs encoding and searching for a subgraph, before then exploring a brute force approach to computing Ramsey Numebrs with two subsequent optimisations of this computation.

3.3.1 Graph Encoding

A graph G can be represented with the following encoding scheme. Every graph of order n has an adjacency matrix of size $n \times n$ where each column and row of the matrix represents a vertex. The value in the matrix at a column c and row r is the existence of the edge between the vertices v_c and v_r. If it is 1, then edge e_{rc} is in $E(G)$, and 0 if e_{rc} is not in $E(G)$.

The encoding can be further compressed by using the definition of a simple graph. As no vertex will share an edge with itself, the diagonal of every matrix is 0, and each edge is undirected, so the matrix is symmetrical about the diagonal. By taking each element of a row, after the diagonal element, and concatenating it to the previous elements, I can build a binary string encoding of a graph. This string will be referred to as the adjacency string of a graph, or notationally as $\langle G \rangle$.

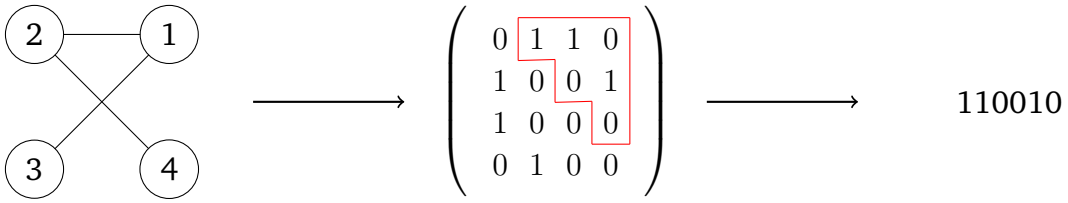


Figure 3.1: The encoding $\langle G \rangle$ of a graph G with 4 vertices.

The benefit of the string $\langle G \rangle$ is that any string will uniquely identify a graph. This occurs because the string must be a fixed width to correctly encode the graph. For example, the strings 000 and 000000 both encode an empty graph, but the first is for an empty graph of 3 vertices and the second is an empty graph of 4 vertices. The exact length of $\langle G \rangle$ is given by $n(n-1)/2$, which as a binary string gives us $2^{n(n-1)/2}$ possible strings for a given n. Appropriately, this also gives us the number of graphs for n vertices, and thus there is a bijective mapping between binary strings of a fixed length and graphs.

It is possible to go further, and reduce the binary string into a decimal number. However, to maintain the uniqueness between a graph and a number, the number of vertices for the graph must also be provided. The strings 000 and 000000 encode different graphs, but their corresponding decimal numbers are both 0. To recover the graph from a decimal number, the number of vertices, e.g. 3, to correctly revert the decimal number 0, to the binary number 0, to the binary string of 000.

Algorithm 1 Constructing a graph from an integer id

```
function GRAPH-FROM-INT(i, n)
    s ← bin(i)
    for j ← ⌈log2(i)⌉ to n(n − 1)/2 do
        s ← 0 ∘ s
    return s
```

Algorithm 1 works by converting the decimal integer i into its binary represention s, and then appending any missings 0's to the front of the string to reach the correct length required for a graph of order n. The worst case of this algorithm is for the integer 0, where it must append $n(n-1)/2 - 1$ 0's to the front of s, which gives the algorithm a runtime of $O(n^2)$.

3.3.2 Clique & Independent Sets Searching

With a given graph G, any *CIS-RAMSEY* algorithm will need to perform some form of search for the k-clique or l-iset. As previously mentioned, a subgraph search has

pre-existing algorithms and implementations [McCreesh *et al.* 2020] [Ullmann 1976]. However, to simplify scope, this algorithm will make use of a simple depth first search tree approach inspired Jeffrey Ullman's recursive algorithm [Ullmann 1976].

The algorithm searches for the subgraph H in the graph G. By enumerating the ways to choose $|V(H)| = k$ vertices from $|V(G)| = n$ vertices using a tree, the algorithm will build the mapping α of vertices in H to the chosen vertices of G. Then it will determine H is a subgraph of G if, every edge between H's vertices and G's chosen vertices are equal. If the edge e_{ij} in H is 1, then the value of edge $e_{\alpha(i)\alpha(j)}$ in G must also be 1. The pseudocode is in Algorithm 2 and has a runtime of

$$T(a, b) \in O(\binom{a}{b} b^2)$$

It diverges from Ullman's algorithm by not performing an optimisation step that reduces the number of nodes in search tree. Ullman's algorithm searches through the list of potential subgraphs by constructing them through a binary tree. It then prunes the tree if the partially constructed subgraph cannot have the desired subgraph. *Has-Subgraph* does not perform the optimising pruning step, but constructs the subgraphs in the same process as Ullman's algorithm.

Searching for a k-clique in G can be performed by calling *Has-Subgraph*(G, K_k), and the l-iset search is calling *Has-Subgraph*(G, E_l). As this algorithm searches for a subgraph H in G, instead of a clique or iset directlty, it easily generalizes any classical *CIS-RAMSEY* implementation that uses this algorithm to a classical implmenetation of *RAMSEY*. This means that the next three algorithms for a Ramsey Number can all work in the general case of $R(F, H)$ if it is modidfied to accept the graphs H and F instead of intergers k and l.

3.3.3 Linear Enumeration

With a subroutine to search for cliques and isets, the last step the algorithm requires is some way to enumerate all the graphs of order n. This can be done easily enough by the decimal encoding of graphs. The algorithm already has access to the number of vertices, so it can easily reconstruct the graph from a natural number. Using a for loop i from 0 to $2^{n(n-1)/2}$, an algorithm can iterate the set of all graphs by converting i to $\langle G \rangle$.

Combining the linear iteration with the subgraph search for the clique K_l and iset E_l for every graph is the simplest algorithm to compute *LCR*. The pseudocode is found in Algorithm 3.

The runtime functions for *Graph-from-Int* (Algorithm 1) and *Has-Subgraph* (Algorithm 2) are $O(n^2)$ and $O(\binom{a}{b} b^2)$ where $a = |V(A)|$ and $b = |V(B)|$. *Graph-from-Int*, *Has-Subgraph* for the k-clique and, *Has-Subgraph* for the l-iset are called for all $2^{n(n-1)/2}$

Algorithm 2 Searching for a subgraph B in a graph A

```
function HAS-SUBGRAPH(A, B)
    initialize choice_stack
    a ← A.order
    b ← B.order
    while choice_stack.size ≠ 0 do
        current_choice ← choice_stack.pop()
        found_subgraph ← True
        if current_choice.size = b then
            for i ← 1 to b do
                α ← current_choice[i]
                for j ← 1 to b do
                    if i = j then
                        continue to next j
                    β ← current_choice[j]
                    index ← −½i² + (b − ½)i + j − i − 1
                    mapped_index ← −½α² + (a − ½)α + β − α − 1
                    found_subgraph ← found_subgraph AND A[mapped_index] =
B[index]
            if found_subgraph then return True
        else
            for i ← 0 to a do
                if i in current_choice then
                    continue to next i
                choice_stack.push( current_choice.push(i) )
    return False
```

graphs. Thus, the runtime function of *Linear-RamNum* (Algorithm 3) is

$$\begin{aligned} T(|\langle n,k,l\rangle|) &= O(2^{n(n-1)/2})\left(O(n^2)+O(\binom{n}{k}k^2)+O(\binom{n}{l}l^2)\right) \\ &= O(2^{n^2/2-n/2}n^2)+O(2^{n^2/2-n/2}\binom{n}{k}k^2)+O(2^{n^2/2-n/2}\binom{n}{k}k^2) \\ &= O(2^{n^2}\cdot\binom{n}{\max(k,l)}\cdot\max(k,l)^2) \end{aligned} \tag{3.2}$$

where k and l are the specified clique and iset size.

3.3.4 Binary Tree Enumeration

The linear enumeration algorithm handles every graph individually, but certain graphs can share edges, and therefore potentially subgraphs. An example of this can be seen in Figure 3.2. Using this property, an algorithm can be optimised to eliminate multiple graphs with only one subgraph search. As the subgraph search is itself an expensive operation, it is useful to minimise the number of times it is called into.

Algorithm 3 Computing a Ramsey Number using linear enumeration

function LINEAR-RAMNUM(n, k, l)
  for $i \leftarrow 0$ to $2^{n(n-1)/2} - 1$ **do**
    $G^i \leftarrow$ GRAPH-FROM-INT(i)
    if HAS-SUBGRAPH(G^i, K_k) equals *False* **then**
      if HAS-SUBGRAPH(G^i, E_l) equals *False* **then return** G
  return -1

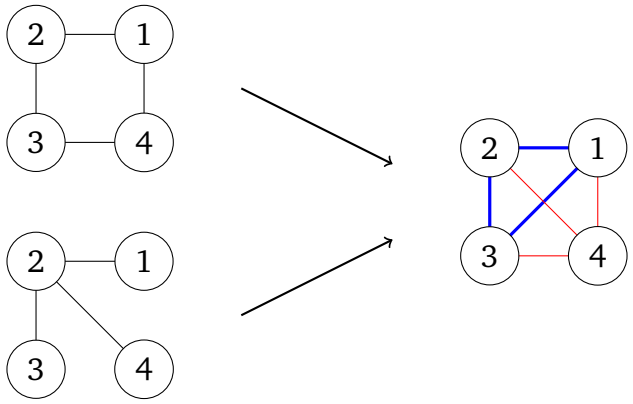


Figure 3.2: Common edges between two different graphs: A blue edge is an edge that exists or does not exist in both graphs, and a red edge is an edge that exists in one graph and, does not exist in the other.

If the algorithm enumerates the graphs by constructing them as a tree, it is possible to exploit this commonality. For every node in the tree, create a new left child node by filling in the next edge as 'empty' (0) and the right child node fills in the edge as 'complete' (1) (Figure 3.3). Then at each node, perform a subgraph search through the unfinished graph, checking if what has already been filled in has the subgraph. A slight modification of *Has-Subgraph* is required to treat any unfilled edges as not satisfying the edge matching constraint. This modification is just a check of the edge value so can be performed in constant time, i.e. not modifying its performance. If it has the subgraph, then the search can 'prune' the subtree as it is guaranteed to have the subgraph as well. The psuedocode for this process is in Algorithm 4.

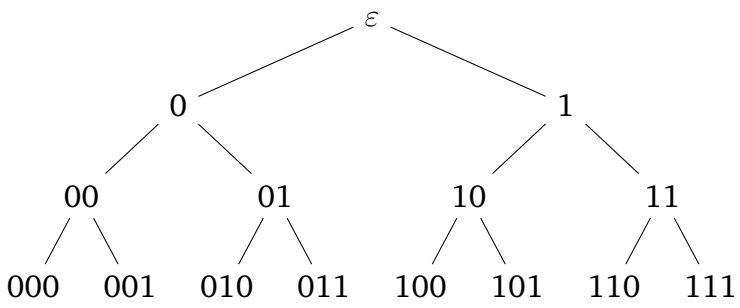


Figure 3.3: The binary tree enumeration of order 3 graphs

Strictly speaking, if there are no subgraphs at any node, then the algorithm will now perform the subgraph search for all the nodes in the binary tree. The depth of our tree is given by the length of the graph string, $n(n-1)/2$, so there are $2 \times 2^{n(n-1)/2} - 1$ nodes. However, the intuition is that the pruning process will overcome this differential in the number of nodes, and thus overall there will be less subgraph searches.

Algorithm 4 Computing a Ramsey Number using binary tree enumeration

function BINTREE-RAMNUM(n, k, l)
  $K \leftarrow$ the complete graph with k vertices
  $E \leftarrow$ the empty graph with l vertices
  initialize graph_stack
  $\varepsilon \leftarrow$ a blank graph
  graph_stack.push(ε)
  while graph_stack.size $\neq 0$ **do**
    $G \leftarrow$ graph_stack.pop()
    if $i \neq n$ **then**
      $G' \leftarrow G \circ 1$ ▷ The symbol $\circ$ is set the next edge to the value, or append 1 to the string $\langle G \rangle$
      if not HAS-SUBGRAPH(G', K) and not HAS-SUBGRAPH(G', E) **then**
        graph_stack.push(G')
      $G'' \leftarrow G \circ 0$
      if not HAS-SUBGRAPH(G'', K) and not HAS-SUBGRAPH(G'', E) **then**
        graph_stack.push(G'')
    else
      if not HAS-SUBGRAPH(G, K) and not HAS-SUBGRAPH(G, K) **then return** $\langle G \rangle$

Importantly, the runtime cost of the Binary Tree enumeration is

$$\begin{aligned} T(n) &= O(2^{n(n-1)/2+1}) \left(O(\binom{n}{k} k^2) + O(\binom{n}{l} l^2) \right) \\ &= O(2^{n^2} \binom{n}{\max(k,l)} \cdot \max(k,l)^2) \end{aligned} \tag{3.3}$$

as the algorithm constructs each graph as a leaf node in the tree and performs the two searches. The number of leaf nodes in a binary tree is 2^d where d is the depth, and the total number of nodes is $2^{d+1} - 1$. The algorithm needs to make a string of length $n(n-1)$, which gives it a depth of $d = n(n-1)/2$, matching the number of leaf nodes to equation 2.1. Even if the algorithm did not perform any pruning it would still have the same runtime complexity function. To get an exact idea on which of these two algorithms has a better runtime, a more in-depth theoretical analysis is required, or alternatively an empirical analysis can be performed.

