Universiteit Leiden

Master Computer Science

Quantum Speedups for Monte Carlo Tree Search

Name:	Orson R. L. Peters
Student ID:	s1412728
Date:	02/08/2021
Specialisation:	Artificial Intelligence
1st supervisor:	Vedran Dunjko
2nd supervisor:	David Elkouss
3rd supervisor:	Casper Gyurik

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Quantum speedups for Monte Carlo Tree Search

Orson R. L. Peters

August 2, 2021

Abstract

Monte Carlo Tree Search (MCTS) is a stochastic planning algorithm that can provide suggestions for moves in two-player games without the need for a heuristic. In this work we describe a quantum algorithm to speed up the stochastic 'random rollout' step performed in a variant of MCTS that does multiple such rollouts. Another quantum algorithm is introduced that speeds up the computation of an ensemble of MCTS instances each ran for k iterations. As a corollary of the techniques developed, a quantum algorithm is presented for estimating the expected value or maximizing first step of an arbitrary (stochastic) fixed-length policy-guided walk in an arbitrary (stochastic) environment. This walk is defined by an initial state, a policy function and a transition function, with a value assigned to such a walk by an arbitrary evaluation function defined on the complete path taken. All speedups found are quadratic with respect to the best known classical algorithm.

Contents

Chapter 1

Introduction

Interest in Monte Carlo Tree Search (MCTS) has dramatically increased after being successfully applied to the game of Go with later variations using Deep Learning methods such as AlphaGo [Sil+16], AlphaGo Zero [Sil+17b] and AlphaZero [Sil+17a] [Sil+18] finally surpassing the best humans in an extraordinary feat of artificial intelligence. Similarly quantum computers are gathering a lot of attention for achieving runtimes to problems often quadratically but even exponentially faster than the best known classical algorithms. It is thus natural to ask oneself if quantum computers can be used to speed up (variants of) MCTS.

In this master thesis we dive into quantum computing, and look in detail at the inner workings of MCTS. From this we form a plan of attack and attempt to speed up (portions of) MCTS using quantum computation techniques. We study and analyze the constructed quantum algorithms to see if and how the resulting techniques speed up MCTS.

1.1 Related work

As the topic of this thesis implies, any literature on fundamental quantum computation or Monte Carlo Tree search can be considered related. Our efforts to list related work would pale in comparison to that of a seasoned domain expert, thus we would recommend reading respectively *Quantum Computation and Quantum Information* by Nielsen & Chuang [NC11] or *A Survey of Monte Carlo Tree Search Methods* by Browne et. al. [Bro+12a] instead.

To our knowledge no direct prior related work exists—that is an application of quantum algorithms or methods to speed up Monte Carlo Tree Search or parts thereof. Theorem 8.1 and its proposed applications in Chapter 8 are closely related to the work by Arjan Cornelissen in his master thesis [Cor18] in which he describes a quantum method for quickly estimating the mean reward of a discounted Markov decision process (MDP). However, our method is subtly different and our scope is different (fixed-length walks rather than a discount factor and allowing arbitrary path reward functions rather than traditional MDP step rewards), which allows for different applications.

Finally, while we are not aware of direct prior related work we must mention that the application of quantum algorithms to the optimal *minimax* solution of two-player games (often described as AND-OR or NAND trees) is well-studied [Chi+07] [FGG08] [Rei11] [Mon16] and prior work on finite-horizon dynamic programming similar to MCTS also exists [Amb+18].

1.2 Contributions

In Chapter 6 we give two different quantum algorithms for quickly estimating the winrate of a random rollout as performed in MCTS. The first method embeds a game tree into a larger tree where each node has 2^k children and then defines a clever value function on the leaf nodes in this embedding such that the function is well-behaved and its mean value over the leaf nodes is related to the random rollout winrate w by a known constant factor. The embedding and value function are both efficiently implemented as a quantum circuit after which the mean can be evaluated using quantum mean estimation in $O\left(\frac{\log 1/\delta}{\epsilon\sqrt{w}}\right)$ time with relative error ϵ and failure probability δ. The second method achieves the same asymptotic complexity and also uses quantum mean estimation to estimate w, however instead of using the previously mentioned embedding it directly prepares a superposition over the leafs of the original game tree where the squared amplitude of a leaf matches the probability the random rollout reaches that leaf.

In Chapter 7 we take a different approach and use a black-box approach by applying a quantum multi-armed bandit algorithm to a quantum circuit implementing k iterations of MCTS to quickly estimate what suggestion an ensemble of 2^ℓ instances of MCTS would be likely to make after k iterations.

In Chapter 8 we provide our most useful contribution, despite it not being directly tied to MCTS itself. By generalizing the superposition preparation technique from Section 6.3.1 and using our previous knowledge of quantum mean estimation and quantum bandits we derive the powerful Theorem 8.1 and Theorem 8.2. These theorems allow one to estimate respectively the expected value or optimal first step of an arbitrary (stochastic) fixed-length policy-guided walk in an arbitrary (stochastic) environment defined by an initial state, a policy function and a transition function, with a value assigned to such a walk by an arbitrary evaluation function defined on the complete path taken, quadratically faster than on classical computers. As mentioned in the related work section, this is closely related to the work by Arjan Cornelissen in his master thesis [Cor18], however our method is more general in allowing arbitrary evaluation functions on complete paths as opposed to only (discounted) MPD rewards, and to our knowledge the application of quantum bandits to find a maximizing first step is completely novel.

In Section 5.1 we derive Lemma 5.1, which is an extension of the often cited [JVV86]. The original lemma (often referred to as the *powering lemma*) shows that if one has $O(\log 1/\delta)$ independent estimates each of which is within certain error bounds with probability at least $\frac{3}{4}$ then their median is within those same error bounds with probability $1-\delta$. We extend this lemma to work with estimation procedures whose success probability is $\frac{1}{2}+\gamma$ rather than at least $\frac{3}{4}$, deriving that in this case $\frac{\log 1/\delta}{2\gamma^2}$ estimates are sufficient. The techniques of this derivation are found in virtually every complexity theory textbook to show that the required success probability of $\frac{2}{3}$ in complexity class BPP is completely arbitrary, nevertheless we were unable to find a prior explicit derivation that exposes the (asymptotic) complexity w.r.t. the *advantage* γ over random chance.

Finally, in Chapter 6 we derive Lemma 6.1 which shows that to estimate p/q with relative error ϵ with failure probability δ, it suffices to compute $\hat{p}/\hat{q}$ where $\hat{p}, \hat{q}$ were independently estimated with relative error $\epsilon/3$ with failure probability $\delta/2$. The techniques used to demonstrate this are completely elementary but we are unaware of a prior explicit derivation of this fact. This lemma is then used in Theorem 6.2 to prove correct a simple but novel quantum algorithm which can, given a fixed-height tree implied by a local neighborhood function, estimate m, the fraction of (strictly downward) paths from root to leaf that end in a marked leaf, in $O\left(\frac{\log 1/\delta}{\epsilon\sqrt{m}}\right)$ time with relative error ϵ and failure probability δ.

1.3 Overview

In Chapter 2 we provide the preliminary knowledge of quantum computing needed for this thesis. Then in Chapter 3 we discuss the problem of strategizing in two player games, introduce the minimax and Monte Carlo Tree Search algorithms and discuss real-world applications of the latter. In Chapter 4 we discuss what exactly qualifies as a quantum improvement to MCTS, discuss possible avenues of such improvements as well as what obstacles can be expected. In Chapter 5 we discuss the last preliminaries for our main methods, in particular two more advanced quantum algorithms as well as how we measure complexity. Then in Chapter 6 we introduce a set of novel methods to quickly estimate random rollout winrate, and in Chapter 7 we give a novel method to quickly predict the choice a certain variant of parallel MCTS makes. In Chapter 8 we state two powerful new theorems for the estimation of mean behavior of stochastic policies in stochastic environments and discuss possible applications of them. Finally in Chapter 9 we conclude the thesis and discuss our findings.

Chapter 2

Quantum computing

2.1 Fundamentals

In this thesis we do not concern ourselves with the physical manifestation of quantum computing, instead working with an idealized quantum machine that has no errors or noise. This is similar to classical computing, where a computer scientist assumes idealized binary circuits with perfect logic gates not concerned with the physical implementation in transistors or otherwise.

What follows is a brief introduction of this idealized quantum world and its operations.

2.1.1 Notation

Linear algebra

We assume the reader is familiar with basic linear algebra. As a quick refresher, in linear algebra our objects of interest, *vectors*, live in a finite dimensional *vector space*, a space of n-tuples over some field $\mathbb{F}$. A vector $\mathbf{z}$ can be written $(z_1, z_2, \ldots, z_n)$ or

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}. \tag{2.1}$$

Addition of vectors and multiplication by a scalar is defined element-wise,

$$\mathbf{x} + \mathbf{y} = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}, \qquad a\mathbf{z} = \begin{bmatrix} az_1 \\ az_2 \\ \vdots \\ az_n \end{bmatrix}. \tag{2.2}$$

Depending on the context the symbol 0 may be used to refer to an appropriately sized vector with only zero entries.

A *matrix* is a generalization of a vector, being an $m \times n$ two-dimensional array of elements of some field $\mathbb{F}$ also denoted as

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,n} \\ a_{2,1} & a_{2,2} & \ldots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \ldots & a_{m,n} \end{bmatrix}. \tag{2.3}$$

$A_{i,j}$ denotes the entry at ith row and jth column of matrix A. We define multiplication by a scalar $[aA]_{i,j} = aA_{i,j}$ and addition over two equal-sized matrices A, B as $[A + B]_{i,j} = A_{i,j} + B_{i,j}$. When A is a $m \times n$ matrix and B is a $n \times p$ matrix we define the product of them as the $m \times p$ matrix with entries

$$[AB]_{i,j} = \sum_{r=1}^{n} A_{i,r} B_{r,j}. \tag{2.4}$$

Since every vector is also a matrix this product also defines a matrix-vector product $A\mathbf{z}$. The matrix-vector product of a $m \times n$ matrix A and a n-vector $\mathbf{z}$ has a one-to-one correspondence with the linear mappings from $\mathbb{F}^n \to \mathbb{F}^m$, thus we will also refer to such matrices as linear mappings, functions or operators.

We define one more product on matrices, the Kronecker product $A \otimes B$ defined as the block matrix

$$A \otimes B = \begin{bmatrix} a_{1,1}B & a_{1,2}B & \dots & a_{1,n}B \\ a_{2,1}B & a_{2,2}B & \dots & a_{2,n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1}B & a_{m,2}B & \dots & a_{m,n}B \end{bmatrix}. \tag{2.5}$$

The notation $A^{\otimes k}$ is used to denote A multiplied by itself k times using the Kronecker product.

The *transpose* of a matrix A is denoted as A^T and defined as $[A^T]_{i,j} = A_{j,i}$. A matrix is called *square* if it has size $n \times n$. The square matrix with only ones on its diagonal and zeroes everywhere else is called the *identity matrix* and denoted as I (with its size implied by context). If a matrix A^{-1} exists satisfying $A^{-1}A = I$ then it is unique and we call it the *inverse* of A.

Complex numbers

In quantum computing we almost exclusively use the field of *complex numbers* $\mathbb{C}$. This field is an extension of the real numbers with the imaginary unit i defined as $i^2 = -1$. A general complex number is written as $a + bi$ where $a, b \in \mathbb{R}$. This gives us the addition and multiplication rules

$$\begin{aligned} (a + bi) + (c + di) &= (a + c) + (b + d)i, \\ (a + bi)(c + di) &= (ac - bd) + (ad + bc)i. \end{aligned} \tag{2.6}$$

The *conjugate* of a complex number $z = a+bi$ is defined as $z^* = a-bi$, and defined over matrices of complex numbers by element-wise application $[A^*]_{i,j} = {A_{i,j}}^*$. The *Hermitian conjugate or adjoint* of a matrix A is defined as $A^\dagger = \left(A^T\right)^*$. A matrix A is *unitary* if $A^\dagger A = I$.

Hilbert space

A Hilbert space is a vector space with an associated inner product that is also a complete metric space with respect to the distances given by the inner product. Without exception the inner product used here is

$$(\mathbf{y}, \mathbf{z}) = \mathbf{y}^\dagger \mathbf{z} = \sum_i {y_i}^* z_i. \tag{2.7}$$

Of extra interest to quantum computing is the finite-dimensional Hilbert space $\left(\mathbb{C}^2\right)^{\otimes n} = \mathbb{C}^{2^n}$. Unless mentioned otherwise this vector space is assumed, with n implied by context. This Hilbert space has an orthonormal basis

$$\text{span}\left\{(1, 0, \dots, 0), (0, 1, \dots, 0), \dots, (0, 0, \dots, 1)\right\} = \mathbb{C}^{2^n}, \tag{2.8}$$

also known as the *computational basis* where the kth computational basis vector is all-zero except for the kth column having a 1.

Brakets

In quantum computing (borrowing from its origins in quantum mechanics) the standard notation for a vector is not a boldface lowercase letter, rather it is a *ket*:

$$|\psi\rangle. \tag{2.9}$$

Here ψ is the equivalent of a variable name for the vector. There is however a second, subtly different meaning. When we write $|x\rangle$ where x is a previously defined integer we mean to describe the xth vector in the computational basis. Depending on the context x can be zero-based ($x \in \{0, 1, \ldots, 2^n - 1\}$) or one-based ($x \in \{1, 2, \ldots, 2^n\}$). Similarly, x can be a string of bits of length n, which is to be interpreted as an unsigned binary integer forming a zero-based index. E.g. for $n = 1$ we have

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \tag{2.10}$$

and for $n = 2$ we have

$$|0\rangle = |00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad |1\rangle = |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad |2\rangle = |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad |3\rangle = |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}. \tag{2.11}$$

The Hermitian conjugate of a vector is also written as a *bra*:

$$\langle\phi| = |\phi\rangle^\dagger. \tag{2.12}$$

When a bra and a ket are written together they form a *braket*, which denotes the inner product of the two vectors,

$$\langle\phi|\psi\rangle = (|\phi\rangle, |\psi\rangle). \tag{2.13}$$

The implied multiplication between any two vectors written together is understood to be the Kronecker product,

$$|\phi\rangle|\psi\rangle = |\phi\rangle \otimes |\psi\rangle, \tag{2.14}$$

in all other cases the implied product is the matrix-matrix product. The *Euclidean norm* of a vector $|v\rangle$ is defined as

$$\||v\rangle\| = \sqrt{\langle v|v\rangle}. \tag{2.15}$$

A vector is *normal* or a *unit vector* if its Euclidean norm is one, and *normalizing* a vector means dividing it by its Euclidean norm. Two vectors $|\phi\rangle, |\psi\rangle$ are considered *orthogonal* if $\langle\phi|\psi\rangle = 0$. A set of vectors is considered *orthonormal* if all the vectors are normal and pairwise orthogonal.

Eigenvectors and eigenvalues

An *eigenvector* of a linear operator A is a vector $|v\rangle$ such that

$$A|v\rangle = v|v\rangle \tag{2.16}$$

where v is the unique associated *eigenvalue* associated with eigenvector $|v\rangle$ w.r.t. A. This uniqueness is only one-way, there might be multiple vectors having eigenvalue v, the set of all such vectors is called the *eigenspace* of v w.r.t. A.

2.1.2 Single-qubit systems

The simplest classical system that can contain information is the *bit*, which lives in a 'state space' of $\{0, 1\}$. That is, a bit can have the states 0 and 1. The simplest quantum system is a *qubit*, which has associated state space $\mathbb{C}^2$. It can be in the state $|0\rangle$ or $|1\rangle$, but also any *superposition* of the two,

$$|\psi\rangle = a|0\rangle + b|1\rangle, \quad \text{where} \quad |a|^2 + |b|^2 = 1, \tag{2.17}$$

where a, b are the respective *amplitudes* of states $|0\rangle, |1\rangle$. While a single bit only allows a single non-trivial operation (boolean negation), we can evolve the state of a single qubit with a lot more freedom, using any unitary operator:

$$|\psi'\rangle = A|\psi\rangle, \quad \text{where} \quad A^\dagger A = I. \tag{2.18}$$

However, a state and its amplitudes can never be directly observed. Instead, when a measurement in the computational basis is made on $|\psi\rangle$ the superposition *collapses* and we observe $|0\rangle$ with probability $|a|^2$ and $|1\rangle$ with probability $|b|^2$. This collapse is unavoidable; the measurement changes the state to be consistent with the observation and the original state is lost. E.g., if we measure the state $|\phi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ w.r.t. the computational basis we have an equal probability of observing $|0\rangle$ or $|1\rangle$, after which the state will have collapsed and thus the new state $|\phi'\rangle \in \{|0\rangle, |1\rangle\}$.

Quantum computing thus gives us a vastly richer continuous state space for our computation to live in, but ultimately we must observe it to extract our desired result, bringing us back into a discrete world. The quantum programmer must be clever, but if we evolve our quantum systems in the right way and observe it at the right moment(s) we can do some powerful computation indeed.

2.1.3 Multi-qubit systems

A brief formalization of *closed* quantum systems follows, summarizing the more complete introduction and formalization found in [NC11]. Closed meaning isolated—no outside interference is allowed to mutate the state unpredictably.

Quantum amplitudes

More generally we say that with a quantum system there's an associated Hilbert space forming the *state space*. The state of a quantum system is a unit vector in this space, or equivalently any state $|\psi\rangle$ must satisfy $\langle\psi|\psi\rangle = 1$. The n-qubit system has associated Hilbert space $\mathbb{C}^{2^n}$. It is generally assumed that at least one state can be constructed directly as an initial state, almost always $|0\rangle$.

If a quantum system is in a state $|\psi\rangle$ which is a linear combination of states

$$|\psi\rangle = \sum_i \alpha_i |\psi_i\rangle, \tag{2.19}$$

we say that it is in a *superposition* of states $|\psi_i\rangle$ where the *amplitude* of state $|\psi_i\rangle$ is α_i.

