Classical and Quantum Algorithms for USO Recognition

Master Thesis
Vitor Bosshard
October 13, 2015

Advisors: Prof. Dr. Bernd Gärtner, Prof. Dr. Stefan Wolf, Ämin Baumeler
Department of Computer Science, ETH Zürich
Abstract

Unique Sink Orientations (USOs) are orientations of the hypercube graph capable of encoding many optimization problems, such as linear programming. When a USO is given in implicit form as an oracle, the USO recognition problem aims to distinguish USO from non-USO oracles.

In this thesis we introduce a new type of orientation, the pseudo-USOs, which are closely related to the USO recognition problem. We derive many of their combinatorial properties, up to and including a full characterization. Using these properties, we design efficient USO recognition algorithms both in the classical and quantum models of computation.
Notation

General

\([n] := \{1, 2, \ldots, n\}\) is the set of integers from 1 to \(n\).
\([\cdot]\) is the indicator function.
\([x, y] \subseteq \mathbb{R}\) is the closed interval between \(x\) and \(y\). \((x, y)\) is the open interval, \([x, y]\) and \((x, y)\) are half-open intervals.
\(A \times B\) is the cartesian product of two sets.
\(\log(x)\) is the logarithm in base 2 (not the natural logarithm).
\(H(p)\) is the binary entropy function, defined as
\[
H(p) = -p \log(p) - (1 - p) \log(1 - p),
\]
where \(0 \log(0) = 0\) by convention.

Words and Functions on Words

A word \(v\) is a finite-length sequence of symbols from some finite set (the alphabet).
\(v_i\) is the \(i\)-th symbol of a word \(v\).

If \(u\) and \(v\) are words, then \(uv\) or \(u \cdot v\) is their concatenation, and \(v^n\) is the \(n\)-fold concatenation of \(v\) with itself. Single symbols are also length one words, so for example \(v \cdot 0\) is a valid word.

Words from the set \(\{0, 1\}^n\) can be thought of as \(n\)-vectors over \(\mathbb{F}_2\) (the finite field with 2 elements), but formally we will consider them only as words.
\(e_i \in \{0, 1\}^n\) is the word \(0^{i-1} \cdot 1 \cdot 0^{n-i-1}\).

For \(x, y \in \{0, 1\}^n\), the symmetric difference \(x \oplus y \in \{0, 1\}^n\) is the element-wise exclusive OR (XOR), i.e.
\[
(x \oplus y)_i := x_i \oplus y_i = x_i + y_i \mod 2,
\]
and the conjunction \(x \land y \in \{0, 1\}^n\) is
\[
(x \land y)_i := x_i \land y_i.
\]
Note that multi-valued functions are considered to map words to words, and we will often define them element-wise, as in the examples above.

The Hamming distance \( d_H : \{0, 1\}^n \mapsto [n] \) between two words \( x, y \in \{0, 1\}^n \) is defined as

\[
d_H(x, y) := \sum_{i=1}^{n} [x_i \neq y_i].
\]

The Hamming weight of \( v \in \{0, 1\}^n \) is \( d_H(v, 0^n) \), i.e. the number of ones in \( v \).

**Graphs**

Let \( G = (V, E) \) be a graph, and \( V' \subseteq V \) be a subset of vertices. The induced subgraph with vertex set \( V' \) is a graph \( G' = (V', E') \) where

\[
E' := E \cap (V' \times V').
\]

**Landau Notation**

Let \( f \) be a function. \( O(f) \) is the set of functions that asymptotically grow at most as fast as \( f \), i.e. they are at most a constant fraction larger than \( f \) for all \( n \) larger than some threshold.

\[
g \in O(f) \iff \exists n_0, c \in \mathbb{R}^+ : \forall n > n_0 : |g(n)| \leq c \cdot |f(n)|.
\]

\( \Omega(f) \) is the set of functions that grow at least as fast as \( f \), i.e.

\[
g \in \Omega(f) \iff f \in O(g),
\]

and \( \theta(f) \) is the set of functions that grow exactly as fast as \( f \), i.e.

\[
\theta(f) = \Omega(f) \cap O(f).
\]

In a slight abuse of notation, we sometimes include these sets in the middle of an expression, with the meaning that an unspecified member of the set belongs there. For example,
\[ f(n) = 2^{n+O(\log(n))} \iff \exists g \in O(\log(n)) \text{ s.t. } f(n) = 2^{n+g(n)}. \]

\(\text{poly}(n)\) is the set of all functions that grow at most polynomially fast, i.e.

\[ \text{poly}(n) := \bigcup_{\epsilon=0}^{\infty} O(n^\epsilon) \]
Contents

Notation .................................................. ii

Contents ................................................. v

1 Introduction ........................................... 1

2 Quantum Computing Basics .......................... 3
   2.1 Model of Computation ............................. 3
   2.2 Quantum Search Algorithms ..................... 5
      2.2.1 Grover’s Algorithm ........................ 6
      2.2.2 Using Grover with heuristics ............... 7
      2.2.3 Generalization 1: Amplitude Amplification 8
      2.2.4 Generalization 2: Search via Quantum Walk 9
      2.2.5 The Element Distinctness Algorithm .......... 11

3 Unique Sink Orientations ........................... 15
   3.1 Definitions ...................................... 15
      3.1.1 The Cube .................................. 15
      3.1.2 Unique Sink Orientations .................. 16
   3.2 Fundamental Properties of USOs ................ 17
   3.3 Inherited Orientations ........................... 18

4 Pseudo-USOs ........................................... 23
   4.1 Algorithmic Motivation .......................... 23
   4.2 Definition and Existence Proofs ................ 25
   4.3 Combinatorial Properties ....................... 28

5 The USO Recognition Problem ....................... 37
   5.1 Overview ........................................ 37
   5.2 Classical Algorithm ............................. 38
      5.2.1 Hamming Balls and Coverings ............... 38
Contents

5.2.2  The Algorithm ................................................. 40
5.2.3  Improving the Algorithm ................................. 42
5.3  Quantum Algorithms ........................................... 44
5.4  Lower Bounds Revisited ....................................... 50

Bibliography ..................................................... 53
Chapter 1

Introduction

Unique sink orientations are orientations of the hypercube with the property that each face (subcube) possesses a unique sink. They are an abstract framework onto which many problems can be reduced.

For example, every linear program can be reduced to a USO in such a way that the solution (or lack of one) is encoded in its global sink. The essential combinatoric nature of the LP is captured in the process, discarding superfluous details like the exact geometric boundaries of the feasible region, but preserving a partial order of the solution candidates which allows the solution to be found. The existence of a polynomial time algorithm for finding the sink of a USO would therefore imply a strongly polynomial time algorithm for linear programming, which is a long-standing open problem in theoretical computer science [11].

Similar reductions are possible for other problems such as smallest enclosing balls [24], smallest enclosing balls of balls [9], and some linear complementarity problems [19]. Given this background, it is not surprising that USOs have been studied extensively. In particular, many sink-finding algorithms have been developed [10] [24].

In this thesis, we have tried to approach the problem from a radically different viewpoint, by attempting to apply quantum algorithms to USO sink-finding and related problems.

Quantum computers are a theoretical model of computation making use of quantum mechanical phenomena to perform computational tasks faster than any classical computer is capable of. Their relevance stems from the fact that they are believed to be implementable in the physical universe, as opposed to other powerful models such as nondeterministic Turing machines.

Even though we did not achieve the original goal of finding efficient quantum algorithms for USO sink-finding, the viewpoint provided by approach-
1. Introduction

ing USOs from an unusual angle led to the discovery of a novel type of combinatorial object, the pseudo-USO. These objects occur in the context of USO recognition, the problem of determining whether or not a given function is an implicit representation (outmap) of a USO.

Unlike USO sink-finding, USO recognition is a problem of purely theoretical interest. It pops up when attempting to place USO sink-finding into a complexity class. Complexity classes in their conventional form categorize decision problems, i.e. problems where there are yes-instances and no-instances. What is swept under the rug is that there fundamentally exists a third type of instance, namely malformed instances which don’t make sense semantically. These are usually rolled into the no-instances, with the implicit understanding that detecting them is trivial [14].

For example, the 3-SAT problem defines yes-instances as strings encoding a 3-SAT formula that is satisfiable. No-instances include both unsatisfiable formulas and strings not encoding 3-SAT formulas at all. Any sensible encoding of 3-SAT formulas is trivial to check for correctness, so when we determine that 3-SAT is NP-complete, this is clearly a statement about the difficulty of separating satisfiable from unsatisfiable formulas, not about separating correctly encoded from incorrectly encoded formulas.

In the case of USOs, recognizing well-formed instances is not trivial at all (in fact it is coNP-hard [12]), so formalizing USO sink-finding as a decision problem is doomed to fail. In some sense, the USO recognition problem “shadows” the actual difficulty of the USO sink-finding problem, and it is worth studying for that reason alone.

In Chapter 2, we give an overview of quantum computing aimed at computer scientists. We introduce Grover’s quantum search algorithm and the generalizations that have been developed from it in the two decades since its discovery.

In Chapter 3, we define unique sink orientations and give their most relevant known properties in a self-contained manner.

In Chapter 4, we introduce pseudo-USOs and derive many of their combinatorial properties, including a complete characterization of their outmaps. This is the main contribution of the thesis.

In Chapter 5, we design both classical and quantum algorithms to solve the USO recognition problem, using some of the properties derived in the previous chapter.
Chapter 2

Quantum Computing Basics

In this thesis, we study quantum computers as purely theoretical constructs. We ignore the issues surrounding their actual construction, currently being attempted by physicists around the world, and treat them just as a model of computation.

In the first section of this chapter, we very briefly describe the model, mostly to give a broad sense of how quantum computers operate and to put the common buzzwords heard around them (superposition, entanglement, interference, collapse) into a minimal amount of context. For a thorough introduction to quantum computing, we suggest [20].

In the rest of the chapter, we introduce the quantum algorithms needed to develop our results. They will be treated as black boxes, that is we assume that they are available to be used, only going into their implementation details as far as necessary to develop our results.

2.1 Model of Computation

The most common model in use is the quantum circuit model. Quantum circuits are like boolean circuits, except that they use quantum bits, qubits, instead of classical bits and quantum gates instead of classical logic gates.

Qubits are a generalization of classical bits; they can take on the classical values of 0 and 1, but also be in a quantum superposition of those values. The classical states 0 and 1 are basis vectors in a linear space (namely $\mathbb{C}^2$), and a qubit is a unitary vector in that space. The complex coefficients of those vectors are called amplitudes. When we have $n$ qubits instead of 1, the space becomes exponentially larger; the classical basis states are all $2^n$ bitstrings of length $n$, i.e. the linear space becomes $\mathbb{C}^{2^n}$. 
Quantum gates are unitary matrices that operate on (a subset of) the qubits. Just like in classical computing, certain sets of quantum gates are universal, in the sense that any computation can be expressed only using those gates. In the quantum circuit model, some universal set of gates is chosen by convention, and the runtime of algorithms is measured in the number of such gates required to execute the algorithm.

It is commonly believed that the possibility of creating superpositions of states is what makes quantum computers fast, but we can also achieve such superpositions in the classical world. Whenever a classical algorithm makes a random decision, we can think of the memory of the computer going into a superposition (i.e. probability distribution, i.e. stochastic vector) of all possible outcomes, and this distribution is transformed by successive computation steps and split further by successive random choices. At the end of the computation we draw a sample from the distribution generated by the algorithm. This process is equivalent to sampling a path in the computation tree of the algorithm. Quantum computers operate similarly, but the process of drawing a sample at the end involves decoherence (“collapse”) of the quantum superposition.

Where quantum computing differs from classical computing, is that, since the coefficients of the linear combinations can be complex numbers, the contributions of different paths in the computation tree can interfere with each other. For example, if a classical computer has two different computation paths leading to the same result, each with 10% chance of occurring, the probability of that result is 20%. For a quantum computer, the final probability can be anywhere between 40% (constructive interference) and 0% (destructive interference) depending on the relative phase (difference in direction) of the amplitudes. This is similar to the interference between waves observed in classical physics, and the art of designing quantum algorithms is controlling the interference patterns in such a way that undesired computation paths cancel each other out.