3.3.5 Reduced Binary Tree Enumeration

Using the binary tree, instead of starting at a completely unfinished graph ε, it is possible to jump start the algorithm at a specific depth in the tree of n. If the search begins

with the following graphs instead of the blank graph, it can reduce the size of the tree and still be correct: Add n incomplete graphs to the search tree, where counting from 0 to $n-1$ as i, fill in the first n edges with i 1s and $n-i$ 0s.

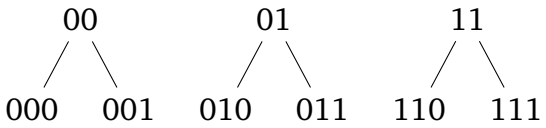


Figure 3.4: The reduced binary tree enumeration of order 3 graphs

This works similarly to the proof process of a Ramsey Number (see Appendix A for more details) where the algorithm examines specific cases that will encounter all the unlabelled graphs of order n. In essence, what is being done is a permutation of the graph labels so that the first vertex will always have a specific degree of $0, 1, 2, ..., n-1$. This will help remove the graphs that are identical under some permutation labelling from the enumeration. This optimisation will still correctly check every graph as a Ramsey Number is not dependent on the vertex labellings.

Algorithm 5 Computing a Ramsey Number using a reduced binary tree enumeration

function REDUCED-BINTREE-RAMNUM(n, k, l)
  $K \leftarrow$ the complete graph with k vertices
  $E \leftarrow$ the empty graph with l vertices
  initialize graph_stack
  for $i \leftarrow 0$ to $n-1$ **do**
    $G \leftarrow$ blank graph
    for $j \leftarrow 1$ to $n-1$ **do**
      $G \circ 0$
    for $j \leftarrow i+1$ to $n-1$ **do**
      $G \circ 1$
    graph_stack.push(G)
  while graph_stack.size $\neq 0$ **do**
    $G \leftarrow$ graph_stack.pop()
    if $i \neq n$ **then**
      $G' \leftarrow G \circ 1$
      if not HAS-SUBGRAPH(G', K) and not HAS-SUBGRAPH(G', E) **then**
        graph_stack.push(G')
      $G'' \leftarrow G \circ 0$
      if not HAS-SUBGRAPH(G'', K) and not HAS-SUBGRAPH(G'', E) **then**
        graph_stack.push(G'')
    else
      if not HAS-SUBGRAPH(G, K) and not HAS-SUBGRAPH(G, K) **then return** $\langle G \rangle$

The standard binary tree iterates through $2^{n(n-1)/2+1}-1$ nodes without pruning, whereas the reduce binary tree will remove the starting portion of the tree, and replace it with

n subtrees of depth $n(n-1)/2 - n + 1$. The exact runtime of Algorithm 5 is therefore

$$\begin{aligned} T(|\langle n,k,l\rangle|) &= O(n^2) + O(n2^{n^2}\binom{n}{\max(k,l)} \cdot \max(k,l)^2) \\ &= O(n2^{n^2}\binom{n}{\max(k,l)} \cdot \max(k,l)^2) \end{aligned} \tag{3.4}$$

Counting the number of nodes in the tree, there are $2n \cdot 2^{n(n-1)/2-n+1}$ nodes in the reduced binary tree and $2 \cdot 2^{n(n-1)/2}$ nodes in the standard binary tree. The exact runtime of Algorithm 5 is therefore

$$\begin{aligned} T(|\langle n,k,l\rangle|) &= O(n^2) + O(n2^{n^2}\binom{n}{\max(k,l)} \cdot \max(k,l)^2) \\ &= O(n2^{n^2}\binom{n}{\max(k,l)} \cdot \max(k,l)^2) \end{aligned} \tag{3.5}$$

Therefore, the reduced binary tree will iterate through

$$\Delta = 2^{n(n-1)/2+1} - 1 - (n2^{n(n-1)/2+1-n+1} - 1) = 2^{n(n-1)/2+1}(1 - n2^{1-n}) \tag{3.6}$$

less nodes each call. Even though the algorithm still has approximately $O(2^{n^2})$ nodes to search through, this is a major reduction in the exact number of nodes. Considering Δ is exponentially dependent on n, this optimisation will also become even more effective at larger n.

As each algorithm has an identical or close to identical runtime complexity, an empirical analysis is required to fully understand their performance differences. Therefore, the next sections details the experimental procedure and results necessary for the analysis.

3.4 Classical Experiments

3.4.1 Methodology

The empirical experiment of each algorithm will follow the outlined methodology below. Each algorithm shall be tested to produce correct results, and then tested on their measured runtimes.

To test whether the algorithms are correct, the Ramsey Numbers $R(3,3) = 6$, $R(2,4) = 4$ and $R(2,5) = 5$ are given as input for at least one value below $R(k,l)$ and $R(k,l)$ itself. In each case, the algorithm is expected to produce some resultant graph if $n < R(k,l)$ and -1 if $n \geq R(k,l)$ (Table 3.1). These cases are chosen from the proofs provided in appendix A.

Due to the exponential runtime of all the algorithms, the experiments cannot run on large test cases and have repeated executions. Thus, runtime measurements are restricted to either a single data point at a given n, or restricted to smaller n with multiple executions.

n	k	l	Expected Result
4	3	3	any $\langle G \rangle$ with 4 vertices with no 3-clique and no 3-iset
5	3	3	any $\langle G \rangle$ with 5 vertices with no 3-clique and no 3-iset
6	3	3	-1
3	2	4	$\langle E_3 \rangle$
4	2	4	-1
4	2	5	$\langle E_4 \rangle$
5	2	5	-1

Table 3.1: Correctness test cases of the classical algorithms

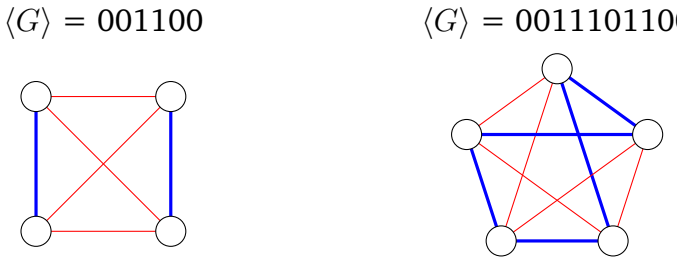


Figure 3.5: The visualised graphs from row 1 and 2 of Table 3.2. A red edge is 0, and a blue edge is 1

If an algorithm is correct, it will run on $n \in [5, 10]$ for $k, l \in [3, 5]$ at least 5 times, where the runtime of each execution will be measured by the wall clock time of its execution, and not its CPU time. I shall take the mean runtimes of these 5 execution, fit them to a regression, and analyse the results to identify which of the correct algorithms produce the fastest results.

3.4.2 Results

Each algorithm, executed on a Xeon E5-2680 CPU and with 32GB RAM, produced correct results, and interestingly enough, they produced identical results for the first two cases (Table 3.2). The code is available at `https://github.com/orwellian225/ramsey-numbers`. The graphs from case 1 and 2 can be seen in Figure 3.5 where neither graph has a 3-clique or 3-iset.

n	k	l	Expected Result	Linear	Bin-Tree	Reduced-Bin-Tree
4	3	3	$\langle G \rangle$	001100	001100	001100
5	3	3	$\langle G \rangle$	0011101100	0011101100	0011101100
6	3	3	-1	-1	-1	-1
3	2	4	$\langle E_3 \rangle$	000	000	000
4	2	4	-1	-1	-1	-1
4	2	5	$\langle E_4 \rangle$	000000	000000	000000
5	2	5	-1	-1	-1	-1

Table 3.2: Correctness test results of the classical algorithms

In Figure 3.6, every algorithm runs faster at a larger k and l for the smaller values of n. As n grows, however, the larger k and l grow far more rapidly than the smaller k and l and this early gap is quickly lost. Each curve is also roughly quadratic on the log scale.

Algorithm 3 is an outlier where the overtaking pattern of did not occur. A reader might notice that the linear case only runs until $n = 8$ instead of $n = 10$. The reason for this is that while trying to run this algorithm, it never produced a result for $n > 8$ once before timing out. Therefore, it is reasonable to assume that this pattern repeats itself, but the linear case is far too slow to be able to observe this with data given the hardware available.

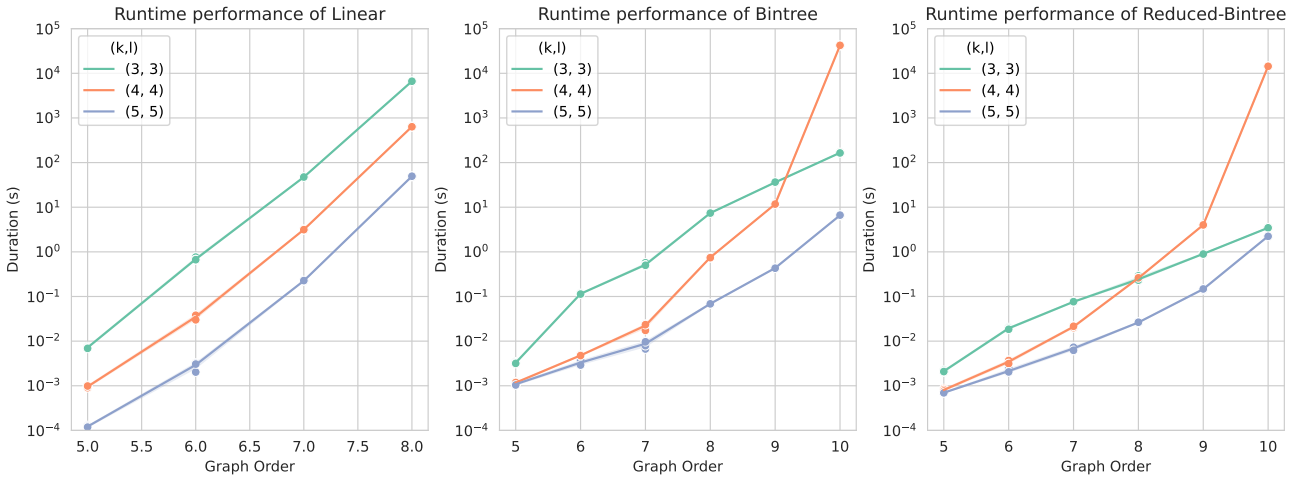


Figure 3.6: The runtime performance of the Linear, Binary-Tree and, Reduced-Binary-Tree algorithms

Between the three algorithms, the linear search (Algorithm 3) performed the worse. It consistently has at least a one order of magnitude greater execution duration than both the Binary Tree algorithms (Algorithms 4 and 5) , and as the size of n grows, this gap becomes even larger. The most interesting case is on the third graph of Figure 3.7 where at $R(5, 5)$ and $n = 5$, the linear algorithm is significantly faster than the other two.

For the binary tree algorithms, their results are basically identical. The Reduced-Binary-Tree (Algorithm 5) was always faster than the standard Binary-Tree (Algorithm 4) but this is a very small difference comparing them at a log scale. Comparing the absolute difference between the two does paint a different picture in Figure 3.6.

The difference Δ of the Binary-Tree runtime minus Reduced-Binary-Tree runtime is increasing with growing n, and it shows the pattern of larger k and l performing better until a sudden jump in the computation time (Figure 3.8). The gap between the different algorithms is increasing, at a near exponential rate (linear on the log scale).

