Memory Compression with Quantum Random-Access Gates

Harry Buhrman ✉
QuSoft, CWI Amsterdam, The Netherlands
University of Amsterdam, The Netherlands

Bruno Loff ✉
University of Porto, Portugal
INESC-Tec, Porto, Portugal

Subhasree Patro ✉
QuSoft, CWI Amsterdam, The Netherlands
University of Amsterdam, The Netherlands

Florian Speelman ✉
QuSoft, CWI Amsterdam, The Netherlands
University of Amsterdam, The Netherlands

Abstract

In the classical RAM, we have the following useful property. If we have an algorithm that uses M memory cells throughout its execution, and in addition is sparse, in the sense that, at any point in time, only m out of M cells will be non-zero, then we may "compress" it into another algorithm which uses only $m \log M$ memory and runs in almost the same time. We may do so by simulating the memory using either a hash table, or a self-balancing tree.

We show an analogous result for quantum algorithms equipped with quantum random-access gates. If we have a quantum algorithm that runs in time T and uses M qubits, such that the state of the memory, at any time step, is supported on computational-basis vectors of Hamming weight at most m, then it can be simulated by another algorithm which uses only $O(m \log M)$ memory, and runs in time $\tilde{O}(T)$.

We show how this theorem can be used, in a black-box way, to simplify the presentation in several papers. Broadly speaking, when there exists a need for a space-efficient history-independent quantum data-structure, it is often possible to construct a space-inefficient, yet sparse, quantum data structure, and then appeal to our main theorem. This results in simpler and shorter arguments.

2012 ACM Subject Classification Theory of computation → Quantum complexity theory; Theory of computation → Quantum complexity theory

Keywords and phrases complexity theory, data structures, algorithms, quantum walk

Digital Object Identifier 10.4230/LIPIcs.TQC.2022.10

Related Version *ArXiv Version*: `https://arxiv.org/abs/2203.05599`

Funding This work was supported by the Dutch Ministry of Economic Affairs and Climate Policy (EZK), as part of the Quantum Delta NL programme.
Harry Buhrman: Harry Buhrman is supported by NWO Gravitation grants NETWORKS and QSC, and EU grant QuantAlgo.
Bruno Loff: Bruno Loff's research is supported by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.
Subhasree Patro: Subhasree Patro is supported by the Robert Bosch Stiftung and additionally supported by NWO Gravitation grants NETWORKS and QSC, and EU grant QuantAlgo.
Florian Speelman: Florian Speelman is supported by NWO Gravitation grants NETWORKS and QSC, and EU grant QuantAlgo.

Acknowledgements We would like to thank Robin Kothari and Ryan O'Donnell for helpful discussions.

17th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2022).
Editors: François Le Gall and Tomoyuki Morimae; Article No. 10; pp. 10:1–10:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

This paper arose out of the authors' recent work on quantum fine-grained complexity [4], where we had to make use of a quantum walk, similar to how Ambainis uses a quantum walk in his algorithm for element distinctness [2]. An essential aspect of these algorithms is the use of a history-independent data-structure. In the context of our paper, we needed three slightly different data structures of this type, and on each of these occasions we saw a similar scenario. If we were only concerned with the time complexity of our algorithm, and were OK with a polynomial increase in the space complexity (the number of qubits used by the algorithm), then there was a very simple data structure that would serve our purpose. If, however, we wanted the algorithm to be space-efficient, as well, then we needed to resort to more complicated data structures.

And we made the following further observation: the simple, yet space-inefficient, data structures were actually *sparse*, in the sense that although M qubits were being used, all the amplitude was always concentrated on computational-basis vectors of Hamming weight $\leq m \ll M$. The analogous classical scenario is an algorithm that uses M memory registers, but at any time step all but m of these registers are set to 0. In the classical case, we know how to convert any such an m*-sparse* algorithm into an algorithm that uses $O(m \log M)$ memory, by using, e.g., a hash table. We wondered whether the same thing could be said of quantum algorithms. This turned out to be possible, and the main purpose of this paper is to explain how it can be done. We will take an arbitrary *sparse* quantum algorithm, and *compress* it into a quantum algorithm that uses little space.