We have not mentioned yet the quantum mechanical phenomenon called entanglement. While entanglement is extremely important in quantum information theory and the philosophical interpretation of quantum mechanics, in our formalism entanglement is the rather mundane observation that sometimes, the state of two qubits cannot be described independently of each other, but only as a whole. In some sense, a pair of entangled qubits is the quantum equivalent of a pair of correlated classical bits, and it has been shown that quantum computers are faster for certain tasks than classical computers even when not allowed to make use of entanglement at all [4].

One final note: While quantum circuits are more efficient than classical computers for many tasks, they cannot compute any function that is not com-
putable by a classical computer. In other words, they are Turing complete model of computation, and any quantum algorithm can be simulated by a classical computer by paying an exponential overhead.

### 2.2 Quantum Search Algorithms

One problem that has been researched extensively in the quantum model is the so-called unstructured search problem. Intuitively, we are looking for a marked element in a database, where by marked element, we mean an element with a property that we can recognize efficiently.

The database is defined implicitly, so we should not imagine that this is a conventional database written down somewhere, but rather a function taking as input the elements of some domain, typically bitstrings i.e. \( \{0, 1\}^n \), and giving as output their marked/unmarked status. The term “database” keeps being used in this context for historical reasons only, though as computer scientist, we really should be more precise and call it a search space.

Mathematically speaking, the unstructured search problem can be defined as follows:

\[
\text{Given a predicate } f : \mathcal{X} \mapsto \{0, 1\}, \text{ find an } x \in \mathcal{X} \text{ such that } f(x) = 1, \text{ or show that no such } x \text{ exists.}
\]

The problem is unstructured, because unlike more typical problems in computer science, there is no guarantee or restriction of any kind limiting the nature of \( f \). We also don’t have access to the implementation of \( f \); it is given as an oracle only, i.e. a black box. The runtime of any algorithm on this problem is measured in the unit cost model, where a query to \( f \) costs 1 and all other costs are ignored.

The only sensible classical algorithm to attack this problem is to search through all elements. This trivial algorithm has a runtime of \( O(|\mathcal{X}|) \) which already matches the lower bound [15], so it seems there is nothing interesting left to be said about the problem.

Enter the Quantum model of computation. If we assume that the oracle giving \( f \) is not only accessible classically, but can also be queried with a quantum superposition of inputs, it turns out that we can do much better. In fact, the runtime can be reduced to \( O(\sqrt{|\mathcal{X}|}) \) by using Grover’s algorithm, which we will discuss in the next section.

It should be noted that the assumption that the oracle gives us access to \( f \) in a way compatible with quantum computers is by no means trivial. However,
if we did have a classical implementation of $f$ available (for example as a circuit of boolean gates) and the capacity to build quantum computers, we would also be capable of building a quantum circuit implementing $f$. This follows from the fact that quantum circuits are a generalization of classical computers [20].

2.2.1 Grover’s Algorithm

Grover’s quantum search algorithm is one of the two best-known quantum algorithms in existence\(^1\). The details of how it operates can be found in the original paper [15] and in more recent treatments [6] [20], so we will not elaborate on them further.

What is relevant for this thesis is that Grover’s algorithm can solve the unstructured search problem over the space $\{0, 1\}^n$ of $n$-bit strings in time $O(\sqrt{2^n}) = O(2^n)$. If more than one marked element exists, namely $2^k$ marked elements in a search space of size $2^n$, then the runtime is $O(2^{n-k})$. Basically, the runtime is the square root of the reciprocal of the solution density (size of search space divided by the number of solutions), and does not actually depend on the size of the search space directly. This fact will be relevant when combining Grover with classical search heuristics.

It is still important to get at least an intuition for how Grover’s algorithm operates, to better understand other algorithms derived from it.

The algorithm begins by creating an equal superposition of all states. Then, it repeatedly applies a small circuit that causes the amplitude of the marked elements to grow. This is achieved by an application of the oracle followed by the so-called Grover diffusion operator. After a sufficient number of iterations depending on the number of marked elements present, the probability of hitting a marked element when measuring has increased to a constant, at which point a measurement is made.

Grover’s algorithm doesn’t depend on knowing the exact number of marked elements in advance, but it does require the order of magnitude of that number to be known. When run for too many iterations, the amplitude of marked elements starts decaying again, so the algorithm is not stable in the sense of converging towards a solution and staying there indefinitely. This is not an obstacle to the full generality of the algorithm, as we can just do an exponential search over the number of solutions (marked elements). We first determine if there is exactly 1 solution, then if there are 2 solutions, 3-4 solutions, 5-8 solutions, etc. This is very simple and incurs only a factor $n$ overhead for a search space of size $2^n$. There are better methods to solve this

---

\(^1\)The other one being Shor’s algorithm for factoring prime numbers in polynomial time [23].
issue of generality, described e.g. in [20], but this one is very simple and sufficient for our purposes.

Grover’s search algorithm was a significant breakthrough in quantum computing, but it left some question unanswered. What if we are dealing with a problem that is not purely a search problem? Every decision problem can be treated as a search problem of course, but when we do, we might leave some performance on the table by not exploiting the problems’ structure.

One situation is that we have access to an efficient search heuristic. This means that we have a way of drawing samples from the search space that is better than drawing uniformly at random, in terms of expected number of samples until we hit a marked element. This situation is discussed in the next section.

After that, we will look at two different ways to generalize Grover’s algorithm, each of which addresses different circumstances where pure Grover search is not enough.

2.2.2 Using Grover with heuristics

In some cases, the runtime provided by Grover’s algorithm is not satisfactory. Consider for example the 3-SAT problem: given a 3-CNF formula\(^2\), decide whether or not there exists a satisfying assignment.

Grover’s search algorithm solves this problem by using a SAT verifier as an oracle. A SAT verifier is a function that given a formula and a satisfying assignment, outputs whether the assignment satisfies the formula. To convert it to an oracle, we fix the first input of the function to the formula we are trying to solve, and allow the quantum algorithm to query the function by passing a quantum superposition of potential assignments as the second argument.

The best classical algorithm for the 3-SAT problem known is PPSZ, named after Paturi, Pudlák, Saks and Zane [21], for which the runtime is known to be \(O(1.308^n) \approx O(2^{0.387 \cdot n})\) [17]. This is substantially better than what the generic quantum search algorithm can offer us.

Ideally, we would want to combine (i.e. stack on top of each other) both the classical and quantum algorithms, to hopefully achieve a runtime of \(O(2^{0.5 \cdot 0.387 \cdot n}) = O(2^{0.194 \cdot n})\).

It turns out that this is possible, but only because PPSZ is not just any algorithm, but it can be classified as a search heuristic. In a nutshell, a search

\(^2\)A 3-CNF formula is a boolean formula in conjunctive normal form with at most 3 variables per clause
2. **Quantum Computing Basics**

Heuristic is a classical algorithm that runs in guaranteed polynomial time and then terminates, returning the correct solution with positive probability, and some wrong solution if it fails. If this probability is large enough, the expected runtime until we hit a solution with high probability is small.

If the algorithm is *exponentially* better than the naive approach of just picking a random sample out of the search space, i.e. for the case of $n$-bit strings it has a success probability of $2^{-kn}$ for $k \in (0, 1)$, then the number of iterations required until hitting a solution with high probability is $2^{kn} \cdot poly(n)$ [26].

We can combine search heuristics with Grover’s algorithm by putting the search heuristic in front of the oracle to build a new oracle. In the case of SAT, this means that the quantum search is performed over the space of random bits to give as secondary input to PPSZ (the first input being the formula, which is fixed), and then the SAT verifier checks whether or not PPSZ’s output satisfies the formula.

One technical detail to note is that we always give the same number of random bits to PPSZ on every run, that is, even if the algorithm consumes different number of random bits on different inputs, we always make as many bits available as it needs in the worst case. This is needed to be able to construct an equal superposition over the search space in an efficient manner, and it doesn’t affect the runtime at all, since what we ultimately care about is the solution density of the search space and not its absolute size.$^3$

Furthermore, a search heuristic by definition runs in time $poly(n)$, so it can never consume more than $poly(n)$ random bits, and thus the oracle is guaranteed to run in polynomial time. For details and formal proofs of these ideas, see [6].

### 2.2.3 Generalization 1: Amplitude Amplification

We have briefly mentioned how Grover’s algorithm works: It has a starting step where it constructs an equal superposition over all basis states, an iteration step that makes the amplitude of all marked elements grow, and a measurement at the end.

Let us assume there is exactly one marked element in the space $\{0, 1\}^n$ of $n$-bit strings, and our goal is to find it. If we run Grover but sample the quantum state after the starting step, the success probability is $2^{-n}$, since every element of the search space is equally likely to be sampled. By allowing Grover to finish running, we instead boost the success probability to a constant, using $2^n$ computation steps in the process.

$^3$For instance, if we were to add 3 unneeded random bits to the input, we would increase the size of the search space eightfold, but also increase the multiplicity of solutions eightfold, which cancels out.
2.2. Quantum Search Algorithms

But what if we have some other quantum algorithm with a success probability of $2^{-p}$ on the same problem? It turns out we can insert this algorithm into the starting and iteration steps of Grover’s algorithm and boost the success probability to a constant in only $2^p$ iteration steps [6].

This is somewhat abstract, but in practice it usually means that we can nest several layers of Grover’s algorithm: We run Grover to find an element in a search space, but the oracle we call is not a classical deterministic algorithm at all, but in fact runs Grover internally to find out if the input it was given is marked.

2.2.4 Generalization 2: Search via Quantum Walk

In some sense, Grover’s algorithm is the quantum equivalent of classical Monte Carlo sampling. By this we mean the trivial algorithm which just draws samples uniformly at random from the search space until it hits a solution. By “drawing a sample” we mean evaluating the oracle on the chosen element to see if it is marked. In practice where we need to actually implement the oracle, “drawing a sample” is often a more involved process, resulting in some knowledge about the sample encoded in a datastructure of some sort.

In classical computing, it is well established that a way to substantially increase the speed of Monte Carlo sampling is by using information gained from taking previous samples to make further samples cheaper to evaluate. This is known as Markov Chain Monte Carlo sampling [16]. As the name suggests, instead of taking successive independent samples from the search space, a Markov chain is used to move randomly through the search space. Each sample taken is the neighbor of the previous sample (in some sense), so in many cases it will be cheaper to evaluate as it shares information with a sample we have already evaluated.

If the Markov chain mixes towards the uniform distribution, and does so fast enough, then sampling in this way covers the search space almost as well as if we were taking independent samples. Depending on how much time we save on evaluating a sample given information about its predecessor in the random walk, there is a tradeoff between convergence time and computation saved, which might be favorable as a whole. Here the idea of a datastructure containing information about the current sample becomes more explicit, and the speedup gained is derived from mutating the information in the datastructure from one sample to its neighbor, instead of rebuilding it from scratch at every sampling step.

It is possible to translate this approach to the quantum setting. The basic idea is that if we have a Markov chain with certain properties that make it
2. Quantum Computing Basics

compatible with the reversible computation requirements of quantum computing, i.e. ergodicity and symmetry, then we can solve a search problem in the space over which the chain is defined, in the square root of the time a classical Markov chain Monte Carlo algorithm would need.

Let us make all of this more precise. Assume that we have a Markov Chain over a space \( \mathcal{X} \), and we are searching for a marked element \( m \in \mathcal{M} \), where \( \mathcal{M} \) is the space of all marked elements. There is a membership oracle deciding for any \( x \in \mathcal{X} \) whether or not it is contained in \( \mathcal{M} \). We need to define three costs associated with the running of any Markov chain Monte Carlo algorithm:

- **S**: The setup cost, i.e. the cost of sampling an element \( x \in \mathcal{X} \) from scratch.
- **U**: The update cost, i.e. the cost of sampling an element \( y \) among all neighbors of an element \( x \in \mathcal{X} \) that we have already sampled.
- **C**: The checking cost, i.e. the cost of evaluating whether a sampled element belongs to \( \mathcal{M} \).

These costs can be used to measure the runtime of the algorithm or, as in our case, the number of queries made to the oracle.

Let \( \delta > 0 \) be the eigenvalue gap of the Markov Chain\(^4\), and \( \varepsilon := \frac{|\mathcal{M}|}{|\mathcal{X}|} \) the solution density of the search space.