Unitary evolution

The state of a quantum system can evolve over time. Continuous-time definitions exist but for the purposes of this thesis we limit ourselves to discrete evolution in steps. In this context a state

$|\psi\rangle$ at step t can evolve into a new state $|\psi'\rangle$ at step $t+1$ with

$$|\psi'\rangle = A|\psi\rangle, \quad \text{where} \quad A^\dagger A = I. \tag{2.20}$$

So it is precisely the unitary mappings that can change our state from one to the next. In Section 2.1.4 the concrete mappings we'll use are shown. A critical implication of this is that until measurement all computation must be reversible. In Section 2.3 we show a technique to embed non-reversible computing in this context.

Measurement

A quantum measurement is taken by defining a set of *measurement operators* $\{M_m\}$, exactly one of which will be applied by the measurement, giving knowledge of which was applied (the outcome of the measurement). The probability of seeing outcome m when measuring state $|\psi\rangle$ is given by

$$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle, \tag{2.21}$$

after which the state of the system (due to collapse) will be

$$|\psi'\rangle = \frac{M_m|\psi\rangle}{\sqrt{\langle\psi|M_m^\dagger M_m|\psi\rangle}}. \tag{2.22}$$

The probabilities must sum to one thus

$$\sum_m p(m) = \langle\psi|\left(\sum_m M_m^\dagger M_m\right)|\psi\rangle = 1 \iff \sum_m M_m^\dagger M_m = I, \tag{2.23}$$

giving the *completeness equation* the set of measurement operators must satisfy.

A special case of measurement is measurement of n qubits w.r.t. the computational basis, in which case the 2^n measurement operators are $M_m = |m\rangle\langle m|$, which are the projections onto the computational basis vectors. Such a measurement is thus also called a *projective measurement*. Note that then

$$M_m^\dagger = \big(|m\rangle\langle m|\big)^\dagger = \langle m|^\dagger |m\rangle^\dagger = |m\rangle\langle m| = M_m, \tag{2.24}$$

and by orthonormality of the computational basis vectors

$$M_m|i\rangle = |m\rangle\,\langle m|i\rangle = \begin{cases} |m\rangle & \text{if } i = m \\ 0 & \text{otherwise} \end{cases}, \tag{2.25}$$

which also implies $M_m^\dagger M_m = M_m^2 = M_m$.

If we express a state using amplitudes over the computational basis

$$|\psi\rangle = \sum_i \alpha_i |i\rangle \tag{2.26}$$

we find that if we observe this state we get outcome m with probability

$$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle = \langle\psi|\sum_i \alpha_i M_m|i\rangle = \alpha_m\,\langle\psi|m\rangle = \alpha_m \alpha_m^\dagger = |\alpha_m|^2. \tag{2.27}$$

Thus the square of the absolute values of an amplitude corresponds to the probability of observing its respective outcome. Note that this also means that two states $|\psi\rangle$ and $e^{i\theta}|\psi\rangle$ are

indistinguishable as all probabilities of observation are unchanged, regardless of θ (which is known as the *global phase*).

After measurement over the computational basis and observing m the new state will be

$$|\psi'\rangle = \frac{M_m|\psi\rangle}{\sqrt{\langle\psi|M_m^\dagger M_m|\psi\rangle}} = \frac{\alpha_m}{|\alpha_m|}|m\rangle, \tag{2.28}$$

which as $\frac{\alpha_m}{|\alpha_m|} = e^{i\theta}$ for some θ is indistinguishable from simply $|m\rangle$. Thus for all intents and purposes our measurement has collapsed our quantum state to simply $|m\rangle$, exactly what we observed.

Kronecker product

The state space of the joint system of two independent quantum systems is described mathematically by the Kronecker product. E.g. if I have two quantum states $|0\rangle$ and $|1\rangle$ then

$$|0\rangle \otimes |1\rangle = |0\rangle|1\rangle = |01\rangle, \tag{2.29}$$

their Kronecker product, is used to denote the combined system of two qubits, the first in state 0 and the second in state 1. Such a system would be a two-qubit system, and in general we can arbitrarily combine to form *n-qubit systems*.

The same holds true for extending a set of unitary operators to apply to a combined system. E.g. if U, V are unitary operators defined to operate on two quantum systems, then $U \otimes V$ is the operator that applies U to the first and V to the second system when combined.

Entanglement

The converse of the previous section also holds true, if we can write a quantum state as

$$|\psi\rangle = |a\rangle|b\rangle \tag{2.30}$$

then we can decompose state $|\psi\rangle$ into two independent systems a, b. For classical computing and classical bits this always holds true, but interestingly for quantum computing this decomposition does **not** always hold.

It's possible to have a multi-qubit quantum system where the value of one bit is correlated with one or more other bits. We say the qubits are *entangled*. A canonical example is

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle). \tag{2.31}$$

Even if you observe only one of the two qubits of $|\psi\rangle$, the other qubit will collapse to the same value as well. These two qubits are thus not independent - they are correlated, even if in the physical world their manifestation might be (seemingly) arbitrarily far separated [Hen+15].

2.1.4 Quantum gates & circuits

Like classical circuits are built out of *classical gates* (e.g. AND, OR, NOT), quantum circuits are built from *quantum gates*. Unlike classical gates however, all quantum gates are reversible, meaning that for any gate G there exists another gate G^{-1} such that $G^{-1}G = I$. Since all our quantum gates are unitary we also write $G^{-1} = G^\dagger$.

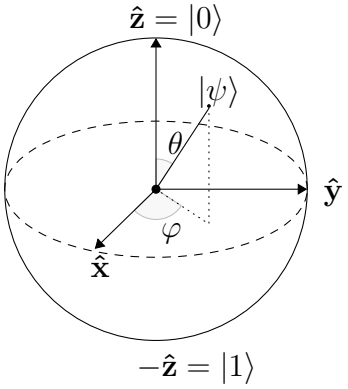
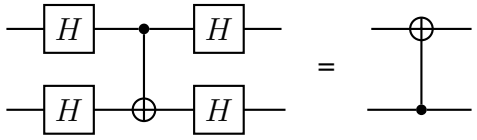


Figure 2.1: Illustration of the Bloch sphere with the X, Y and Z axes shown, by TEX StackExchange user *Sebastiano*. Any single qubit state $|\psi\rangle = a|0\rangle + b|1\rangle$ can be taken with a real as global phase is unobservable, and expressed as $a = \cos(\theta/2), b = e^{i\varphi}\sin(\theta/2)$.

These quantum gates can be described in multiple ways, here we'll describe them by their associated unitary matrix as well as their behavior for the computational basis states (which by linearity defines their behavior completely).

Quantum circuits are shown in a visual notation. On the left hand side are inputs, with each wire representing on (register of) qubit(s), with the output on the right hand side. On each wire quantum gates may be applied, which is understood to happen serially from left-to-right. An example (showing an equivalence between two circuits):



Quantum gates and sub-circuits are drawn using square blocks, the other notation is explained below. Finally, two or more wires or gates that are above/below each other are combined using the Kronecker product, thus the left hand side of the above circuit could also have been written as

$$(H \otimes H) \cdot \mathrm{CNOT} \cdot (H \otimes H). \tag{2.32}$$

X gate

The X gate is a single-qubit gate, roughly the quantum equivalent of the classical NOT gate:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \qquad X|0\rangle = |1\rangle, \qquad X|1\rangle = |0\rangle. \tag{2.33}$$

X is its own inverse as $X^2 = I$.

Y and Z gates

Whereas a classical bit is discrete and one-dimensional, a qubit is a unit vector in $\mathbb{C}^2$. This is often thought of as a point on the *Bloch sphere* (as seen in Figure 2.1). Then the X gate is a $180°$

rotation about the X-axis in the Bloch sphere, and similarly for the Y and Z gates

$$\begin{aligned} Y &= \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \qquad Y|0\rangle = i|1\rangle, \qquad Y|1\rangle = -i|0\rangle, \\ Z &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \qquad Z|0\rangle = |0\rangle, \qquad Z|1\rangle = -|1\rangle. \end{aligned} \tag{2.34}$$

Like X, Y and Z are its own inverse, $Y^2 = Z^2 = I$.

H gate

The H gate or Hadamard gate introduces a superposition from the computational basis states:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \qquad H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \qquad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \tag{2.35}$$

These two particular superposition states are quite common and are often written as

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \qquad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \tag{2.36}$$

H is also its own inverse, $H^2 = I$. Notable is that applying the Hadamard gate to the all-zero n-qubit state creates the uniform n-qubit superposition

$$H^{\otimes n}|0\rangle^{\otimes n} = \frac{1}{\sqrt{2^n}} \sum_{i=1}^{2^n} |i\rangle. \tag{2.37}$$

Finally the Hadamard gate can be thought of as a rotation in the Bloch sphere around the diagonal axis between X and Z, and thus $H = \frac{1}{\sqrt{2}}(X + Z)$.

S, T and R_k gates

The phase gate or S gate induces a change in phase only for a set bit,

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \qquad S|0\rangle = |0\rangle, \qquad S|1\rangle = i|1\rangle. \tag{2.38}$$

Unlike the previous gates the S gate is not its own inverse, instead we have $S^2 = Z$. In some sense we can thus say $S = \sqrt{Z}$. In a similar fashion we retrieve the $T = \sqrt{S}$ gate:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/8} \end{bmatrix}, \qquad T|0\rangle = |0\rangle, \qquad T|1\rangle = e^{2\pi i/8}|1\rangle. \tag{2.39}$$

All of these can be understood as rotations around the Z-axis in the Bloch sphere. The Z, S, T gates respectively rotate about the Z-axis by $180, 90$, and 45 degrees. We generalize this into an arbitrary power-of-two rotation operator

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix}, \qquad R_k|0\rangle = |0\rangle, \qquad R_k|1\rangle = e^{2\pi i/2^k}|1\rangle. \tag{2.40}$$

We can now rotate a qubit about the Z-axis by an arbitrary angle θ approximated to n bits of precision by expressing θ as a binary fractional number

$$\theta \approx (0.a_1 a_2 \dots a_n)_2 = \sum_{k=1}^{n} 2^{-k} a_k \tag{2.41}$$

and precisely applying those R_k where $a_k = 1$, giving

$$R_{\hat{\mathbf{z}}}(\theta) \approx {R_n}^{a_n} {R_{n-1}}^{a_{n-1}} \cdots {R_1}^{a_1} \tag{2.42}$$

If θ is a dyadic rational and can thus be written as $x/2^n$ for some $n \in \mathbb{N}, x \in [0, 2^n)$ then the above approximation is exact.

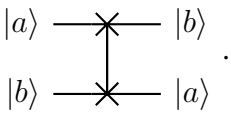
Swap gate

The quantum circuits in use in quantum algorithms can be quite large, but all gates defined here are just for one, two or three qubit systems. Using the identity gate and the Kronecker product we can apply these gates selectively to the qubits we desire inside a larger system if they are adjacent, but they might not be.

Thus we introduce one more gate, the swap gate,

$$\mathrm{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad \mathrm{SWAP}|a\rangle|b\rangle = |b\rangle|a\rangle, \tag{2.43}$$

denoted in quantum circuit diagrams as



.

While not necessarily the best implementation in practice, at least theoretically it's possible to bring any combination of qubits to be adjacent in the n-qubit system using a series of swaps, apply the local gate and swap back to the original positions. From now on we assume we can simply apply multi-qubit gates to arbitrary collections of qubits, knowing that if not directly possible in the physical manifestation one can always emulate it using purely local gates and swaps.

CNOT gate

The CNOT gate is often the most primitive (but a terrifically useful) two-qubit gate. On the computational basis states it leaves its first argument (the *control bit*) unchanged and maps the second argument (the *target bit*) to the XOR (denoted $\oplus$) of the two:

$$\mathrm{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \qquad \mathrm{CNOT}|a\rangle|b\rangle = |a\rangle|a \oplus b\rangle. \tag{2.44}$$

It is denoted as

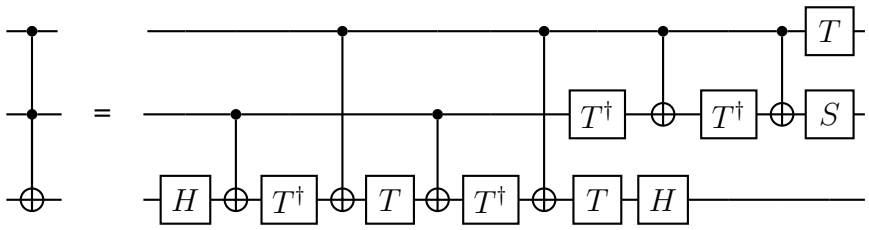
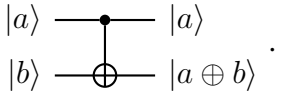


Figure 2.2: Implementation of the Toffoli gate from more primitive gates.

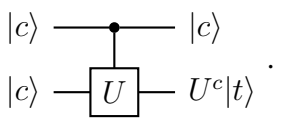


Alternatively one can understand it as flipping the second bit if the first is true (hence conditional NOT or CNOT).

More generally, any gate U can be turned into a controlled version by the block matrix

$$\text{c-}U = \begin{bmatrix} I & 0 \\ 0 & U \end{bmatrix}, \qquad \text{c-}U|c\rangle|t\rangle = |c\rangle U^c|t\rangle, \qquad \text{where } c \in \{0,1\}. \tag{2.45}$$

A complete circuit can be made into a controlled version by replacing every basic gate with a controlled version. A controlled version of a gate or subcircuit is denoted as



Toffoli gate

The Toffoli (CCNOT) gate is similar to the CNOT gate, however it is a three-qubit gate and it only flips its target bit if its *two* control bits are both set,

$$\text{CCNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \qquad \text{CCNOT}|c_1\rangle|c_2\rangle|t\rangle = |c_1\rangle|c_2\rangle|t \oplus c_1c_2\rangle. \tag{2.46}$$

Unlike the CNOT gate, the Toffoli gate can be built from previously seen gates, as seen in Figure 2.2.

2.2 Quantum phase and amplitude estimation

Many (if not most) quantum algorithms create some unitary operator U whose eigenvectors or eigenvalues have specific properties of interest. Using the algorithms presented in this section we can observe these properties efficiently, giving rise to more efficient algorithms than previously possible.

2.2.1 Quantum Fourier transform

The *Fourier Transform* is a classic transform from signal processing theory that allows transforming a signal between a time domain (a waveform) and a frequency domain (a spectrum analysis). The theory is very deep and touches many fields, however we won't get into any of that theory here—we simply use the Fourier transform for its computational properties.

We're interested in one particular variant, the *discrete Fourier Transform*. It takes as input a vector of N complex numbers $x_0, \ldots, x_{N-1}$ and outputs a vector $y_0, \ldots, y_{N-1}$ with coefficients

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi i jk/N}. \tag{2.47}$$

The *quantum Fourier Transform* is a similar linear transformation defined on the computational basis states (using zero-based indexing for convenience) as

$$\text{QFT}: \quad |j\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i jk/N} |k\rangle. \tag{2.48}$$

It can be seen to be equivalent to (2.47) by viewing its action on an arbitrary state:

$$\text{QFT}: \quad \sum_{j=0}^{N-1} x_j |j\rangle \mapsto \sum_{k=0}^{N-1} y_k |k\rangle. \tag{2.49}$$

If we take $N = 2^n$ and allow writing j as a string of binary digits $j = j_1 j_2 \ldots j_n = \sum_{l=1}^{n} 2^{n-l} j_l$ and use binary fraction shorthand $0.abc\ldots$ to mean $a/2 + b/4 + c/8 + \cdots$, then we can show that

$$|j\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{2\pi i jk/2^n} |k\rangle \tag{2.50}$$

$$= \frac{1}{\sqrt{2^n}} \sum_{k_1=0}^{1} \cdots \sum_{k_n=0}^{1} e^{2\pi i j \left(\sum_{l=1}^{n} k_l 2^{-l}\right)} |k_1 \ldots k_n\rangle \tag{2.51}$$

$$= \frac{1}{\sqrt{2^n}} \sum_{k_1=0}^{1} \cdots \sum_{k_n=0}^{1} \bigotimes_{l=1}^{n} e^{2\pi i j k_l 2^{-l}} |k_l\rangle \tag{2.52}$$

$$= \frac{1}{\sqrt{2^n}} \bigotimes_{l=1}^{n} \left[\sum_{k_l=0}^{1} e^{2\pi i j k_l 2^{-l}} |k_l\rangle \right] \tag{2.53}$$

$$= \frac{1}{\sqrt{2^n}} \bigotimes_{l=1}^{n} \left[|0\rangle + e^{2\pi i j 2^{-l}} |1\rangle \right] \tag{2.54}$$

$$= \frac{1}{\sqrt{2^n}} \Big(|0\rangle + e^{2\pi i 0.j_n} |1\rangle \Big) \Big(|0\rangle + e^{2\pi i 0.j_{n-1} j_n} |1\rangle \Big) \cdots \Big(|0\rangle + e^{2\pi i 0.j_1 j_2 \cdots j_n} |1\rangle \Big), \tag{2.55}$$

where (2.55) is the *product representation* of the QFT.