Our main theorem is as follows (informally stated):

▶ **Theorem 1.** *Any m-sparse quantum algorithm using time T and M qubits can be simulated with ε additional error by a quantum algorithm running in time $O(T \cdot \log(\frac{T}{\varepsilon}) \cdot \log(M))$, using $O(m \log M)$ qubits.*

We will prove this result using quantum radix trees in Section 3. The result can also be proven, with slightly worse parameters, using hash tables, but we will not do so here. The sparse algorithm is allowed to use quantum random-access gates (described in the preliminaries Section 2), and the *compressed* simulation requires such gates, even if the original algorithm does not.

The $\log M$ factor in the time bound can be removed if we assume that certain operations on $O(\log M)$ bits can be done at $O(1)$ cost. This includes only simple operations such as comparison, addition, bitwise XOR, or swapping of two blocks of $O(\log M)$ adjacent qubits.[1] All these operations can be done at $O(1)$ cost in the usual classical Random-Access Machines.

The techniques used to prove our main theorem are not new: quantum radix-trees first appeared in a paper by Bernstein, Jeffery, Lange and Meurer [3] (see also Jeffery's PhD thesis [5]). One contribution of our paper is to present BJLM technique in full, as in currently available presentations of the technique, several crucial aspects of the implementation are missing or buggy[2].

But our main contribution is to use these techniques at the right level of abstraction. Theorem 1 is very general, and can effectively be used as a black box. One would think that Theorem 1, being such a basic and fundamental statement about quantum computers, and being provable essentially by known techniques, would already be widely known. But this

[1] The qubits in each block are adjacent, but the two swapped blocks can be far apart from each other.

[2] For example, some operations are defined which are not unitary. Or, there is no mention of error in the algorithms, but they actually cannot be implemented in an error-free way using a reasonable number of gates from any standard gate set.

appears not to be the case, as papers written as recently as a year ago could be significantly simplified by appealing to such a theorem. Indeed, we believe that the use of Theorem 1 will save researchers a lot of work in the future, and this is our main motivation for writing this paper.

To illustrate this point, in Section 4 we will overview three papers [2, 1, 4] that make use of a quantum walk together with a history-independent data structure. These papers all use complicated but space-efficient data structures. As it turns out, we can replace these complicated data structures with very simple tree-like data structures. These new, simple data structures are memory inefficient but sparse, so we may then appeal to Theorem 1 to get similar upper bounds. The proofs become shorter: we estimate each of these papers could be cut in size by 4 to 12 pages. And furthermore, using simpler (memory inefficient but sparse) data-structures allows for a certain *separation of concerns*: when one tries to describe a space-efficient algorithm, there are several bothersome details that one needs to keep track of, and they obscure the presentation of the algorithm. By using simpler data structures, these bothersome details are disappear from the proofs, and are entrusted to Theorem 1.

2 Definitions

We let $[n] = \{1, \ldots, n\}$, and let $\binom{[n]}{\le m}$ be the set of subsets of $[n]$ of size at most m.

We let $\mathcal{H}(N)$ denote the complex Hilbert space of dimension N, and we let $\mathcal{U}(N)$ denote the space of unitary linear operators from $\mathcal{H}(N)$ to itself (i.e. the unitary group). We let $\mathcal{B}$ denote a set of universal quantum gates, which we will fix to containing the $\mathsf{CNOT} \in \mathcal{U}(4)$ and all single-qubit gates, but which we could have been chosen from among any of the standard possibilities.

Of particular importance to this paper will be the set $\mathcal{Q} = \mathcal{B} \cup \{\mathsf{RAG}_n \mid n \text{ a power of } 2\}$ which contains our universal set together with the *random-access gates*, so that $\mathsf{RAG}_n \in \mathcal{U}(n2^{1+n})$ is defined on the computational basis by:

$$\mathsf{RAG}_n|i, b, x_0, \ldots, x_{n-1}\rangle = |i, x_i, x_0, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_{n-1}\rangle \qquad \forall i \in [n], b, x_0, \ldots, x_{n-1} \in \{0,1\}$$

We now give a formal definition of what it means to solve a Boolean relation $F \subseteq \{0,1\}^n \times \{0,1\}^m$ using a quantum circuit. This includes the special case when F is a function.