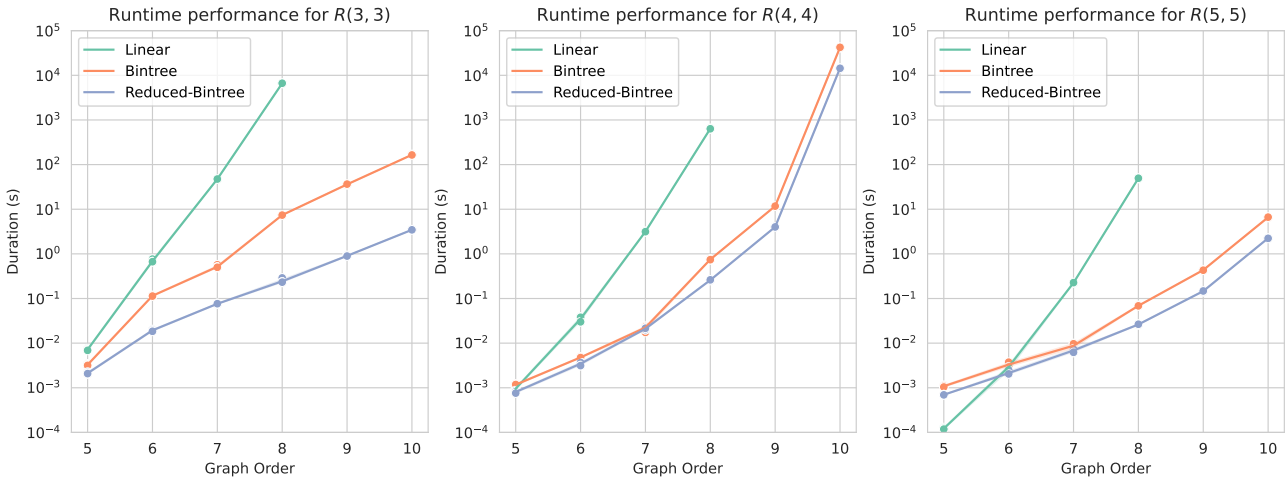


Figure 3.7: The runtime performance of each algorithm for a specific case of $R(k,l)$

3.5 Analysis

The identical results achieved for the first two cases of each algorithm is an interesting coincidence, but is explainable. While each algorithm does have a specified search order, the exact ordering of the encountered graphs is dependent on implementation details. For example, if the depth first search (DFS) for Algorithms 4 and 5 go down the right branch instead of the left, it will encounter the graphs with more edges first, and possibly lead to a different result. The DFS detail of left branch or right branch will specify if the graphs are encountered in ascending ID order (as in the linear algorithm) or descending order.

As this matching result is dependent on how likely the graph without either subgraphs existing is, it is unlikely to keep happening at larger values of n. This does highlight a potential future optimisation where the most likely candidates for the graph without either subgraph, are in the centre of the graphs ordered by their IDs. By searching this centre first using some form of heuristic, the algorithm might gain some speedup.

Comparing the runtime performance of each algorithm, there is a clear demonstration that the optimisations are working correctly. At a theory level, all the algorithms match their runtime functions of approximately $O(2^{n^2}\binom{n}{\max(k,l)})$, but there is a clear gap in performance from the linear to binary tree algorithms. This suggests that the intuition of reducing the number of subgraph searches with the pruning is very effective at improving performance. The influence of the $\binom{n}{\max(k,l)}$ is also visible, but counter-intuitively, the larger k,l's perform better at smaller n.

The reason for this is that the larger k and l's are more likely to have a suitable graph G that satisfies *LCR* so it is easier for smaller n. However, as n gets closer to the actual value of $R(k,l)$ and is more likely to match *UCR*, it needs to search through more and more graphs, until it must search through every single graph. This makes the algorithm faster for small n with higher k and l as they will push the actual value of $R(k,l)$ into larger values of n.

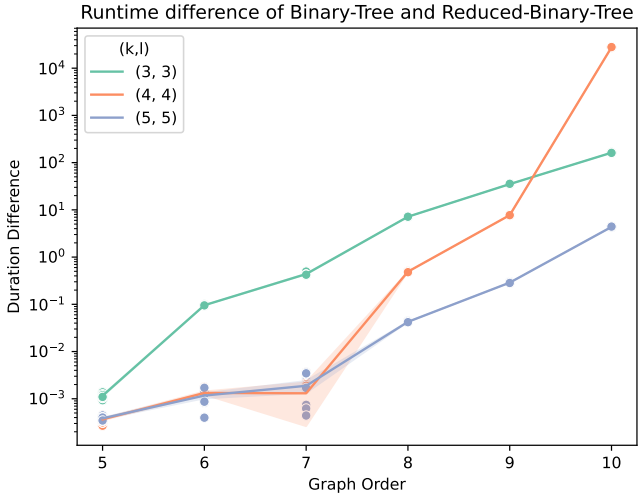


Figure 3.8: The absolute difference of measured runtimes for the Binary Tree and Reduced Binary Tree algorithms with a mean estimation over the data points, with the associated distribution

Between the three algorithms, the Reduced-Binary-Tree is the best performing. Although it effectively matches the Binary-Tree algorithm, the difference in performance Δ does grow much larger with increasing n, again matching the theoretical result for Δ.

Although the Reduced-Binary-Tree algorithm is significantly better than the Linear algorithm, and an improvement over the Binary-Tree algorithm, it is still incredibly slow. By $n = 10, k = 4, l = 4$, it took 4 hours to get a result. Fitting the measured results to a regression (Figure 3.9), the projected runtime to compute the value of $R(5, 5)$, i.e. testing the values of $n \in \{43, 44, 45, 46\}$ will take 1.2×10^{142} seconds (3×10^{132} centuries).

The classical algorithm does work, and it will produce correct results. It is just incredibly unreasonable to let this computation run. There are potential optimisations in using a search heuristic or improving the subgraph searching, but these optimisations are not expected to bring the computation into the more reasonable timescale of a researcher's lifetime. Even considering a parallelization strategy of putting a search subtree on separate processors with every processor doing as much work as possible, the computation would be reduced by a factor of the number of processors and thus not reduce the computation away from 10^{100} order of magnitude.

A Ramsey Number computation is slow, and requires significant resources. Clearly, the classical approach to computing a Ramsey Number is insufficient, and an alternate approach is required to even dream of a reasonable computation.

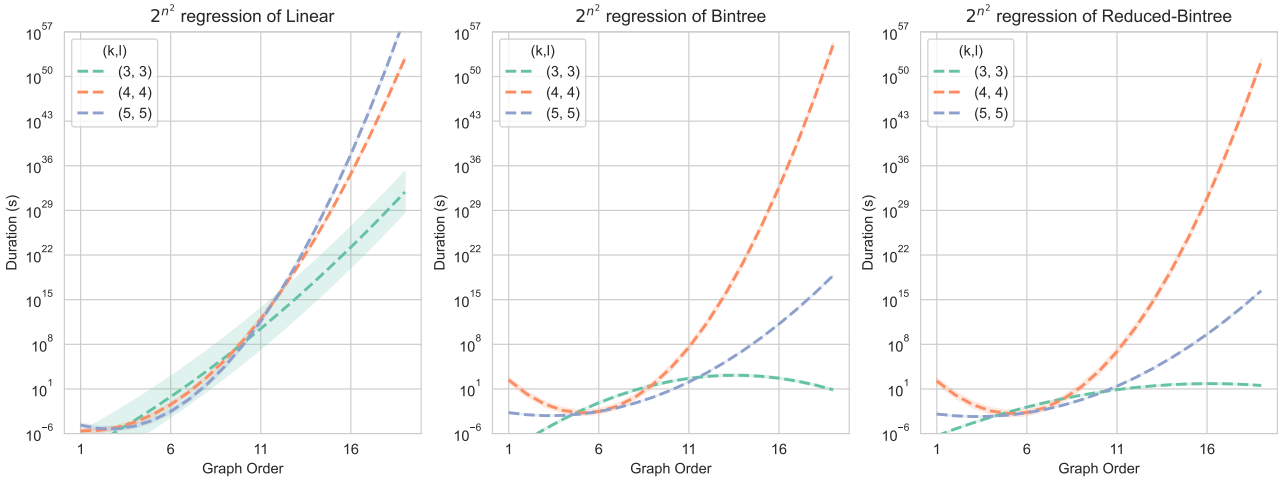


Figure 3.9: Fitting the performance of each algorithm to the regression 2^{an^2+bn+c}, drawn with the bounds of the regression fit

Chapter 4

Quantum Ramsey Numbers

Quantum Computers are a new approach to computation problems. They process information differently to a classical computer and are able to leverage quantum mechanics principles such as superposition to solve problems. These changes in information processing has lead to algorithms that are in *P* for classically *NP* problems. The most famous example is Shor's algorithm [Shor 1994] where Shor is able to exploit a quantum-computing specific technique called phase-kickback to factorize an integer in polynomial time.

On the other hand, quantum computers are not a silver bullet. For search algorithms, the best known quantum improvement is only quadratic with Grovers Search [Bennett *et al.* 1997]. The lack of theoretical results for quantum computers have led to researchers trying to find the 'Quantum Advantage' empirically. The Quantum Advantage is the concept that a quantum computer has an overwhelming performance advantage over a classical computer. Whether a quantum advantage exists is an open question, with successes [Zhu *et al.* 2022], and then refutations [Oh *et al.* 2024] being common.

This variability in the success of quantum computers therefore requires an investigation into their uses for solving Ramsey Numbers. Previous work has shown some success [Gaitan and Clark 2012] [Wang 2016], but these are limited to simulated results or on out-of-date hardware by the time of writing.

This section will investigate how a quantum computer works, how quantum algorithms modify the classical complexity of Ramsey Numbers, and lastly, implement and test two different quantum algorithms.

4.1 How Quantum Computers Work

Quantum computers differ from classical computers because they provide a different way of expressing computation [Deutsh 1989]. Classical computers operate with two states, 0 and 1. The basic tool of classical computing, a bit, exists in only one of these two states. Any complex structure must be representable as a concatenation of bits, a bit-string, with some interpretation of the bit-string to represent the structure. Any

computation is then performed by a series of logical gates that transform a bit-string into a different bit-string.

Quantum computers instead have access to a more complex primitive, the qubit. A qubit can exist in a probabilistic superposition of bit states 0 and 1, and is described as the linear combination of some chosen basis vectors.

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,,\ \text{where}\ \alpha^2 + \beta^2 = 1$$

These gates can modify the probabilities of a qubit's state, instead of just the state itself. Individual qubits can encode more information into an object, and have different behaviour to a classical bit (cbit), which can be used to provide additional computation capability. Qubits are built using a spin-$\frac{1}{2}$ particle [Lloyd 1996], and any qubit or concatenation of qubits is a quantum system, and obeys the rules of quantum mechanics.

Providing confirmation for Feynman's Conjecture, quantum computers can be used to simulate the evolution of any quantum system that is consistent with special and general relativity [Lloyd 1996]. When a system is time-dependent, the vector $|\psi(t)\rangle$ that describes the system, is evolved by the repeated application of a Hamiltonian through Schrödinger's Equation.

$$\hat{H}\,|\psi(t)\rangle = i\hbar\frac{\mathrm{d}}{\mathrm{d}t}\,|\psi(t)\rangle \tag{4.1}$$

When studying a quantum system, the problem is centred around some observable property of the system. This is unitary operator that that describes the system in some way, for example the mass of the system, or amount of energy. The expected value of an observable $\hat{O}$ can be retrieved by

$$\langle\psi|\,\hat{O}\,|\psi\rangle = o \in \mathbb{R}$$

If the chosen basis of $|\psi\rangle$ is the eigenvectors of $\hat{O}$, then the expected value of a basis statevector is the corresponding eigenvalue.

Quantum computing works with the energy observable, the Hamiltonian $\hat{H}$, and as a result all statevectors are in the basis of the problem Hamiltonian, unless otherwise stated. The energy of a system E in some state is therefore

$$E = \langle\psi|\,\hat{H}\,|\psi\rangle$$

A quantum system may have two states that share an eigenvalue of an observable. When this occurs, the states are said to be degenerate, as they are indistinguishable from each other.

To use a quantum computer, any algorithm needs to be encoded into an operator that modifies an initial statevector. IBM's Qiskit makes use of a 'Quantum Circuit' to achieve this. A quantum circuit is similar to the classical boolean circuit, where it is composed of a series of wires and gates that modify some value. The difference between a classical circuit, and a quantum circuit is that the quantum circuit gates are unitary operators that modify a quantum system.

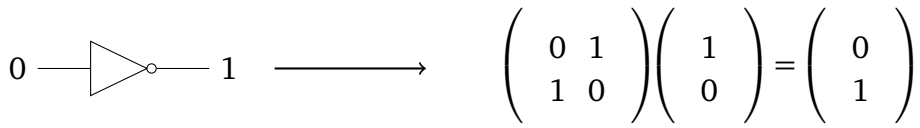


Figure 4.1: A not gate represented as matrix multiplication

The evolution of any quantum system $|\psi\rangle$ described by equation 4.1 can be encoded into a quantum circuit if the Hamiltonian can be decomposed into a linear combination of the quantum gates. The system itself is encoded into the qubits used by the circuit, so a quantum system of 3 particles has 3 qubits. The set of gates that can describe the Hamiltonian are the Pauli Operators,

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

the Identity matrix I, the Hadamard Operator

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

and several more [qis a].