It costs at most

\[
O(S + \frac{1}{\delta\varepsilon}(U + C))
\]

for a classical computer to determine if \( \mathcal{M} \) is empty. For a quantum computer, the cost is at most

\[
O(S + \frac{1}{\sqrt{\delta\varepsilon}}(U + C)).
\]

For the details of the algorithms accomplishing this, as well as fully formal definitions and proofs, see [18].

One non-obvious point necessary to bring up is that the datastructure we only mentioned in passing for the classical case actually needs to be implementable on a quantum computer. It turns out to be possible to implement

---

\(^4\)The eigenvalue gap of a Markov chain is closely linked to its mixing time. It is a known quantity for the specific graphs we will be working with, so we will not elaborate on its derivation.
arbitrary data structures on quantum computers if we have enough qubits available for them. However, in order to create the interference patterns required for a specific quantum algorithm to work properly, it is required that the data structure be deterministic, in the sense that the same set of information always gets stored in memory in the exact same way. There are data structures that do not fulfill this requirements, for instance most self-balancing binary trees, whose memory layout depends on the insertion order of the elements.

See [2, Section 6] for more discussion of this and other implementation issues, all of which can ultimately be overcome.

2.2.5 The Element Distinctness Algorithm

The rather abstract framework of quantum search via random walk, which has applications to many problems, came about by studying the element distinctness problem, which is defined as follows:

*Given a function $f: [n] \to \mathbb{N}$ as an oracle, find $x, y \in [n]$ such that $f(x) = f(y)$ or show that no such $x, y$ exists.*

It is clear that in the classical world we need to query all the inputs of the function to answer this question. Attempting to use Grover’s algorithm for this problem fails completely: The search space of $x, y$ pairs has size $n^2$, so even with the quantum search speedup, we are no better off than with the classical algorithm.

A clever search algorithm using amplitude amplification was able to cut this time down to $O(n^{\frac{3}{4}})$ for quantum computers [7], and a breakthrough by Ambainis in 2003 allowed the problem to be solved in only $O(n^{\frac{3}{2}})$ time [2], matching the lower bound for the problem [1].

The ideas of Ambainis (and parallel ideas by Szegedy et al [25]) were later developed into the general framework described above [18].

Let us elaborate a bit on how Ambainis’ element distinctness algorithm works, since we will be using this later in the thesis to provide a refined runtime analysis for related algorithms on specific inputs. We have already established that any quantum walk based search algorithm has a runtime of $O(S + \frac{1}{\sqrt{n}}(U + C))$.

The element distinctness algorithm works as follows:
2. Quantum Computing Basics

- The setup step consists of evaluating \( f(x) \) for all \( x \) in a u.a.r. subset \( X \subset [n] \) of size \( n^\frac{2}{3} \), storing all the \( x, f(x) \) pairs in a data structure.

- The update step consists of removing one element \( x \in X \) u.a.r from the data structure, evaluating \( f(x') \) for another element \( x' \in [n] \setminus X \), and adding \( x', f(x') \) to the data structure. In other words, the set \( X \) is updated to \( (X \setminus \{x\}) \cup \{x'\} \).

- The checking step consists of testing whether \( f(x) = f(y) \) for every pair \( x, y \in X \times X \).

Recall that we are measuring the performance of the algorithm only by the number of times the oracle is called, and the cost of any other computations is ignored. This makes it easy to determine the cost of the 3 steps:

- \( S = n^\frac{2}{3} \),
- \( U = 1 \),
- \( C = 0 \).

The Johnson Graph

To get a definite runtime for the algorithm, we need to know the values of 5 variables, namely \( S, U, C, \varepsilon \) and \( \delta \). Of these, we have already derived all but the last two, which depend on the graph induced by the Markov chain. This graph has a vertex for each subset of \([n]\) of size \( n^\frac{2}{3} \), and two vertices share an edge if their associated subsets differ by exactly one element.

Graphs with this structure are known as Johnson Graphs, written as \( J(\mathcal{X}, s, l) \), characterized by 3 parameters: \( \mathcal{X} \) is the universe of elements, \( s \) is the size of subsets associated with the vertices, and \( l \) is the size of the intersection between subsets required to form an edge. This graph is known to have an eigenvalue gap \( \delta = \theta(n^\frac{1}{2}) \) [18].

See Figure 2.1 for an example of a Johnson graph. The Graph we are dealing with in the algorithm is \( J([n], n^\frac{2}{3}, 1) \). The solution density \( \varepsilon \) can be derived from elementary combinatorics. To keep the analysis more general, we will consider the graph \( f = J([n], n^r, 1) \) for some \( r \in (0, 1) \).

There are \( \binom{n}{s} \) vertices in the graph, each of which contains \( n^r \) elements, so there are \( \binom{n}{s} \cdot n^r \) elements contained in vertices in total. Since the graph is symmetric, each element is contained in the same number of vertices, namely \( \binom{n}{s} \cdot n^r / n \). Looking at a single vertex of \( f \), the probability of it containing a specific element is therefore \( n^{r-1} \), and the probability of containing \( k \) specific elements (where \( k \) is a constant independent of \( n \)) is \( \theta(n^{(r-1)k}) \).
2.2. Quantum Search Algorithms

Figure 2.1: An example Johnson graph, namely \( J([4], 2, 1) \), the graph of all 2-element subsets of \([4]\) where two vertices are connected if they differ by exactly 1 element.

The latter statement should be intuitively clear; it follows from the fact that the subgraph of \( J \) induced by all vertices containing the element \( x \) is isomorphic to \( J([n] \setminus \{x\}, n' - 1, 1) \) which is \( \theta(n'^{-1}) \) times the size of the original Johnson Graph. Each vertex of this induced graph has a probability \( \theta(n'^{-1}) \) of containing a specific second element, so we get a second induced graph that is smaller by another factor of \( \theta(n'^{-1}) \) and so on, \( k \) times.

If our instance of element distinctness has at least one pair \( x, y \) such that \( f(x) = f(y) \), the solution density \( \epsilon \) is at least \( \theta(n^{(\frac{1}{3}-1)\cdot 2}) = \theta(n^{-\frac{2}{3}}) \). The runtime of Ambainis’ element distinctness algorithm is therefore

\[
S + \frac{1}{\sqrt{\delta \epsilon}}(U + C) \\
= n^\frac{2}{3} + \frac{1}{\sqrt{\delta}} \frac{1}{\sqrt{\epsilon}} \\
\leq n^\frac{2}{3} + \theta(n^{\frac{1}{3}})\theta(n^{\frac{1}{3}}) \\
= \theta(n^{\frac{2}{3}})
\]

This shows that the size of subsets \( n^{\frac{2}{3}} \) was not chosen arbitrarily, but rather
is the exact value that balances the contributions of the startup and update costs to the total cost.

**From Element Distinctness to Quantum Pair Finding**

We have studied the element distinctness algorithm in detail, but why are we focusing so much on this specific algorithm? A priori it has nothing at all to do with unique sink orientations, the topic of this thesis.

It turns out that the algorithm is not specific to the element distinctness problem at all. In fact, we can use the same algorithm to determine if there is a pair \( x, y \) fulfilling an arbitrary predicate \( p(x, y, f(x), f(y)) \), by using that predicate whenever we would perform the check \( f(x) = f(y) \) in the element distinctness algorithm, but leaving the rest of the algorithm unchanged [8].

The generalized version of the element distinctness algorithm is called the quantum pair finding algorithm, since it finds any marked pair in some search space, just how Grover’s algorithm finds any marked single element. The algorithm can be further generalized to the subset finding algorithm, which finds a subset of size \( k \) with an arbitrary property in time \( O(n^{\frac{k}{k+1}}) \) [2].
As mentioned in the introduction, unique sink orientations are orientations of the hypercube with the property that each face possesses a unique sink. Behind this deceptively simple statement there lurks a complex class of combinatorial object with many interesting properties. In order to understand USOs, we must first make the informal definition above precise.

3.1 Definitions

3.1.1 The Cube

Definition 3.1 (Cube) The $n$-cube is an undirected graph $Q_n = (V, E)$, where

$$V := \{0, 1\}^n = \{0, 1\} \times \ldots \times \{0, 1\}$$

$$E := \{\{u, v\} \mid u, v \in V, d_H(u, v) = 1\},$$

with $d_H$ being the Hamming distance.

Definition 3.2 (Face) Let $w \in \{0, 1, *\}^n$ be a word. The Face (or subcube) with label $w$ is the induced subgraph of $Q_n$ with vertex set

$$V' := f(w_1) \times f(w_2) \times \ldots \times f(w_n)$$

where

$$f(i) := \begin{cases} 
\{0\} & \text{if } i = 0, \\
\{1\} & \text{if } i = 1, \\
\{0, 1\} & \text{if } i = *.
\end{cases}$$
3. Unique Sink Orientations

$w$ can be interpreted as a pattern similar to a regular expression, where all vertices that match it are part of the face. It is clear that a face of a cube is a cube as well. More precisely, a face with label $w$ is isomorphic to $Q_i$ where $i = \sum_{j=1}^{n} w_j = \ast$.

The face labeled $\ast^n$ corresponds to the cube itself. On the other extreme, a 0-dimensional face i.e. one labeled with a word $w \in \{0,1\}^n$, i.e. not containing $\ast$, corresponds to a single vertex. 1-dimensional faces are known as edges, and they correspond to the edges of the cube in the graph-theoretic sense. Finally, $(n-1)$-dimensional faces are known as facets.

How many faces does the cube have? Each face is labeled by a different word $w$, and no two words $w$ and $w'$ correspond to the same face, since their sets of vertices don’t match by definition. Therefore, there must be exactly $3^n$ faces.

Furthermore, there are exactly

$$\binom{n}{k} \cdot 2^{n-k}$$

faces of dimension $k$. This expression follows from elementary combinatorics: $\binom{n}{k}$ is the number of ways in which we can position the $k$ stars in the word $w$, while $2^{n-k}$ is the number of ways to fill the gaps between the stars with zeros and ones.

3.1.2 Unique Sink Orientations

When we take a cube and make each of its edges directed, we obtain an orientation of the cube.

Definition 3.3 (Outmap & Orientation) Let $Q_n = (V, E)$ be the $n$-cube. An outmap is a function $\phi : \{0,1\}^n \mapsto \{0,1\}^n$ mapping each vertex $v \in V$ to the orientations of its incident edges.

$\phi$ induces an orientation $O = (V, E_{out})$, which is a directed graph with

$$E_{out} := \{(v, v \oplus e_i) \mid v \in V, i \in [n], \phi(v)_i = 1\}.$$ 

$\phi(v)$ is known as the outvalue of $v$.

See Figure 3.1 for a few examples. Note that a proper orientation always has exactly one edge connecting a pair of vertices at Hamming distance 1. This is the same as saying that every pair of neighbors in the cube agree (through
Figure 3.1: Example orientations on the 2-cube, drawn in the top row as directed graphs according to definition 3.3, and in the bottom row as cubes with oriented edges. Left: A proper orientation. Center: A degenerate orientation. Right: A unique sink orientation.

their respective outvalues) how the edge between them is oriented. If this doesn’t hold we get a degenerate orientation, where either both endpoints claim that the edge is pointing towards them, resulting in no directed edge between them, or both claim the edge is oriented away from them, resulting in both possible directed edges existing at the same time.

We can now finally define unique sink orientations with full precision.

**Definition 3.4 (USO)** A unique sink orientation is an orientation of the n-cube, such that the orientation induced on each face has exactly one sink.

It is easy to see that a USO is always a proper orientation. However, allowing degenerate orientations to exist in principle is useful for keeping the formalism as simple as possible.

### 3.2 Fundamental Properties of USOs

Having formally defined them, we are now ready to explore the structure of USOs. We begin by stating some known properties originally found in [24], and updated to the notation of this thesis.

**Lemma 3.5 ([24], Lemma 2.1)** Let \( \phi \) be an outmap. Then for any vector \( z \),

\[
\phi'(x) := \phi(x) \oplus z
\]

is the outmap of a USO if and only if \( \phi \) is.

**Proof** It suffices to show the case where \( z = e_i \) for some \( i \).
3. Unique Sink Orientations

Assume $\phi$ is the outmap of a USO. Consider the two facets of the cube along direction $i$. Let their sinks under $\phi$ be $u$ and $v$ respectively. One of them, say $u$, must be the global sink. Under $\phi'$, $v$ becomes the global sink because its only outgoing edge is flipped. At the same time, $u$ stops being the global sink.