A *quantum circuit* over a gate set $\mathcal{G}$ (such as $\mathcal{B}$ or $\mathcal{Q}$) is a tuple $C = (n, T, S, C_1, \ldots, C_T)$, where $T \ge 0$, $n, S \ge 1$ are natural numbers, and the C_t give us a sequence of instructions. Each instruction C_t comes from a set $\mathcal{I}_{\mathcal{G}}(S)$ of possible instructions, defined below. The number n is the input length, the number T is the *time complexity*, and S is the *space complexity*, also called the *number of wires* or the *number of ancillary qubits* of the circuit. Given an input $x \in \{0,1\}^n$, at each step $t \in \{0, \ldots, T\}$ of computation, the circuit produces an S-qubit state $|\psi_T(x)\rangle \in \mathcal{H}(2^S)$, starting with $|\psi_0(x)\rangle = |0\rangle^{\otimes s}$, and then applying each instruction C_t, as we will now describe.

For each possible q-qubit gate $G \in \mathcal{G} \cap \mathcal{U}(2^q)$, and each possible ordered choice $I = (i_1, \ldots, i_q) \in [S]^q$ of distinct q among S qubits, we have an instruction $\mathsf{APPLY}_{G,I} \in \mathcal{I}_{\mathcal{G}}(S)$ which applies gate G to the qubits indexed by I, in the prescribed order. The effect of executing the instruction $\mathsf{APPLY}_{G,I}$ on $|\psi\rangle \in \mathcal{H}(2^S)$ is to apply G on the qubits indexed by I, tensored with identity on the remaining $S - q$ qubits. I.e., $\mathsf{APPLY}_{G,I} \in \mathcal{U}(2^S)$ corresponds to the unitary transformation defined on each basis state by:

$$\mathsf{APPLY}_{G,I} \cdot |y_I\rangle \otimes |y_J\rangle = (G|y_I\rangle) \otimes |y_J\rangle,$$

where $J = [S] \setminus I$.

Furthermore, for each possible ordered choice $I = (i_1, \ldots, i_{\lceil \log n \rceil}) \in [S]^{\lceil \log n \rceil}$ of distinct $\lceil \log n \rceil$ among S qubits, and each $i \in [S] \setminus I$, we have an instruction $\mathsf{READ}_{I,i} \in \mathcal{I}_{\mathcal{G}}(S)$, which applies the query oracle on the qubits indexed by I and i. I.e., given an input $x \in \{0,1\}^n$, the instruction $\mathsf{READ}_{I,i} \in \mathcal{U}(2^S)$ applies the unitary transformation defined on each basis state by:

$$\mathsf{READ}_{I,i} \cdot |y_I\rangle \otimes |y_i\rangle \otimes |y_J\rangle = |y_I\rangle \otimes |y_i \oplus x_{y_I}\rangle \otimes |y_J\rangle,$$

where $J = [S] \setminus (I \cup \{i\})$.

Hence if we have a sequence $C_1, \ldots, C_T$ of instructions and an input x, we may obtain the state of the memory at time step t, on input x, by $|\psi_0(x)\rangle = |0\rangle^{\otimes S}$ and $|\psi_{t+1}(x)\rangle = C_{t+1}|\psi_t(x)\rangle$.

We say that a quantum circuit $C = (n, T, S, C_1, \ldots, C_T)$ *computes* or *solves a relation* $F \subseteq \{0,1\}^n \times \{0,1\}^m$ *with error* ε if C is such that, for every input $x \in \{0,1\}^n$, if we measure the first m qubits of $|\psi_T(x)\rangle$ in the computational basis, we obtain, with probability $\geq 1 - \varepsilon$, a string $z \in \{0,1\}^m$ such that $(x, z) \in F$.

2.1 Quantum Random-Access Machine (QRAM)

Generally speaking, a quantum circuit is allowed to apply any of the basic operations to any of its qubits. In the definition given above, a quantum random-access gate can specify any permuted subset of the qubits to serve as its inputs. This allows for unusual circuit architectures, which are undesirable.