These gates provide computational capability by first reproducing the gates of a classical circuit (σ_x is the quantum NOT gate, Figure 4.1) and then extending the capabilites to manipulate the superposition of the qubits. As an example, counting the number of spin-up states for any amount of qubits, the Hamiltonian $\hat{H} = \frac{1}{2}(I + \sigma_z)$ is applied to every qubit. Taking the expected value of $\hat{H}$ for any statevector $|\psi\rangle$ will be the number of zero bits, for example with two qubits

$$\hat{H} = \frac{1}{2}(I \otimes I - \sigma_z \otimes I)\frac{1}{2}(I \otimes I - I \otimes \sigma_z) = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

and the expected measurements of each state becomes

$$\langle 00| \hat{H} |00\rangle = 2, \langle 01| \hat{H} |01\rangle = 1$$
$$\langle 10| \hat{H} |10\rangle = 1, \langle 11| \hat{H} |11\rangle = 0$$

Circuits have their complexity measured in the number of gates required to execute the operation. This is related to the runtime complexity of a Turing Machine by a polynomial factor [Chi-Chih Yao 1993]. Therefore, if a circuit only requires a polynomial number of gates to compute the problem, then a quantum Turing Machine can compute the same problem in polynomial runtime.

With an understanding of how a quantum computer works, it is clear that they do offer some potential improvements at a theoretical scale. To understand the effects of these improvements on Ramsey Numbers, a discussion on Quantum Complexity with the Polynomial Hierarchy, and subsequently Ramsey Numbers, is required.

4.2 Quantum Complexity of Ramsey Numbers

The impact of quantum computers on computational complexity is still an open question [Bennett *et al.* 1997]. Most classical classes have a quantum equivalent, for example *NP* and Quantum-Merlin-Arthur *QMA*, *BPP* and *BQP*, but the actual relationship between them depends on the same open questions of classical complexity [Bernstein and Vazirani 1993]. There is evidence that a quantum computer is more powerful than a classical computer, *BPP* $\subseteq$ *BQP*, but nothing conclusive as *BPP* $\neq$ *BQP* requires the open question *P* vs *PSPACE*.

The complexity class of *QMA* is of interest because it generalizes classical *NP* into a quantum framework. The difference between the two is that *QMA* must accept an input string $|x\rangle$ with certificate $|\psi\rangle$ with $1-\varepsilon$ probability with $\varepsilon > 0$, instead of input string x with certificate w. Specifically, *NP* handles a classical input string, but *QMA* can accept any quantum state. Informally, a problem is in *QMA* if a quantum algorithm exists that can verify a solution in polynomial time with a high probability of correctness. *NP* is a subset of *QMA* as restricting $|x\rangle$ and $|\psi\rangle$ to only classical states is the definition of *NP*.

For Ramsey Numbers specifically, it is possible to reduce the problem to trying to find the state with the least energy Hamiltonian that only acts on k-local qubits [Gaitan and Clark 2012]. The specifics for this algorithm are covered in the next section 4.3.2, but for now, I will assume that this can be done in polynomial time. *QMA* is closed under karp-reducibility (See appendix B.2 for details), which means that if one problem can be converted into another problem, and the second problem is in *QMA*, then the first problem must be in *QMA*. The k-local Hamiltonian problem is in QMA [Kempe *et al.* 2005] and *CIS-RAMSEY* can be transformed into a k-local Hamiltonian problem in polynomial time (Section 4.3.2), thus

Thereom 2: *CIS-RAMSEY* $\in$ *QMA*

A number n can be verified as a Ramsey Number $R(k,l)$ in polynomial time on a quantum computer.

Theorem 2 is a piece of evidence that quantum computers are more powerful than classical computers as *CIS-RAMSEY* is in classical $\mathbf{\Pi}_2^P$, the level above *NP*, but in the quantum *NP*. However, this is not a direct proof of $\mathbf{\Pi}_2^P \subseteq$ *QMA*, and should be investigated more thoroughly as future work. Additionally, determining if *CIS-RAMSEY* is *QMA*-complete is the next step, but also as future work, possibly as part of the proof for $\mathbf{\Pi}_2^P \subseteq$ *QMA*.

With an understanding of how quantum computers work, it is now possible to design and analyse a quantum algorithm for computing Ramsey Numbers.

4.3 Quantum Algorithms

Implementing a quantum algorithm can be done in two ways. The first is to use a classical algorithm that exploits a feature of quantum mechanics like superposition to increase performance. The second is to reframe the problem into a quantum mechanics

framework, that can then more directly exploit the quantum features. In this section, I explore both these approaches, how they work and the complexity function of the algorithms.

4.3.1 The classical algorithms on a Quantum Computer

A quantum computer can implement any classical algorithm without modifications. It does this by restricting all of its operations to the quantum analogue of the classical instruction. At the very least, then, any classical problem remains unmodified if directly ported to the quantum device. This is very restrictive, and does not exploit any of the additional quantum power given to the device.

Some algorithms such as Grovers Search [Grover 1996] are able to exploit quantum behaviour and get some speedup over the classical counterpart. As Ramsey Numbers are a search problem, Grover's Search is an excellent candidate for improving the classical algorithms. However, using Grover's Search with its quadratic speedup, the search space goes from $O(2^{n^2})$ classically, to $O(2^{\frac{n^2}{2}})$ which is still bounded below by 2^n which is still exponential. Therefore, the use of a quantum computer with any of the classical algorithms does not improve the performance of *CIS-RAMSEY*, and another way of viewing the problem is required.

4.3.2 *CIS-RAMSEY* as an optimisation problem

In the classical section, all the algorithms approach Ramsey Numbers as a search problem, however it can also be modelled as an optimisation problem. By defining the cost function $\mathcal{C}(G, k, l)$ that counts the number of k-cliques and l-isets in G, the definition 2 for *UCR* and *LCR* can be modified to the following. *UCR* is defined so that any graph G with n vertices must have a non-zero value for $\mathcal{C}$ and, *LCR* is that there is at least one graph with n vertices with a zero value for $\mathcal{C}$.

$$UCR = \{\langle n, k, l\rangle : \forall G \text{ of order } n, \mathcal{C}(G, k, l) > 0\}$$
$$LCR = \{\langle n, k, l\rangle : \exists G \text{ of order } n, \mathcal{C}(G, k, l) = 0\}$$

These definitions can be read as all graphs G of order n must have a non-zero value for the function $\mathcal{C}$ for *UCR* and, *LCR* as there must be at least one graph G of

without changing *CIS-RAMSEY*. The algorithms for *CIS-RAMSEY*'s complexity remain unchanged, except where instead of evaluating if $K_k \subseteq G$ or $E_l \subseteq G$ using *SIP*, the algorithm evaluates the cost function $\mathcal{C}$.

A classical implementation of this cost function would be the counting problem *#SIP*, where instead of deciding if an instance of the subgraph exists, it would count the number of subgraphs that occur. Any counting problem is at least as hard as its equivalent decision problem [Valiant 1979], so the complexity of *CIS-RAMSEY* remains the same.

To get around this, it is possible to encode the value of $\mathcal{C}$ as a Hamiltonian constructed by Pauli operators. By using the same encoding scheme of a graph as in section 3.3.1, I

can model our graph as the quantum state vector $|\langle G\rangle\rangle$. Hence, to model a graph of n vertices, the quantum computer needs $n(n-1)/2$ qubits. Each graph is then a quantum state that is not in superposition. If the algorithms then encodes the value of $\mathcal{C}(G,k,l)$ into a Hamiltonian $\hat{H}$ such that

$$\hat{H}\,|\langle G\rangle\rangle = \mathcal{C}(G,k,l)\,|\langle G\rangle\rangle \tag{4.2}$$

the optimisation problem is now about finding the lowest energy (eigenvalue) of the Hamiltonian and its associated state (eigenvector).

Algorithm 6 Clique Hamiltonian encoding

function CLIQUE-HAMILTONIAN(n, k)
  $H^k \leftarrow 0 \cdot I$ ▷ I is the identity matrix
  for $i \leftarrow 0$ to $\binom{n}{k} - 1$ **do**
    $H_\alpha \leftarrow I$
    for $j \leftarrow 0$ to $\binom{k}{2} - 1$ **do**
      $H_\alpha \leftarrow H_\alpha \cdot \frac{1}{2}(I^e - \sigma_z^e)$
    $H^k \leftarrow H^k + H_\alpha$

The procedure to encode a clique into a Hamiltonian $\hat{H}^k$ is in algorithm 6 and the independent set Hamiltonian $\hat{H}^l$ is in algorithm 7. An example of each can be seen in Figure 4.2. The resultant Hamiltonian for k-cliques is

$$\hat{H}^k = \sum_{\alpha=0}^{\binom{n}{k}} \prod_{e\in E_\alpha} \frac{1}{2}(I^e - \sigma_z^e) \tag{4.3}$$

where E_α is the edges generated from the α-th choice of k vertices from graph G, σ_z^e is the Z pauli operator for qubit e corresponding to edge e in $\langle G\rangle$. An operator like σ_z^e actually looks like $I^{\otimes e} \otimes \sigma_z \otimes I^{n-e-1}$ where σ_z is acting only on the e-th qubit of the whole tensor product space of n-qubits.

For l-isets, the Hamiltonian is similar to $\hat{H}^k$ with

$$\hat{H}^l = \sum_{\alpha=0}^{\binom{n}{k}} \prod_{e\in E_\alpha} \frac{1}{2}(I^e + \sigma_z^e) \tag{4.4}$$

The runtime of both these algorithms is $O(\binom{n}{k})$ and $O(\binom{n}{l})$ for the clique and iset respectively, as there are $O(\binom{n}{k})$ and $O(\binom{n}{l})$ terms that need to be summed over. An $O(\binom{a}{b})$ runtime function is difficult to reason about, so it can be simplified to

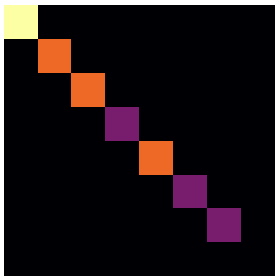
$$O(\binom{a}{b}) = O(\left(\frac{ea}{b}\right)^b)$$

where e is the base of the natural logarithm, which is a polynomial bound [Cormen *et al.* 2022, p1181]. Thus, the runtimes of *Clique-Hamiltonian* and *ISet-Hamiltonian* are polynomials.

Algorithm 7 Independent set Hamiltonian encoding

function ISET-HAMILTONIAN(n, l)
  $H^l \leftarrow 0 \cdot I$ ▷ I is the identity matrix
  for $i \leftarrow 0$ to $\binom{n}{l} - 1$ **do**
    $H_\alpha \leftarrow I$
    for $j \leftarrow 0$ to $\binom{l}{2} - 1$ **do**
      $H_\alpha \leftarrow H_\alpha \cdot \frac{1}{2}(I^e + \sigma_z^e)$
    $H^l \leftarrow H^l + H_\alpha$

(a) Hamiltonian of a 2-clique for 3 vertices



(b) Hamiltonian of a 2-iset for 3 vertices

Figure 4.2: Examples of the independent set and clique Hamiltonians for 3 vertices with $k = l = 2$ as a heatmap

Adding these two Hamiltonians together gives the Hamiltonian $\hat{H}^{kl}$. $\hat{H}^{kl}$ encodes the number of k-cliques and l-isets for a graph G as the eigenvalue for eigenstate $|\langle G \rangle\rangle$. In Figure 4.2 this encoding is visible, each column represents a unique graph, and as it is diagonal, the value in the diagonal corresponds to the energy of that graph i.e. the number of cliques or isets.

To compute *CIS-RAMSEY*, the last step required is to extract the groundstate energy of the Hamiltonian. If this state has 0 energy, then there exists at least one graph without either the clique or independent set, and n is in *LCR* else if the groundstate energy is greater than 0, then n is in *UCR*.

A point of interest is that if the Hamiltonian $\hat{H}^{kl}$ can be constructed in polynomial time, why can a classical algorithm not find the eigenvalue in polynomial time. There are two reasons this does not work on a classical computer.