This argument holds for any face spanning dimension $i$. Faces not spanning $i$ are not affected by the change from $\phi$ to $\phi'$. Therefore, the USO property is preserved on all faces.

The same argument holds when switching the roles of $\phi$ and $\phi'$.

\textbf{Lemma 3.6 ([24], Lemma 2.2)} Let $\phi$ be the outmap of a USO. Then $\phi$ is bijective.

\textbf{Proof} It suffices to show injectivity. Let $u$ and $v$ be vertices such that $\phi(u) = \phi(v)$. Consider the orientation $\phi'(x) := \phi(x) \oplus \phi(u)$, which is a USO by Lemma 3.5. $u$ and $v$ are both sinks in $\phi'$, so it follows that $u = v$.

The above lemma of course holds for all faces of a USO, per definition. It turns out that this property of recursive bijectivity is a complete characterization of USO outmaps. It can be expressed in symbols as follows:

\textbf{Corollary 3.7 ([24], Lemma 2.3)} Let $\mathcal{O}$ be an orientation with outmap $\phi$. Then $\mathcal{O}$ is a USO if and only if

$$((\phi(u) \oplus \phi(v)) \land (u \oplus v)) \neq 0^n \quad \forall u, v \in \{0, 1\}^n. \quad (3.1)$$

\textbf{Proof} If $\mathcal{O}$ is a USO, then for any pair $u, v$ of vertices consider the face on which $u$ and $v$ are antipodals. This face spans exactly the dimensions $i$ where $u_i \oplus v_i \neq 0$, and in one of those dimensions it must be the case that $\phi(u)_i \neq \phi(v)_i$ as well, otherwise $\phi$ wouldn't be bijective on that face, violating Lemma 3.6.

For the other direction, there exists a pair $u, v$ of vertices for which (3.1) doesn't hold. Consider the outmap

$$\phi'(x) := \phi(x) \oplus \phi(u).$$

The face on which $u$ and $v$ are antipodals has two sinks under $\phi'$, namely $u$ and $v$, which means that $\phi'$ is not the outmap of a USO, and (per Lemma 3.5) neither is $\phi$.

3.3 Inherited Orientations

One important property of unique sink orientations is the fact that we can construct smaller USOs from a given USO, by collapsing some of their di-
3.3. Inherited Orientations

dimensions and determining the orientation of the resulting smaller cube according to a specific rule. The resulting USO is called an inherited orientation.

To create an inherited orientation we basically take the cube, split it into $2^{n-k}$ $k$-dimensional faces (let's call these the "inner cubes"), noting that this set of faces disjointly covers the cube. Then each of these inner cubes takes the role of a logical vertex in an "outer cube" of dimension $n-k$. The outvalue of a logical vertex is determined by finding the sink of the inner cube, taking the outvalue of that vertex (considered on the entire original cube) and keeping only the dimensions corresponding to the outer cube.

First of all, we define a new type of bitstring, the so-called face patterns, which tell us along which dimensions the cube should be collapsed, i.e. which dimensions correspond to inner cubes and which to the outer cube.

**Definition 3.8 (Face Pattern)** A Face Pattern of the $n$-cube is a bitstring $p \in \{\_*, 0, 1\}^n$. A face $f$ of the cube is said to match $p$ iff

$$p_i = * \iff f_i = * \quad \forall i \in [n].$$

The inner dimension of $p$ is $\sum_{i=0}^{n} [p_i = *]$ and its outer dimension is $\sum_{i=0}^{n} [p_i = \_]$.

Just like a face is in some sense a regular expression describing a set of vertices, a face pattern is a regular expression describing a set of faces, a meta-regex if you will. Since the * symbol is already used up on the definition of faces, we need to introduce a new symbol \_ as a second wildcard. There is nothing holding us back from allowing 0 or 1 in our meta-regex, i.e. allowing any $p \in \{0, 1, \_, *\}^n$, but in practice we are only interested in sets of faces that completely cover the cube, i.e. where every vertex belongs to exactly one face matching the pattern.

With this definition established, we can now formalize inherited orientations.

**Definition 3.9 (Inherited Orientation)** Let $O$ be an orientation on the cube $Q_n$ with outmap $\phi$, let $p \in \{0, 1, \_, *\}^n$ be a face pattern with outer dimension $k$, let $\text{sink}_O : \{0, 1, \_, *\}^n \mapsto \{0, 1\}^n$ be the function mapping faces of $O$ to their sinks, and let

$$\mathcal{P} := \{f \in \{0, 1, \_\}^n \mid f \text{ matches } p\}$$

be a domain. Then, $O'$ is the $p$-inherited orientation of $O$ on the cube $Q_k$, with
Figure 3.2: Examples of inherited orientations. Left: a USO. Center: showing the faces matched by 3 different patterns, with their sinks highlighted. The patterns, from top to bottom, are $p_1 = _* _$, $p_2 = _* _$ and $p_3 = _* _$. Right: Inherited orientations generated by the patterns.

$\textsf{outmap}\quad \phi' : \mathcal{P} \mapsto \mathcal{P}$

$\phi'(f)_i := \begin{cases} * & \text{if } p_i = *, \\ \phi(\text{sink}_\mathcal{O}(f))_i & \text{otherwise.} \end{cases}$

See Figure 3.2 for a couple of examples of inherited orientations.

There is one important thing we need to remark here: $\mathcal{P}$ is not the usual domain $\{0,1\}^k$ we use for the cube graph but it is isomorphic to it, in the obvious way of replacing all * found in the words of $\mathcal{P}$ with the empty word, thus reducing the length of the words from $n$ to $k$, the outer dimension of $\rho$. The same isomorphism holds for the outvalues themselves. It is therefore reasonable to actually call $\phi'$ the outmap of an orientation, and we implicitly
3.3. Inherited Orientations

assume that the translation to the standard domain is made whenever we use inherited orientations.

Let us train our intuition a bit: If we choose the pattern \( p = \_^n \), then all \( 2^n \) vertices match the pattern and the \( p \)-inherited orientation of \( O \) is simply \( O \) itself. On the other extreme, a pattern \( p = *^n \) is only matched by the entire cube, and the \( p \)-inherited orientation is a single vertex. The most frequent patterns we will encounter have inner dimension exactly 1; the inner cubes are the edges along a specific direction, and if we repeatedly apply such a pattern, we can produce any inherited orientation. For example, the \((\_ * \_ *)\)-inherited orientation of \( O \) is the same as the \((\_ * \_ *)\)-inherited orientation of the \((\_ * \_ *)\)-inherited orientation of \( O \).

The reason we have chosen to formalise inherited orientations in this way is that the face patterns give a clear mental picture of their structure, and it is made explicit that we are collapsing a set of complementary faces, treating them as if they were vertices by ignoring their inner structure.

On a more technical note, inherited orientations can be well defined even if the orientation we are dealing with is not a USO. We just need to make sure that the orientation is a USO on all faces matching the face pattern, because otherwise the sink function is not well-defined.

Inherited orientations are interesting to us because, as their name already suggests, they in fact inherit the USO property from their progenitor:

**Lemma 3.10 ([24], Lemma 3.1)** Let \( O \) be a USO with outmap \( \phi \). Then, for all \( p \in \{\_ *, \_ * \}^n \), the \( p \)-inherited orientation \( O' \) with outmap \( \phi' \) is a USO.

**Proof** If \( O' \) is not a USO, we restrict our attention to some face on which bijectivity is violated, which must exist according to Lemma 3.6. The orientation on this face is an inherited orientation of some face of the original cube. Take a pair \( f, g \) of logical vertices with the same outvalue \( \phi'(f) = \phi'(g) \), and consider the vertices \( u := sink_O(f) \) and \( v := sink_O(g) \). Note that \( \phi(u)_i = \phi(v)_i \) for all \( i \) where \( p_i = * \), by virtue of being sinks of \( f \) and \( g \) which match the pattern \( p \), and for all other \( i \) by assumption. This means the original orientation cannot be a USO, which is a contradiction. \( \square \)

Note that the only-if direction of this lemma does not hold. It is also clear that the sink does not have a privileged position in this lemma. By applying Lemma 3.5 we can construct an inherited orientation by taking the sources of all logical vertices, or indeed vertices with any other fixed out-value.

The ability to take inherited orientations is extremely important for the development of efficient sink-finding algorithms [24, Section 3], and it is the first nontrivial structure that was found in USOs.
3. **Unique Sink Orientations**

For a different, more mathematical but perhaps simpler take on the subject of inherited orientations, see [24].
Chapter 4

Pseudo-USOs

4.1 Algorithmic Motivation

In this thesis, we are interested in analysing the USO recognition problem, defined as follows:

\[ \phi : \{0, 1\} \rightarrow \{0, 1\} \]

Given a function \( \phi : \{0, 1\} \rightarrow \{0, 1\} \) as an oracle, determine whether or not it is the outmap of a USO.

When we have a negative instance of the USO recognition problem, Corollary 3.7 tells us that we can find a pair of vertices proving that the USO condition is violated. The two vertices have the same outvalue on the face spanned between them, i.e. the subcube of which they are antipodals.

We will call such a vertex pair a certificate. The existence of a certificate for no-instances shows that the problem is contained in the complexity class \( \text{coNP} \) [3].

It’s not immediately clear what can be said about the algorithmic difficulty of this problem. If \( \phi \) is given as a boolean circuit, the problem is in fact known to be \( \text{coNP} \)-complete [12]. However, defining it in such a way leaves open the theoretical possibility of solving the problem by looking at the structure of the circuit implementing \( \phi \), and in such a setting it is exceedingly hard to prove lower bounds. In particular, proving an exponential lower bound would imply that \( P \neq \text{coNP} \), which would answer one of the biggest open questions in complexity theory, closely related to the famous \( P = NP \) question [3].
4. Pseudo-USOs

We can sidestep the whole issue by giving $\phi$ as a black box oracle, which is also the appropriate choice in the context of quantum algorithms. The runtime of an algorithm is measured in the unit cost model, that is we count the number of queries to the oracle it takes to solve the problem, while the costs of all other computations performed are ignored.

We can immediately show both a classical and quantum lower bound for the problem, separated by a factor of $O(n^2)$. The proof contains a straightforward construction which is also used in [12] to show coNP-completeness.

Lemma 4.1 Let $\phi : \{0, 1\}^n \mapsto \{0, 1\}^n$ be an outmap given as an oracle. It takes $2^n$ queries to $\phi$ in the worst case to determine whether or not it is the outmap of a USO. For a quantum computer, $\Omega(\sqrt{2^n})$ queries are required.

Proof We show that an unstructured search problem can be mapped onto the USO recognition problem. Given a function $g : \{0, 1\}^{n-1} \mapsto \{0, 1\}$, we construct an outmap $\phi$ as follows:

$$
\begin{align*}
\phi : \{0, 1\}^n &\mapsto \{0, 1\}^n \\
\phi(0 \cdot x) &:= g(x) \cdot x \\
\phi(1 \cdot x) &:= 1 \cdot x
\end{align*}
$$

If $g \equiv 0$, this yields an orientation with outmap $\phi(x) = x$, which is easily seen to be a USO, as the sink of any face is the unique vertex with lowest Hamming weight contained in that face. This orientation is known as the uniform orientation.\(^1\)

If $g \not\equiv 0$, there exists a $y$ such that $g(y) = 1$, and $\phi$ is not the outmap of a USO, as certified by the pair of vertices $0 \cdot y$ and $1 \cdot y$.

Therefore, the USO recognition problem is at least as hard as the unstructured search problem, and the claimed bounds are well known lower bounds for the latter [15].

This proof makes use of a construction where the certificate consists of two vertices that are very close together, namely at Hamming distance 1. There are only $2^n \cdot poly(n)$ such pairs, among the $4^n$ possible pairs of vertices.

If we condition on the distance between the certificate vertices being 1 (or any other constant), this lower bound is actually matched by the trivial algorithm (i.e. exhaustive search classically, Grover search quantumly). The upper bound provided by the trivial algorithm gets worse as we allow the distance between the certificate vertices to increase\(^2\), but at the same time the lower bound disappears. It makes sense intuitively to analyse algorithms

\(^1\)Sometimes also called the downwards orientation or the trivial orientation

\(^2\)In the classical case, the runtime stays at $2^n$ if we cache all oracle calls in memory, of course.
4.2 Definition and Existence Proofs

and lower bounds for this problem with regards to the minimum allowed
distance between certificate vertices.