Note that applying a Hadamard gate on $|j_l\rangle$ gives

$$|j_l\rangle \mapsto \frac{1}{\sqrt{2}} \big(|0\rangle + e^{2\pi i 0.j_l} |1\rangle \big), \tag{2.56}$$

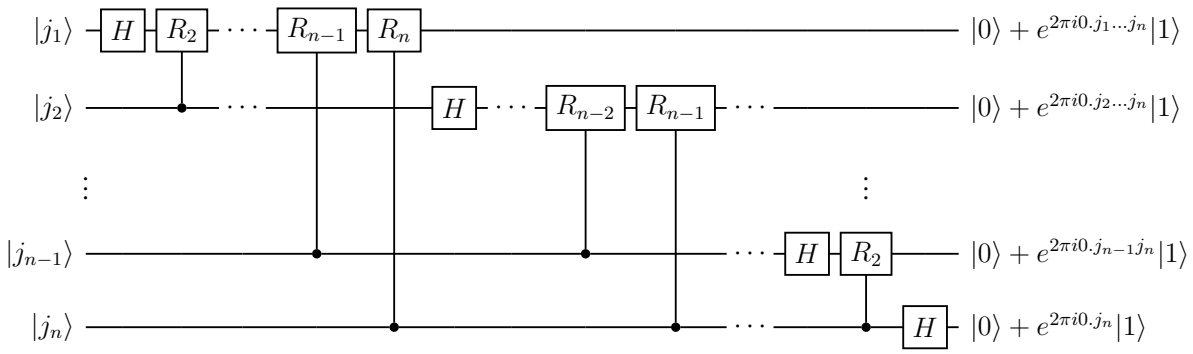


Figure 2.3: Circuit implementing the quantum Fourier transform. For brevity the swap gates at the end of the circuit reversing the output order are omitted, as well as the normalization factor $\frac{1}{\sqrt{2^n}}$ of the results, but besides that the circuit is a direct implementation of (2.55).

as $e^{2\pi i 0.j_l} = e^{\pi i j_l} = (-1)^{j_l}$. After that it's possible to apply a R_2 gate controlled by a qubit in the state $|j_{l+1}\rangle$, giving

$$\frac{1}{\sqrt{2}}\big(|0\rangle + e^{2\pi i 0.j_l}|1\rangle\big) \mapsto \frac{1}{\sqrt{2}}\big(|0\rangle + e^{2\pi i 0.j_l j_{l+1}}|1\rangle\big), \tag{2.57}$$

as $e^{2\pi i 0.j_l j_{l+1}} = e^{2\pi i 0.j_l} e^{2\pi i j_{l+1}/2^2}$ and recalling (2.40). Repeating this process for $R_3, \dots, R_{n+1-l}$ controlled by respectively $|j_{l+2}\rangle, \dots, |j_n\rangle$ allows for the production of the phases required in (2.55). The resulting circuit can be seen in Figure 2.3. Note that it uses $O(n^2)$ gates whereas the classical Fast Fourier Transform algorithm is $O(N \log N)$. The QFT is thus 'exponentially faster' (recall that $N = 2^n$), although different as you can not simply read out the values.

Inverse

The inverse of the quantum Fourier transform is the mapping

$$\mathrm{QFT}^\dagger : \quad \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i j k/N} |k\rangle \mapsto |j\rangle. \tag{2.58}$$

When N is a power of two it can be constructed similar to Figure 2.3, by reversing the order of all the gates and replacing each gate with its inverse.

2.2.2 Phase kickback and quantum phase estimation

Suppose we have a unitary operator U that has eigenvector $|u\rangle$ and eigenvalue $e^{2\pi i \varphi_u}$. Applying a controlled version of this operator on a control bit plus the eigenvector gives

$$\text{c-}U|0\rangle|u\rangle = |0\rangle|u\rangle, \qquad \text{c-}U|1\rangle|u\rangle = e^{2\pi i \varphi_u}|1\rangle|u\rangle. \tag{2.59}$$

We used a controlled operator on a state which we know will be left unchanged, which caused a phase to be applied to the control bit. This is known as the *phase kickback*. We can construct a mapping

$$|j\rangle|u\rangle \mapsto |j\rangle U^j|u\rangle \tag{2.60}$$

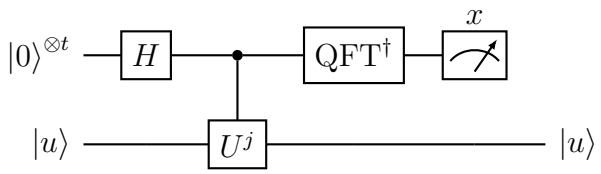


Figure 2.4: Circuit for quantum phase estimation of a unitary operator U with eigenvector $|u\rangle$.

by representing $|j\rangle$ in binary using a t qubit register and using c-U^{2^k} on $|u\rangle$ controlled by the kth bit of $|j\rangle$. If we then produce a uniform superposition of t qubits in our first register and $|u\rangle$ in our second register and apply (2.60) we get

$$\frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}|j\rangle U^j|u\rangle = \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}e^{2\pi ij\varphi_u}|j\rangle|u\rangle. \tag{2.61}$$

If we assume that φ_u can be exactly represented using t bits, that is $\varphi_u = x/2^t$ for some integer $x \in [0, 2^t)$ and we apply QFT† to the first register we get exactly $|x\rangle|u\rangle$ as per (2.58). Thus measuring the first register allows us to read x and thus φ_u. The overall procedure for quantum phase estimation is visualized in Figure 2.4.

In [NC11] Section 5.2.1 it is proven that when φ_u can *not* be written as $x/2^t$ but we apply the above procedure regardless then the resulting estimate $\widetilde{\varphi_u}$ is accurate to $t - \left\lceil \log\left(2 + \frac{1}{2\epsilon}\right)\right\rceil$ bits with a probability of success of at least $1 - \epsilon$. This thus gives us an efficient probabilistic algorithm to estimate φ_u to any desired precision.

Finally, if one applies *QPE* (quantum phase estimation) to a state $|\psi\rangle$ which is not an eigenvector of U one can first express $|\psi\rangle$ as a superposition in the *eigenbasis* of U:

$$|\psi\rangle = \sum_i a_i|u_i\rangle. \tag{2.62}$$

Then with probability $|a_i|^2 = \langle u_i|\psi\rangle$ the outcome will be an estimate of the eigenvalue λ_i corresponding to eigenvector $|u_i\rangle$, which due to the collapse from the observation can be found in the second register.

2.2.3 Quantum amplitude estimation

Suppose we have a quantum circuit $\mathcal{A}$ that prepares some state of interest $|\psi\rangle = \mathcal{A}|0\rangle$ when applied to the all-zero state and a function $f(x) \to \{0, 1\}$ that given some input x determines whether it is *good* (1) or not (0). We would like to estimate the probability of observing a good bit string when $|\psi\rangle$ is measured w.r.t. the computational basis. We describe a procedure due to Brassard et. al. [Bra+02].

A circuit implementing f can always be transformed to a *phase oracle*

$$S_f|x\rangle = (-1)^{f(x)} = \begin{cases} |x\rangle & \text{if } f(x) = 0 \\ -|x\rangle & \text{if } f(x) = 1 \end{cases} \tag{2.63}$$

either directly or through a classical oracle

$$O_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle, \tag{2.64}$$

an ancillary qubit, and the following circuit:

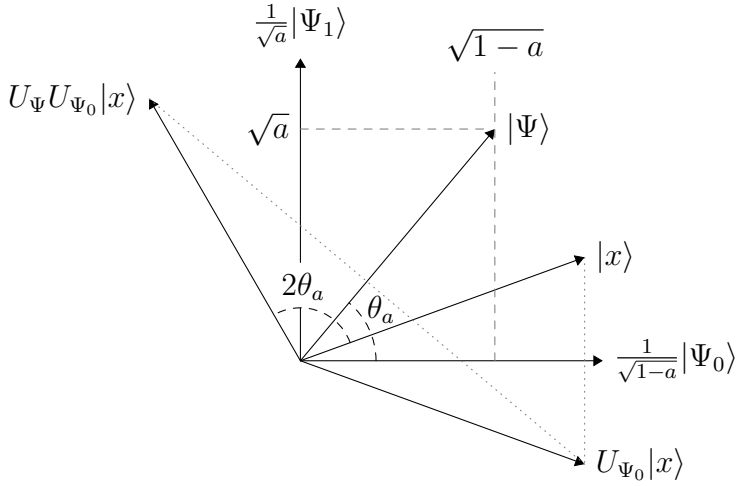
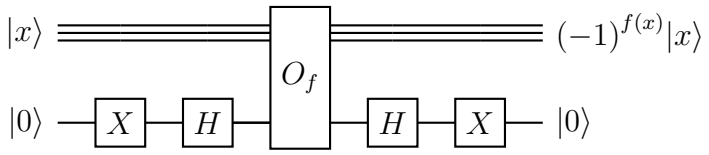


Figure 2.5: $Q = U_\Psi U_{\Psi_0}$ applied to $|x\rangle$ visualized as two reflections in the two-dimensional subspace $\mathcal{H}_\Psi$ (with only real coefficients). First we reflect over $|\Psi_0\rangle$ followed by a reflection over $|\Psi\rangle$. The net effect is a rotation of $|x\rangle$ by angle $2\theta_a$.



If $f(x) = 0$ the above is trivially the identity and when $f(x) = 1$ we find that

$$O_f|x\rangle(|0\rangle - |1\rangle) = |x\rangle(|1\rangle - |0\rangle) = -|x\rangle(|0\rangle - |1\rangle) \tag{2.65}$$

and thus a sign flip is induced. We similarly introduce a phase oracle S_0 that flips the sign of $|0\rangle$ and nothing else. Such a phase oracle can also be viewed as a reflection over the state it leaves unchanged, or a reflection + negation over the state it flips the sign of. E.g.

$$S_0 = -(2|0\rangle\langle 0| - I). \tag{2.66}$$

We write $|\Psi\rangle = \mathcal{A}|0\rangle = |\Psi_0\rangle + |\Psi_1\rangle$ where $|\Psi_0\rangle, |\Psi_1\rangle$ are respectively the superpositions of the bad and good vectors in $|\Psi\rangle$ and $a = \langle\Psi_1|\Psi_1\rangle$ is the probability of measuring a good state.

We can now study the unitary operator

$$Q = -\mathcal{A}S_0\mathcal{A}^\dagger S_f \tag{2.67}$$

on Hilbert space $\mathcal{H}$. If we assume $0 < a < 1$ then Q can be understood geometrically in the two-dimensional subspace $\mathcal{H}_\Psi$ spanned by $|\Psi_0\rangle, |\Psi_1\rangle$ as the product of two reflections

$$\begin{aligned} U_{\Psi_0} &= S_f = \frac{2}{1-a}|\Psi_0\rangle\langle\Psi_0| - I, \\ U_\Psi &= -\mathcal{A}S_0\mathcal{A}^\dagger = \mathcal{A}(2|0\rangle\langle 0| - I)\mathcal{A}^\dagger = 2|\Psi\rangle\langle\Psi| - I, \\ Q &= U_\Psi U_{\Psi_0}, \end{aligned} \tag{2.68}$$

where U_Ψ reflects through $|\Psi\rangle$ and U_{Ψ_0} reflects through $|\Psi_0\rangle$. We find

$$\begin{aligned} Q|\Psi_0\rangle &= U_\Psi|\Psi_0\rangle = (2|\Psi\rangle\langle\Psi| - I)|\Psi_0\rangle = 2(1-a)|\Psi\rangle - |\Psi_0\rangle \\ &= (1-2a)|\Psi_0\rangle + 2(1-a)|\Psi_1\rangle, \\ Q|\Psi_1\rangle &= -U_\Psi|\Psi_1\rangle = (I - 2|\Psi\rangle\langle\Psi|)|\Psi_1\rangle = |\Psi_1\rangle - 2a|\Psi\rangle \\ &= -2a|\Psi_0\rangle + (1-2a)|\Psi_1\rangle. \end{aligned} \tag{2.69}$$

From (2.69) one can verify by direct substitution that

$$|\Psi_\pm\rangle = \frac{1}{\sqrt{2}}\left(\pm\frac{i}{\sqrt{1-a}}|\Psi_0\rangle + \frac{1}{\sqrt{a}}|\Psi_1\rangle\right) \tag{2.70}$$

are two eigenvectors of Q with respective eigenvalues

$$\lambda_\pm = e^{\pm 2i\theta_a} \tag{2.71}$$

where $\sin(\theta_a)^2 = a$. Similarly one can verify that

$$\mathcal{A}|0\rangle = |\Psi\rangle = -\frac{i}{\sqrt{2}}\left(e^{i\theta_a}|\Psi_+\rangle - e^{-i\theta_a}|\Psi_-\rangle\right). \tag{2.72}$$

Thus if we perform quantum phase estimation on $|\Psi\rangle$ with respect to unitary Q we can estimate one of the eigenphases $\pm 2\theta_a$ from which we can derive a, our desired probability.

Theorem 2.1 (Quantum amplitude estimation)**.** *[Bra+02] Given quantum circuit $\mathcal{A}$, its inverse $\mathcal{A}^\dagger$ and a projector P one can estimate $a = \langle\psi|P|\psi\rangle$ using the above procedure, where $|\psi\rangle = |\mathcal{A}\rangle|0\rangle$ and $S_f = I - 2P$. The resulting estimate $\tilde{a}$ satisfies*

$$|\tilde{a} - a| \leq 2\pi\frac{\sqrt{a(1-a)}}{t} + \frac{\pi^2}{t^2} \tag{2.73}$$

with probability at least $\pi^2/8$, using t invocations of $\mathcal{A}$, $\mathcal{A}^\dagger$ and P each.

Proof. See Theorem 12 of [Bra+02]. □

Despite all of the above, the connection between Fourier analysis, phase estimation and amplitude estimation might seem a bit too abstract and magical. To provide some intuition as to what's going on, the clever operator Q performs a rotation in the two-dimensional subspace with basis vectors $\frac{1}{\sqrt{1-a}}|\Psi_0\rangle$ (the 'all bad' state) and $\frac{1}{\sqrt{a}}|\Psi_1\rangle$ (the 'all good' state) with an angle that depends on $|\Psi\rangle$ itself (the given state), as in Figure 2.5. If one repeats a rotation sufficiently many times, you will exactly (for rational angles) or very closely (for irrational angles) reach the starting position. Thus a rotation operator is *periodic*, which is where Fourier analysis steps in to find that period. If we know the period we can derive the angle between $|\Psi\rangle$ and $|\Psi_0\rangle$ and thus derive the probability of a good state.

2.3 Classical universality

In 1913 Henry M. Sheffer proved [She13] that the NAND gate ($\neg(a \wedge b)$) was *universal*: any Boolean expression (or classical circuit) could be expressed using purely NAND operations and fanouts. From this it's quite easy to see how a quantum circuit can be universal for classical computation, by using the fact that

$$\begin{aligned}\mathrm{TOFF}|a\rangle|b\rangle|1\rangle &= |a\rangle|b\rangle|\neg(a \wedge b)\rangle,\\ \mathrm{TOFF}|a\rangle|1\rangle|0\rangle &= |a\rangle|1\rangle|a\rangle.\end{aligned} \tag{2.74}$$

Any classical circuit on k bits can thus be transformed into a circuit using n NAND plus fanout gates for some number n, which can be transformed into a quantum circuit using $k + n + 1$ qubits

using just an initial state of the input, $|1\rangle^{\otimes(n+1)}$ (or optionally $|0\rangle^{\otimes(n+1)}$ and $n+1$ X gates) and n Toffoli gates.

Generally it's possible to be more efficient than this trivial encoding. First, there are many other gates (CNOT, X, etc) that can be used to implement a particular classical circuit much more efficiently than a naive NAND to Toffoli conversion would provide. This reduces the total number of gates necessary. The number of temporary work bits (also known as *ancillas*) also does not always have to contain the complete temporary result history. If a particular bit of information is not part of the final output, then after it has been computed and used it can be *uncomputed*, by inverting the gates that were used to compute it and applying them in reverse. Usually ancillas which can be uncomputed can be disregarded in a computational cost analysis as the ancillas can be re-used for other circuits when composing quantum circuits, as only the maximum number of simultaneously required ancillas are needed.

Even though we can often eliminate many ancillas, as quantum computing is reversible, not every classical circuit on k bits can be expressed using an equivalent quantum circuit on k qubits. Any remaining ancillas that can not be eliminated are also often referred to as *garbage*—we don't want that information anymore, but we can't destroy it, so we must keep it around.

A more formal analysis is done in [BTV01] where it's shown that an irreversible classical computation using time T and space S can be simulated reversibly using $T' = 3^k 2^{O\left(T/2^k\right)} S$ time and $S' = S(1 + O(k))$ space where k is a freely chosen parameter $0 \leq k \leq \log T$.

Interestingly, even circuits that *can* theoretically be implemented using no ancillary qubits at all can benefit from having ancillas available regardless. A classical example is the $C^n(U)$ gate: U controlled by n control bits. A trivial construction is possible using $2n-2$ chained Toffoli gates and $n-1$ ancillas while an implementation without ancillas can be done [Gid15] but is significantly more involved.

Chapter 3

Monte Carlo Tree Search

3.1 Introduction

3.1.1 Two player games

Monte Carlo Tree Search is a strategy for finding moves in *games*. While generalizations for almost every assumption listed below exist, we assume that we are playing a game where

- there are always two players which we'll call Alice (A) and Bob (B),
- the game is strictly *turn based*, with the players alternating their turns,
- the game is *deterministic* and *well-defined*, meaning there is no randomness and in any situation it should be clear which moves a player can make,
- both players have *perfect information*, meaning there is no hidden game state invisible to either Alice or Bob (or a spectator that watched the whole game),
- some game states are *terminal* meaning neither player can make another move and there is some associated objective *value* associated with a terminal game state, and
- the game is always *finite*, meaning in any game state there are at most b move options to pick from and there must exist some h such that no legal sequence of moves exists with a length bigger than h.

If the set of possibles values of a terminal position is $\{\text{A wins}, \text{B wins}\}$ then such a game is traditionally known as a *combinatorial game* [Con76]. We will use the term *combinatorial game* to refer to any such game even though we allow the values of terminal games to take on values in $\mathbb{R}$ (unless mentioned otherwise).

The goal of a game depends on the player. For Alice, the goal is to maximize the terminal value, for Bob the goal is to minimize the terminal value of the game. In a game where one can only win or lose it is typical to score a win for Alice as $+1$ and a win for Bob as -1. If draws are possible one can score those as 0.

To formalize a combinatorial game, we define a finite set of states S corresponding to game states/positions, where some are terminal $T \subseteq S$, and exactly one is the initial state. There is a value function on terminal positions $V : T \to \mathbb{R}$, and for each state $s \in S$ there is an associated set of actions $A(s)$, corresponding to the moves a player can make in that position. If $s \in T$ then $|A(s)| = 0$. Finally, for each $s \in S, a \in A(s)$ the transition function $\delta(s, a) \in S$ is defined, giving the next state.

Game trees

A combinatorial game can be visualized as a *game tree*. In such a game tree each node corresponds to a game state and each edge corresponds to a move. At its root it has the initial game state, with a child edge for each possible move the starting player can make leading to a child with the corresponding resulting game state. This continues recursively, each node has a corresponding game state and for each move one can make in this game state there exists a child node with the resulting game state after making that move. The leafs of the resulting tree will be the terminal game states.