The first reasons is that the construction of $\hat{H}^{kl}$ can be done in two ways, the first with a Pauli string representation [Li *et al.* 2021] where the operations are performed on the coefficients of the matrices, and the other with the actual matrices. Using the Pauli string, $\hat{H}^k$ can be done in $O(\binom{n}{k})$, but using the matrix representation will force it into $O(2^{n^2} \cdot \binom{n}{k})$. Hilbert spaces grow exponentially under the tensor product, so an operator over one state particle is in $\mathbb{C}^{2\times2}$, two state particles are in $\mathbb{C}^{4\times4}$, three is $\mathbb{C}^{8\times8}$ and so on. The encoding of n requires $O(n^2)$ qubits, therefore every matrix is in $\mathbb{C}^{2^{n^2}\times2^{n^2}}$. Constructing $\hat{H}^{kl}$ therefore requires matrix operations over $O(2^{n^2})$ elements, and is thus exponential.

The second reason is that eigensolvers require the matrix to computer over. Even if $\hat{H}^{kl}$ is built with a Pauli string representation, it would still need to be evaluated to the matrix i.e $O(2^{n^2})$ and then have the eigensolver applied. The standard classical eigensolvers work in $O(n^3)$ time [Press *et al.* 1992], where n is the size of the matrix. There are $O(2^{n^2})$ elements, thus the actual algorithm would be $O(2^{3n^2})$.

Any classical implementation of this optimisation problem would immediately be an *EXPTIME* algorithm that is worse than a brute force classical approach.

However, quantum computers are able to use the Pauli String encoding of a Hamiltonian [Li *et al.* 2021] and avoid the dimensionality explosion of the matrices by working directly in the Hilbert space. This makes them natural candidates for exploiting $\hat{H}^{kl}$'s polynomial time construction.

4.3.3 Adiabatic Quantum Optimisation

The first approach to finding the eigenvalues of a Hamiltonian is the Adiabatic Quantum Optimisation (AQO) algorithm [Farhi *et al.* 2000]. By the adiabatic theorem, if a system is slowly changed under specific conditions over time, then the state will remain in its initial state even through the change [Messiah 1981]. In practice, what this means is that if we slowly evolve a groundstate vector of system $\hat{H}_a$ to system $\hat{H}_b$, when it is fully described by $\hat{H}_b$, the statevector should be in the groundstate of $\hat{H}_b$.

The only condition for the adiabatic evolution to occur is that there are no degenerate groundstates. The runtime of the algorithm T is a specifiable parameter that indicates for how long to run the evolution, and T must be larger than the inverse of the energy gap g_{min} between the two lowest states, E_0 and E_1.

$$T \gg 1/g_{\text{min}} = 1/(E_1 - E_0) \tag{4.5}$$

If there is a degenerate groundstate, g_{min} becomes 0, and therefore the required T becomes unpredictable and effectively infinite. The specific details of this problem are an open question [Farhi *et al.* 2000]. The algorithm will still work with degenerate states, but there is a very high chance that the specified T is insufficient, and the adiabatic evolution is not finished.

To use this as a quantum algorithm, there are two required components: The initial Hamiltonian $\hat{H}_i$ with an easily identifiable groundstate to start in, and a problem Hamiltonian $\hat{H}_f$ that encodes the desired property of the problem.

For computing a Ramsey Number, the initial Hamiltonian is

$$\hat{H}_i = \sum_{l=0}^{n(n-1)/2-1} \frac{1}{2}(I^l - \sigma_x^l)$$

which has a ground state of

$$|\psi\rangle = \frac{1}{2^{n(n-1)/4}} \sum_{i=0}^{2^{n(n-1)}-1} |\text{bin}(i)\rangle$$

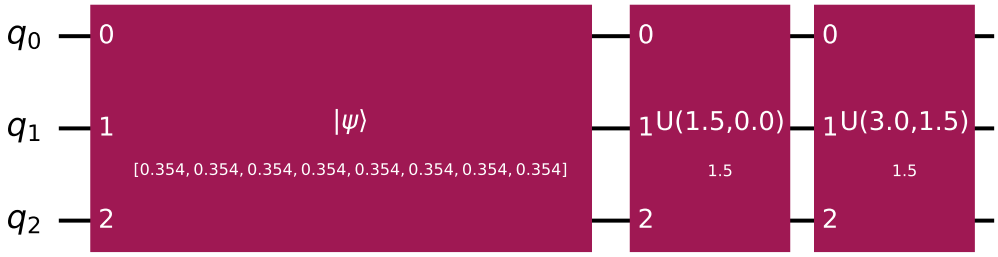


Figure 4.3: The simplified qiskit circuit implementation of a trotterized AQO on a 3-qubit system with $\Delta = 1.5$ and $T = 3$, i.e. for two discrete steps

where bin(i) is the binary representation of the number i, and therefore a graph G. The problem Hamiltonian $\hat{H}_f$ is

$$\hat{H}_f = \hat{H}^{kl} = \hat{H}^k + \hat{H}^l$$

where $\hat{H}^k$ and $\hat{H}^l$ are the Hamiltonians from *Clique-Hamiltonian*(n, k) and *ISet-Hamiltonian*(n, l) from the previous section. By parametrising $\hat{H}$ as $\hat{H}(t) = (1-\frac{t}{T})\cdot\hat{H}_i + \frac{t}{T}\cdot\hat{H}_f$, the change in the system is represented by moving t from 0 to T.

An asterisk attached to the AQO algorithm of Ramsey Numbers is that there is a very high chance that multiple graphs do not feature the desired subgraphs. This creates degenerate states of energy $E_0 = 0$, requiring large T values for finding these instances. When the groundstate is non-zero, i.e. no graph without either subgraph exists, there are degenerate states. These states are guaranteed to exist because there are two sources of energy that are complementary, $\hat{H}^k$ and $\hat{H}^l$. Every single possible state encodes one graph, so the all-zero state and empty graph will have some number of l-isets, and some other graph will have an equivalent number of k-cliques as edges are added to the graph.

The AQO algorithm is a continuous process, but it can be represented with discrete circuits through the Trotterization process. By discretising $\hat{H}(s)$ into distinct steps $\hat{H}^t_{t+\Delta}$ that progress t by a specified difference Δ, the continuous system evolution can be represented as the application of several unitary operators $U^{t+\Delta}$. The exact number of timesteps and circuit depth is determined by the application time T and the distance to increase t with each step (Figure 4.3). This circuit can then be decomposed into the primitive gates supported by the quantum computer (Figure 4.4).

To read the raw circuit, a line is a qubit through the time of the circuit, and a block with a symbol such as R_z or X represent a parametrized rotation of a single qubit around the subscript axis of its spin, or the labelled Pauli Operator. The gates with a symbol on one qubit, and then a coloured line to another qubit are controlled gates. The behaviour of a controlled gate require that the first qubit (the small dot) has some value before modifying the change on the second qubit. The controlled plus-symbol is a CNOT gate, it will flip the value of the second qubit around its $\hat{x}$ axis if the first qubit is in a superposition or $|1\rangle$ state.

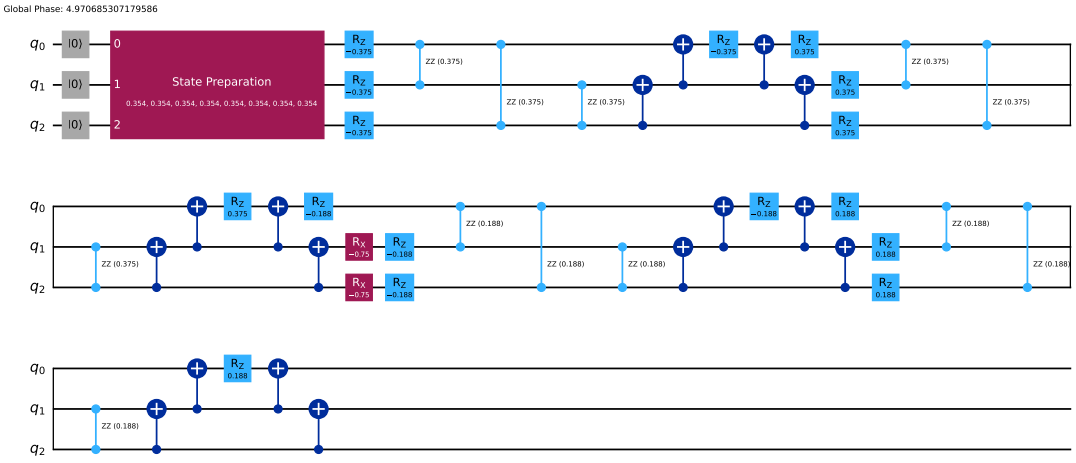


Figure 4.4: The decomposed circuit of figure 4.3 into primitive gate instructions.

The circuit complexity is $O(n^2)$ gates per $U^{t+\Delta}$ [Li *et al.* 2021] and there are $O(\frac{T}{\Delta})$ $U^{t+\Delta}$ gates. However, there are degenerate states in the problem Hamiltonian $\hat{H}_f$, so the exact number of $U^{t+\Delta}$ gates is unknown. This makes the circuit complexity infinite for correct computation. The circuit complexity is therefore

$$C(n) = \lim_{T\to\infty} O(\frac{T}{\Delta}n^2) \tag{4.6}$$

AQO is a quantum computation, but modern quantum computers have significant challenges with long computations and interference [Preskill 2018]. These challenges are typically keeping qubits stable for the whole duration of the computation, and are regularly experience interference from the outside environment effecting their actual state. New algorithms were developed that were a mix of quantum and classical steps to make Noisy Intermediate-Scale Quantum (NISQ) devices useful. One of the more successful algorithms has been the Variational Quantum Eigensolver to identify the eigenvalues of an observable property like energy with Hamiltonians.

4.3.4 Variational Quantum Eigensolvers

The Variational Quantum Eigensolver (VQE) is a mixed quantum-classical algorithm that can find the eigenvalues of an observable property [Peruzzo *et al.* 2014]. As Hamiltonians are the observable property of the energy in a quantum system, VQE can extract out the minimum eigenvalue of a Hamiltonian.

The decision problem of *CIS-RAMSEY* can be solved by finding the ground state energy of the Hamiltonian $\hat{H}_f = \hat{H}^k + \hat{H}^l$ and testing if it is zero. Thus, VQE can be applied to *CIS-RAMSEY*.

The VQE algorithm works by parametrising the statevector $|\psi\rangle$ with parameters $\vec{\theta}$ as $\left|\psi(\vec{\theta})\right\rangle$ and then optimising

$$\min_{\vec{\theta}} \left\langle \psi(\vec{\theta}) \right| \hat{H} \left| \psi(\vec{\theta}) \right\rangle$$

The evaluation of the cost function $\left\langle \psi(\vec{\theta}) \right| \hat{H} \left| \psi(\vec{\theta}) \right\rangle$ is performed on the quantum computer, and then optimised on the classical computer with Gradient Descent.

Gradient Descent requires $\nabla \hat{H}$ which can be found as $\hat{H}$ is the sum of pauli operators σ, hence

$$\nabla \hat{H} \left| \psi(\vec{\theta}) \right\rangle = \nabla \sum \sigma \left| \psi(\vec{\theta}) \right\rangle = \sum \nabla \sigma \left| \psi(\vec{\theta}) \right\rangle$$

Specifically for $\hat{H}_f$, the terms $\hat{H}^k$ and $\hat{H}^l$ from equations 4.3 and 4.4 are derivable so VQE can be applied to *CIS-RAMSEY*. The exact runtime of Gradient Descent is dependent on the chosen optimser, but all are in the optimisation equivalent of *NP*, *PPAD* [Fearnley *et al.* 2020].

To implement the evaluation on the quantum computer, a circuit is still required. However, pre-existing research suggests that very specific circuits can influence the correctness and speed of results. Therefore, by Qiskits recommendation, I use the EfficientSU2 circuit [East *et al.* 2023] with a polynomial complexity (Figure 4.5).

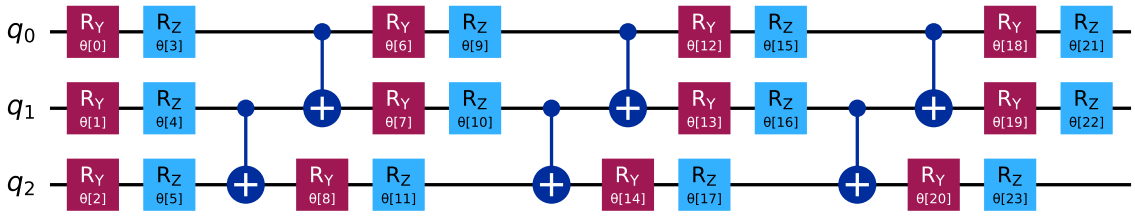


Figure 4.5: The EfficientSU(2) circuit for 3 qubits

The complexity of this algorithm is difficult to analyse. The mixed quantum-classical back and forth between optimising and measuring means that while the individual quantum and classical components may be analysed, the whole algorithm's complexity is unclear. This is made worse by the cloud computing nature of IBM's computers which means that a runtime analysis is effectively meaningless. The point of interest for this algorithm is not about how fast it works, instead it is if it can produce correct results where other algorithms might fail. A focus on the empirical results instead of the theoretical is therefore required for VQE.