One may then define a more restricted class of circuits, as follows. We think of the qubits as divided into two parts: work qubits and memory qubits. We have M memory qubits and $W = O(\log M)$ work qubits, for a total space complexity $S = W + M$. We restrict the circuit so that any unitary gate $G \in \mathcal{B}$, or read instruction, must be applied to work qubits only. And, finally, any random-access gate must be applied in such a way that the addressing qubits (i) and the swap qubit (b) are always the first $\log M + 1$ work qubits, and the addressed qubits $(x_0, \ldots, x_{M-1})$ are exactly the memory qubits, and are always addressed in the same, fixed order, so one can speak of *the first memory qubit, the second memory qubit, etc.* We may then think of a computation as alternating between doing some computation on the work registers, then swapping some qubits between work and memory registers, then doing some more computation on the work registers, and so forth. The final computational-basis measurement is also restricted to measuring a subset of the work qubits.

Under these restrictions, a circuit of time complexity T may be encoded using $O(T \log S)$ bits, whereas in general one might need $\Omega(TS)$ qubits in order to specify how the wires of the circuit connect to the random-access gates.

We will then use the term a *quantum random-access machine algorithm*, or *QRAM algorithm*, for a family of circuits that operate under these restrictions.[3]

2.2 Sparse QRAM algorithms

In classical algorithms, we may have an algorithm which uses M memory registers, but such that, at any given time, only m out of these M registers are non-zero. In this case we could call such an algorithm *m-sparse*. The following definition is the quantum analogue of this.

[3] Such a computational model has been referred to by several names in the past. For instance, the term *QRAQM* appears in several publications, starting with [6], and *QAQM* has also been used [7].

▶ **Definition 2.** *Let $\mathcal{C} = (n, T, W, M, C_1, \dots, C_T)$ be a* QRAM *algorithm using time T, W work qubits, and M memory qubits. Then, we say that C is m-sparse, for some $m \leq M$, if at every time-step $t \in \{0, \dots, T\}$ of the algorithm, the state of the memory qubits is supported on computational basis vectors of Hamming weight $\leq m$. I.e., we always have*

$$|\psi_t\rangle \in \mathsf{span}\left(|u\rangle|v\rangle \;\middle|\; u \in \{0,1\}^W, v \in \binom{[M]}{\leq m} \right)$$

In other words, if $|\psi_t\rangle$ is written in the computational basis:

$$|\psi_t\rangle = \sum_{u \in \{0,1\}^W} \sum_{v \in \{0,1\}^M} \alpha_{u,v}^{(t)} \cdot \underbrace{|u\rangle}_{\textit{Work qubits}} \otimes \underbrace{|v\rangle}_{\textit{Memory qubits}},$$

then $\alpha_{u,v}^{(t)} = 0$ whenever $|v| > m$.

2.3 Time complexity of simple operations (the constant γ)

Throughout the paper we will often describe algorithms that use certain simple operations over a logarithmic number of bits. These may include comparison, addition, bitwise XOR, swapping, and others. In a classical random-access machine, all of these operations can be done in $O(1)$ time, as in such machines it is usually considered that every memory position is a register that can hold $O(\log n)$ bits, and such simple operations are taken to be machine instructions.

We do not necessarily wish to make such an assumption for quantum algorithms, since we do not really know what a quantum computer will look like, just yet. So we will broadly postulate the existence of a quantity γ, which is an upper-bound on the time complexity of doing such simple operations. We then express our time upper-bounds with γ as a parameter. Depending on the precise architecture of the quantum computer, one may think of γ as being $O(1)$, or $O(\log n)$. In all our bounds, the simple operations that we will make use of can always be implemented using $O(\log M)$ elementary gates.

2.4 Controlled unitaries

Sometimes we will explain how to implement a certain unitary, and we wish to have a version of the same unitary which can be activated or deactivated depending on the state of an additional control bit. We will make free use of the following lemma, which we state without proof.