Any subtree of a game tree can be considered a game in of itself, it simply has a different initial game state compared to the original game. This correspondence highlights the recursive nature of combinatorial games.

Examples and strategy

Some of the most well-known combinatorial games are Chess, Checkers, Go and Tic-Tac-Toe. While a child can learn the optimal strategy to the latter, the others have fascinated people for ages in how they should best be played. It may come as a surprise that this fascination remains while an algorithm to compute the optimal strategy has already long been mathematically known, we will discuss it in Section 3.1.2.

The problem is that this computation quickly becomes infeasible. For Tic-Tac-Toe such a computation takes a fraction of a second on a modern computer, but for Checkers it took dozens of computers almost 20 years to calculate that with optimal play from both sides the game plays out to a draw [Sch+07]. The number of Chess positions ($\approx 10^{46}$, [Chi96]) and Go positions ($\approx 10^{170}$, [TF07]) is orders of magnitude higher than the number of Checkers positions ($\approx 10^{20}$, [Sch+07]), so such a similar feat likely won't be repeated for those games any time soon. Simply analyzing the total number of game positions to conclude a game is hard to solve is somewhat flawed of course, as one can trivially construct a game with a single correct strategy yet arbitrarily many game positions. But unless a game is highly limiting in its possibly optimal move choices a large portion of the game positions will have to be explored.

For this reason for real-world games we often resign to not computing the *optimal* strategy, but simply a *good* one based on heuristics. Monte Carlo tree search is an algorithm doing just that. Its goal is not to solve the game completely (in as fast as time as possible), rather, its goal is to give as good a suggestion for a move in a fixed amount of time.

3.1.2 Minimax

Proven as the optimal strategy in 1928 by von Neumann [Neu28] *minimax* is a very simple idea taken to its limit: my best move is whichever move is worst for my opponent. It defines two mutually recursive functions computing respectively the maximum and minimum values achievable by a player in a given position,

$$\begin{aligned} M_{\max}(s) &= \begin{cases} V(s) & \text{if } s \in T \\ \max_{a \in A(s)} M_{\min}(\delta(s,a)) & \text{otherwise} \end{cases}, \\ M_{\min}(s) &= \begin{cases} V(s) & \text{if } s \in T \\ \min_{a \in A(s)} M_{\max}(\delta(s,a)) & \text{otherwise} \end{cases}. \end{aligned} \tag{3.1}$$

Then to compute the game-theoretic value of a position s we use $M_{\min}(s)$ if it is Alice's turn (as she is a maximizing player and would like to minimize her opponents score) and similarly $M_{\max}(s)$

if it is Bob's turn. This simple algorithm can compute who is winning for any combinatorial game position, taking $O(N)$ time where $N = b^h$.

Alpha-beta pruning

A discussion of minimax is not complete without mentioning *alpha-beta pruning*. Independently invented many times, this optimization maintains two bounds, α and β, initially $-\infty, \infty$ respectively. α forms a lower bound on the value the maximizing player can achieve, and β forms an upper bound on the value the minimizing player can achieve. Then if at any point we notice that $\beta < \alpha$ we can abort our search in our current subtree as with optimal play this subtree would never be reached (both players have better alternatives). In a pair of papers [Pea80] [Pea82] Judea Pearl proved that minimax with alpha-beta pruning is asymptotically optimal for complete game trees of degree b and height h with random leaf values (so no game-specific heuristics exist), visiting approximately $N^{3/4}$ nodes on average.

3.1.3 Multi-armed bandit problem

Before introducing Monte Carlo Tree Search we must look at one more related problem, the *multi-armed bandit problem*. In this problem we are faced with k slot machines (named *bandits* because slot machines steal your money). Each slot machine is idealized: it has an arm we can pull and when we pull it we get some reward. That is, when we pull the ith arm we sample a random real-valued variable X_i with a hidden but stationary distribution, independent from any other arm. We want to find an optimal strategy to decide which arm to pull next based on the previous outcomes, one that maximizes our total expected value over n pulls. There are no costs in this model, we simply assume we get n pulls.

In this problem there is an inherent exploration-exploitation trade-off. After trying every arm once we might be extremely greedy and simply keep pulling the arm with the highest reward from this first sequence of pulls. But it is (easily) possible that this arm is not truly the best arm, but got 'lucky' while another, better arm on average, got 'unlucky'. So a smart strategy must balance the number of pulls used for exploiting the current conjectured best arm with the number of pulls used for finding a possibly better arm. Worse, in some versions of the problem we're not told in advance how many times we get to pull an arm so one constantly has to balance exploitation with further exploration.

In 1985 Lai and Robbins [LR85] proved that the *regret* of any strategy as the number of pulls n goes to infinity must grow with at least $\Theta(\log n)$ for a large family of distributions X_i. The regret of a strategy is defined as

$$R_n = n\mu^* - \sum_{i=1}^{k} E[T_i(n)]\mu_i, \quad \text{where} \quad \mu^* = \max_i \mu_i, \qquad \mu_i = E[X_i], \tag{3.2}$$

and $E[T_i(n)]$ is the expected number of times the ith arm gets pulled in n total pulls by the chosen strategy. In other words, the regret of a strategy is how much worse that strategy performs compared to an ideal strategy that (somehow) always pulls the best arm.

UCB1

In 2002 Auer et. al. [ACF02] came up with an algorithm that has a regret of

$$R_n = \left(\sum_{i:\mu_i<\mu^*} \frac{8}{\mu^* - \mu_i} \right) \ln n + \left(1 + \frac{\pi^2}{3} \right) \left(k\mu^* - \sum_{i=1}^{k} \mu_i \right). \tag{3.3}$$

This algorithm is thus *asymptotically optimal*, as it achieves a regret within a constant factor of the lower bound (3.2). This algorithm named UCB1 (since it gives *upper confidence bounds* for μ^*) works as follows. For each arm i we have $\bar{x}_i$, the average reward of that arm, and n_i, the total number times that arm was pulled. Then the UCB or upper confidence bound of that arm is

$$U_i = \bar{x}_i + \sqrt{\frac{2 \ln n}{n_i}}. \tag{3.4}$$

The strategy is then to simply compute U_i for each arm and pull $\arg\max_i U_i$. When n_i is zero we define $U_i = \infty$, and thus the algorithm always pulls unexplored arms first. In practice often a variant of the UCB1 formula is used,

$$U_i = \bar{x}_i + C_p \sqrt{\frac{\ln n}{n_i}}, \tag{3.5}$$

where C_p is a parameter used to tweak the exploitation-exploration trade-off for the problem at hand.

3.2 Algorithm

Finally we can introduce *Monte Carlo tree search* (MCTS) [Cou07]. As said before, MCTS is an algorithm used to find a good move in a game. The algorithm is an *anytime* algorithm—it can run either for a specified amount of time, or simply until it is stopped, after which it will suggest a move. It does this by repeatedly gathering more and more information in small iterations. In each iteration a complete *playout* occurs: a move sequence is generated from the start of the game until a terminal position. This move sequence is partially informed based on previous iterations, and partially random (from which the name Monte Carlo tree search derives). After the playout the algorithm uses the outcome to update the information it has stored, and starts a new iteration. After the final iteration is done the stored information is used to make a final decision—which move to suggest.

Monte Carlo tree search stores a portion of the complete game tree of a game. Initially it simply stores one node, the game tree root. At each node two integers are stored, w and n. n is simply the total number of times this node was visited during a playout and w is the total number of times these playouts were winning for the player that last made a move.

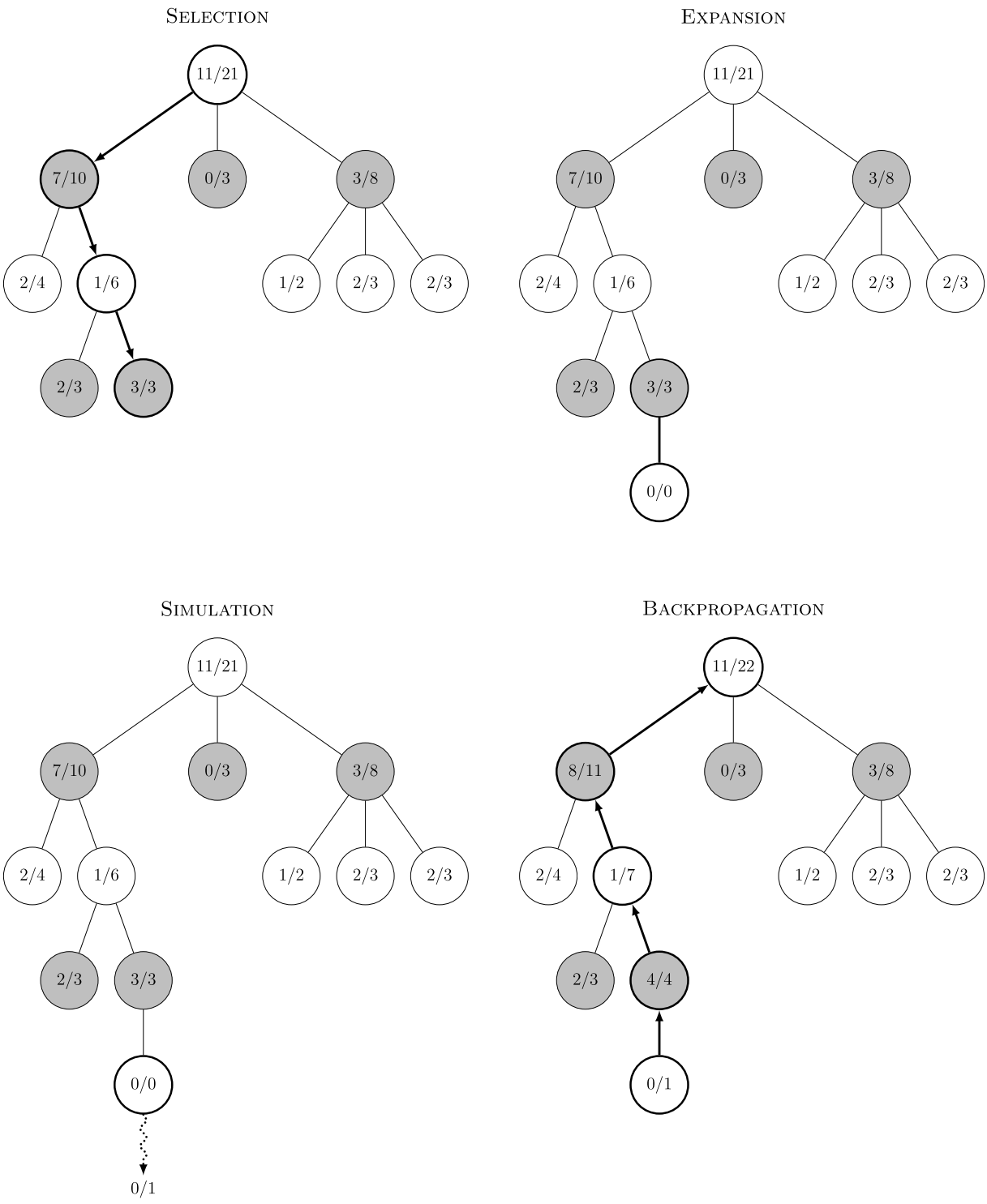


Figure 3.1: Illustration by Wikipedia user *Rmoss92* showing the four steps of one iteration of MCTS: selection, expansion, simulation and backpropagation. In this figure black makes the first move, and ends up winning in this playout. The w/n statistics are updated in the last step, thus incrementing n for each node along the chosen path, but w only for the black nodes.

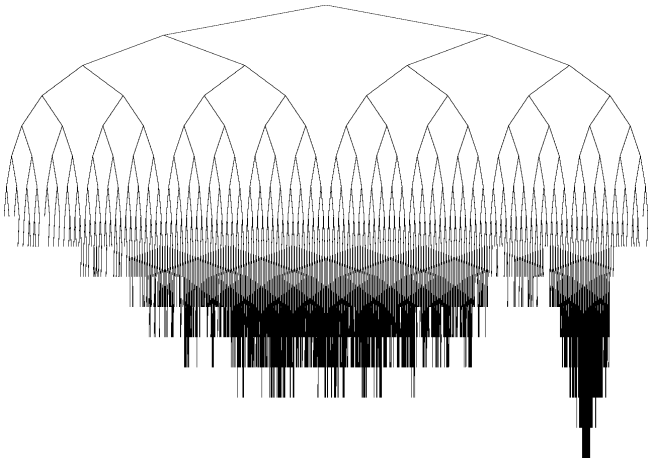


Figure 3.2: Illustration from [CM07] showing asymmetric tree growth exhibited by MCTS.

Then each iteration of MCTS performs four steps:

1. We start at the root of our stored partial tree. We *select* a move (details in Section 3.2.1). If that move corresponds to a child in our partial tree we move to that node and repeat this step, otherwise we
2. *expand* our stored partial tree with a new node corresponding to the selected move, initialized with $w = n = 0$. Then,
3. uniformly *random* moves are chosen until we reach a terminal position. We record which player won and
4. *backpropagate* this information along the nodes from steps 1 and 2, updating their w/n statistics.

This process is visualized in Figure 3.1. Note how in each iteration we (assuming we didn't hit an early terminal node) expand once, adding one node to our growing partial game tree. MCTS thus requires space linear in the number of iterations.

3.2.1 Selection

The first step in MCTS is to do selection. Starting at the root, we repeatedly select moves and walk down our partial game tree until we select a move that doesn't have a corresponding node in our partial game tree or until we hit a terminal node. In the event we hit a terminal node we skip the next two steps (expansion and simulation) and go straight to backpropagation (Section 3.2.4).

The true strength of MCTS comes from the selection rule. It decides based on the information from previous iterations what is considered an interesting move to explore. If the result was a win, it will be more likely to explore that option again, if not, less likely. This means that even without some sort of a priori policy guiding its decisions, it will automatically focus on the 'more interesting' portions of the game tree, and spend relatively little time on moves that are quickly shown to be poor. This is visualized in Figure 3.2.

UCT

The first description of MCTS [Cou07] used an ad-hoc rule to select moves. Shortly after Kocsis and Szepesvári introduced a more justified rule that became the standard for many MCTS

implementations: *UCB for trees* or just UCT in short. The core idea is to reinterpret the MCTS selection problem as a (recursive) multi-armed bandit problem and use the UCB1 formula (3.5).

In this interpretation we view each possible move as an arm in the multi-armed bandit. If we pull an arm we are choosing a move, and we may win or we may not after that, depending on the random moves chosen in the simulation step (Section 3.2.3). This is thus a random binary variable X_i. By storing and updating the w and n statistics in each node in the backpropagation step we can compute $\bar{x}_i = w_i/n_i$. Then $\bar{x}_i$ is exactly our current belief on the expected chance of winning after picking move i, just from our stored past experience. Thus we can apply the UCB1 formula as our selection rule—we pick the move

$$\arg\max_i \ \frac{w_i}{n_i} + C_p\sqrt{\frac{\ln \sum_i n_i}{n_i}}, \tag{3.6}$$

which as we saw earlier balances between exploration of lesser known conjectured poor moves and deeper exploitation of better known conjectured good moves automatically, no matter how often we sample.

Note that if any $n_i = 0$, that is, we are selecting a move in a node where a particular move has never been selected before, then that move is always selected before re-selecting a previously explored move. Finally, to reduce any biases in move ordering it is customary to break ties in (3.6) randomly, where all $\{i : n_i = 0\}$ are considered tied among each other.

Finally, despite its real-world success our usage of the UCT1 formula is not fully justified. Its proof of convergence (speed) depends on the random distributions being sampled being *stationary*, but in MCTS they are anything but—each iteration of MCTS changes the underlying distributions. In the recent [SXX20] the authors tackle this problem with a modified UCT-like rule and show that as $n \to \infty$ MCTS converges to the minimax values when using this rule (as well as non-asymptotic convergence results).

3.2.2 Expansion

The expansion step is the simplest of them all. In our previous step we have selected a move, but found that it does not yet have a corresponding node in our partial game tree. So we add one, initializing it with $w = n = 0$.

3.2.3 Simulation

We are now in a leaf of a partial game tree with an associated state. But this leaf is not terminal. In order to find out whether this leaf leads to a winning or a losing game we perform a *simulation*. This can be any process which chooses moves repeatedly for both players until the game ends (and as we've defined our games to be *finite*, this will always happen).

Random rollout

The most common simulation strategy is *random rollout*. It is as simple as can be: it simply chooses moves uniformly at random. It is this step that makes Monte Carlo tree search 'Monte Carlo'[1], at least traditionally [Brü93].

[1]The author would like to argue that the name 'Monte Carlo tree search' may be suboptimal, as MCTS makes complete sense even when using a deterministic simulation step and deterministic tie-breaking in the selection step.

3.2.4 Backpropagation

Recall that each node has an associated player, based on whoever made the move to get to that node. The children of the root node are thus associated with the player who gets to play first, and the associations alternate for each level of the tree afterwards. For consistency the root node is associated with the player who either just made a move, or, at the start of a game, the player who moves second.

Having reached a terminal game state in the simulation step we now know whether Alice or Bob won. As a final step MCTS will now *backpropagate* that information along all the nodes in the partial game tree that lie on the traversed path from the root to this state. For each such node the corresponding n value gets incremented. Assuming that Alice was the winner in the terminal node, we increment w for each node associated with Alice, and vice versa if Bob was the winner. Alternatively, when backpropagating real values (e.g. in AlphaZero) rather than simply wins/losses, a moving average can be stored directly.

3.2.5 Move suggestion

After the last iteration of MCTS is done we have to suggest a move using the information gathered. In particular, we look at w_i, n_i *node statistics* at each of the children of the root.

There are then three main methods [Bro+12b] to make this suggestion: we suggest either the move with the highest *winrate* w_i/n_i, the one maximizing a *lower confidence bound* (a formula similar to UCB) or simply the move that was selected the most times (n_i). It is unclear which strategy is the best; an argument can be made for each of them, as they come from a continuum of choices balancing between strict maximization of the expectation and robustness of the choice through variance.