The rest of this chapter shows that a new class of interesting combinatorial objects arises quite naturally when following this line of thought. The algorithmic side of things will be developed further in Chapter 5.

4.2 Definition and Existence Proofs

Consider all non-USO orientations on the $n$-cube. Each such orientation has
a finite number of certificates per Corollary 3.7, and therefore also has a
smallest distance between the vertex pair of any such certificate. We classify
these orientations according to that minimum distance, which we express
in terms of $n$, i.e. every non-USO has a parameter $c \in (0, 1]$ such that the
distance is $cn$. Note that even though $c$ is a real-valued parameter, it can
take only $n$ distinct values.

Given a non-USO orientation $\phi$ with parameter $c$, it follows that all its faces
of dimension $d < cn$ are USOs. At the same time, there exists at least one
face of dimension exactly $cn$ which is not a USO, having as a certificate two
of its antipodals. The orientation on this face must obey the restriction that
the orientation on each of its facets is a USO, while the whole orientation is
not. We will analyse this kind of orientation, which we call a pseudo-USO.

Definition 4.2 (Pseudo-USO) A Pseudo-USO is an orientation of the $n$-cube
such that it is not a USO, but all its faces of dimension $d < n$ are.

Every non-USO orientation with parameter $c$ contains a pseudo-USO of
dimension $cn$. Therefore, analyzing the properties of pseudo-USOs might
yield insights that result in better algorithms for the USO recognition prob-
lem.

Before going further, we need to show that such objects even exist for arbi-
trary $n$, which is not obvious a priori.

Observation 4.3 There exists a family of functions that are the outmaps of pseudo-
USOs.

Proof We claim that functions of the form

$$\phi_n : \{0, 1\}^n \mapsto \{0, 1\}^n$$

$$\phi_n(x)_i := x_i \oplus x_{(i-1) \mod n}$$

are outmaps of pseudo-USOs. Such an outmap clearly doesn’t represent a
USO, since $\phi(0^n) = \phi(1^n) = 0^n$. 

25
We prove that all faces except the cube itself are USOs, by induction on the dimension $k$ of the face.

The base case $k = 0$ is trivial, since all orientations on a single vertex are USOs.

For the step case, consider a face with label $w$. Choose a $j \in [n]$ such that $w_{((j−1) \mod n)} \in \{0, 1\}$ and $w_j = \ast$. This $j$ must exist because otherwise the face would either be a single vertex or the whole cube. Assume WLOG that $w_{((j−1) \mod n)} = 0$ (the other case is analogous). For any vertex $v$ in the face labeled $w$, it follows that $\phi_n(v)_j = v_j$. This means that the orientation on the face is combed in direction $j$. An orientation consisting of two USOs with all edges between them combed in the same direction is known to be a USO [22, Section 4].

The first few members of this family of pseudo-USOs are shown in Figure 4.1. This proof shows that the USOs occurring on the facets of $\phi_n$ are so-called decomposable USOs, recursively defined as orientations consisting of 2 facets which are themselves decomposable USOs, connected by a set of combed edges.

More specifically, the facets of $\phi_n$ are recursively combed USOs, a subclass of decomposable USOs where the dimensions in which the faces are combed are globally aligned with each other. These USOs are exactly those whose outmaps can be generated by deterministic finite state automata (epsilon-free letter transducers to be precise, see [5] for details).

Note that it is possible to embed pseudo-USOs anywhere in the cube:

**Observation 4.4** For every face $f$ of the $n$-cube, there is a non-USO that contains a pseudo-USO embedded in $f$ and no other pseudo-USOs.

**Proof** Construct the non-USO in question by taking the uniform orientation$^3$ and changing the edges within $f$ to form any pseudo-USO. Faces dis-

---

$^3$Recall that this is the orientation with outmap $\phi(x) = x$
4.2. Definition and Existence Proofs

Joint from $f$ are USOs by construction, and strict subfaces of $f$ are USOs by definition of pseudo-USOs. Strict superfaces of $f$ are not pseudo-USOs, again by definition.

The remaining faces intersect $f$ but neither strictly contain or are contained by $f$. Within all such faces, there exists at least one dimension not spanned by $f$, which is combed per definition. An orientation consisting of 2 USO facets connected with combed edges is always a USO [22, Section 4].
4. Pseudo-USOs

Figure 4.3: Base case of Lemma 4.7: These are the only 2 pseudo-USOs that exist on the 2-cube, up to isomorphism.

4.3 Combinatorial Properties

Pseudo-USOs are quite similar to USOs in their combinatorial structure. In order to establish some properties that are unique to them, we first observe that 2 results previously established for USOs, Lemma 3.5 and Lemma 3.10 also hold for Pseudo-USOs. The proofs are omitted as they contribute no additional insight.

Observation 4.5 Let \( \phi \) be an outmap. Then for any vector \( z \),

\[
\phi'(x) := \phi(x) \oplus z
\]

is the outmap of a pseudo-USO if and only if \( \phi \) is.

Observation 4.6 Let \( O \) be a pseudo-USO with outmap \( \phi \). Then, for all \( p \in \{\pm, \ast\}^n \setminus \{\pm^n\} \), the \( p \)-inherited orientation \( O' \) is a pseudo-USO.

Note that there is a slight difference between Lemma 3.10 and Observation 4.6, namely that the latter doesn’t hold for the \( \pm^n \) face pattern. This is because the whole concept of inherited orientations is not well defined in this case, since a global unique sink is not guaranteed to exist. In the case of a USO, taking its \( \pm^n \)-inherited orientation is well-defined but useless (the result is the original orientation).

Looking at examples of Pseudo-USOs such as the one given in Figures 4.1 and 4.2, an interesting property stands out, namely that vertices always have the same outvalue as their antipodals. As it turns out, this is not a coincidence, but a property that must be shared among all pseudo-USOs.

Lemma 4.7 Let \( \phi : \{0,1\}^n \mapsto \{0,1\}^n \) be the outmap of a pseudo-USO \( O \). Then \( \forall x, \phi(x) = \phi(x \oplus 1^n) \).

Proof The proof is by induction on \( n \). The base case with \( n = 2 \) is trivial and can be seen in Figure 4.3.

For the induction step, take a pair of antipodals \( a, a' \) with \( \phi(a) \neq \phi(a') \), which is assumed to exist for the sake of contradiction. Let \( i \) be some direction in which \( \phi(a)_i \) and \( \phi(a')_i \) differ (WLOG, \( \phi(a')_i = 1 \)), and let \( b := a' \oplus e_i \). The situation is shown in Figure 4.4.
4.3. Combinatorial Properties

Consider the \((i^{-1} \ast n^{-i})\)-inherited orientation \(O'\) obtained by collapsing the cube along direction \(i\). This is a \((n - 1)\)-pseudo-USO per Observation 4.6. \(a\) and \(b\), being the sinks of their respective 1-cubes along dimension \(i\), determine the outvalue of their logical vertices \(\tilde{a}\) and \(\tilde{b}\) in \(O'\). By the induction hypothesis, those logical vertices have the same outvalue in \(O'\), and therefore \(a\) and \(b\) must have the same outvalue in \(O\) in all directions \(j \neq i\). This implies that the facet spanned by \(a\) and \(b\) in the original cube is not a USO, in contradiction to the definition of \(O\). \(\square\)

More properties of pseudo-USOs can be derived from this one, but this is the only one which we will directly use in the development of algorithms. Readers interested only in the algorithms for this problem might want to skip the rest of this chapter.

Let us look at some of the implications of Lemma 4.7, with an eye towards giving a complete characterization of pseudo-USOs in the spirit of what Corollary 3.7 is for USOs.

The outmaps of pseudo-USOs are not bijective as we have seen above. Interestingly, they turn out to have a similarly strong global constraint on the outvalues of their vertices.

**Lemma 4.8** Let \(O\) be a pseudo-USO of the \(n\)-cube with outmap \(\phi : \{0, 1\}^n \rightarrow \mathcal{P}\). Then the outdegrees of the vertices of \(O\) all have the same parity.

In symbols, \(\mathcal{P} \in \{\mathcal{P}_{\text{even}}, \mathcal{P}_{\text{odd}}\}\) where

\[
\mathcal{P}_{\text{even}} := \{x \in \{0, 1\}^n \mid \sum_{i=1}^{n} x_i \equiv 0 \pmod{2}\}, \text{ and}
\]

\[
\mathcal{P}_{\text{odd}} := \{x \in \{0, 1\}^n \mid \sum_{i=1}^{n} x_i \equiv 1 \pmod{2}\}.
\]

**Proof** We first show that the outvalues of non-antipodal vertices must differ in at least 2 coordinates. Let \(a\) and \(b\) be vertices. Assume that \(\phi(a)\) differs from \(\phi(b)\) in only 1 coordinate \(i\) (the case of 0 coordinates is trivial). \(a\) and \(b\) span a face between them, as do \(a\) and \(b'\), where \(b'\) is the antipodal of \(b\).
These faces span a complementary sets of coordinates, i.e. they split the \( n \) coordinates between them in some fashion (see Figure 4.5). Since coordinate \( i \) can only be spanned by one of these two faces, it follows that either \( a \) and \( b \) or \( a \) and \( b' \) have the same outvalue when restricted to the face spanned between them, which is a contradiction of the USO property.

Next we use this property to prove the claim.

Consider the facet \( f := \ast_{n-1} 0 \) of \( \mathcal{O} \). The orientation on \( f \) is a USO with outmap \( \phi' \). We build a cube graph out of the vertices of \( f \), but instead of connecting two vertices by an edge if \( d_H(a, b) = 1 \) as usual, we connect them if \( d_H(\phi'(a), \phi'(b)) = 1 \). This yields a cube since \( \phi' \) is a bijection.

Color each vertex of this new cube according to the parity of its outvalue in the original orientation \( \mathcal{O} \). We show by contradiction that the cube is monochromatic. Assume that the graph contains 2 vertices of different colors. Trivially (because the graph is connected) there must exist 2 neighboring vertices \( a \) and \( b \) of different colors.

Note that \( \phi(a)_n \neq \phi(b)_n \), because otherwise \( \phi(a) \) and \( \phi(b) \) would only differ in 1 coordinate, which is not allowed as shown in the first part of the proof. But also \( \phi(a)_n = \phi(b)_n \) because otherwise \( a \) and \( b \) would have the same color. \( \square \)

**Corollary 4.9** Let \( \phi \) be the outmap of a pseudo-USO. Then \( \phi \) is a 2-to-1 function, that is

\[
\forall y \in \{0, 1\}^n : |\phi^{-1}(y)| \in \{0, 2\}
\]

where \( \phi^{-1}(y) \) is the preimage of \( y \) under \( \phi \).

**Proof** Each facet of the orientation is a USO and therefore has \( 2^{n-1} \) distinct outvalues. Each of those outvalues occurs twice due to Lemma 4.7. \( \square \)

These properties imply that a Pseudo-USO is fully determined (up to isomorphism) by specifying one facet plus a parity bit. The opposing facet has
the same orientation, except flipped “upside down”, and the edges between
are constrained by the parity chosen. See Figure 4.6 for an illustration. It
seems relevant to study the facets of pseudo-USOs if we want to understand
pseudo-USOs themselves better, leading us to the following definition:

Definition 4.10 A PUSO-facet is a USO that can occur as a facet in a pseudo-
USO.

Each PUSO-facet creates exactly one pseudo-USO with the construction of
Figure 4.6, but a pseudo-USO can have up to \( n \) different PUSO-facets as
predecessors, so this is not a 1-to-1 equivalence between PUSOs and PUSO-
Facets, but it comes close.

What properties does a PUSO-facet have? First of all, we notice that our
Lemmas 3.5 and 3.10, already stated for USOs and pseudo-USOs, also apply
to PUSO-facets.

Since PUSO-facets are in fact USOs, it must be clarified that the two lemmas
not only hold, but that the orientations obtained from them are specifically
PUSO-facets, not just any USOs. The proofs are omitted again.