▶ **Lemma 3.** *If a unitary U can be implemented using T gates from $\mathcal{Q}$, then the unitary*

$$|b\rangle|x\rangle \mapsto \begin{cases} |b\rangle(U|x\rangle) & \text{if } b = 1 \\ |b\rangle|x\rangle & \text{if } b = 0 \end{cases}$$

can be implemented (without error) using $O(T)$ gates from $\mathcal{Q}$.

3 Compressing sparse algorithms using quantum radix trees

Let $\mathcal{C} = (n, T, W, M, C_1, \dots, C_T)$ be the circuit of an m-sparse QRAM algorithm computing a relation F with error ε and let the state of the algorithm at every time-step t, when written in the computational basis, be

$$|\psi_t\rangle = \sum_{u \in \{0,1\}^W} \sum_{v \in \binom{[M]}{\leq m}} \alpha_{u,v}^{(t)} \underbrace{|u\rangle}_{\text{Work qubits}} \otimes \underbrace{|v\rangle}_{\text{Memory qubits}}. \tag{1}$$

Using the description of $\mathcal{C}$ and the assumption that this algorithm is m-sparse we will now construct another QRAM algorithm $\mathcal{C}'$ that uses much less space ($O(m \log M)$ qubits) and computes F with almost same error probability with only $O(\log M \log T)$ factor worsening in the run time.

Main observation

As the state of the memory qubits in $|\psi_t\rangle$ for any t is only supported on computational basis vectors of Hamming weight at most m, one immediate way to improve on the space complexity is to succinctly represent the state of the sparsely used memory qubits. The challenge, however, is that every instruction C_i in $\mathcal{C}$ might not have an *easy* analogous implementation in the succinct representation. So we will first present a succinct representation and then show that, for every instruction C_i in the original circuit $\mathcal{C}$, there is an analogous instruction or a series of instructions that evolve the state of the succinct representation in the same way as the original state evolves due to the application of C_i.

A succinct representation

Let $v \in \{0,1\}^M$ be a vector with $|v| \leq m$ (with $|v\rangle$ being the corresponding quantum state that uses M qubits). Whenever m is significantly smaller than M (i.e., $m \log M < M$) we can instead represent the vector v using the list of indices $\{i\}$ such that $v[i] = 1$. Such a representation will use much fewer (qu)bits. Let S_v denote the set of indices i such that $v[i] = 1$. We will then devise a quantum state $|S_v\rangle$, that represents the set S_v using a quantum data structure. This representation will be unique, meaning that for every sparse computational-basis state $|v\rangle$ there will be a unique corresponding quantum state $|S_v\rangle$, and $|S_v\rangle$ will use much fewer qubits. Then for every time-step t, the quantum state $|\psi_t\rangle$ from Equation (1) has a corresponding *succinctly represented* quantum state $|\phi_t\rangle$ such that

$$|\phi_t\rangle = \sum_{u \in \{0,1\}^w} \sum_{v \in \binom{[M]}{\leq m}} \alpha_{u,v}^{(t)} |u\rangle \otimes |S_v\rangle. \tag{2}$$

By using such a succinct representation, we will be able to simulate the algorithm $\mathcal{C}$ with $O(m \log M)$ qubits, with an $O(\gamma \log \frac{T}{\delta})$ additional factor overhead in time and an additional δ probability of error.

To obtain the desired succinct representation $|S_v\rangle$, we use the quantum radix trees appearing in an algorithm for the subset-sum problem by Bernstein, Jeffery, Lange, and Meurer [3] (see also [5]). Several crucial aspects of the implementation were missing or buggy, and required some amount of work to complete and fix. The resulting effort revealed, in particular, that the data-structure is unlikely to be implementable efficiently without error (as it relies on a particular gate which cannot be implemented in an error-free way using the usual basic gates). So we here include all the required details.

3.1 Radix Tree

A quantum radix tree is a quantum data structure inspired by the classical radix tree whose definition is as follows.

▶ **Definition 4.** *A radix tree is a rooted binary tree, where the edges are labeled by non-empty binary strings, and the concatenation of the labels of the edges along any root-to-leaf path results in a string of the same length ℓ (independent of the chosen root-to-leaf path). The value ℓ is called the* word length *of the tree.*

There is a bijective correspondence between radix trees R of word length ℓ and subsets $S \subseteq \{0,1\}^\ell$. Given R, we may obtain S as follows. Each root-to-leaf path of R gives us an element $x \in S$, so that x is the concatenation of all the edge labels along the path.