4.4 Quantum Experiments

4.4.1 Methodology

Each quantum algorithm is first executed on the Quantum Aer Simulator for the cases in table 4.1. The simulator is used to tune and identify the set of hyperparameters used in the quantum computers execution that will lead to the most accurate results. If the algorithms are correct, they will then be run on an IBM Quantum Computer via the cloud with the exact same cases and hyperparameters. The exact chosen device is dependent on the availability of IBM's services but the list of potential computers is

- Heron - 133 available qubits
- Eagle - 127 available qubits

Quantum computers are noisy and probabilistic devices. A consequence of this is that any algorithm will produce results within some margin of error ε. For example, if a computation should result in 0, but it actually produces 0.1, this should still be considered a correct result. An important part of this however is that ε should not be overly large, and an error of $\varepsilon > 0.5$ should be disregarded as an incorrect result.

The computation of the AQO is dependent on the parameters T and Δ to correctly induce the adiabatic evolution. Before computing the correctness results of this algorithm, a 'tuning' step is required to identify the optimal performance of the algorithm. This tuning process will be performed on the $R(2, 4)$ test case. The potential T values are 1, 2 and, 3 with a Δ ranging from 0 to 1. For each pair of parameter values, the AQO circuit is run on $n = 3, 4, 5$ to correlate the performance of the algorithm at groundstate energy E_0 of 0 and greater than 0 (the two expected conditions that determine the result). The correctness test cases will be run with the best performing values of T and Δ.

If either AQO or VQE is correct on the quantum device as well as the simulator, then the runtime performance of the algorithms will be measured on the following test cases of $n \in [5, 10]$ and $k, l \in [3, 5]$. The runtime performance of these executions is the number of 'shots' i.e. how many execution attempts were needed to achieve the correct distribution and result. However, as the use of a quantum device is very costly and slow, this is a stretch goal to create a more ideal empirical comparison between the quantum algorithms and classical algorithms and not strictly necessary.

n	k	l	Expected Result
4	3	3	$E_0 = 0 + \varepsilon$
5	3	3	$E_0 = 0 + \varepsilon$
6	3	3	$E_0 > 0 + \varepsilon$
3	2	4	$E_0 = 0 + \varepsilon$
4	2	4	$E_0 > 0 + \varepsilon$
4	2	5	$E_0 = 0 + \varepsilon$
5	2	5	$E_0 > 0 + \varepsilon$

Table 4.1: Correctness test cases of the quantum algorithms

4.4.2 Results

The simulated results were computed on an i7-7700 with 24GB RAM, and the code is available at https://github.com/orwellian225/ramsey-quantum.

The tuning process of the AQO algorithm was not successful. In Figure 4.6, the performance of the algorithm was only able to achieve a successful result once for $T = 3$, regardless of Δ. In every other variable assignment and test case, the algorithm was off its expected value by at least 0.5, breaking the error limit. For $T = 1$ or 2, the algorithm

performs worse for larger Δ as expected, but for $T = 3$, the $n = 5$ case improves, and $n = 3, 4$ worsen.

For the correctness checking, although all tuning results were unsuccessful, the most successful case of $T = 3$ and $\Delta = 0.1$ was chosen.

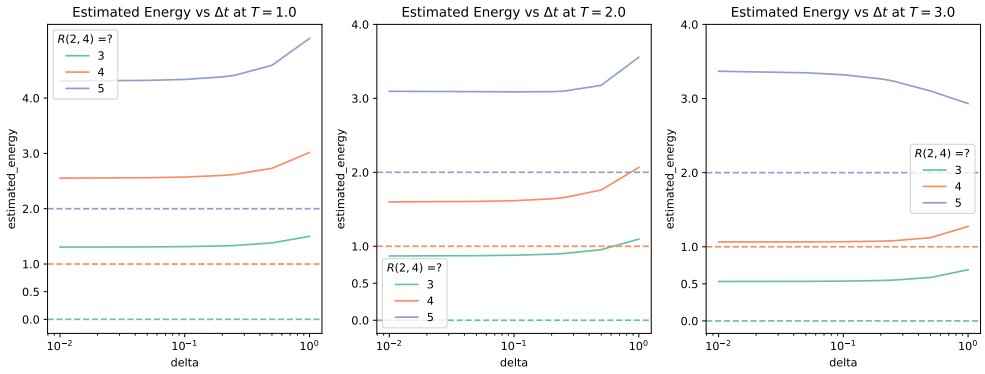


Figure 4.6: The results of tuning the adiabatic simulation on computing $R(2, 4)$. The solid line is the measured energy from the algorithm, and the dashed line is the expected energy. The expected energy was computed with a classical eigensolver for accuracy.

The AQO did not produce any successful *LCR* cases. Every single reading is greater than $0 + \varepsilon$ with $\varepsilon \leq 0.5$. Even if the ε limit is ignored, the second case of $(n, k, l) = (5, 3, 3)$ is still incorrect. Due to this, the algorithm was not executed on a noisy simulator, or the actual quantum computer.

n	k	l	Expected Result	AQO	VQE
4	3	3	$E_0 = 0 + \varepsilon$	0.75	0.02
5	3	3	$E_0 = 0 + \varepsilon$	2.24	0.72
6	3	3	$E_0 > 0 + \varepsilon$	4.79	2.89
3	2	4	$E_0 = 0 + \varepsilon$	0.54	0.00
4	2	4	$E_0 > 0 + \varepsilon$	1.07	1.01
4	2	5	$E_0 = 0 + \varepsilon$	0.59	0.06
5	2	5	$E_0 > 0 + \varepsilon$	1.09	1.30

Table 4.2: Correctness test results for simulating the quantum algorithms on a noiseless simulator

The VQE algorithm produced a correct result for every case except $(n, k, l) = (5, 3, 3)$ (Table 4.2). The introduction of noise into the simulator also did not change the results, as it still only failed on the $(5, 3, 3)$ case again.

There are no results for the VQE algorithm on a quantum computer. Most Qiskit based quantum computing make use of a system of libraries supported by IBM directly. The implementation of the VQE algorithm used, is in the Qiskit-Algorithms library. Recently this library has had its official support dropped and is maintained by the community. Recent changes and deprecations in Qiskit with the quantum computation primitives

n	k	l	Expected Result	AQO	VQE
4	3	3	$E_0 = 0 + \varepsilon$	/	0.02
5	3	3	$E_0 = 0 + \varepsilon$	/	1.09
6	3	3	$E_0 > 0 + \varepsilon$	/	2.70
3	2	4	$E_0 = 0 + \varepsilon$	/	0.00
4	2	4	$E_0 > 0 + \varepsilon$	/	1.02
4	2	5	$E_0 = 0 + \varepsilon$	/	0.00
5	2	5	$E_0 > 0 + \varepsilon$	/	1.26

Table 4.3: Correctness test results for simulating the quantum algorithms on with a noisy simulator. Algorithm I was not tested

has left the Qiskit-Algorithms library out of date for use on a quantum device. I did attempt to apply a custom patch and update the VQE implementation, but my access to the quantum computer was no longer available by the time it was implemented.

4.5 Analysis

The performance of both algorithms agree with the theoretical and previously established results for each algorithm respectively. The AQO algorithm has degenerate states, and thus is an open question to identify the needed runtime for the algorithm. This can be seen clearly in both the tuning and correctness step of the algorithm as it was never able to actually achieve correct results. It does get close, as it fails only on the $\varepsilon \leq 0.5$ restriction, as all the $E_0 = 0 + \varepsilon$ cases are just failing for all but one case. A modification to the algorithm that can better handle the degenerate states will likely lead to more accurate results [Wang 2016], but implementing and testing this improvement was not in the scope of this report.

*T*he VQE algorithm was very effective at getting results, failing only a single case on both the noiseless and noisy algorithm. VQE has been previously shown to be effective on NISQ computers [Peruzzo *et al.* 2014] at small scale, and this result is no different. While it is unfortunate that the VQE could not run on an actual quantum device, there is no reason to assume that it will fail.

The failure of both VQE and AQO on the case $(n, k, l) = (5, 3, 3)$ is very interesting. The failing of both algorithms on $R(3, 3) - 1$ suggests that there is either an issue in the encoding of $\hat{H}^{kl}$ or that there is a specific property in the $R(k, l) - 1$ case that makes it difficult. This same failing does not occur for $R(2, 4) - 1$ and $R(2, 5) - 1$ however, so there is minimal information to draw a good hypothesis from.

Of the two algorithms, the AQO is a potentially viable quantum algorithm that can be improved, but the VQE algorithm was able to achieve good results. The scalability of either algorithm remains to be seen, as neither were able to produce any results at large values of n. The open question of AQO's runtime with degenerate groundstates, and VQE's optimisation process This leaves two doors open for future work investigating the $R(3, 3) - 1$ anomaly and improving the VQE to potentially find a value for the next Ramsey Number on a quantum device.

An important note, however, is that the use of a quantum computer is expensive. The VQE execution took 4 minutes on a single test case that already have a known answer, at an average price of $\$48$ per minute, this computation cost approximately $200. Given the mixed quantum-classical (*QMA* and *PPAD*) nature of the algorithm, this makes the cost of the computation unpredictable and very expensive, at least on NISQ hardware.

Chapter 5

Classical vs Quantum Ramsey Numbers

A comparative analysis of quantum and classical algorithms solving the same problem is a difficult procedure. The theoretical analysis of quantum and classical problems do intersect, but in a very broad manner. Empirical analysis is even more difficult. The poor performance of quantum computers and their restricted access makes an empirical analysis after correctness tests effectively worthless. Quantum simulation is not an alternative either, as the simulation is itself expensive, which masks the runtime of the quantum algorithm.

Previous work to compare classical and quantum algorithms focus on comparing how well classical and quantum algorithms perform at solving a problem encoded in the same way [Ibarrondo *et al.* 2022]. For example, how well can a classical genetic algorithm solve a Hamiltonian optimisation problem compared to a quantum genetic algorithm. While this is useful, it has its shortcomings. The most obvious is that a Hamiltonian may not be the best encoding of the problem for the algorithm to solve, and if a more classical encoding of the problem was used, it may perform better. In parallel computation, comparing a parallel algorithm to a sequential algorithm requires that the sequential algorithm is not artificially limited in some way [Snyder and Lin 2011, Chapter 5]. Extending this idea, comparing a quantum algorithm and a classical algorithm solving the same problem should be compared at their best known performance.

As a result, I will make use of the following framework to perform the comparison: the membership and relationship of *CIS-RAMSEY*'s quantum and classical complexity classes, a comparison of the runtime complexity functions and lastly, contrasting the empirical results of each algorithm.

The purpose of this next section is to identify if a quantum or classical approach is more effective for computing a Ramsey Number, when considering the performance now at the time of writing, and any future changes created with improvements in quantum hardware.

5.1 Complexity of Ramsey Numbers

CIS-RAMSEY is in the classical class of $\mathbf{\Pi}_2^P$, and the quantum class of *QMA*. The exact relationship of these classes is unknown but, it is possible to gain an idea of their relationship. As *QMA* $\subseteq$ *PSPACE* [Bernstein and Vazirani 1993], and $\mathbf{\Pi}_2^P \subseteq$ *PSPACE* [Sipser 2012], there is a common property for *CIS-RAMSEY*. Regardless of whether using a Quantum or Classical Turing Machine, *CIS-RAMSEY* can be computed in a polynomial amount of space.

To compare the amount of time needed for *CIS-RAMSEY* independent of a Quantum or Classical Turing Machine, the relationship between *QMA* and $\mathbf{\Pi}_2^P$ needs to be established. A known relationship is *NP* $\subseteq$ *QMA* (Section 4.2), but this is not useful as *NP* $\subseteq \mathbf{\Sigma}_2^P$. It is possible that by using *QMA*, the Quantum Polynomial Hierarchy (QPH) can be defined in the same way the Classical Polynomial Hierarchy (CPH) is to *NP* [Gharibian *et al.* 2022]. By restricting the second-level of the quantum polynomial hierarchy to classical inputs only, the second level of the CPH becomes a subset of the second level of the QPH.

$$\mathbf{\Sigma}_2^P = NP^{NP} \subseteq QMA^{QMA}$$
$$\mathbf{\Pi}_2^P = \text{co}NP^{NP} \subseteq \text{co}QMA^{QMA}$$

However, this does not properly provide a bound on $\mathbf{\Pi}_2^P$ and *QMA*, as it is between co*NP* and co*QMA*^{QMA}. Either $\mathbf{\Pi}_2^P \subseteq$ *QMA* or *QMA* $\subseteq \mathbf{\Sigma}_2^P \cup \mathbf{\Pi}_2^P$, so determining if the quantum *CIS-RAMSEY* problem is solvable faster than the classical *CIS-RAMSEY* remains unknown.