3.3 Applications

Monte Carlo tree search has many applications for finding strategies in games, as we will discuss here. A focus is on Go, as MCTS has a strong mutual history this game with the first AI successes in Go coming from MCTS and the first applications of MCTS occurring in Go.

3.3.1 Go and other games

Go is a combinatorial game traditionally played on a 19×19 board where players alternate placing stones on the board grid of their own color. When your stones completely surround a group of stones of your opponent, those stones are captured and removed from the board. Whichever player surrounds more empty undisputed territory plus captured stones at the end of the game wins.

Go has traditionally been very hard for computers due to the large branching factor. The first move has 361 options, the second 360 and so forth (minus some illegal moves), so the state space of Go very quickly explodes. There also did not exist a good heuristic evaluation function for Go—it is hard to determine which territory is 'undisputed', and if disputed, which player will manage to conquer it. This means the traditional alpha-beta search engines struggled to beat strong amateur players in 2000 whereas in Chess they were arguably already beyond human performance with the display of Deep Blue.

Mogo

The first public MCTS based engine that follows the structure as presented here is MoGo [GWT06] in 2006. Unlike alpha-beta search engines it saw success on the 9×9 board even against strong amateurs. A boom in computing power and optimization of the engines meant that in 2012 the MCTS-based Go engines could compete with strong amateurs on the full-sized board as well.

AlphaGo, AlphaGo Zero, and AlphaZero

The true explosion in playing strength did not come until DeepMind's efforts in building AlphaGo [Sil+16], AlphaGo Zero [Sil+17b] and AlphaZero [Sil+17a] [Sil+18], the latter being a single algorithm capable of learning to playing Chess, Go and Shogi at a beyond-human level starting from nothing but the game's rules. Their idea was to combine the power of deep-learning and self-play to create a very powerful 'intuition' or heuristic evaluation with the tree exploration power of MCTS to refine this intuition. We cite Silver et. al.:

> "AlphaZero evaluates positions using non-linear function approximation based on a deep neural network, rather than the linear function approximation used in typical chess programs. This provides a much more powerful representation, but may also introduce spurious approximation errors. MCTS averages over these approximation errors, which therefore tend to cancel out when evaluating a large subtree. In contrast, alpha-beta search computes an explicit mini-max, which propagates the biggest approximation errors to the root of the subtree. Using MCTS may allow AlphaZero to effectively combine its neural network representations with a powerful, domain-independent search."

While it would take way too much time to explain the full AlphaZero algorithm, we do want to mention the two crucial variations AlphaZero makes on traditional MCTS. They use a formula similar to (3.5) which is a modified variant inspired by the PUCT ('predictor' UCT) from [Ros11]:

$$U_i = \bar{x}_i + p_i \cdot \frac{\sqrt{\sum_i n_i}}{1 + n_i} \cdot \left[c_{\text{init}} + \log \frac{1 + \sum_i n_i + c_{\text{base}}}{c_{\text{base}}} \right]. \tag{3.7}$$

Everything in square brackets is an exploration 'constant' that scales with the logarithm of the number of visits to the current node, but the interesting part is p_i. This is a prior probability generated by the deep neural network, representing probability of searching child i. Thus the deep neural network guides the search towards good nodes with a prior bias based on its past experience.

The second key difference is that AlphaZero does not use a simulation step at all. Instead, whenever a new node gets added during an expansion step (and it is not already a terminal node) the neural network provides a value evaluation (between -1 and 1), which is then used as the value being backpropagated instead of the simulation result.

3.3.2 General planning

MCTS can not just be used for games, but for other real-world problems too.

Modelling

The idea is very simple: turn the real-world problem into a game. Any sequential problem where there are states, actors that can make choices, rules for how actor's choices evolve the states and certain terminal states with associated evaluations can be modelled as a game. This can include single-player games, where a single person tries to explore a sequence of options to find a satisfactory solution.

An example problem might be the renowned NP-hard Traveling Salesperson Problem. In this problem a salesman is trying to visit a set of cities exactly once, returning to the starting city, minimizing the total distance travelled. This is an important real-world question in e.g. shipping. This problem is difficult to solve optimally, so it's ripe for heuristic methods.

By choosing the set of already visited cities (initially empty) as the 'game state', one can compute the list of 'moves' (travelling to another city) and when hitting the 'terminal game state' (all cities visited), giving an evaluation based on the total distance travelled. At this point one can run MCTS to try and find a good move (sequence), since we have turned the problem into a game. In [NST20] the authors do exactly this for a variant where the salesman is a shipping truck containing drones which can be used to quickly serve cities in a radius around its current location, meaning those cities don't have to be visited.

MuZero

Beyond modelling the problem as a game by describing exactly the states, state transition rules and evaluations by hand, DeepMind extended AlphaZero with a machine learning powered abstraction layer that *learns* the behavior behavior of a system to create MuZero [Sch+20]. The rules of the game are not embedded in the algorithm, instead, the algorithm learns how actions evolve the state and then uses this to 'play a game in its head' with MCTS, strictly 'imagining' the moves and what their results would be. The exact same algorithm was used to not only learn Chess, Shogi, Go but 57 real-time Atari games as well from nothing but reinforcement (e.g. being told if the game was a win or a loss). With such a diversity in strategies needed it is thus likely to be useful for planning in almost any sort of environment with states, actions and rewards.

Chapter 4

Quantum improvements

4.1 Criteria & desiderata

Before being able to claim any method is a (quantum) improvement, we must first define our criteria for a valid method, and what constitutes an improvement. This is inherently a bit subjective, as MCTS is more a family of algorithms with a wide variety of subtle (or not so subtle) changes than a specific well-defined algorithm.

So while recognizing its subjectivity we approach the problem with two main criteria in mind:

- Any quantum algorithm must be a direct equivalent to a 'reasonable' MCTS variant on a classical computer. That is, it must compute the same result, or draw results from the same probability distribution. Unfortunately we do not see a way to avoid the subjective nature of what exactly is a 'reasonable' MCTS variant and will simply assume good faith.

- Any quantum algorithm must be effectively realizable in time and space and the analysis to show this must be complete. Thus an improvement in one aspect that would result in a time/space blowup in an other in any real-world instance is unacceptable, nor is one that neglects a portion of the analysis such as circuit size or approximation errors. Finally, to be 'effectively realizable' any quantum gates used must be reasonably available or efficiently approximable on currently available quantum machines. For the purposes of this thesis we assume the gates listed in Section 2.1.4 are efficiently realizable, but ultimately this depends on the hardware used.

With this in mind, we define a quantum algorithm an improvement if it matches the above criteria and has an asymptotic advantage in either time or space over the best equivalent classical algorithm for at least a family of inputs/parameters, and preferably all possible inputs/parameters.

4.2 Parallel MCTS

Quantum computers have, in the right circumstances, the capability of performing many computations seemingly in parallel through the use of superposition, allowing them to answer very particular queries about their results, as we'll see in Chapter 5. For this reason we take a look at some of the parallelizations of MCTS described in [Bro+12a] section 6.3, to look for possible quantum improvements.

4.2.1 Leaf parallelization

In *leaf parallelization* (only) the simulation step of MCTS is parallelized.

k-MCTS

In k-MCTS we replace the simple simulation step with the mean outcome of k simulation steps. This is studied in [CJ07] [CWH08]. Both studies are rather limited, but found that performing 8 simulations in each iteration of MCTS was a significant improvement compared to just a single simulation. In Chapter 6 we study how quantum computers can more efficiently estimate the mean outcome of a simulation step than classical k-MCTS does.

∞-MCTS

However, we argue that leaf parallelization is very much limited in the sense that adding more and more computational power will not get you significant returns, at least for the traditional random rollouts. For that we consider the hypothetical ∞-MCTS, an algorithm that would get you the exact mean behavior of the simulation step.

The problem is that ultimately, the random agent isn't very good. It's trivial to construct counterexamples where the random agent reports arbitrarily small winrates for a game that is actually a guaranteed win with optimal play. E.g. consider a long path along a cliff with a pot of gold at the end, where the two move options are 'take a step forward' and 'jump into the cliff'. Thus even if you invested an arbitrarily high amount of computation into estimating the exact mean behavior of the random agent, it would not result in an arbitrary improvement of MCTS.

Nevertheless, the methods developed in Chapter 6 are interesting, and we explore other uses for them in the corollary Chapter 8.

4.2.2 Root parallelization

Root parallelization is arguably the simplest method of parallelizing MCTS: you simply run k instances of MCTS in parallel and adapt the move suggestion step to take into account the statistics of all instances. Due to the stochastic nature of MCTS each instance's statistics of the children of the root are random variables sampling from the same distribution. If MCTS were to be deterministic (e.g. using a deterministic simulation step), then the variance would be zero and root parallelization would simply not provide any improvement.

In [SKW10] this strategy is studied in detail, with two move suggestion strategies evaluated: majority voting and best average. In *majority voting* each of the k MCTS instances votes for a particular best move independently and the majority is accepted, in *best average* all k statistics for each move option are averaged, and then a single decision is made using these averages. The authors found that the tree parallelization performed better with the same amount of computation. However, in an earlier study [CWH08] root parallelization performed better, using best average. In Chapter 7 we explore a quantum method for root parallelization using best average.

4.2.3 Tree parallelization

A natural classical method for parallelization is *tree parallelization* [Mir+18]. In this method, unlike root parallelization, multiple computational threads perform the MCTS algorithm, sharing the same tree containing the node winrate statistics.

In order to prevent each thread from following the exact same path down the tree during the selection step (which would effectively reduce it to leaf parallelization), a common strategy is to split up the backpropagation. During the selection step, after the UCT formula is evaluated, the algorithm immediately increments the number of plays of a node, instead of doing that during backpropagation. Then the algorithm proceeds as normal, and during backpropagation (if the simulation was a win), the win counter gets incremented. This effectively means that any node selected during the selection step is counted as a temporary loss (also known as *virtual loss* [Mir+17]), discouraging other threads from selecting this exact same path.

Unfortunately, this method is not immediately compatible with quantum parallelism due to the inherent use of shared mutable memory, so we did not investigate it further.

4.3 Obstacles

4.3.1 Iterative nature

After much exploration we did not manage to find other avenues for quantum computers to speed up MCTS than parallelism. We identified the iterative nature of MCTS as a main obstacle for this. In each iteration of MCTS information about the game is gathered, which improves the quality of the information gathered in future iterations. There exists a long 'dependency chain' of information, where decisions in the kth iteration ultimately rely on the results of each iteration before it.

We don't know if this limitation is fundamental or not. Quantum algorithms exist for answering questions about other seemingly iterative processes, so we do not rule out that smarter people will overcome this obstacle.

4.3.2 Linear memory

A more fundamental limitation is that of memory. If one wishes to perform n iterations of MCTS, one needs $O(n)$ memory to store the winrate statistics of each node created in the expansion step. This is not a limitation specific to quantum MCTS, classical MCTS also suffers from this. However, at the time of writing there appear to be little to no obstacles to scaling memory for classical computers, but physical implementations of quantum computers are very much limited in the number of qubits they have available.

Chapter 5

Quantum primitives

In this chapter we will discuss some more advanced quantum computing algorithms that we will use as primitive operations in our results.

5.1 (ϵ, δ) query complexity

Before discussing the primitives themselves we want to spend some time emphasizing how the computational complexity of many quantum primitives is measured. Since most quantum algorithms are probabilistic, it would not be enough to simply state a runtime. Furthermore, most quantum algorithms are not just probabilistic, they are approximations as well, so a probability of success plus runtime is not enough either. This leads rise to (ϵ, δ) complexity, which is a method to quantify the runtime algorithms that are probably approximately correct. There is nothing inherently quantum about this, and in fact the same measure can be applied to classical randomized approximation algorithms.

In this notion the quantum algorithm *succeeds* with probability $1 - \delta$. We say that δ is the *failure probability*. When an algorithm does not succeed its output is generally undefined and may be arbitrarily bad. When an algorithm does succeed and approximates the correct answer μ we say that the algorithm outcome $\tilde{\mu}$ must satisfy

$$|\mu - \tilde{\mu}| \leq \epsilon. \tag{5.1}$$

This is known as *absolute error*. There is another common metric of error, *relative error*, giving

$$|\mu - \tilde{\mu}| \leq \epsilon|\mu|. \tag{5.2}$$

Why would anyone want an algorithm that can fail? Obviously no one likes failure, but given access to a probabilistic algorithm that succeeds at least half the time we can (with some overhead) make the failure chance arbitrarily low.

Lemma 5.1 (Powering lemma[1])**.** *Let $\mathcal{A}$ be a probabilistic algorithm giving estimate $\tilde{\mu}$ of constant μ satisfying $|\mu - \tilde{\mu}| \leq c$ with probability $\frac{1}{2} + \gamma, \gamma > 0$. Then, if we take $O\left(\frac{\log 1/\delta}{\gamma^2}\right)$ independent estimates of μ using $\mathcal{A}$ their median m will satisfy $|\mu - m| \leq c$ with probability at least $1 - \delta$.*

Proof. Let $X = X_1 + X_2 + \cdots + X_n$ be the sum of independent Bernoulli random variables with $P[X_i = 1] = p$. Then,

$$P[X \geq k] = \sum_{i=k}^{n} \binom{n}{i} p^i (1-p)^{n-i}, \tag{5.3}$$

[1]We prove an extension of [JVV86], which assumes the success probability of $\mathcal{A}$ is at least $\frac{3}{4}$.

which for $np < k < n$ and $p < \frac{1}{2}$ the Chernoff-Hoeffding theorem [Hoe63] bounds as

$$P[X \geq k] \leq \exp\left(-\frac{(k-np)^2}{n}\frac{1}{1-2p}\log\frac{1-p}{p}\right). \tag{5.4}$$

Note that if we perform $n = 2k+1$ independent estimates using $\mathcal{A}$ giving $\tilde{\mu}_1, \ldots, \tilde{\mu}_n$ and at least $k+1$ satisfy $|\mu - \tilde{\mu}_i| \leq c$ then the median of them must also satisfy this inequality. Thus if we output the median as our robust estimate we only have a chance to fail if at least $k+1$ estimates fail. Now let $X_i = 1 \iff |\mu - \tilde{\mu}_i| > c$ (giving $p = \frac{1}{2} - \gamma$) and we can bound our failure probability δ as

$$\delta \leq P[X \geq k+1] \leq \exp\left(-\frac{(k+1-np)^2}{n}\frac{1}{1-2p}\log\frac{1-p}{p}\right). \tag{5.5}$$

We can relax the inequality by substituting $(k+1-np)^2$ with $\left(\frac{1}{2}n - np\right)^2$ simplifying to

$$\delta \leq \exp\left(-\frac{n}{4}(1-2p)\log\frac{1-p}{p}\right). \tag{5.6}$$

Now we can solve for n, substituting $p = \frac{1}{2} - \gamma$, and thus using

$$n \geq \left\lceil 2 \cdot \frac{\log 1/\delta}{\gamma} \cdot \frac{1}{\log(1+2\gamma) - \log(1-2\gamma)} \right\rceil, \tag{5.7}$$

estimates suffices. As $\frac{1}{2}(\log(1+x) - \log(1-x)) = \tanh^{-1}(x) \geq x$ for $0 < x < \frac{1}{2}$ we find that the right-hand denominator is bigger than 4γ and we can relax the inequality and simplify to

$$n \geq \left\lceil \frac{\log 1/\delta}{2\gamma^2} \right\rceil. \tag{5.8}$$

Thus $O\left(\frac{\log 1/\delta}{\gamma^2}\right)$ estimates suffices as $\delta, \gamma \to 0$.

□

Using the above lemma with either $c = \epsilon$ or $c = \epsilon|\mu|$ shows that probabilistic algorithms providing either relative or absolute estimates can be made to fail with an arbitrarily small chance as long as the original algorithm fails less than half the time.

5.2 Mean estimation

The work in this thesis heavily relies on the basic quantum mean estimation procedure described in [Mon15] by Ashley Montanaro.

Let $\mathcal{A}$ be a quantum circuit that when applied to $|0^n\rangle$ prepares a state of interest and let $\phi(x) : \{0,1\}^k \to [0,1]$ be a function mapping some measurement x to a real value. Then $v(\mathcal{A})$ is a random variable with a distribution determined by preparing $\mathcal{A}|0\rangle$, observing the last k qubits as x and outputting $\phi(x)$. We define W as a unitary on $k+1$ qubits as

$$W|x\rangle|0\rangle = |x\rangle\left(\sqrt{1-\phi(x)}|0\rangle + \sqrt{\phi(x)}|1\rangle\right). \tag{5.9}$$

Then if we perform quantum amplitude estimation (Section 2.2.3) using unitary

$$U = (I \otimes W)(\mathcal{A} \otimes I) \tag{5.10}$$

and projector $P = I \otimes |1\rangle\langle 1|$ we get an estimate of $E[v(\mathcal{A})]$. We are thus able to estimate the mean of a function bounded between 0 and 1 over an arbitrary superposition of inputs (if we have a circuit preparing that superposition from $|0^n\rangle$).

Theorem 5.2. *[Mon15] Let $\mu = E[v(\mathcal{A})]$. Using $O(t \log 1/\delta)$ invocations of $U, U^\dagger$ each we can get estimate $\tilde{\mu}$ such that*

$$|\tilde{\mu} - \mu| \leq 2\pi \frac{\sqrt{\mu(1-\mu)}}{t} + \frac{\pi^2}{t^2} \tag{5.11}$$

with probability at least $1 - \delta$.