If R corresponds to S, we say that R stores*, or* represents *S, and write $R(S)$ for the radix tree representing S, i.e., for the inverse map of what was just described (see below).*

An example of a radix tree appears in Figure 1.

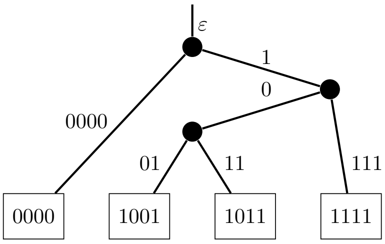

Figure 1 A radix tree storing the set $\{0000, 1001, 1011, 1111\}$.

Given a set $S \subseteq \{0,1\}^\ell$, we obtain $R(S)$ recursively as follows: The empty set corresponds to the tree having only the root and no other nodes. We first find the longest common prefix $p \in \{0,1\}^{\leq \ell}$ of S. If $|p| > 0$, then we have a single child under the root, with a p-labeled edge going into it, which itself serves as the root to $R(S')$, where S' is the set of suffixes (after p) of S. If $|p| = 0$, then the root will have two children. Let $S = S_0 \cup S_1$, where S_0 and S_1 are sets of strings starting with 0 and 1, respectively, in S. The edges to the left and right children will be labeled by p_0 and p_1, respectively, where $p_0 \in \{0,1\}^{\leq \ell}$ is the longest common prefix of S_0 and $p_1 \in \{0,1\}^{\leq \ell}$ is the longest common prefix of S_1. The left child serves as a root to $R(S_0')$, where S_0' is the set of suffixes (after p_0) of S_0. Analogously, the right child serves as a root to $R(S_1')$, where S_1' is the set of suffixes (after p_1) of S_1.

Basic operations on radix trees

The allowed basic operations on a radix tree are insertion and removal of an element. Classically, an attempt at inserting an element already in S will result in the identity operation. Quantumly, we will instead allow for *toggling* an element in/out of S.

Representing a radix tree in memory

We now consider how one might represent a radix tree in memory. For this purpose, suppose we wish to represent a radix tree $R(S)$ for some set $S \subseteq \{0,1\}^\ell$ of size $|S| \leq m$. Let us assume without loss of generality that m is a power of 2, and suppose we have at our disposal an array of $2m$ memory blocks.

Each memory block may be used to store a node of the radix tree. If we have a node in the tree, the contents of its corresponding memory block will represent a tuple (z, p_1, p_2, p_3). The value $z \in \{0,1\}^{\leq \ell}$ stores the label in the edge from the node's parent, the values $p_1, p_2, p_3 \in \{0, 1, \ldots, 2m\}$ are pointers to the (block storing the) parent, left child, and right child, respectively, or 0 if such an edge is absent.

It follows that each memory block is $O(\ell + \log m)$ bits long. In this way, we will represent $R(S)$ by a binary string of length $O(m(\ell + \log m))$. The root node is stored in the first block, empty blocks will be set to 0, and the only thing that needs to be specified is the *memory layout*, namely, in which block does each node get stored. For this purpose, let $\tau : R(S) \to [2m]$ be an injective function, mapping the nodes of $R(S)$ to the $[2m]$ memory blocks, so that $\tau(\text{root}) = 1$. For any $S \subseteq \{0,1\}^\ell$ of size $|S| \leq m$, we then let

$$R_\tau(S) \in \{0,1\}^{O(m(\ell+\log m))}$$

denote the binary string obtained by encoding $R(S)$ as just described.

BJLM's quantum radix tree

We see now that although there is a unique radix tree $R(S)$ for each S, there is no obvious way of making sure that the representation of $R(S)$ in memory is also unique. However, this bijective correspondence between S and its memory representation is a requirement for quantum algorithms to use interference. The idea of Bernstein et al [3], then, is to represent S using a superposition of *all possible layouts.* I.e., S is to be uniquely represented by the (properly normalized) quantum state:

$$\sum_\tau |R_\tau(S)\rangle.$$

The trick, then, is to ensure that this representation can be efficiently queried and updated. In their discussion of how this might be done, the BJLM paper [3] presents the broad idea but does not work out the details, whereas Jeffery's thesis [5] glosses over several details and includes numerous bugs and omissions. To make their idea work, we make use of an additional data structure.