5.2 Runtime Complexities

The runtime function of the Linear (Algorithm 3) and Binary-Tree (Algorithm 4) are both

$$O(2^{n^2} \cdot \binom{n}{\max(k,l)} \cdot \max(k,l)^2)$$

from equations (3.2) and (3.3), and the Reduced-Binary-Tree (Algorithm 5) is equation (3.5)

$$O(n2^{n^2} \cdot \binom{n}{\max(k,l)} \cdot \max(k,l)^2)$$

The AQO quantum algorithm has a circuit complexity function of

$$C(n) = \lim_{T\to\infty} O(\frac{T}{\Delta} n^2)$$

which becomes

$$T(n) = \lim_{T\to\infty} O(\text{poly}(\frac{T}{\Delta} n^2))$$

as circuit complexity is related to time complexity by a polynomial factor [Chi-Chih Yao 1993]. The VQE algorithm has no specific complexity function.

Comparing just the runtime functions, the AQO algorithm with its quadratic complexity is the best performing algorithm. Factoring in the $\lim_{T\to\infty}$ part of the function however makes the AQO unreliable for an actual computation.

Instead, the best theoretical algorithm is the Reduced-Binary-Tree. Between the classical algorithms, the Reduced-Binary-Tree has the lowest runtime complexity, although the n factor of the term may make it appear to be larger. Comparing the Reduced-Binary-Tree to the VQE, the VQE does have the potential to reach a specific instance of the problem faster through the optimisation process, there will not hold true for every case of the problem. Hence, the Reduced-Binary-Tree is able to more consistently and predictably test for a Ramsey Number.

The Reduced-Binary-Tree algorithm is the best algorithm theoretically, but this results is not always the same in an empirical study. To fully understand whether the classical or quantum approach is better, a comparison of their empirical results is required.

5.3 Empirical Comparison

As a runtime experiment for the quantum algorithms could not be performed, the empirical comparison is limited to the correctness tests for each algorithm. This comparison will therefore identify which of the algorithms is the most consistently correct, and how verifiable the correct result is.

Comparing the result of an algorithm's execution, the classical algorithms return an actual graph G that, with enough patience for large n, can be manually verified. The use of external subgraph searching software like the Glasgow Solver [McCreesh *et al.* 2020] can also be used to audit the results and independently search for the subgraphs K_k or E_l in the returned graph. The quantum algorithms only produce an energy reading value, which is less verifiable. It is possible to retrieve a potential graph G by measuring all the qubits of a circuit, but this produces probabilistic results and requires repeated execution of the circuit.

The quantum algorithms also produce a result with an error ε that could leave the result up for debate. For example, in table 4.2, the results of the ideal simulated execution for case $(n, k, l) = (4, 2, 5)$ is 0.06 for the VQE algorithm, and 0.59 for the AQO. It is reasonable to say the VQE algorithm gives a correct result, as ε is only 0.06 but the AQO algorithm would need ε at 0.6 to be correct.

The consistent correctness and the verifiably correct results suggest that the classical algorithm is a better approach to computing a new Ramsey Number. Additionally, although the runtime of the classical algorithm is large, it is predictable. The runtime of the quantum algorithms cannot be properly demonstrated due to simulations, the cost of running the algorithm on quantum hardware, and the general issues of quantum hardware. This makes the classical algorithms more useful than the quantum algorithms as they better demonstrate the difficulty of the problem, and make it easier to understand how an improvement can affect the computation in the future.

Chapter 6

Conclusion

The computation of Ramsey Numbers is a difficult task. On a classical computer, Ramsey Numbers are harder than *NP* problems, in the second level of the Polynomial Hierarchy $\mathbf{\Pi}_2^P$ specifically. The quantum computer does seemingly offer additional power in the computation, bringing Ramsey Numbers into the quantum *NP*, *QMA*. The exact relationship between *QMA* and $\mathbf{\Pi}_2^P$ is not known, and therefore makes it difficult to know if the quantum computer does theoretically improve the computation.

Implementing an algorithm for Ramsey Numbers reinforces the difficulty of the problem. A non-trivially optimized classical algorithm for solving Ramsey Numbers will take 10^{134} years to reach a new answer, which is more time than it takes for a supermassive black hole to decay (10^{116} years) [Frautschi 1982]. There is still room for optimisation in the classical algorithms, by using search heuristics or implementing a more recent algorithm for subgraph searching, and it can still be parallalized. However, the incredibly large runtime of the classical algorithms do suggest that improvements will help, but not in a way to bring the computation away from the length of black hole decay to the more reasonable result of a human lifetime. The amount of time did not improve on a quantum computer either, as they run slower and produce correct results less reliably.

While the quantum algorithms can produce correct results, it comes at significant cost. To produce a correct result, the algorithm needs to run multiple times, and the results averaged. Every run of the device costs a significant amount of money, and while the access to these computers is low and their price high, they remain an even less effective approach to computing a Ramsey Number. Even if the hardware improved, the most successful quantum algorithm was a mixed classical-quantum approach. The full quantum algorithm requires more research and solving an open question to get working correctly.

Comparing the quantum and classical algorithms did not help show which algorithm is better. Identifying if the quantum computer improves the computation performance over a classical computer at a theoretical level is dependent on the relationship of *QMA* to $\mathbf{\Pi}_2^P$, but this is still unexplored. For implementing the algorithms, the challenges faced by NISQ quantum computers, and the unknown behaviour of degenerate states

mean that a classical computer is better than a quantum computer for Ramsey Numbers as of 2024.

The aim of this report was to establish if a Ramsey Number can be computed in a reasonable amount of time on either a classical or quantum computer. The answer is therefore no, Ramsey Numbers are not reasonably computable on either device. Even under ideal circumstances with parallelization or unlimited access to a quantum device, neither algorithm is able to produce an answer for the next unknown Ramsey Number $R(5, 5)$ within 50 years.

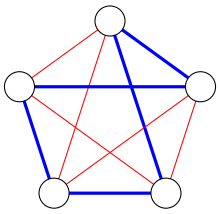
Appendix A

Ramsey Number Proofs

A.1 The Party Problem: $R(K_3, E_3) = 6$

The party problem is the colloquial name for the $R(3,3)$ Ramsey Number. The standard formulation is how many people must attend a party before at least three people all know each other, or three people have never met.

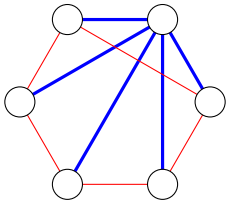
To prove $R(3,3) = 6$, it must first be show that $R(3,3) > 5$, and this can be done with the below coloring of the K_5 graph:



This coloring does not have a blue K_3 subgraph nor does it have a red K_3 which satisfies the lowerbound of $R(3,3)$.

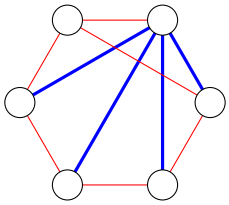
Showing $R(3,3) \geq 6$ can be done with the following contradictions: Assume that there exists a 2-coloring c of K_6 that has neither a red K_3 or blue K_3

Case 1: A vertex v has no red edges



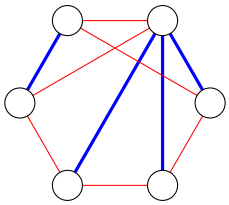
If you draw either a blue edge or red edge from any vertex to any other vertex, the edge will form a blue K_3 or red K_3. This contradicts the assumption.

Case 2: A vertex v has one red edges



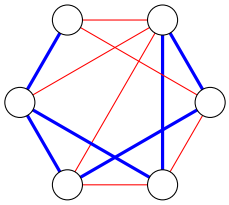
If you draw either a blue edge or red edge from any vertex to any other vertex, the edge will form a blue K_3 or red K_3. This contradicts the assumption.

Case 3: A vertex v has two red edges



If you draw either a blue edge or red edge from any vertex to any other vertex, the edge will form a blue K_3 or red K_3. This contradicts the assumption.

Case 3: A vertex v has three red edges



If you draw either a blue edge or red edge from any vertex to any other vertex, the edge will form a blue K_3 or red K_3. This contradicts the assumption.

The blue and red colorings are complimentary, therefore a vertex with n edges, i of which are red, has $n - i$ blue edges. This means that the above cases hold if you invert every color from red to blue, and this will still produce a contradiction for all cases of a vertex with 0, 1, 2, 3, 4, or 5 red edges. Therefore, by proof from contradiction and cases, every 2-coloring of K_6 will have a red K_3 or blue K_3.

Every 2-coloring of a K_n can also represent an uncolored graph of order n, thus every graph with 6 vertices either has a 3-clique or 3-iset.

A.2 $R(K_2, E_m) = m$

Let G be a complete graph of order n

Let $n < m$,
let c be a 2-coloring where for every edge $e_{ij} \in E(G), c(e_{ij}) = 2$
$\implies G = K_n$ and has all blue edges
$\implies K_m \nsubseteq G$
$\therefore \forall n < m, \exists c$ that has no red K_2 and no blue K_m
$\therefore R(K_2, K_m) > m - 1$

Let $n \geq m$,
Case I: let c be 2-coloring where for every edge $e_{ij} \in E(G), c(e_{ij}) = 2$
$\implies G = K_n$and has all blue edges
$\implies$ a blue $K_m \subseteq G$
Case II: let c be a 2-coloring where for at least one edge $e_{kl} \in E(G), c(e_{kl}) = 1$ and all other edges $e_{ij} \in E(G), c(e_{ij}) = 2$
$\implies$ Graph $(\{v_k, v_l\}, \{e_{kl}\}) = K_2$ and all edges are red
$\implies$ a red $K_2 \subseteq G$
$\therefore \forall n \geq m, \forall c$ there is either a red $K_2 \subseteq G$ or a blue $K_m \subseteq G)$
$\therefore R(K_2, K_m) \leq m$

$R(K_2, K_m) > m - 1$ and $R(K_2, K_m) \leq m$, implies that $R(K_2, K_m) = m$

Appendix B

Complexity Proofs

B.1 *P* and *EXPTIME* are closed for complementary languages

A complexity class $\mathcal{C}$ is defined as closed under complementary languages if for every language L in $\mathcal{C}$, its complement $\bar{L}$ is also in $\mathcal{C}$.

Thereom 3: The class P is closed for complementary languages

Let L be a language with a TM M with strings $\langle x \rangle$ of length n, that runs in time $T_M(n)$ that is polynomial function. By definition of P, $L \in P$.

Define a TM that decides $\bar{L}$,

define TM N on input $\langle\langle x \rangle\rangle$

1: Simulate $M(\langle x \rangle)$
If $M(\langle x \rangle) = 1$, return 0, else return 1

TM N simulates M such that

$$N(\langle x \rangle) = 1 \iff M(\langle x \rangle) = 0 \iff \langle x \rangle \notin L$$

and

$$N(\langle x \rangle) = 0 \iff M(\langle x \rangle) = 1 \iff \langle x \rangle \in L$$

thus deciding $\bar{L}$.

The runtime of N is the runtime of M and the step to invert the answer. Thus N decides $\bar{L}$ in the polynomial runtime function $T_N(n) = T_M(n)$ and therefore $\bar{L} \in P$. Therefore, P is closed for complementary languages.

Without loss of generality, *EXPTIME* is closed for complimentary languages.

B.2 *QMA* is closed under Karp-Reducibility

A complexity class $\mathcal{C}$ is defined as closed under Karp-reducibility ($\leq_m^p$) if for all languages A_1 and A_2 whenever $A_1 \leq_m^p A_2$ and $A_2 \in \mathcal{C}$, then $A_1 \in \mathcal{C}$

Thereom 4:

Let A be a language in *QMA*, and B as a language such that $B \leq_m^p A$. There exists a Quantum Turing Machine N on input $|y\rangle$ that decides if $|y\rangle \in A$ in polytime with probability greater than $\geq \frac{3}{4}$

$B \leq_m^p A \implies$ there exists a polytime computable function f such that $|x\rangle \in B \implies f(|x\rangle) \in A$

Construct a Quanutm Turing Machine M that non-deterministically decides $|x\rangle \in B$ in polytime with probability greather than $\geq \frac{3}{4}$

define TM M on input $\langle |x\rangle, |c\rangle \rangle$

1: Compute $|y\rangle = f(|x\rangle)$
2: Simulate N on input $|y\rangle$
If $N(y) = 1$ then return 1 else return 0

Step 1 is polytime by assumption, and the computation of N is non-deterministically polytime with probability $\geq \frac{3}{4}$, therefore M decides $|x\rangle \in B$ in non-deterministic polynomial time with probability $\geq \frac{3}{4}$.