Proof. We apply the above procedure. We note that

$$A|0^n\rangle = \sum_x \alpha_x |\psi_x\rangle|x\rangle \tag{5.12}$$

and if we apply U to the all-zero state we get

$$|\psi\rangle = (I \otimes W)(A \otimes I)|0^{n+1}\rangle = \sum_x \alpha_x |\psi_x\rangle|x\rangle \Big(\sqrt{1-\phi(x)}|0\rangle + \sqrt{\phi(x)}|1\rangle \Big), \tag{5.13}$$

and thus the quantum amplitude estimation estimates

$$\langle\psi|P|\psi\rangle = \sum_x |\alpha_x|^2 \phi(x) = E[v(\mathcal{A})] = \mu. \tag{5.14}$$

From Theorem 2.1 we get bound (5.11) using t invocations of $U, U^\dagger$ with a fixed failure probability of $\pi^2/8$. With Lemma 5.1 we boost this to an arbitrary probability $1 - \delta$ giving a total complexity $O(t \log 1/\delta)$. □

Corollary 5.2.1. *Using $t = O(1/\epsilon)$ we can estimate $\mu = E[v(\mathcal{A})]$ up to absolute error ϵ.*

Proof. We note that $\mu \leq 1$, implying $\sqrt{\mu(1-\mu)} \leq \frac{1}{2}$, and thus (5.11) can be relaxed to

$$|\tilde{\mu} - \mu| \leq \frac{\pi}{t} + \frac{\pi^2}{t^2}. \tag{5.15}$$

Substituting $t = \frac{\pi}{c\epsilon}$ gives us

$$|\tilde{\mu} - \mu| \leq c\epsilon + c^2\epsilon^2. \tag{5.16}$$

Finally we note that only the domain $\epsilon < \frac{1}{2}$ is sensible (otherwise one can trivially output $\tilde{\mu} = \frac{1}{2}$), and thus $\epsilon^2 \leq \frac{1}{2}\epsilon$ and $c = \sqrt{3} - 1$ as the solution to the equation $c + \frac{1}{2}c^2 = 1$ suffices. □

Corollary 5.2.2. *Using $t = O(1/(\epsilon\sqrt{\mu}))$ we can estimate $\mu = E[v(\mathcal{A})]$ up to relative error ϵ.*

Proof. Substituting $t = \frac{\pi}{c\epsilon\sqrt{\mu}}$ into (5.11) gives us

$$|\tilde{\mu} - \mu| \leq 2c\epsilon\mu\sqrt{1-\mu} + c^2\epsilon^2\mu \leq \big(2c\epsilon + c^2\epsilon^2\big)\mu, \tag{5.17}$$

and thus by similar logic to the previous corollary we find that $c = \sqrt{6} - 2$ suffices. □

5.3 Best bandit selection

We saw the multi-armed bandit problem earlier in Section 3.1.3. Using quantum computers we can solve a similar problem we'll dub *quantum bandits*. In this problem we have n arms, each with an unknown probability of success p_i, and wish to identify the arm with the highest chance of success.

To help place the multi-armed bandit problem in an (ϵ, δ) context we first modify the problem statement. Our goal is to identify arm i such that $|p^* - p_i| \leq \epsilon$ with probability $1 - \delta$, where $p^* = \max_i p_i$. In [GGL12] the authors showed this problem can be solved classically using a query complexity of

$$O\left(\sum_{i=1}^{n} \min\left\{\epsilon^{-2}, \Delta_i^{-2}\right\} \cdot \log\left(\frac{n}{\delta \Delta_2}\right)\right), \tag{5.18}$$

where $\Delta_i = p_1 - p_i$, assuming without a loss of generality that p_i is sorted in descending order.

Suppose we have a quantum oracle implementing these bandits in the form of the unitary

$$U|i\rangle|0\rangle|0\rangle = |i\rangle\left(\sqrt{1-p_i}|u_i\rangle|0\rangle + \sqrt{p_i}|v_i\rangle|1\rangle\right) \tag{5.19}$$

where u_i, v_i are arbitrary states and p_i is the probability of success of bandit i. With this oracle we can perform the algorithm from [Wan+20] to identify the best arm faster than (5.18). The algorithm is quite technical, but at a very high level the algorithm uses quantum amplitude estimation to count the number of arms that have $p_i > l$ for some l. Using binary search they find an appropriately large value of l with a confidence interval dictated by ϵ. Then using quantum amplitude amplification based on l they find such an i. The authors actually use *variable-time* amplitude estimation/amplification, allowing the overall complexity scale with Δ_i and ϵ.

Theorem 5.3. *Given quantum oracle U we can find arm i such that $|p^* - p_i| \leq \epsilon$ with probability $1 - \delta$ with query complexity*

$$O\left(\sqrt{\min\left\{\frac{n}{\epsilon^2}, \sum_{i=2}^{n} \Delta_i^{-2}\right\}} \cdot \text{poly}\left(\log\left(\frac{n}{\delta \Delta_2}\right)\right)\right). \tag{5.20}$$

Proof. See Corollary 1 in [Wan+20]. □

This provides roughly a quadratic asymptotic speedup, as the polylogarithmic term (as opposed to just logarithmic in (5.18)) for a constant failure probability δ gets dominated by the square root term.

Chapter 6

Mean estimation in QMCTS

In this chapter we'll describe the various strategies we've explored to apply mean estimation to quantum Monte Carlo tree search. All of them apply on the same principle: enhancing the simulation step of MCTS.

It's a common optimization [Bro+12a] to perform not one random rollout during the simulation step, but rather k random rollouts, averaging over their outcomes. This variant, which we called k-MCTS in Section 4.2.1, has a trade-off. It uses a factor k more computation (in the simulation step), but in return gets less variance in the winrate given by the simulation. Note that this doesn't have to mean MCTS becomes k times slower overall, as we can use *parallelism* to perform many simulations at the same time. In this chapter we'll explore *quantum parallelism* to sample from the same random process that the simulation step in k-MCTS samples from, except quadratically faster.

In order to facilitate this, each algorithm in this chapter makes the same assumption: we know h (*maximum tree height* minus one), an upper bound on the maximum number of moves in a game or equivalently a bound on the depth of the game tree minus one, and b (*maximum branching factor*), an upper bound on the number of actions in any node. In addition to this, we assume that the game tree is extended from terminal positions with line graphs until each path from the root to a terminal position has exactly h steps. This can be done with a trivial modification of the transition function δ and associated actions A (see Section 3.1.1), by only allowing each player to pass once a terminal position has been reached, until h moves have been played in total.

6.1 Ratio estimation

Our first approach to speed up random rollouts was to embed the game tree inside a larger power-of-two sized game tree amenable to quantum computing, and to directly estimate the number of winning leaves in this tree by only counting the subset of the embedding that is found in the original tree. However, this was subtly wrong. Consider the following two random processes:

1. Select a random move with uniform probability at each node, until a terminal position is reached. Return whether this terminal position is winning for Alice or not.

2. Select a random valid move sequence from root to a terminal node uniformly at random from all such move sequences. Return whether the terminal position is winning for Alice or not.

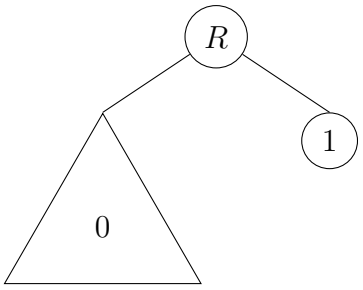


Figure 6.1: Illustration of an unbalanced tree. The left child of root R is a large subtree in which all paths lead to losing terminal states. The right child is immediately terminal, and winning. A random rollout from R has a winrate of 50%, whereas choosing a valid move sequence at random from all valid move sequences has a winrate of $1/N$ where N is the number of terminal nodes, which can be arbitrarily small. This illustrates the difference between estimating the random rollout winrate probability, and the probability that a leaf is winning.

We initially (mistakingly) assumed that these two random processes sample from the same underlying distribution. They do for complete binary trees, but they do not in general, a counterexample is shown in Figure 6.1.

Despite this we will present the result here, even if not applicable to MCTS in general. The method is valid and achieves a speedup over a classical algorithm, it simply computes the wrong quantity for our main problem. First we require a lemma and then we show the method.

6.1.1 Relative error of ratios

Lemma 6.1. *If one can estimate quantities $r, s > 0$ each once, independently, with respective relative errors $\epsilon_r, \epsilon_s < 1$ and individual failure probability $\delta/2$, it's possible to estimate r/s with failure probability δ and relative error $\epsilon = (\epsilon_r + \epsilon_s)/(1 - \epsilon_s)$.*

Proof. Let $\hat{r}, \hat{s}$ denote our observed estimations for r, s and let our estimate of r/s be simply $\hat{r}/\hat{s}$. With probability $1 - \delta/2$ we have

$$|r - \hat{r}| \leq \epsilon_r \cdot r \quad \Longrightarrow \quad (1 - \epsilon_r)r \leq \hat{r} \leq (1 + \epsilon_r)r, \tag{6.1}$$

and similarly for $s, \hat{s}, \epsilon_s$. Since $(1 - \delta/2)^2 \geq 1 - \delta$ holds trivially we can safely assume both inequalities hold simultaneously, as our estimation error probabilities are independent.

We must show that our relative error is at most ϵ, that is

$$\left| \frac{r}{s} - \frac{\hat{r}}{\hat{s}} \right| \leq \epsilon \cdot \frac{r}{s}. \tag{6.2}$$

Note that (6.1) implies $u = \frac{(1-\epsilon_r)r}{(1+\epsilon_s)s}$ is the smallest possible underestimation, and $o = \frac{(1+\epsilon_r)r}{(1-\epsilon_s)s}$ is the biggest possible overestimation. Therefore if we can show that

$$\frac{r}{s} - u \leq \epsilon \cdot \frac{r}{s} \quad \text{and} \quad o - \frac{r}{s} \leq \epsilon \cdot \frac{r}{s} \tag{6.3}$$

both hold, then Equation 6.2 must also hold and we are done. Dividing by r/s and some basic simplification gives us

$$\frac{\epsilon_r + \epsilon_s}{1 + \epsilon_s} \leq \epsilon \quad \text{and} \quad \frac{\epsilon_r + \epsilon_s}{1 - \epsilon_s} \leq \epsilon \tag{6.4}$$

The right-hand inequality is tighter thus if we choose $\epsilon = (\epsilon_r + \epsilon_s)/(1 - \epsilon_s)$ our relative error bound is valid. □

Corollary 6.1.1. *If one can estimate quantities $r, s > 0$ each once, independently, with relative error $\epsilon/3$ and individual failure probability $\delta/2$, it's possible to estimate r/s with failure probability δ and relative error $\epsilon < 1$.*

Proof. Choosing $\epsilon_r = \epsilon_s = \epsilon/3$ for the above lemma gives relative error bound

$$\frac{\epsilon/3 + \epsilon/3}{1 - \epsilon/3} = \frac{\epsilon + \epsilon}{3 - \epsilon} < \frac{\epsilon + \epsilon}{3 - 1} = \epsilon. \tag{6.5}$$

□

6.1.2 Winning move sequences through double estimation

There is a fairly trivial quantum algorithm for estimating the probability a move sequence chosen uniformly at random is winning if the game tree is a complete binary tree. We can create a unitary T that uses transition function δ to apply a certain move choice a to a state,

$$T : |s\rangle|a\rangle|0\rangle|0\rangle \mapsto |s\rangle|a\rangle|g\rangle|\delta(s, a)\rangle, \tag{6.6}$$

where g is some unspecified garbage dependent on s, a in order to make the mapping reversible. We can apply unitary T h times to construct a path from a root state $|R\rangle$ to a terminal state for some sequence of choices $|a_1\rangle, |a_2\rangle, \ldots, |a_h\rangle$. If we define $s_0 = R$ and $s_i = \delta(s_{i-1}, a_i)$ we can describe this process as unitary T^*,

$$T^* : |R\rangle|a_1\rangle \cdots |a_h\rangle|0\rangle \mapsto |s_0\rangle|a_1\rangle|g_1\rangle|s_1\rangle \cdots |s_{h-1}\rangle|a_h\rangle|g_h\rangle|s_h\rangle. \tag{6.7}$$

If the game tree is a complete binary tree we can use a single qubit to represent a_i and initialize $|a_i\rangle = \frac{1}{2}(|0\rangle + |1\rangle)$ using Hadamard gates, creating a uniform superposition over all paths from root to leaf,

$$|\psi\rangle = T^* \left(I \otimes H^{\otimes h} \otimes I\right) |R\rangle|0^h\rangle|0\rangle. \tag{6.8}$$

We can now use Corollary 5.2.2, choosing ϕ to be a Boolean function that determines whether s_h is winning for Alice. This gives us an estimate of the probability w of her winning in a move sequence chosen uniformly at random with a relative error ϵ, failure probability δ, using $O\left(\frac{\log 1/\delta}{\epsilon\sqrt{w}}\right)$ queries to T^* and ϕ.

To extend this result, we *embed* any tree into a binary tree. We represent any move choice a_i using a $\lceil \log_2 b \rceil$ qubit register. Note that we can then still initialize $|a_i\rangle$ to a uniform superposition using Hadamard gates, but this may result in spurious non-existent moves when $a_i > |A(s_i)|$. Nevertheless, we build our state $|\psi\rangle$ as before, by extending δ's behavior to be arbitrary but consistent when asked to perform a non-existent move.

We define that a certain move sequence $a_1, a_2, \ldots, a_h$ is *valid* when $\forall_i(a_i \leq |A(s_i)|)$ and note that we can compute this directly. This, in combination with a circuit that computes whether s_h is winning for Alice allows us to compute whether a certain move sequence as in (6.7) is winning *and* valid. This allows us to estimate the probability that a valid move sequence is winning as

$$P(W \mid V) = \frac{P(W \cap V)}{P(V)}. \tag{6.9}$$

Theorem 6.2. *Let v be the probability that a path from root to leaf in a game tree embedded in a binary tree is valid. Using $O\left(\frac{\log 1/\delta}{\epsilon\sqrt{wv}}\right)$ queries to T^*, a Boolean function ϕ_w determining whether s_h is winning for Alice, and a Boolean function ϕ_v determining whether $s_0, a_1, s_1, \ldots, a_h, s_h$*

is a valid action/move sequence, we can get an estimate of the probability w that a valid move sequence chosen uniformly at random is winning for Alice, with relative error ϵ and failure probability δ.

Proof. The ratio of two estimates from Corollary 5.2.2 with respectively $\phi_{wv}(x) = \phi_w(x)\phi_v(x)$ and ϕ_w on the state $|\psi\rangle$ have the right expectation as per (6.9) and the right asymptotic (ϵ, δ) query complexity as per Corollary 6.1.1. □

There are two issues with this approach. First, as mentioned before, it estimates a different quantity than random rollout, but nevertheless one that might be useful as a heuristic for MCTS. The bigger problem is that for trees where the number of options in each node is not a power of two it requires exponential time complexity, hidden in the variable v. E.g. even if our game tree has 15 options in each node, you would still find that the chance a randomly generated path from root to leaf is valid is $v = (15/16)^h$; rapidly approaching zero as the tree gets deeper. We fix both issues in our next approach.

6.2 Overcounting estimation

In a proper random rollout the probability of any specific path occurring is determined by the number of move options available on its path. If at each node there are many other options available, the chance of picking this specific path is very low, whereas if there are few if any choices along the path, a random rollout will be likely to choose that path.

To account for this and create a binary tree embedding simultaneously we will duplicate each move option at a node many times until the number of options is a power of two. Doing this carefully so that each option is duplicated roughly the same number of times we get a binary tree embedding on which random rollouts closely resemble rollouts on the original tree. By carefully keeping track of how many times each node was duplicated (or if you were enumerating all paths, how many times you are *overcounting* that path), we can correct any biases introduced in the embedding and get an accurate estimate of the random rollout winrate.

6.2.1 Random rollout winrate

Let $\mathbf{a} = (a_1, a_2, \ldots, a_h)$ represent a series of choices (or *actions*) forming a path from the root of the game tree to a terminal leaf. Let the set $\mathcal{P}$ represent all such possible paths $\mathbf{a}$. Let the function $S(\mathbf{a}, i)$ represent the state reached after i steps along the path $\mathbf{a}$,

$$\begin{aligned} S(\mathbf{a}, 0) &= R, \\ S(\mathbf{a}, i) &= \delta(S(\mathbf{a}, i-1), a_{i-1}). \end{aligned} \tag{6.10}$$

We define $W(\mathbf{a})$ as the Boolean function indicating whether the final leaf node of path $\mathbf{a}$ ($S(\mathbf{a}, h)$) is winning for Alice, and

$$M(\mathbf{a}, i) = |A(S(\mathbf{a}, i-1))|, \tag{6.11}$$

the number of move options that were available for the ith step along the path $\mathbf{a}$. Then, the probability that a specific path $\mathbf{a}$ is chosen during random rollout is

$$p(\mathbf{a}) = \frac{1}{M(\mathbf{a}, 1)} \cdot \frac{1}{M(\mathbf{a}, 2)} \cdot \cdots \cdot \frac{1}{M(\mathbf{a}, h)} = \frac{1}{\prod_{i=1}^{h} M(\mathbf{a}, i)}. \tag{6.12}$$

This allows us to calculate the overall probability of a random rollout being winning for Alice as

$$w = \sum_{\mathbf{a} \in \mathcal{P}} W(\mathbf{a})\, p(\mathbf{a}). \tag{6.13}$$

6.2.2 Embedding in complete binary tree

As we did before, we shall embed our paths in a binary tree (or at least a tree where each node has the same number of children, and that number being a power of two). We do this by choosing $\mathbf{b} = (b_1, b_2, \ldots, b_h)$ as a series of k-bit integers, and extending transition function δ to interpret the move choice as a zero-indexed offset modulo the number of move choices,

$$\delta'(s, b) = \delta\big(s, 1 + (b \bmod |A(s)|)\big), \tag{6.14}$$

as opposed to simply letting invalid moves be undefined but consistent behavior. We also redefine S, M, W, and p to respect this extension.

As an example, for $M(\mathbf{b}, i) = 5$ and $k = 4$ we would find that the 16 possible states of the move register b_i would map to actions

$$b_i \equiv (0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0)_{b_i}. \tag{6.15}$$

We say that the *overcounting factor* of x is the number of states of the move register that are equivalent to x. In the above example the overcounting factor of 0 is four, and all the other states have overcounting factor three. We define $F(\mathbf{b}, i)$ as the overcounting factor of b_i, or to be specific

$$F(\mathbf{b}, i) = \left\lfloor \frac{2^k}{M(\mathbf{b}, i)} \right\rfloor + \begin{cases} 1 & \text{if } b_i \bmod M(\mathbf{b}, i) < 2^k \bmod M(\mathbf{b}, i) \\ 0 & \text{otherwise} \end{cases}. \tag{6.16}$$

We extend F to be defined on just a path as well, being the product of the overcounting factors of each move along its path:

$$F(\mathbf{b}) = \prod_{i=1}^{h} F(\mathbf{b}, i). \tag{6.17}$$

Now $F(\mathbf{b})$ represents the number of times you would encounter a path equivalent to $\mathbf{b}$ in the original game tree if you were to enumerate all possible 2^{kh} paths in the extended binary tree.