3.2 Prefix-Sum Tree

In our implementation of the Quantum Prefix Tree, we will need to keep track of which blocks are empty and which are being used by a node. For this purpose, we will use a data-structure that is famously used to (near-optimally) solve the dynamic prefix-sum problem.

▶ **Definition 5.** *A prefix-sum tree is a complete rooted binary tree. Each leaf node is labelled by a value in* $\{0,1\}$*, and each internal node is labelled by the number of* 1*-valued leaf nodes descending from it.*

Let $F \subseteq [\ell]$ *for* ℓ *a power of* 2*. We use* $P(F)$ *to denote the prefix-sum tree where the* i^{th} *leaf node of the tree is labelled by* 1 *iff* $i \in F$*.*

A prefix-sum tree $P(F)$ will be represented in memory by an array of $\ell - 1$ blocks of memory, holding the labels of the inner nodes of $P(F)$, followed by ℓ bits, holding the labels of the leaf nodes. The blocks appear in the same order as a breadth-first traversal of $P(F)$. Consequently, for every $F \in \{0,1\}^\ell$ there is corresponding binary string of length $(\ell - 1)\log \ell + \ell$ that uniquely describes $P(F)$.

We will overload notation, and use $P(F)$ to denote this binary string of length $(\ell - 1)\log \ell + \ell$.

Allocating and deallocating

The idea now is to use the prefix tree as an memory allocator. We have $2m$ blocks of memory, and the set F will keep track of which blocks of memory are unused, or "free".

We would then like to have an operation that allocates one of the free blocks. To implement Bernstein et al's idea, the choice of which block to allocate is made in superposition over all possible free blocks. I.e., we would like to implement the following map U_{alloc} and also its inverse, U_{free}.

$$U_{\text{alloc}} : |P(F)\rangle|0\rangle|0\rangle \to \frac{1}{\sqrt{|F|}} \sum_{i \in F} |P(F \setminus \{i\})\rangle|i\rangle|0\rangle, \tag{3}$$

The second and third registers have $O(\log m)$ bits. We do not care for what the map does when these registers are non-zero, or when $F = \varnothing$. We will guarantee that this is never the case.

Note that each internal node of the prefix tree stores the number of elements of F that are descendants to that node. In particular, the root stores $|F|$. In order to implement U_{alloc}, we then start by constructing the state

$$\frac{1}{\sqrt{|F|}} \sum_{j=1}^{|F|} |j\rangle. \tag{4}$$

While this might appear to be simple, it actually requires us to use a gate

$$U_{\text{superpose}} : |k\rangle|0\rangle \mapsto \frac{1}{\sqrt{k}} \sum_{j=1}^{k} |k\rangle|j\rangle. \tag{5}$$

This is much like choosing a random number between 1 and a given number k on a classical computer. Classically, such an operation cannot be done exactly if all we have at our disposition are bitwise operations (since all achievable probabilities are then dyadic rationals). Quantumly, it is impossible to implement $U_{\text{superpose}}$ efficiently without error by using only the usual set of basic gates.

So the reader should take note: it is precisely this gate which adds error to BJLM's procedure. This gate can be implemented up to distance ε using $O(\log \frac{m}{\varepsilon})$ basic gates, where m is the maximum value that k can take. I.e., using so many gates we can implement a unitary U such that the spectral norm $\|U - U_{\text{superpose}}\| \leq \varepsilon$.[4] We will need to choose $\varepsilon \approx \frac{1}{T}$, which is the inverse of the number of times such a gate will be used throughout our algorithm.

Once we have prepared state (4), we may then use binary search, going down through the prefix tree to find out which location i corresponds to the j^{th} non-zero element of F. Using i, as we go up we can remove the corresponding child from $P(F)$, in $O(\gamma \cdot \log m)$ time, while updating the various labels on the corresponding root-to-leaf path. This requires the use of $O(\log m)$ work bits, which are $|0\rangle$ at the start and end of the operation. During this process, the register holding j is also reset to $|0\rangle$, by subtracting the element counts we encounter during the deletion process from this register. The inverse procedure U_{free} is implemented in a similar way.