Observation 4.11 Let \( \phi \) be an outmap. Then for any vector \( z \),

\[ \phi'(x) := \phi(x) \oplus z \]

is the outmap of a PUSO-facet if and only if \( \phi \) is.

Observation 4.12 Let \( O \) be a PUSO-facet with outmap \( \phi \). Then, for all \( p \in \{ -, * \}^n \), the \( p \)-inherited orientation \( O' \) is a PUSO-facet.

Could it be that all USOs are actually PUSO-facets? It turns out this is not
the case, since each vertex in a PUSO-facet, when actually embedded in a
pseudo-USO, is adjacent to a copy of its antipodal through an edge, which
contributes 0 to the parity of one endpoint and 1 to the parity of the other.
Since both these vertices must end up having the same parity due to Lemma
4.8, it follows that the parities must have been different when restricted to
the PUSO-facet.

We will formalize this requirement and then immediately derive a general-
ization of the statement.

Lemma 4.13 Let \( a \) and \( a' \) be antipodals in a PUSO-facet \( F \) on the \( n \)-cube. Then \( \phi(a) \) and \( \phi(a') \) have different parities.

Proof Extend \( F \) to a pseudo-USO \( O \) on the \(( n + 1)\)-cube by taking a copy
of \( F \) and connecting both copies along dimension \( n + 1 \). The copy of \( F \) is
flipped such that each vertex trades places with its antipodal. The edges
connecting both copies are oriented in such a way that all outvalues have the same parity\textsuperscript{4}. This construction is shown in Figure 4.6.

Let $\phi'$ be the outmap of $\mathcal{O}$. Let $b$ be the neighbor of $a$ along coordinate $n + 1$. $\phi'(a)$ and $\phi'(b)$ have the same parity per Lemma 4.8, but since $a$ and $b$ share an edge running along dimension $n + 1$, they have different parities when ignoring the contribution of that edge.

Since $a'$ and $b$ are antipodals, we know from Lemma 4.7 that $\phi'(a') = \phi'(b)$, and in particular they have the same parity when ignoring coordinate $n + 1$. The claim follows.

\textbf{Lemma 4.14} Let $a$ and $b$ be two vertices of a PUSO-facet, such that it is possible to split the coordinates of the cube into two distinct groups, the inner group and the outer group, as follows: For each coordinate $i$, $\phi(a)_i = \phi(b)_i$ if $i$ belongs to the inner group and $a_i \neq b_i$ if $i$ belongs to the outer group.

Then, the parities of the outdegrees of $a$ and $b$ are different if we only consider the coordinates in the outer group.

In symbols, if
\[
\forall i \in [n] \quad (\phi(a)_i = \phi(b)_i) \lor (a_i \neq b_i),
\]
then
\[
\sum_{i=1}^{n} (\phi(a)_i + \phi(b)_i) \cdot [a_i \neq b_i] \equiv 1 \pmod{2}.
\]

\textbf{Proof} The edges belonging to the inner group are pointing towards $a$ and $b$ WLOG\textsuperscript{5}, i.e. $\phi(a)_i = \phi(b)_i = 0$ for all coordinates $i$ in the inner group. We

\textsuperscript{4}The construction is always possible since $\mathcal{F}$ is a PUSO-facet and can per definition be extended to a pseudo-USO, and its complementary facet must be a flipped version of $\mathcal{F}$ by Lemma 4.7.

\textsuperscript{5}We can use Lemma 3.5 to make sure of this.
4.3. Combinatorial Properties

use Lemma 3.10 to construct an inherited orientation with pattern $p$ defined as

$$p_i := \begin{cases} * & \text{if } \phi(a)_i = \phi(b)_i = 0, \\ - & \text{otherwise.} \end{cases}$$

Note that $p$ collapses the faces spanned in the coordinates of the inner group into logical vertices, such that $a$ and $b$ are the sinks of an antipodal pair of such vertices. Lemma 4.13 can now be applied to the facets of the resulting pseudo-USO, and the claim follows. □

See Figure 4.7 for an example of this lemma.

Lemma 4.13 is strictly weaker than Lemma 4.14, as can be seen by the example of the uniform orientation. The implications of the former lemma hold for odd-dimensional uniform orientations, but not even ones. The implications of the latter never hold, because an even-dimensional uniform orientation can always be inherited from an odd-dimensional one. This proves that uniform orientations are not PUSO-facets.

In some sense, Lemma 4.14 is the transitive closure of Lemma 4.13 over the transformations allowed by Observations 4.11 and 4.12.

Lemma 4.14 is important because, as it turns out, satisfying it is not just necessary to be a PUSO-facet, but also sufficient. This means that we have achieved a complete characterization of PUSO-facets.

**Lemma 4.15** Let $F$ be a USO on the $n$-cube satisfying the implications of Lemma 4.14. Then $F$ is a PUSO-facet.

**Proof** Extend $F$ to an orientation $O$ on the $(n+1)$-cube with outmap $\phi$ in the same way that was done in the proof of Lemma 4.13. See Figure 4.6 for an illustration.

We need to show that $O$ is a pseudo-USO. By construction, all antipodal pairs have the same outvalue, violating the USO condition on the whole...
cube, so we must only show that all orientations on smaller faces are USOs. This is clear for faces strictly contained in \( \mathcal{F} \) or its copy, which are USOs by definition.

To establish the USO property for the remaining faces, we show that all pairs of non-antipodal vertices fulfill the equation of Corollary 3.7. Assume for the sake of contradiction that some pair \( u, v \) violates the property, i.e.

\[
(\phi(u) \oplus \phi(v)) \land (u \oplus v) = 0^n.
\]

This means that for each coordinate \( i \), at least one of \( \phi(u)_i = \phi(v)_i \) and \( u_i = v_i \) holds. We split the coordinates into two subsets \( D \subset [n+1] \) and \( D' := [n+1] \setminus D \), such that

\[
i \in D \Rightarrow \phi(u)_i = \phi(v)_i \land u_i \neq v_i,
\]

\[
i \in D' \Rightarrow u_i = v_i.
\]

(4.1)

Note that \( |D'| > 0 \) because otherwise \( u \) and \( v \) would be antipodals, which by definition they are not.

\( u \) and \( v \) have the same outvalues along coordinates in \( D \), so their outvalues must have the same parity when restricted to coordinates in \( D' \), because the overall outvalue parity is the same for all vertices of \( \mathcal{O} \) by construction.

Let \( v' \) be the antipodal of \( v \). Stated in terms of \( v' \), (4.1) implies

\[
i \in D \Rightarrow \phi(u)_i = \phi(v')_i
\]

\[
i \in D' \Rightarrow u_i \neq v'_{i}.
\]

so \( u \) and \( v' \) fulfill the conditions of Lemma 4.14 with inner group \( D \) and outer group \( D' \). Therefore, their outvalues must have different parities when restricted to coordinates in \( D' \).

The parity requirements imposed on \( v \) and \( v' \) are contradictory, because \( \phi(v) = \phi(v') \) by construction.

\( \square \)

We summarize the last 3 lemmas as a theorem:
Theorem 4.16 Let $O$ be a USO of the $n$-cube with outmap $\phi$. Then $O$ is a PUSO-facet if and only if the following holds:

$$\forall a, b \in \{0, 1\}^n$$

if

$$\forall i \in [n] \quad (\phi(a)_i = \phi(b)_i) \lor (a_i \neq b_i),$$

then

$$\sum_{i=1}^{n} (\phi(a)_i + \phi(b)_i) \cdot [a_i \neq b_i] \equiv 1 \pmod{2}.$$
Chapter 5

The USO Recognition Problem

5.1 Overview

In this chapter, we will take the theoretical results from the last chapter and develop several algorithms that solve the USO recognition problem, both in the classical and the quantum setting.

As mentioned in Section 4.1 of the previous chapter, when given a non-USO orientation, we can classify it according to the minimum distance $k = c \cdot n$ separating any pair of vertices contained in a certificate proving its non-USO status\footnote{1}. As $n \to \infty$, we will look at the runtime of our algorithms on instances with fixed parameter $c$. This decision is justified, since for any given $n$ there are only polynomially many such $c \in (0, 1]$ and all algorithms whose runtime depends on this $c$ can easily be adjusted such that their runtime is input sensitive.

What this means is that when we have an algorithm $A$ with runtime $f(n, c)$ that depends on knowing $c$ beforehand, we can assume that this runtime is achieved even if we do not know $c$. If $f$ is a known, computable function, given an instance of dimension $n$, we sort the possible values of $c$ according to their expected runtime $f(n, c)$, and then run the algorithm once for each $c$, in ascending order. This ensures that when we arrive at the true value of $c$, we will have done at most $n$ extra calls to $A$, all of which ran for less time than the final call to $A$, for a total overhead that is at most a factor of $n$ on top of $f(n, c)$. In this setting our algorithms have exponential runtimes, so this is a very small overhead. If for some reason $f$ was unknown or uncomputable, we could still achieve input sensitivity by running $A$ in paralell for all values of $c$, though this is a much less elegant solution.

\footnote{1}{Recall that a certificate is a pair of vertices that violates Lemma 3.7.}
5. The USO Recognition Problem

The runtime plots of different algorithms found in this chapter always show the input dependent parameter $c$ plotted against the runtime $f(c)$, where $f(c) := f(n, c)$ is the expected runtime of the algorithm as $n \to \infty$.

It should also be noted that we will have several algorithms at our disposal, and we may freely mix and match them, so the true algorithm we are building towards in this chapter is a meta-algorithm choosing the best algorithm in its arsenal for each specific value of $c$. This can again be done without incurring meaningful overhead.

5.2 Classical Algorithm

Lemma 4.1 from the previous chapter gives us a strict lower bound on the USO recognition problem matching the upper bound given by the trivial algorithm. We might think that no further analysis is needed or even possible in the classical case. However, while an algorithm better than the trivial algorithm can’t exist in general, if we restrict ourselves to the case where there is a constant $c \in (0, 1]$ such that no pair of vertices closer than $cn$ violates Corollary 3.7, we can do better.

**Theorem 5.1** Let $c \in (0, 1]$ be a constant, and let $\phi : \{0, 1\}^n \mapsto \{0, 1\}^n$ be the outmap of an orientation of the $n$-cube not containing a pseudo-USO of dimension less than $cn$ embedded in one of its faces. As $n \to \infty$, it takes at most

$$\min_{0 \leq d < \frac{c}{2}} 2^{(1-cH(\frac{d}{c})+H(\frac{d}{c}))n}. \text{poly}(n)$$

queries to $\phi$ to determine whether or not $\phi$ is the outmap of a USO, where $H$ is the binary entropy function.

See Figure 5.1 for a plot of the runtime, which is is exponentially better than $2^n$ for all $c > 0$, and decreases monotonically as $c$ increases.

We will prove this theorem by showing that there is an algorithm for USO recognition with the claimed runtime. In order to do this, we must first introduce a couple of concepts that will be important for the analysis.

5.2.1 Hamming Balls and Coverings

If we take a graph and consider the shortest path connecting any pair of vertices, this forms a metric over the set of vertices of that graph\(^2\). In case of the $n$-cube, this metric coincides with the Hamming distance. Whenever we have a metric, we can define balls in a natural way.

---

\(^2\)This metric is known as the geodesic graph distance.
5.2. Classical Algorithm

Figure 5.1: Runtime of classical algorithm for USO recognition, given as a blue solid line. The trivial algorithm is shown as a dotted line. As explained in the introduction of this chapter, we plot $f(c)$ which implies a runtime of $2^{f(c)n}$ as $n \to \infty$.

**Definition 5.2** Let $Q_n = (V, E)$ be the $n$-cube. The Hamming ball $B(v, r)$ of radius $r \in \mathbb{R}^+$ centered around $v \in V$ is the set

$$B(v, r) := \{ v' \in V \mid d_H(v, v') \leq r \}$$

of vertices at a Hamming distance at most $r$ from $v$.

$\text{vol}(n, r)$ is the volume of a Hamming ball on $Q_n$ with radius $r$.

Note that the volume is defined independent of the balls’ center. This is not an oversight, since all balls of the same radius have the same size, due to the symmetry of the cube. The volume can easily be seen to be

$$\text{vol}(n, r) := \sum_{i=0}^{r} \binom{n}{i}.$$

Let us state some properties of Hamming balls paraphrased from [26, Chapter 5].