6.2.3 Overcounting correction

Now that we have F we can use it to correct our overcounting and compute w exactly as

$$w = \sum_{\mathbf{b} \in \{0,1\}^{kh}} \frac{W(\mathbf{b}) p(\mathbf{b})}{F(\mathbf{b})}. \tag{6.18}$$

To make it amenable to quantum computation, we note that any sum can be expressed as the mean over its terms scaled by the number of terms,

$$w = \frac{1}{2^{kh}} \sum_{\mathbf{b} \in \{0,1\}^{kh}} \frac{2^{kh} W(\mathbf{b}) p(\mathbf{b})}{F(\mathbf{b})}. \tag{6.19}$$

In principle this mean be computed by Theorem 5.2, since the superposition over all possible $\mathbf{b}$ is trivial to prepare using Hadamard gates and W, p, and F are all computable by classical circuits (trivially so if preparing a state like (6.8) first, also keeping track of the number of move options at each node along the path).

Theorem 6.3. *We can estimate the probability w that Alice is the winner after a random rollout using $O\left(\frac{\log 1/\delta}{\epsilon\sqrt{w}}\right)$ or $O\left(\frac{\log 1/\delta}{\epsilon}\right)$ queries to W, p, F for respectively relative error or absolute error ϵ with failure probability δ when $k = O(\log b + \log h)$.*

Proof. Naively we could simply use Theorem 5.2 and be done. However there is one caveat: the function we're computing the mean over must be bounded to $[0, 1]$. So to check, we let

$$\phi(\mathbf{b}) = \frac{2^{kh} W(\mathbf{b}) p(\mathbf{b})}{F(\mathbf{b})}, \tag{6.20}$$

and find

$$\phi(\mathbf{b}) = W(\mathbf{b}) \cdot \frac{2^{kh}}{\prod_{i=1}^{h} M(\mathbf{b}, i) F(\mathbf{b}, i)}. \tag{6.21}$$

The non-negative bound is trivial, so we only have to prove $\phi(\mathbf{b}) \leq 1$. Thus we can ignore Boolean factor $W(\mathbf{b})$ and note that in the worst case we have $F(\mathbf{b}, i) = \left\lfloor \frac{2^k}{M(\mathbf{b},i)} \right\rfloor$ giving

$$\phi(\mathbf{b}) \leq \frac{2^{kh}}{\prod_{i=1}^{h} M(\mathbf{b}, i) \left\lfloor \frac{2^k}{M(\mathbf{b},i)} \right\rfloor}. \tag{6.22}$$

Using the identity $x \cdot \left\lfloor \frac{n}{x} \right\rfloor = n - (n \bmod x)$ we find

$$\phi(\mathbf{b}) \leq \prod_{i=1}^{h} \frac{2^k}{2^k - (2^k \bmod M(\mathbf{b}, i))}, \tag{6.23}$$

which simplifies further using the maximum branching factor b to

$$\phi(\mathbf{b}) \leq \left(\frac{2^k}{2^k - b}\right)^h. \tag{6.24}$$

Unfortunately, this is bigger than 1, but not by much if we choose h sufficiently big. We can thus *rescale*

$$\begin{aligned} \phi'(\mathbf{b}) &= \phi(\mathbf{b}) \cdot \left(\frac{2^k - b}{2^k}\right)^h, \\ \phi'(\mathbf{b}) &= W(\mathbf{b}) \cdot \prod_{i=1}^{h} \frac{2^k - b}{M(\mathbf{b}, i) F(\mathbf{b}, i)}, \end{aligned} \tag{6.25}$$

and rest assured that $0 \leq \phi'(\mathbf{b}) \leq 1$. After our mean estimation procedure is over we can multiply the estimate by our scaling factor again to get an accurate estimate of w.

This rescaling only doesn't affect our asymptotic complexities for the relative and absolute error if our scaling factor is constant. Thus we want our scaling factor to be at most some constant c, giving

$$\left(\frac{2^k}{2^k - b}\right)^h \leq c \tag{6.26}$$

and solving to

$$k \geq \log_2(b) + \log_2(c)/h - \log_2\left(c^{1/h} - 1\right). \tag{6.27}$$

Using the approximation $e^x \approx x + 1 \implies c^{1/h} = e^{\ln(c)/h} \approx \ln(c)/h + 1$ we find that $k = O(\log b + \log h)$ suffices.

Now Theorem 5.2 and its corollaries fully apply, as we met its prerequisites and the complexities are unaffected. □

The intuition for why the above works is two-fold:

1. When k becomes large the bias due to overcounting imbalances vanishes. This is because when there are n move options, every option is counted exactly $\left\lfloor \frac{2^k}{n} \right\rfloor$ times, plus sometimes once more. As k gets large the difference between being counted maybe once more or not becomes negligible.

   In fact, if we'd let $k \to \infty$ we'd find that

   $$w \approx \frac{1}{2^{kh}} \sum_{\mathbf{b} \in \{0,1\}^{kh}} W(\mathbf{b}) \tag{6.28}$$

   and no overcounting correction would be necessary. We explore this idea more in the next section.

2. We don't see occasional blowups in ϕ's value and only see a slight increase above 1 because $F(\mathbf{b}, i)$ and $M(\mathbf{b}, i)$ are very much correlated. When the number of move options is small, the overcounting factor is big. Vice versa, when the number of move options is large, the overcounting factor is small. Thus the product of the two stays relatively constant. And when $M(\mathbf{b}, i)$ is always a power $\leq 2^k$ we even have $F(\mathbf{b}, i) \cdot M(\mathbf{b}, i) = 2^k$, and thus $\phi(\mathbf{b})$ would simplify to just $W(\mathbf{b})$, showing that no overcounting correction is necessary for binary trees. This recovers the algorithm from Section 6.1.2, which was valid for complete binary trees.

6.3 Direct amplitude estimation

So far we've described processes with which we can use one or more estimates of means of certain functions to compute the random rollout winrate indirectly. In this section we will generalize and simplify, preparing a superposition in which a single qubit can be observed to be $|1\rangle$ with a probability approximating the random rollout winrate arbitrarily closely. We also lay the groundwork to go beyond simple random rollouts.

6.3.1 Policy-guided stochastic walk state preparation

Previously we considered the case of exactly simulating a random rollout. We will now simultaneously extend and limit this:

- We will be able to simulate an arbitrary classical policy π that for some given state s and action $1 \leq a \leq |A(s)|$ gives the probability $\pi(s, a)$ of the policy picking that action in this state. Random rollout would be the uniform distribution policy in this context.

- However, we will only accept π that outputs probabilities that can be written in the form $x/2^k$ with $x \in [0, 2^k]$.

The second limitation might make one think that this algorithm is not a strict superset of the previous one. It theoretically means that some rational probabilities occurring in the random rollout winrate computation will be impossible to simulate exactly, but we can get arbitrarily close. And as we'll prove later, choosing $k = O(\log h + \log b)$ suffices for the noise due to rounding to dip below the noise floor inherent in our estimation procedure, matching the bound of the previous algorithm exactly.

However, even if that weren't the case, we note that in a lot of classical implementations of MCTS the underlying random number generator will have similar limitations, and that MCTS is a heuristic algorithm to begin with, so the value of exact simulations with arbitrary rational probabilities for MCTS is questionable.

Theorem 6.4. *Given a policy function $\pi(s,a)$ providing a probability $x/2^k$ that in state s action $1 \leq a \leq b$ is taken with $x \in \{0,1,\ldots,2^k\}$, a transition function $\delta(s,a)$ returning the next state, and an initial state R it's possible to prepare a superposition over all paths of length h (with some garbage dependent on the path) where the squared absolute amplitude of each path matches the probability a walk of h steps using policy π and transition function δ starting at R would take that path, using $O(hb)$ queries to π, $O(h)$ queries to δ and $O(hk)$ auxiliary space.*

Proof. We define classically a cumulative policy function

$$\pi_c(s,a) = \sum_{i=1}^{a} \pi(s,i). \tag{6.29}$$

Naturally, we expect the probability distribution to be well-behaved such that $\pi_c(s,b) = 1$. We finally classically define $f(s,r)$ as the unique solution to

$$f(s,r) = i \quad \Longrightarrow \quad 2^k \pi_c(s,i-1) \leq r < 2^k \pi_c(s,i) \tag{6.30}$$

for any $r \in [0,2^k)$. We then note that if r is an uniformly chosen integer random variable $0 \leq r < 2^k$ that

$$P[f(s,r) = a] = \frac{2^k \pi_c(s,a) - 2^k \pi_c(s,a-1)}{2^k} = \pi(s,a). \tag{6.31}$$

We can implement f as a unitary operation

$$F : |s\rangle|r\rangle|0\rangle|0\rangle \mapsto |s\rangle|r\rangle|g\rangle|f(s,r)\rangle \tag{6.32}$$

with some garbage g that depends on s, r and define

$$P = \big(I \otimes H^{\otimes k} \otimes I \otimes I\big)F. \tag{6.33}$$

Using P we can now prepare a state that is a superposition over our move choices with the appropriate probability distribution π:

$$P|s\rangle|0\rangle|0\rangle|0\rangle = |s\rangle \frac{1}{\sqrt{2^k}} \sum_{r=0}^{2^k-1} |r\rangle|g\rangle|f(s,r)\rangle. \tag{6.34}$$

Writing $|g'\rangle = \sum_r |r\rangle|g\rangle$ as garbage we're not interested in, we simplify using (6.31) to

$$= |s\rangle \sum_{a=1}^{b} |g'\rangle \sqrt{\pi(s,a)}|a\rangle. \tag{6.35}$$

In a similar fashion to (6.6) we can now define a policy-guided transition function

$$T_\pi : |s\rangle|0\rangle|0\rangle|0\rangle \mapsto |s\rangle \sum_{a=1}^{b} |g\rangle \sqrt{\pi(s,a)}|a\rangle|\delta(s,a)\rangle \tag{6.36}$$

by combining P and a quantum circuit for δ. For simplicity we have unified the garbage from P and δ into just g which depends on s, a. Finally, similar to (6.7), we can now chain T_π h times, giving

$$\begin{aligned} T_\pi^* : |R\rangle|0^*\rangle \mapsto |s_0\rangle \sum_{a_1=1}^{b} |g_1\rangle\sqrt{\pi(s_0,a_1)}|a_1\rangle|s_1\rangle \cdots |s_{h-1}\rangle \sum_{a_h=1}^{b} |g_h\rangle\sqrt{\pi(s_{h-1},a_h)}|a_h\rangle|s_h\rangle \\ = \sum_{a_1,\ldots,a_h} \sqrt{\pi(s_0,a_1)\cdots\pi(s_{h-1},a_h)}|s_0\rangle|g_1\rangle|a_1\rangle|s_1\rangle\cdots|s_{h-1}\rangle|g_h\rangle|a_h\rangle|s_h\rangle \end{aligned} \tag{6.37}$$

where $s_0 = R$, $s_i = \delta(s_{i-1}, a_i)$, as earlier. This superposition has exactly the amplitudes for each path as required. □

6.3.2 Direct random rollout winrate estimation

The above theorem is very powerful, and makes the algorithm of this section trivial. Analysing the error introduced by rounding the random rollout probabilities to $x/2^k$ is a bit more tricky, however.

Theorem 6.5. *We can estimate the probability w that Alice is the winner after a random rollout using $O\left(\frac{\log 1/\delta}{\epsilon\sqrt{w}}\right)$ queries to T_π^* for relative error $\epsilon < 1$ with failure probability δ, choosing $k = O(\log h + \log b)$.*

Proof. We apply Theorem 5.2 using T_π^* to prepare our superposition, and a function ϕ_w that returns whether a specific path is winning for Alice. We request it with relative error ϵ_e, and get back an estimate of w', which is the winrate of Alice in a 'random rollout' where the probability of each move isn't uniform, but instead rounded to the closest $x/2^k$ where possible (while ensuring the probabilities sum to one). However, we wish to estimate w and we are thus dealing with a second error due to rounding,

$$|w - w'| \leq \epsilon_q w. \tag{6.38}$$

The errors from rounding $\frac{1}{M(s,i)}$ to $x/2^k$ are most significant when $M(s,i)$ is as big as possible, thus when it equals b. The absolute error of each individual number can be at most $\frac{1}{2^k}$. This means our rounded probability of picking a specific move can at most be off by a factor of

$$q = \frac{\frac{1}{b} \pm \frac{1}{2^k}}{\frac{1}{b}} = 1 \pm \frac{b}{2^k} \tag{6.39}$$

compared to the true probability as observed in random rollout. Since the winrate can be viewed as the sum of the probabilities of the winning paths, and the probability of each path is the product of the probabilities of its moves we know that q^h is the worst possible factor between w' and w. From Bernoulli's inequalities $1 + hx \leq (1+x)^h \leq e^{hx}$ which are valid for $h > 0, x > -1$, we can establish that in order for q and thus relative error ϵ_q to become constant we need to choose $k = O(\log h + \log b)$ at most.

Finally, choosing $\epsilon_e = \epsilon_q = \epsilon/3$ suffices, as

$$1 - \epsilon \leq (1 - \epsilon/3)^2 \leq (1 + \epsilon/3)^2 \leq 1 + \epsilon \tag{6.40}$$

holds for any $0 < \epsilon < 1$, and thus the combined estimation and rounding errors satisfy our overall desired relative error ϵ. □

Chapter 7

Bandit selection in QMCTS

7.1 Parallel MCTS as bandits

As we described in Section 4.2, MCTS is a prime target for parallelization. Of interest here is *root parallelization*, where one runs multiple completely independent instances of some stochastic MCTS variant and then return an answer based on all of the gathered statistics. One particular variant of this approach can be sped up using quantum computers: root parallelization with best average.

In this variant one runs n independent MCTS instances, each for k iterations. One then averages the n statistics for each child i of the root, and return child i with the maximum average as the move suggestion. The statistic of choice here is the robust one as discussed in Section 3.2.5, the node with the most visits. This is due to the average number of visits being well-behaved, but the average winrate less so (consider the average of win/loss statistics $1/2$ and $4/100$, the correct answer $5/102$ is not an arithmetic mean and thus incompatible with our upcoming quantum algorithm).

Since MCTS is stochastic, we can view this as a bandit problem where each arm is one of the children, giving a 'reward' of the number of visits divided by k, a value thus in $[0, 1]$. The two key differences are that a classical algorithm must pull every single arm at once, and we simply wish to maximize the chance of returning a good arm. Nevertheless, as seen from comparing equations (5.18), (5.20) even if we could pull each arm individually classically, the quantum algorithm we'll use provides a speedup regardless.

7.2 Quantum parallelism

Our algorithm is a simple application of Theorem 5.3 to a direct translation of MCTS to a quantum circuit. To this end we consider a classical circuit for k iterations of MCTS. That entire computational process is viewed as one (very) large classical circuit. Whatever stochastic choices the classical algorithm makes are considered to be generated by a pseudo-random number generator that was initialized using a w-bit seed r. Finally, the circuit takes in another input i and is expected to report the number of visits MCTS had to the ith child of the root as a $\lceil \log_2(k) \rceil$-bit integer. If the pseudo-random number generator is of sufficient quality each different seed r results in a different run of MCTS and thus a bandit 'arm pull'.

The above quantum circuit can be described as unitary

$$U : |i\rangle|r\rangle|0\rangle|0\rangle \mapsto |i\rangle|r\rangle|v_{i,r}\rangle|g_{i,r}\rangle \tag{7.1}$$

where $v_{i,r}$ is the number of visits to child i with random seed r and $g_{i,r}$ is some unspecified garbage that depends on i, r. Using a series of Hadamard gates to form a $\lceil \log_2 k \rceil$-qubit register $|q\rangle$ in the uniform superposition and a gate flipping an output bit when $q < v_{i,r}$ we can get unitary

$$U' : |i\rangle|r\rangle|0\rangle|0\rangle|0\rangle|0\rangle \mapsto |i\rangle|r\rangle|v_{i,r}\rangle|g_{i,r}\rangle \left(\sqrt{p_{i,r}} \sum_{q=0}^{v_{i,r}-1} |q\rangle|1\rangle + \sqrt{1-p_{i,r}} \sum_{q=v_{i,r}}^{2^{\lceil \log_2(k) \rceil}-1} |q\rangle|0\rangle \right) \tag{7.2}$$

where $p_{i,r} = \frac{v_{i,r}}{2^{\lceil \log_2(k) \rceil}}$ scales linearly with $v_{i,r}/k$ and is off by at most a factor of 2 (independent of i, r, only dependent on k).

Theorem 7.1. *Let the random process R be k iterations of a stochastic variant of MCTS. It is possible to find move option i such that v_i, the mean number of times option i is chosen during 2^ℓ samples to R is at most ϵk less than the move option that was chosen the most in those samples. This can be done with probability $1 - \delta$ with*

$$O\left(\sqrt{\min\left\{ \frac{n}{\epsilon^2}, \sum_{i=2}^{n} \Delta_i^{-2} \right\}} \cdot \operatorname{poly}\left(\log\left(\frac{n}{\delta \Delta_2} \right) \right) \right) \tag{7.3}$$

queries to unitary U', where n is the number of move options being considered and Δ_i scales linearly with the difference between v_i/k and $\max_i v_i/k$. Unitary U' scales linearly in size with k and ℓ.

Proof. If we bring register $|r\rangle$ of unitary U' into an uniform superposition the probability of observing the output register as $|1\rangle$ is $p_i = \frac{1}{2^\ell} \sum_{r=0}^{2^\ell - 1} p_{i,r}$, which is equivalent (ignoring the previously mentioned constant factor only dependent on k) to the mean observation of v_i/k in 2^ℓ instances of MCTS ran for k iterations.