QMA is closed under Karp-Reducibility.

References

[Bennett *et al.* 1997] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and weaknesses of quantum computing. *SIAM Journal on Computing*, 26(5):1510–1523, 1997.

[Bernstein and Vazirani 1993] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 11–20, New York, NY, USA, 1993. Association for Computing Machinery.

[Burr 1987] Stefan A Burr. What can we hope to accomplish in generalized ramsey theory? *Discrete Mathematics*, 67(3):215–225, 1987.

[Chi-Chih Yao 1993] A. Chi-Chih Yao. Quantum circuit complexity. In *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pages 352–361, 1993.

[Cook 1971] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[Cormen *et al.* 2022] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, fourth edition*. MIT Press, 2022.

[Deutsh 1989] David Elieser Deutsh. Quantum computational networks. *Proceedings of the Royal Society*, 425(1868):73–90, 1989.

[East *et al.* 2023] Richard D. P. East, Guillermo Alonso-Linaje, and Chae-Yeun Park. *All you need is spin: SU(2) equivariant variational quantum circuits based on spin networks*, 2023.

[Farhi *et al.* 2000] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. *Quantum Computation by Adiabatic Evolution*, 2000.

[Fearnley *et al.* 2020] John Fearnley, Paul W. Goldberg, Alexandros Hollender, and Rahul Savani. The complexity of gradient descent: CLS = PPAD $\cap$ PLS. *CoRR*, abs/2011.01929, 2020.

[Frautschi 1982] Steven Frautschi. Entropy in an expanding universe. *Science*, 217(4560):593–599, 1982.

[Gaitan and Clark 2012] Frank Gaitan and Lane Clark. Ramsey numbers and adiabatic quantum computing. *Phys. Rev. Lett.*, 108:010501, Jan 2012.

[Gharibian *et al.* 2022] Sevag Gharibian, Miklos Santha, Jamie Sikora, Aarthi Sundaram, and Justin Yirka. Quantum generalizations of the polynomial hierarchy with applications to qma(2). *computational complexity*, 31(2):13, Sep 2022.

[Gonthier and others 2008] Georges Gonthier et al. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[Grover 1996] Lov K. Grover. *A fast quantum mechanical algorithm for database search*, 1996.

[Harary and Palmer 1973] F. Harary and E.M. Palmer. *Graphical Enumeration*. Academic Press, 1973.

[Hattingh *et al.* 2023] Johannes H. Hattingh, Elizabeth Jonck, and Ronald J. Maartens. The k -ramsey number of two five cycles. *AKCE Int. J. Graphs Comb.*, 21:57–62, 2023.

[Ibarrondo *et al.* 2022] Rubén Ibarrondo, Giancarlo Gatti, and Mikel Sanz. Quantum vs classical genetic algorithms: A numerical comparison shows faster convergence. In *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 947–954, 2022.

[Kempe *et al.* 2005] Julia Kempe, Alexei Kitaev, and Oded Regev. *The Complexity of the Local Hamiltonian Problem*, 2005.

[Knuth 2000] Donald E. Knuth. *Dancing links*, 2000.

[Li *et al.* 2021] Gushu Li, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, and Yuan Xie. *Paulihedral: A Generalized Block-Wise Compiler Optimization Framework For Quantum Simulation Kernels*, 2021.

[Lloyd 1996] Seth Lloyd. Universal quantum simulators. *Science*, 273(5278):1073–1078, 1996.

[McCreesh *et al.* 2020] Ciaran McCreesh, Patrick Prosser, and James Trimble. The glasgow subgraph solver: Using constraint programming to tackle hard subgraph isomorphism problem variants. In Fabio Gadducci and Timo Kehrer, editors, *Graph Transformation*, pages 316–324, Cham, 2020. Springer International Publishing.

[Messiah 1981] A. M. L. Messiah. Quantum mechanics, volume ii. 1981.

[Oh *et al.* 2024] Changhun Oh, Minzhao Liu, Yuri Alexeev, Bill Fefferman, and Liang Jiang. Classical algorithm for simulating experimental gaussian boson sampling. *Nature Physics*, 20(9):1461–1468, Sep 2024.

[Peruzzo *et al.* 2014] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5(1), July 2014.

[Planck Collaboration *et al.* 2016] Planck Collaboration, Ade, P. A. R., Aghanim, N., Arnaud, M., Ashdown, M., Aumont, J., Baccigalupi, C., Banday, A. J., Barreiro, R. B., Bartlett, J. G., Bartolo, N., Battaner, E., Battye, R., Benabed, K., Benoît, A.,

Benoit-Lévy, A., Bernard, J.-P., Bersanelli, M., Bielewicz, P., Bock, J. J., Bonaldi, A., Bonavera, L., Bond, J. R., Borrill, J., Bouchet, F. R., Boulanger, F., Bucher, M., Burigana, C., Butler, R. C., Calabrese, E., Cardoso, J.-F., Catalano, A., Challinor, A., Chamballu, A., Chary, R.-R., Chiang, H. C., Chluba, J., Christensen, P. R., Church, S., Clements, D. L., Colombi, S., Colombo, L. P. L., Combet, C., Coulais, A., Crill, B. P., Curto, A., Cuttaia, F., Danese, L., Davies, R. D., Davis, R. J., de Bernardis, P., de Rosa, A., de Zotti, G., Delabrouille, J., Désert, F.-X., Di Valentino, E., Dickinson, C., Diego, J. M., Dolag, K., Dole, H., Donzelli, S., Doré, O., Douspis, M., Ducout, A., Dunkley, J., Dupac, X., Efstathiou, G., Elsner, F., Enßlin, T. A., Eriksen, H. K., Farhang, M., Fergusson, J., Finelli, F., Forni, O., Frailis, M., Fraisse, A. A., Franceschi, E., Frejsel, A., Galeotta, S., Galli, S., Ganga, K., Gauthier, C., Gerbino, M., Ghosh, T., Giard, M., Giraud-Héraud, Y., Giusarma, E., Gjerløw, E., González-Nuevo, J., Górski, K. M., Gratton, S., Gregorio, A., Gruppuso, A., Gudmundsson, J. E., Hamann, J., Hansen, F. K., Hanson, D., Harrison, D. L., Helou, G., Henrot-Versillé, S., Hernández-Monteagudo, C., Herranz, D., Hildebrandt, S. R., Hivon, E., Hobson, M., Holmes, W. A., Hornstrup, A., Hovest, W., Huang, Z., Huffenberger, K. M., Hurier, G., Jaffe, A. H., Jaffe, T. R., Jones, W. C., Juvela, M., Keihänen, E., Keskitalo, R., Kisner, T. S., Kneissl, R., Knoche, J., Knox, L., Kunz, M., Kurki-Suonio, H., Lagache, G., Lähteenmäki, A., Lamarre, J.-M., Lasenby, A., Lattanzi, M., Lawrence, C. R., Leahy, J. P., Leonardi, R., Lesgourgues, J., Levrier, F., Lewis, A., Liguori, M., Lilje, P. B., Linden-Vørnle, M., López-Caniego, M., Lubin, P. M., Macías-Pérez, J. F., Maggio, G., Maino, D., Mandolesi, N., Mangilli, A., Marchini, A., Maris, M., Martin, P. G., Martinelli, M., Martínez-González, E., Masi, S., Matarrese, S., McGehee, P., Meinhold, P. R., Melchiorri, A., Melin, J.-B., Mendes, L., Mennella, A., Migliaccio, M., Millea, M., Mitra, S., Miville-Deschênes, M.-A., Moneti, A., Montier, L., Morgante, G., Mortlock, D., Moss, A., Munshi, D., Murphy, J. A., Naselsky, P., Nati, F., Natoli, P., Netterfield, C. B., Nørgaard-Nielsen, H. U., Noviello, F., Novikov, D., Novikov, I., Oxborrow, C. A., Paci, F., Pagano, L., Pajot, F., Paladini, R., Paoletti, D., Partridge, B., Pasian, F., Patanchon, G., Pearson, T. J., Perdereau, O., Perotto, L., Perrotta, F., Pettorino, V., Piacentini, F., Piat, M., Pierpaoli, E., Pietrobon, D., Plaszczynski, S., Pointecouteau, E., Polenta, G., Popa, L., Pratt, G. W., Prézeau, G., Prunet, S., Puget, J.-L., Rachen, J. P., Reach, W. T., Rebolo, R., Reinecke, M., Remazeilles, M., Renault, C., Renzi, A., Ristorcelli, I., Rocha, G., Rosset, C., Rossetti, M., Roudier, G., Rouillé d'Orfeuil, B., Rowan-Robinson, M., Rubiño-Martín, J. A., Rusholme, B., Said, N., Salvatelli, V., Salvati, L., Sandri, M., Santos, D., Savelainen, M., Savini, G., Scott, D., Seiffert, M. D., Serra, P., Shellard, E. P. S., Spencer, L. D., Spinelli, M., Stolyarov, V., Stompor, R., Sudiwala, R., Sunyaev, R., Sutton, D., Suur-Uski, A.-S., Sygnet, J.-F., Tauber, J. A., Terenzi, L., Toffolatti, L., Tomasi, M., Tristram, M., Trombetti, T., Tucci, M., Tuovinen, J., Türler, M., Umana, G., Valenziano, L., Valiviita, J., Van Tent, F., Vielva, P., Villa, F., Wade, L. A., Wandelt, B. D., Wehus, I. K., White, M., White, S. D. M., Wilkinson, A., Yvon, D., Zacchei, A., and Zonca, A. Planck 2015 results - xiii. cosmological parameters. *AA*, 594:A13, 2016.

[Preskill 2018] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, August 2018.

[Press *et al.* 1992] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Numerical Recipes in C book set. Cambridge University Press, 1992.

[qis a] *Circuit Library*. Accessed: 2024-11-09.

[qis b] *You need 100 qubits to accelerate discovery with quantum*. Accessed: 2024-11-09.

[Radziszowski 2021] Stanisław P. Radziszowski. Small ramsey numbers. *Electronic Journal of Combinatorics*, 2021. [Online; accessed 8-May-2024].

[Ramsey 1930] F. P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, s2-30(1):264–286, 1930.

[Roberts 1984] Fred S. Roberts. Applications of ramsey theory. *Discrete Applied Mathematics*, 9(3):251–261, 1984.

[Rosta 2004] Vera Rosta. Ramsey theory applications. *Electronic Journal of Combinatorics*, 2004.

[Shor 1994] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[Sipser 2012] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[Snyder and Lin 2011] L. Snyder and C. Lin. *Principles of Parallel Programming*. Pearson Education, 2011.

[Ullmann 1976] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[Valiant 1979] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[Wang 2016] Hefeng Wang. Determining ramsey numbers on a quantum computer. *Phys. Rev. A*, 93:032301, Mar 2016.

[Willsch *et al.* 2022] Dennis Willsch, Madita Willsch, Carlos D. Gonzalez Calaza, Fengping Jin, Hans De Raedt, Marika Svensson, and Kristel Michielsen. Benchmarking advantage and d-wave 2000q quantum annealers with exact cover problems. *Quantum Information Processing*, 21(4):141, Apr 2022.

[Wolfram 2020] Stephen Wolfram. A class of models with the potential to represent fundamental physics. *Complex Systems*, 29(2):107–536, June 2020.

[Zhu *et al.* 2022] Qingling Zhu, Sirui Cao, Fusheng Chen, Ming-Cheng Chen, Xiawei Chen, Tung-Hsun Chung, Hui Deng, Yajie Du, Daojin Fan, Ming Gong, Cheng Guo, Chu Guo, Shaojun Guo, Lianchen Han, Linyin Hong, He-Liang Huang, Yong-Heng Huo, Liping Li, Na Li, Shaowei Li, Yuan Li, Futian Liang, Chun Lin, Jin Lin, Haoran Qian, Dan Qiao, Hao Rong, Hong Su, Lihua Sun, Liangyuan Wang, Shiyu Wang, Dachao Wu, Yulin Wu, Yu Xu, Kai Yan, Weifeng Yang, Yang Yang, Yangsen

Ye, Jianghan Yin, Chong Ying, Jiale Yu, Chen Zha, Cha Zhang, Haibin Zhang, Kaili Zhang, Yiming Zhang, Han Zhao, Youwei Zhao, Liang Zhou, Chao-Yang Lu, Cheng-Zhi Peng, Xiaobo Zhu, and Jian-Wei Pan. Quantum computational advantage via 60-qubit 24-cycle random circuit sampling. *Science Bulletin*, 67(3):240–245, 2022.