**Lemma 5.3 ([26], Lemma 5.6)** Let $r \in [0, \frac{1}{2}]$. Then

$$\text{vol}(n, rn) = 2^{nH(r)} \text{poly}(n).$$

**Lemma 5.4 ([26], Lemma 5.9)** Let $Q_n = (V, E)$ be the $n$-cube and $r \in [0, 1]$.
Then, for a set \( V' \subseteq V \) of size \(|V'| \geq \frac{2^n \text{poly}(n)}{\text{vol}(n, rn)}\)
of vertices chosen uniformly at random,
\[
\Pr \left[ \bigcup_{v \in V'} B(v, rn) = V \right] > 1 - \left( \frac{2}{e} \right)^n.
\]
The second Lemma states that Hamming balls can be used to cover the cube efficiently. By choosing only polynomially more vertices than strictly required by a volume argument, we can cover the cube, i.e. with high probability every vertex is at most at distance \( rn \) from one of the chosen vertices.

### 5.2.2 The Algorithm

The idea behind our classical USO recognition algorithm is simple: If an orientation contains a large pseudo-USO, the strong structure imposed by Lemma 4.7 will be difficult to hide. If we take a cleverly chosen set of Hamming balls and evaluate all vertices within those balls, we will hit a pair of antipodals of the pseudo-USO with high probability, even though the number of balls is insufficient to cover the cube.

The algorithm takes a parameter \( c \in [0, 1] \), and finds a pseudo-USO of dimension \( cn \) in an orientation, if it exists. It works as follows:

1. Compute \( d \in [0, \frac{c}{2}) \) that will yield the optimal runtime.
2. Choose \( 2^{(1-cH(\frac{1}{2}))n} \cdot \text{poly}(n) \) vertices u.a.r.
3. Evaluate all vertices contained in a ball of radius \( \frac{dn}{2} \) from one of the vertices chosen in step 1.
4. For each pair of evaluated vertices, test if they violate Corollary 3.7.

Analysing this algorithm is enough to prove the theorem.

**Proof (of Theorem 5.1)** We first show the correctness of the algorithm for an arbitrary choice of \( d \) in step 1. We claim that the number of vertices is chosen such that every face of dimension \( cn \) has a covering of radius \( dn \) with high probability. If we consider a specific face of the cube, the number of vertices belonging to that that face chosen by step 2 is, on expectation,
\[ \frac{2^cn}{2^n} \cdot 2^{(1-cH(\frac{c}{2}))n} \cdot \text{poly}(n) \]
\[ = 2^{c-cH(\frac{c}{2})n} \cdot \text{poly}(n) \]
\[ = 2^{(1-H(d'))n'} \cdot \text{poly}(n') \]
\[ = 2^{n'/\text{vol}(n',d'n')} \cdot \text{poly}(n'). \]

where \( n' := cn \) and \( d' := \frac{d}{c} \).

The last equality uses Lemma 5.3 and only holds when \( d' \leq \frac{1}{2} \), i.e. \( d \leq \frac{c}{2} \).

Using a Chernoff bound, it turns out that sampling at least this many vertices happens “with high probability”, not just “on expectation”, without incurring additional overhead.\(^3\)

It follows from Lemma 5.4 that with high probability, the algorithm will construct a covering of radius \( d'n' = dn \) on any face of dimension \( cn \), no matter where it is embedded in the cube.\(^4\)

Given such a covering on a pseudo-USO of dimension \( cn \), consider a vertex \( v \) that is the center of one of the balls, and its antipodal \( v' \). \( v' \) has at most distance \( dn \) to the center \( u \) of a ball in the covering. If we take any vertex \( w \) halfway between \( u \) and \( v' \), it has distance at most \( \frac{dn}{2} \) from the ball center \( u \), and its antipodal \( w' \) has distance at most \( \frac{dn}{2} \) from the ball center \( v \). See Figure 5.2 for an illustration of the idea.

It follows from this analysis that if a pseudo-USO of dimension \( cn \) exists in the cube, with high probability the algorithm will evaluate two of its antipodals, which by Lemma 4.7 violate Corollary 3.7. This establishes the correctness of the algorithm.

The runtime of the algorithm claimed in Theorem 5.1 is easily shown. We can use Lemma 5.3 to estimate the volume of the balls generated in step 3, since \( d \leq \frac{c}{2} \leq \frac{1}{2} \). By multiplying the number of vertices from step 2 that are the centers of balls with the size of each ball we get an upper bound on the number of vertices evaluated in total, namely

\[ 2^{(1-cH(\frac{c}{2}))n} \cdot \text{vol}(n, \frac{dn}{2}) \cdot \text{poly}(n) = 2^{(1-cH(\frac{c}{2})+H(\frac{c}{2}))} \cdot \text{poly}(n) \]  \hspace{1cm} (5.1)

Steps 1 and 4 perform no oracle evaluations at all, so their runtime is 0. Step 1 consists of analytically computing the expected runtime according to (5.1).\(^3\)

\(^3\)We will make this notion formal after the proof.

\(^4\)Note that we are not claiming to construct such a covering on every face at the same time w.h.p., rather on any fixed face considered on its own.
for every possible choice of \( d \) (of which there are only \( O(n) \) many), and setting \( d \) accordingly. \( \square \)

It remains to explain why and how we use the Chernoff bound in the proof. Lemma 5.4 holds when a certain number of vertices are sampled, but it doesn’t guarantee anything when we sample less than that number. Therefore, it’s not enough to hit enough vertices of the \((cn)\)-cube on average, we actually need to estimate the probability that we hit the average number of vertices or more.

The Chernoff bound is one way to establish such a tail estimate. We will use a sampling version of the bound, found in [13] and stated as follows:

**Lemma 5.5** ([13], Lemma 2.1) Consider a set \( A \) and its subset \( B \subset A \). Suppose we pick a subset \( S \subset A \) u.a.r. from all subsets of \( A \) of size \( k \). Then for \( \delta \in (0, 1) \) we have

\[
\Pr \left[ |S \cap B| < (1 - \delta) \cdot \frac{|B| \cdot |S|}{|A|} \right] < \exp \left( - \frac{|B| \cdot |S| \cdot \delta^2}{2 |A|} \right)
\]

Let us formalize this for our algorithm, where \( A \) is the entire cube, \( B \) is a face of dimension \( cn \) and \( S \) is the set of vertices sampled in step 2 of the algorithm. Therefore, \(|A| = 2^n\), \(|B| = 2^{cn}\) and \(|S| = 2^{n-cH(\frac{d}{c})n} poly(n)\). The expression inside \( \exp(.) \) becomes

\[
- \frac{|B| \cdot |S| \cdot \delta^2}{2 |A|} = - \frac{2^{cn} \cdot 2^{n-cH(\frac{d}{c})n} \cdot \delta^2}{2} = - \frac{2^{(1-H(\frac{d}{c}))cn} \cdot \delta^2}{2} = - \frac{2^{(1-H(\frac{d}{c}))cn} \cdot poly(n)}{2}
\]

where \( \varepsilon = \varepsilon(c,d) > 0 \) is a constant because \( H(\frac{d}{c}) < 1 \). Therefore, by oversampling only by a constant factor (which is folded into the \( poly(n) \) factor in the algorithm analysis), the probability of hitting too few vertices of \( B \) to form a covering goes to 0 double exponentially fast as \( n \) goes to infinity.

5.2.3 Improving the Algorithm

Looking at Figure 5.2, it is clear that a slightly better algorithm exists. \( u \) doesn’t need to be inside the pseudo-USO itself; its center could lie up to \( \frac{dn}{2} \)
5.2. Classical Algorithm

Figure 5.2: Given a pseudo-USO with a covering of radius $dn$, we can always find a pair of antipodals both at maximum distance $\frac{dn}{2}$ of the centers of balls belonging to the covering.

Figure 5.3: Slight improvement of the idea shown in Figure 5.2. $u$ is allowed to lie outside the pseudo-USO by a distance of up to $\frac{dn}{2}$, reducing the number of vertices needed to achieve the required covering.
distance outside the pseudo-USO and both \( w \) and \( w' \) would still lie inside the correct face. This allows to make the covering slightly sparser; it is only required that a face of size \( (c + \frac{d}{2})n \) has a covering of radius \( cn \). The idea is shown in Figure 5.3.

The runtime does become better when using this trick, but only marginally so, at the expense of a considerably more involved analysis, which we omit here. Keep in mind that the main point of this classical algorithm is not its exact performance, but rather the fact that it shows the lower bound from Lemma 4.1 to be tight only when the violation of the USO condition is very localized. We believe this algorithm is very far from the best that can be done classically with regards to this problem.

5.3 Quantum Algorithms

Let us step out of the world of classical algorithms and analyze the quantum case. First of all, we notice that the pair finding algorithm mentioned in Section 2.2.5 can be directly used to solve the USO recognition problem, by searching for a certificate, i.e. a pair of vertices with the property of contradicting Corollary 3.7.

**Observation 5.6** There exists a \( O(2^{\frac{2}{5}n}) \) quantum query algorithm deciding the USO recognition problem.

This is not the end of the story, however. If \( c \) is very small, i.e. \( c \to 0 \), the runtime of a sensible algorithm should converge to \( 2^{0.5n} \), since finding a marked pair essentially boils down to finding one vertex of the pair, then searching a polynomially-sized neighborhood for the other vertex completing the pair.

Similarly, when \( c \) is close to 1 the runtime should also become fast: As shown in Section 3.1.1, the number of faces of dimension \( cn \) in the \( n \)-cube is

\[
\binom{n}{c \cdot n} \cdot 2^{(1-c)n},
\]

which using Lemma 5.3 to approximate the binomial coefficient yields

\[
\binom{n}{c \cdot n} \cdot 2^{(1-c)n} \approx 2^{(1-c+H(c))n}, \quad (5.2)
\]

From this it follows that \( f(c) \to 0 \) as \( c \to 1 \). In fact, as soon as \( c \geq 1 - \frac{\text{poly}(\log(n))}{n} \) there are only polynomially many faces of dimension \( cn \), so we will indeed be able to search through all of them in only polynomial time.
5.3. Quantum Algorithms

Figure 5.4: Runtime of two quantum algorithms for USO recognition. Red dashed line: Generic quantum pair finding algorithm, not tailored specifically to this problem. Blue solid line: Grover search over the space of cube faces. As explained in the introduction of this chapter, we plot \( f(c) \) which implies a runtime of \( 2^{f(c) n} \) as \( n \to \infty \).

Is there an algorithm that properly captures these two extremal runtimes, while behaving reasonably in the range between? In fact there is, namely a simple Grover search over all faces of dimension \( cn \), where a face is marked if it is a pseudo-USO. This can be determined in constant time by evaluating an arbitrary pair of antipodals of the face, as per Lemma 4.7. The size of the search space is determined by (5.2), and the runtime is the square root of that expression.

Observation 5.7 Let \( c \in (0, 1] \) be a constant, and let \( \phi : \{0, 1\}^n \to \{0, 1\}^n \) be the outmap of an orientation of the n-cube not containing a pseudo-USO of dimension less than \( cn \) embedded in one of its faces. It takes at most

\[
O(2^{\frac{1-cn H(c)}{2^{\frac{1}{2}n}}})
\]

queries to \( \phi \) to determine whether or not \( \phi \) is the outmap of a USO.

These first two algorithms have been plotted against each other in Figure 5.4. This is already an improvement, but we have not yet used the combinatorial properties of pseudo-USOs in a meaningful way. In order to further improve the runtime, we must look into the details of the pair finding algorithm instead of treating it as a black box.

Recall that the general case analysis only assumes the existence of a single pair with the property we are searching for. This is too conservative,
5. The USO Recognition Problem

since a pseudo-USO of dimension $c \cdot n$ has $2^{c \cdot n}/2$ antipodal pairs, any of which certify that it is not a USO. Using this increased solution density we can achieve a better runtime analysis, still neglecting the structured way in which the pairs are found in the cube.