We thus define $T = (I \otimes H^{\otimes \ell} \otimes I \otimes I \otimes I \otimes I)U'$ and apply Theorem 5.3 using it. This provides exactly the complexity bounds as desired. □

Chapter 8

Other applications

While the subject of this thesis is quantum speedups for Monte Carlo tree search, Theorem 6.4 is significantly more powerful than just its application to MCTS, so we felt it to be necessary to explore some corollary results not directly related to MCTS.

Theorem 8.1. *Let $\pi(s,a)$ be a policy function providing a probability $x/2^k$ that in state s action $1 \leq a \leq b$ is taken with $x \in \{0,1,\dots 2^k\}$, let $\delta(s,a)$ be a transition function returning the next state, and let R be an initial state. Let $\phi : \{1,2,\dots,b\}^h \to [0,1]$ be an arbitrary function defined over paths of length h starting at R. Then it is possible to estimate the mean value μ of ϕ evaluated over all possible paths of length h, weighted by the probability a walk of h steps using policy π and transition function δ starting at R would take that path. This can be done using $O\left(\frac{\log 1/\delta}{\epsilon\sqrt{\mu}}\right)$ or $O\left(\frac{\log 1/\delta}{\epsilon}\right)$ queries to T_π^* for respectively relative error or absolute error ϵ with failure probability δ.*

Proof. We apply the procedure of Theorem 5.2 using T_π^* from Theorem 6.4 to prepare our superposition over all paths of length h. The complexity bounds come from the corollaries of Theorem 5.2. □

Theorem 8.2. *Let $\mathrm{Arm}_i = (\pi_i, \delta_i, R_i, \phi_i)$ be defined similarly to Theorem 8.1 for all $1 \leq i \leq n$. Then, if μ_i is the mean value associated with Arm_i as in Theorem 8.1, we can find i such that $|\mu^* - \mu_i| \leq \epsilon$ with probability $1-\delta$ with*

$$O\left(\sqrt{\min\left\{\frac{n}{\epsilon^2}, \sum_{i=2}^{n} \Delta_i^{-2}\right\}} \cdot \mathrm{poly}\left(\log\left(\frac{n}{\delta\Delta_2}\right)\right)\right), \tag{8.1}$$

queries to a unitary U' where $\mu^* = \max_i \mu_i$ and $\Delta_i = \mu^* - \mu_i$.

Proof. We construct a circuit entirely analogous to T_π^* with an additional register $|i\rangle$ controlling which π_i, δ_i get applied and which R_i forms the initial state. The result is a unitary $T' : |i\rangle|0\rangle \mapsto |i\rangle|S_i\rangle$ preparing superposition S_i associated with Arm_i. We similarly modify unitary W from (5.9) to apply ϕ_i:

$$W'|i\rangle|x\rangle|0\rangle = |i\rangle|x\rangle\left(\sqrt{1-\phi_i(x)}|0\rangle + \sqrt{\phi_i(x)}|1\rangle\right). \tag{8.2}$$

The concatenation of T' and W' creates unitary

$$U'|i\rangle|0\rangle|0\rangle = |i\rangle\left(\sqrt{1-\mu_i}|u_i\rangle|0\rangle + \sqrt{\mu_i}|v_i\rangle|1\rangle\right), \tag{8.3}$$

which allows us to apply Theorem 5.3 to get the claimed query complexity. □

Everything that follows are direct application of these theorems.

8.1 Policy reward evaluation

If we have a policy for achieving rewards in a certain environment, and wish to evaluate the mean reward achieved by this policy if either or both of the policy and the environment are stochastic, we can do this efficiently using Theorem 8.1. Interestingly, unlike the traditional Markov decision processes [Bel57] these rewards do not have to be associated with actions, but can be globally determined based on the entire path taken as the reward function ϕ can see the whole path at once. A concrete example of this is simulating a truck picking up and delivering goods, where the reward is based on the *maximum* weight the truck had during a path.

Note that we can simulate stochastic environments by adapting the policy function π and transition function δ through the introduction of *stochastic nodes*. If $\delta(s, a)$ should result in the state s_k with probability p_k one can create a modified transition and policy function δ', π' by creating a stochastic node $t_{s,a}$, setting $\delta'(s, a) = t_{s,a}$, $\delta'(t_{s,a}, k) = s_k$ and $\pi'(t_{s,a}, k) = p_k$ to achieve just that.

8.2 Adversarial two-player strength evaluation

If we have two candidate policies for a two-player game and wish to evaluate which wins the most on average in games between the two, we can also do this efficiently with Theorem 8.1. One merely has to set policy π to Alice's policy when it is her turn, and to Bob's policy when it is his turn. To eliminate any first-player advantage one can add a stochastic node at the start of the game that lets Alice or Bob go first with 50% odds. Finally, one must define ϕ to return 1 if Alice won and 0 if Bob won.

8.3 Policy or environment selection

In the scenario in which there are multiple candidate policies for an environment (or similarly, multiple environments in which to deploy a policy), one can quickly estimate which is a good choice using Theorem 8.2.

8.3.1 Policy parameter optimization

A special case of the above is when a policy has a parameter that can be tweaked. If one proposes n candidates for this parameter then using Theorem 8.2 we can find a good candidate out of this set. Interestingly, the query complexity scales with the square root of n which means many options can be evaluated quickly, if an efficient circuit can be constructed that encodes the ith policy π_i. This is trivial in the case one does a structured search over a real parameter, as then the parameter can be computed within the circuit from i.

8.3.2 Adversarial self-play policy parameter optimization

This is a combination of the ideas in the previous section and Section 8.2. If one has a policy for playing a two-player game, and this policy has (many) parameters that can be tweaked one can propose a set of candidate *replacement* parameter values and let the original policy play against replacement policy i as in Section 8.2. Then using Theorem 8.2 one can quickly find a good replacement policy, that is, one that is not much worse than the replacement policy most likely to beat the original policy.

One can then iterate this to learn a policy step by step. However, one might quickly find itself in a local optimum or a rock-paper-scissors scenario if one always optimizes to simply beat the previous policy. It is possible to alleviate this using stochastic nodes once more. In addition to a stochastic node determining who goes first, it is possible to add stochastic nodes to choose which opponent policy out of a set of possible opponent policies gets played. These candidate opponent policies could then be for example the previous policy, the policy k iterations back for a small set of different k, as well as some heuristic baselines a good policy is expected to beat.

8.4 Conditional key parameter identification

Most modern learning algorithms are gradient-based, allowing one to estimate for many parameters how sensitive the output prediction is to changes in that parameter. If a policy has this form (or another form that allows computing contribution factors of parameters w.r.t. decisions), its possible using Theorem 8.2 to quickly identify which parameter is highly significant in arbitrary conditional situations and environments.

As an example, suppose we are performing the adversarial self-play policy parameter optimization from Section 8.3.2, and have just identified a new policy π' that often beats a set of baseline policies. To identify a further area of improvement one can choose a set of parameters from π' and choose ϕ_i as a function that is zero in the event policy π' won, and the mean contribution of parameter i to all decisions made in the event of a loss, and apply Theorem 8.2 to find a highly significant decision parameter specifically during losses.

Chapter 9

Conclusion

In this report we studied the application of quantum computational methods to attempt to speed up Monte Carlo Tree Search. We found moderate success in this regard, although the developed techniques have a wide range of other applications.

The first speedup found was an improvement in estimating the winrate of a random rollout in a variant of MCTS where one does k random rollouts to reduce the variance that a singular random rollout estimate would have. With Theorem 6.5 it was shown that as the desired uncertainty of the random rollout winrate decreases that a quantum algorithm requires quadratically less time to provide such a precise estimate than a classical algorithm. However, the value of such precise estimates was questioned, as by a simple counterexample in Section 4.2.1 it was shown that even perfectly accurate random rollout winrate information can contribute very little to the improvement of MCTS's move suggestion overall for some problems.

The second speedup found can with high probability estimate which moves a large ensemble of instances of MCTS ran for k iterations would suggest using quadratically less computational resources than such an ensemble would take to run on a (cluster of) classical computer(s). While the constructed quantum circuits are not viable for near-term quantum computers, they scale linearly in k and thus the technique is in theory a viable solution in the future for a fixed-budget fixed-time setting, where one has a certain amount of (quantum) computational resources, but also a fixed time budget that prevents the application of those resources in a very long and serial manner. This exact setting occurs in computer Chess, where one might have a very large data center to compute moves, but only a fixed time budget to compute those moves.

Finally the techniques developed were generalized beyond MCTS to two very powerful theorems in Chapter 8, which allows for estimation of the expected value or maximizing first step of an arbitrary (stochastic) fixed-length policy-guided walk in an arbitrary (stochastic) environment defined by an initial state, a policy function and a transition function, with a value assigned to such a walk by an arbitrary evaluation function defined on the complete path taken, quadratically faster than on classical computers. Various applications were also proposed in the same chapter, and there is a broad opportunity for future research in those and other applications of the theorems.

Bibliography

[ACF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem." In: *Machine Learning* 47 (May 2002), pp. 235–256. DOI: 10.1023/A:1013689704352.

[Amb+18] Andris Ambainis et al. *Quantum Speedups for Exponential-Time Dynamic Programming Algorithms*. 2018. arXiv: 1807.05209 [quant-ph].

[Bel57] Richard Bellman. "A Markovian Decision Process." In: *Journal of Mathematics and Mechanics* 6.5 (1957), pp. 679–684. ISSN: 00959057, 19435274. URL: http://www.jstor.org/stable/24900506.

[Bra+02] Gilles Brassard et al. "Quantum amplitude amplification and estimation." In: *Quantum Computation and Information* (2002), pp. 53–74. ISSN: 0271-4132. DOI: 10.1090/conm/305/05215. arXiv: quant-ph/0005055.

[Bro+12a] Cameron Browne et al. "A Survey of Monte Carlo Tree Search Methods." In: *IEEE Transactions on Computational Intelligence and AI in Games* 4:1 (Mar. 2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810. URL: https://www.researchgate.net/publication/235985858_A_Survey_of_Monte_Carlo_Tree_Search_Methods.

[Bro+12b] Cameron B. Browne et al. "A Survey of Monte Carlo Tree Search Methods." In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[Brü93] Bernd Brügmann. *Monte Carlo Go*. 1993.

[BTV01] Harry Buhrman, John Tromp, and Paul Vitányi. "Time and space bounds for reversible simulation." In: *Journal of Physics A: Mathematical and General* 34.35 (2001), pp. 6821–6830. ISSN: 1361-6447. DOI: 10.1088/0305-4470/34/35/308. arXiv: quant-ph/0101133.

[Chi+07] Andrew Childs et al. "Every NAND formula of size N can be evaluated in time $N^{1/2+o(1)}$ on a quantum computer." In: (Mar. 2007). arXiv: quant-ph/0703015.

[Chi96] S. Chinchalkar. "An Upper Bound for the Number of Reachable Positions." In: *J. Int. Comput. Games Assoc.* 19 (1996), pp. 181–183. DOI: 10.3233/ICG-1996-19305.

[CJ07] Tristan Cazenave and Nicolas Jou. *On the Parallelization of UCT*. 2007.

[CM07] Pierre-Arnaud Coquelin and Rémi Munos. "Bandit Algorithms for Tree Search." In: *CoRR* abs/cs/0703062 (2007). arXiv: cs/0703062.

[Con76] John Conway. *On numbers and games*. London New York: Academic Press, 1976. ISBN: 0-12-186350-6.

[Cor18] Arjan Cornelissen. "Quantum gradient estimation and its application to quantum reinforcement learning." MA thesis. 2018. URL: http://resolver.tudelft.nl/uuid:26fe945f-f02e-4ef7-bdcb-0a2369eb867e.

[Cou07] Rémi Coulom. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search." In: *Computers and Games*. Springer Berlin Heidelberg, 2007, pp. 72–83. DOI: 10.1007/978-3-540-75538-8_7.

[CWH08] Guillaume M. J. -B. Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. "Parallel Monte-Carlo Tree Search." In: *Computers and Games*. Springer Berlin Heidelberg, 2008, pp. 60–71. DOI: 10.1007/978-3-540-87608-3_6.

[FGG08] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. "A Quantum Algorithm for the Hamiltonian NAND Tree." In: *Theory of Computing* 4.1 (2008), pp. 169–190. DOI: 10.4086/toc.2008.v004a008. arXiv: quant-ph/0702144.

[GGL12] Victor Gabillon, Mohammad Ghavamzadeh, and Alessandro Lazaric. "Best Arm Identification: A Unified Approach to Fixed Budget and Fixed Confidence." In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/8b0d268963dd0cfb808aac48a549829f-Paper.pdf.

[Gid15] Craig Gidney. "Creating bigger controlled NOTs from single qubit, Toffoli, and CNOT gates, without workspace." In: *StackExchange* (2015). URL: https://cs.stackexchange.com/q/44292.

[GWT06] Sylvain Gelly, Yizao Wang, and Olivier Teytaud. "Modification of UCT with Patterns in Monte-Carlo Go." In: (2006).

[Hen+15] B. Hensen et al. "Loophole-free Bell inequality violation using electron spins separated by 1.3 kilometres." In: *Nature* 526.7575 (2015), pp. 682–686. DOI: 10.1038/nature15759.

[Hoe63] Wassily Hoeffding. "Probability Inequalities for Sums of Bounded Random Variables." In: *Journal of the American Statistical Association* 58.301 (1963), pp. 13–30. DOI: 10.1080/01621459.1963.10500830.

[JVV86] Mark R. Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. "Random generation of combinatorial structures from a uniform distribution." In: *Theoretical Computer Science* 43 (1986), pp. 169–188. DOI: 10.1016/0304-3975(86)90174-X.

[LR85] T.L Lai and Herbert Robbins. "Asymptotically efficient adaptive allocation rules." In: *Advances in Applied Mathematics* 6.1 (1985), pp. 4–22. ISSN: 0196-8858. DOI: 10.1016/0196-8858(85)90002-8.

[Mir+17] S Ali Mirsoleimani et al. "An Analysis of Virtual Loss in Parallel MCTS." In: Jan. 2017, pp. 648–652. DOI: 10.5220/0006205806480652.

[Mir+18] S. A. Mirsoleimani et al. "A Lock-free Algorithm for Parallel MCTS." In: *ICAART*. 2018.

[Mon15] Ashley Montanaro. "Quantum speedup of Monte Carlo methods." In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 471.2181 (Sept. 2015), p. 20150301. ISSN: 1471-2946. DOI: 10.1098/rspa.2015.0301. arXiv: 1504.06987 [quant-ph].

[Mon16] Ashley Montanaro. *Quantum walk speedup of backtracking algorithms*. 2016. arXiv: 1509.02374 [quant-ph].

[NC11] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. 10th. USA: Cambridge University Press, 2011. ISBN: 1107002176. DOI: 10.1017/CBO9780511976667.

[Neu28] John von Neumann. "Zur Theorie der Gesellschaftsspiele." In: *Mathematische Annalen* 100 (1928), pp. 295–320. DOI: 10.1007/BF01448847.

[NST20] Minh Anh Nguyen, Kazushi Sano, and Vu Tu Tran. "A Monte Carlo tree search for traveling salesman problem with drone." In: *Asian Transport Studies* 6 (2020), p. 100028. DOI: 10.1016/j.eastsj.2020.100028.

[Pea80] Judea Pearl. "Asymptotic properties of minimax trees and game-searching procedures." In: *Artificial Intelligence* 14.2 (1980), pp. 113–138. ISSN: 0004-3702. DOI: 10.1016/0004-3702(80)90037-5.

[Pea82] Judea Pearl. "The Solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and Its Optimality." In: *Commun. ACM* 25.8 (Aug. 1982), pp. 559–564. ISSN: 0001-0782. DOI: 10.1145/358589.358616.

[Rei11] Ben W. Reichardt. "Faster quantum algorithm for evaluating game trees." In: *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2011. DOI: 10.1137/1.9781611973082.43. arXiv: 0907.1623 [quant-ph].

[Ros11] Christopher D. Rosin. "Multi-armed bandits with episode context." In: *Annals of Mathematics and Artificial Intelligence* 61.3 (2011), pp. 203–230. DOI: 10.1007/s10472-011-9258-6.

[Sch+07] Jonathan Schaeffer et al. "Checkers Is Solved." In: *Science* 317.5844 (2007), pp. 1518–1522. ISSN: 0036-8075. DOI: 10.1126/science.1144079.

[Sch+20] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model." In: *Nature* 588.7839 (2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4.

[She13] Henry Maurice Sheffer. "A Set of Five Independent Postulates for Boolean Algebras, with Application to Logical Constants." In: *Transactions of the American Mathematical Society* 14.4 (1913), pp. 481–488. ISSN: 00029947. DOI: 10.2307/1988701.

[Sil+16] David Silver et al. "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961.

[Sil+17a] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm." In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815.

[Sil+17b] David Silver et al. "Mastering the game of Go without human knowledge." In: *Nature* 550.7676 (2017), pp. 354–359. DOI: 10.1038/nature24270.

[Sil+18] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play." In: *Science* 362.6419 (2018), pp. 1140–1144. DOI: 10.1126/science.aar6404.

[SKW10] Yusuke Soejima, Akihiro Kishimoto, and Osamu Watanabe. "Evaluating Root Parallelization in Go." In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.4 (2010), pp. 278–287. DOI: 10.1109/tciaig.2010.2096427.

[SXX20] Devavrat Shah, Qiaomin Xie, and Zhi Xu. "Non-Asymptotic Analysis of Monte Carlo Tree Search." In: Association for Computing Machinery, 2020, pp. 31–32. DOI: 10.1145/3393691.3394202. arXiv: 1902.05213 [stat.ML].

[TF07] John Tromp and Gunnar Farnebäck. "Combinatorics of Go." In: *Computers and Games*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 84–99. DOI: 10.1007/978-3-540-75538-8_8. URL: https://tromp.github.io/go/gostate.pdf.

[Wan+20] Daochen Wang et al. *Quantum exploration algorithms for multi-armed bandits*. 2020. arXiv: 2007.07049 [quant-ph].