Let $J$ be the Johnson graph $J(2^n, 2^{r-n}, 1) = (V, E)$, and let $A_i$ be the subset of vertices containing the $i$-th solution pair. Note that all $A_i$ have the same size, since the Johnson Graph is completely symmetric. If we translate our previous analysis of the general case in Section 2.2.5 to the size of this Johnson graph and assume that there is only one solution pair, the solution density would be

$$
\varepsilon := \frac{|A_1|}{|V|} = \theta \left( \frac{2^{r-n}}{2^n} \right)^2.
$$

In our scenario of $\theta(2^{c \cdot n})$ certificates, the density is higher. Per definition, all vertices that contain at least one solution pair are marked, so the solution density $\varepsilon'$ is

$$
\varepsilon' := \frac{|\bigcup_{i=1}^{2^{c \cdot n} - 1} A_i|}{|V|}.
$$

It is difficult to compute the exact size of the set $|\bigcup_{i=1}^{2^{c \cdot n} - 1} A_i|$, but we can bound the size from below as follows:

$$
\varepsilon' = \frac{|\bigcup_{i=1}^{2^{c \cdot n} - 1} A_i|}{|V|} \\
\geq \sum_i \frac{|A_i|}{|V|} - \sum_{i \neq j} \frac{|A_i \cap A_j|}{|V|} \\
= \sum_i \theta \left( \frac{2^{r-n}}{2^n} \right)^2 - \sum_{i \neq j} \theta \left( \frac{2^{r-n}}{2^n} \right)^4 \\
= \theta \left( 2^{c \cdot n} \cdot \frac{2^{2r \cdot n}}{2^{2n}} - 2^{2c \cdot n} \cdot \frac{2^{4r \cdot n}}{2^{4n}} \right) \\
= \theta \left( 2^{(2r-2+c)n} - 2^{(4r-4+2c)n} \right).
$$

The inequality of the second line uses the inclusion-exclusion principle, and the equality on the third line uses the fact that $A_i \cap A_j$ is the set of all vertices containing 4 specific cube vertices (i.e. the vertices forming two certificates).

By incorporating this bound for $\varepsilon'$ into the analysis of the pair finding algorithm, we can show an improved runtime:
Lemma 5.8 Let \( c \in (0, 1] \) be a constant, and let \( \phi : \{0, 1\}^n \mapsto \{0, 1\}^n \) be the outmap of an orientation of the \( n \)-cube not containing a pseudo-USO of dimension less than \( cn \) embedded in one of its faces. It takes at most

\[
O(2^{\frac{2-c}{3}n})
\]

queries to \( \phi \) to determine whether or not \( \phi \) is the outmap of a USO.

Proof The algorithm achieving the claimed bound is quantum pair finding. As already shown in Section 2.2.5, the runtime of the algorithm is

\[
S + \frac{1}{\sqrt{\delta \epsilon'}} (U + C)
\]

where \( S = 2^n \) is the startup cost, \( U = 1 \) is the update cost and \( C = 0 \) is the checking cost. \( \delta \) and \( \epsilon' \) take the values of \( \delta = \theta(2^{-m}) \) and \( \epsilon' \geq \theta \left(2^{(2r-2+c)n} - 2^{(4r-4+2c)n}\right) \).

Note that the expression for \( \epsilon' \) is dominated by the first term as long as \( r < 1 - \frac{c}{2} \), so the runtime is

\[
S + \frac{1}{\sqrt{\delta \epsilon'}} (U + C)
\]

\[
= 2^n + \frac{1}{\sqrt{\delta \epsilon'}}
\]

\[
\leq 2^n + 2^{\frac{n}{2}} 2^{-(1-\frac{c}{2})n}
\]

The value of \( r \) that minimizes the runtime is \( r = \frac{2-c}{3} \), yielding the claim. □

This is a significant improvement over the naive analysis of the pair finding algorithm when \( c \) is large, but it is still noticeably suboptimal when \( c \) is near 0. If we think about the structure of the Johnson graph, it becomes clear that this is not a problem with the analysis anymore, but a suboptimal behaviour of the algorithm itself.

If we consider a random vertex in the Johnson graph, each cube vertex has a probability to be included or excluded independently of all others. This is not optimal, since the combinatorics of the cube are such that pairs of vertices far away from each other vastly outnumber pairs of vertices close together\(^5\). If we looked at a randomly chosen Johnson vertex, we would find that the vertex pairs we are actually interested in seeing, namely certificates, are severely underrepresented. If the Johnson graph was somehow

\(^5\)The vast majority of vertex pairs are at distance \( \frac{n}{2} \).
5. The USO Recognition Problem

Figure 5.5: Runtime of two more algorithms. Red dashed line: Generic quantum pair finding algorithm, with analysis adjusted to consider the multiplicity of solution pairs in this problem. Blue solid line: The algorithm has been further augmented to constrain the search in small regions when \( c \) is small. The previous algorithms are shown as dotted lines.

constrained to only consider pairs of vertices closer together, even at the expense of only covering a limited neighborhood, we might get a better solution density and hence a better runtime.

We have developed an algorithm that manages to overcome this combinatorial obstacle, which works as follows:

1. Choose a face of dimension \( d \cdot n \) u.a.r., where \( d = d(c) \) is a parameter to be determined in the analysis.

2. Use the pair finding algorithm on the selected face.

3. Perform amplitude amplification on steps 1-2.

We are quite happy with our previous algorithm when \( c \) is large, so we artificially create this condition by restricting ourselves to a small region of the cube, namely some face of dimension \( dn \). We can then use the generalization of Grover’s search algorithm to find the correct region to look at more efficiently than just going through them classically, using the technique of amplitude amplification already discussed in Section 2.2.3.

Let us analyse the algorithm in detail.

For step 2, we can use the tighter analysis of the pair finding algorithm provided by Lemma 5.8 to get a runtime of \( 2^{\frac{2}{3} c \cdot dn} \).
5.3. Quantum Algorithms

For step 1, the query complexity is obviously 0, but we need to determine the probability $p$ that the selected face contains the pseudo-USO of dimension $cn$ we are searching for. Since we are choosing the face u.a.r., this probability is exactly the number of $(cn)$-cubes contained in each $(dn)$-cube divided by the total number of $(cn)$-cubes contained in the $n$-cube.

We will apply (5.2) to estimate both of these quantities. Recalling that there are $\binom{n}{k}2^{n-k}$ faces of dimension $k$ in the $n$-cube, we have that

$$p := \frac{2(d-c)n \binom{dn}{cn}}{2(1-c)n \binom{n}{cn}}$$

$$= 2(d-1)n \cdot \frac{dn}{cn} \cdot \frac{n}{cn}$$

$$\approx 2(d-1)n \cdot 2^{dnH(\frac{d}{2})} \cdot 2^{-nH(c)}$$

$$= 2^{(d-1) + dH(\frac{c}{d}) - H(c)}n$$

Steps 1 and 2 combined are a quantum algorithm with probability $p$ of finding the pseudo-USO. In step 3 we need to repeat it $\frac{1}{\sqrt{p}}$ times to boost the probability of finding the solution to a constant, so the total runtime of the algorithm is

$$p^{-\frac{1}{2}} \cdot 2^{\frac{2-c}{2}dn}$$

$$= 2^{-\frac{1}{2}(d-1 + dH(\frac{c}{d}) - H(c))n} \cdot 2^{\frac{2-c}{2}dn}$$

$$= 2^{\frac{1}{2}(1 - d - dH(\frac{c}{d}) + H(c) + \frac{4c - 2d}{d})n}$$

$$= 2^{\frac{1}{2}(1 - dH(\frac{c}{d}) + H(c) + \frac{4c - 2d}{d})n}$$

To minimize this expression, for each $c$ we attempt to find the best $d$, under the constraint that $c \leq d \leq 1$. This constraint ensures that the approximation of the binomial coefficient we used is valid.

In Figure 5.5, the final runtime of the algorithm is plotted, considering the best $d$ for each $c$. The plot has been computed numerically, and leads us to the following statement based on the worst runtime obtained at $c \approx 0.12$:

**Theorem 5.9** There exists a $O(2^{0.6218n})$ quantum query algorithm deciding the USO recognition problem.

In the end we were only able to compute an approximate runtime of the algorithm, which is somewhat unsatisfying. The final runtime as per (5.6) is
difficult to compute analytically. We can at least give some intuition about how the parameters $c$ and $d$ behave.

Figure 5.6 shows the optimal $d$ chosen for each $c$. $d$ increases exponentially with $c$, until it is capped at 1 upon reaching the threshold $c \approx 0.14948$.

Figure 5.7 shows how the runtime changes as we vary $d$, for some example values of $c$ both above and below the threshold. For a fixed $c$, the runtime curve is convex, with a local minimum in the unit interval when $c$ is below the threshold value. Above the threshold, the local minimum moves beyond 1, and hence we end up with $d = 1$. This explains why in Figure 5.5, the runtime converges with the previous algorithm at the threshold: It is exactly what we expect when $d = 1$, as we only consider a single face (the cube itself) in step (1) of the algorithm, which must contain the pseudo-USO we are searching for if it exists at all.

### 5.4 Lower Bounds Revisited

The formulation of classical and quantum lower bounds in the previous chapter (Lemma 4.1) was a good starting point for analyzing the USO recognition problem, since it allowed us to notice the fact that violations of the USO property could be highly localized in the cube, but maybe didn’t need to be. This led us to developing algorithms taking into account the distance
5.4. Lower Bounds Revisited

Figure 5.7: How $d$ affects the runtime for fixed $c$. Given is the example of $c \approx 0.14948$, where the optimal $d$ is exactly 1. For smaller $c$, there is a local minimum to be found in the unit interval, while for larger $c$, the curve decreases monotonically.

$cn$ between the vertices of a certificate and how this could affect their runtime.

With the insights gained over the last two chapters, we now come full circle and provide a generalized lower bound that takes into account this distance, as embodied by the parameter $c$.

**Lemma 5.10** Let $c \in (0, 1]$ be a constant, and let $\phi : \{0, 1\}^n \mapsto \{0, 1\}^n$ be the outmap of an orientation not containing a pseudo-USO of dimension less than $cn$ embedded in one of its faces. It takes at least $2^{(1-c)n}$ queries to $\phi$ in the worst case to determine whether or not $\phi$ is the outmap of a USO. For a quantum computer, $\Omega(2^{\frac{1}{2}(1-c)n})$ queries are required.

**Proof** We reduce the unstructured search problem on a function $g : \{0, 1\}^{n-cn} \mapsto \{0, 1\}$ to the problem of finding a pseudo-USO of dimension $cn$ in a $n$-cube, from which the claim follows.

Let $\phi' : \{0, 1\}^{cn} \mapsto \{0, 1\}^{cn}$ be the outmap of an arbitrary pseudo-USO, e.g. the one guaranteed to exist by Observation 4.3. Using $\phi'$, we construct an outmap $\phi$ as follows:
5. The USO Recognition Problem

\[ \phi : \{0,1\}^n \rightarrow \{0,1\}^n \]
\[ \phi(y \cdot x) := \begin{cases} 
\phi'(y) \cdot x & \text{if } g(x) = 1, \\
 y \cdot x & \text{otherwise,} 
\end{cases} \]

where \( y \) is a \( cn \)-bitstring and \( x \) is a \((n - cn)\)-bitstring. If \( g \equiv 0 \), \( \phi \) is the outmap of the uniform USO. If \( g \not\equiv 0 \), there exists a \( y \) such that \( g(y) = 1 \) and \( \phi \) is not the outmap of a USO, as certified by the vertices \( 0^{cn} \cdot y \) and \( 1^{cn} \cdot y \).

The pseudo-USOs embedded in the cube in this manner are disjoint from each other, so by the same argument used to prove Observation 4.4, no pseudo-USOs of dimension not exactly \( cn \) can exist.

It can be easily seen that this proof is just a generalization of the proof of Lemma 4.1, but creating a contradiction in a face larger than a single edge to mark the spot where \( g \) has a solution.

The resulting lower bound for the quantum case is compared with existing algorithms in Figure 5.8. As discussed at the start of this chapter, we are free to mix and match all the quantum algorithms, so the upper bound is achieved by taking the best algorithm for each specific value of \( c \), which in the figure translates to taking the lower envelope of all the plotted runtime curves.
Bibliography


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

CLASSICAL AND QUANTUM ALGORITHMS
FOR USO RECOGNITION

Author(s): (in block letters):

Name(s):

BOSSHARD

First name(s):

VITOR

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Zürich, 13.10.2015

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.