3.3 Quantum Radix Tree

We may now define the quantum radix tree.

[4] This is done by using Hadamard gates to get a superposition between 1 and the smallest power of 2 which is greater than $\frac{m}{\varepsilon}$, and then breaking this range into m equal intervals plus a remainder of size $< m$. The *remainder subspace* will have squared amplitude $\leq \varepsilon$.

▶ **Definition 6** (Quantum Radix Tree). *Let ℓ and m be powers of 2, $S \subseteq \{0,1\}^\ell$ be a set of size $s = |S| \leq m$, and let $R(S)$ be the classical radix tree storing S. Then, the quantum radix tree corresponding to S, denoted $|R_Q(S)\rangle$ (or $|R_Q^{\ell,m}(S)\rangle$ when ℓ and m are to be explicit), is the state*

$$|R_Q(S)\rangle = \frac{1}{\sqrt{N_S}} \cdot \sum_\tau |R_\tau(S)\rangle |P(F_\tau)\rangle,$$

where τ ranges over all injective functions $\tau : R(S) \to [2m]$ with $\tau(root) = 1$, of which there are $N_S = \frac{(2m-1)!}{(2m-|R(S)|)!}$ many, and $F_\tau = [2m] \setminus \tau(R(S))$ is the complement of the image of τ.

Basic operations on quantum radix trees

The basic allowed operations on a quantum radix trees are look-up and toggle, where the toggle operation is analogous to insertion and deletion in classical radix tree. Additionally, we also define a swap operation which will be used to simulate a RAG gate.

▶ **Lemma 7.** *Let $|R_Q(S)\rangle = |R_Q^{\ell,m}(S)\rangle$ denote a quantum radix tree storing a set $S \subseteq \{0,1\}^\ell$ of size at most m. We then define the following data structure operations.*

1. ***`Lookup`***. *Given an element $e \in \{0,1\}^\ell$, we may check if $e \in S$, so for each $b \in \{0,1\}$, we have the map*

   $$|e\rangle|R_Q(S)\rangle|b\rangle \mapsto |e\rangle|R_Q(S)\rangle|b \oplus (e \in S)\rangle.$$

2. ***`Toggle`***. *Given $e \in \{0,1\}^\ell$, we may add e to S if S does not contain e, or otherwise remove e from S. Formally,*

   $$|e\rangle|R_Q(S)\rangle \mapsto \begin{cases} |e\rangle|R_Q(S \cup \{e\})\rangle, & \text{if } e \notin S, \\ |e\rangle|R_Q(S \setminus \{e\})\rangle, & \text{if } e \in S. \end{cases}$$

3. ***`Swap`***. *Given an element $e \in \{0,1\}^\ell$, $b \in \{0,1\}$ and a quantum radix tree storing a set S, we would like `swap` to be the following map,*

   $$|e\rangle|R_Q(S)\rangle|b\rangle \mapsto \begin{cases} |e\rangle|R_Q(S \cup \{e\})\rangle|0\rangle, & \text{if } e \notin S \text{ and } b = 1, \\ |e\rangle|R_Q(S \setminus \{e\})\rangle|1\rangle, & \text{if } e \in S \text{ and } b = 0, \\ |e\rangle|R_Q(S)\rangle|b\rangle, & \text{otherwise.} \end{cases}$$

These operations can be implemented in worst case $O(\gamma \cdot \log m)$ time and will be error-free if we are allowed to use an error-free gate for $U_{superpose}$ (defined in Equation 5), along with other gates from set $\mathcal{Q}$.

Proof. Let $|b\rangle, |e\rangle$ denote the quantum states storing the elements $b \in \{0,1\}$ and $e \in \{0,1\}^\ell$, respectively. The data structure operations such as `lookup`, `toggle` and `swap` can be implemented reversibly in $O(\gamma \cdot \log m)$ time in the following way.

`Lookup`

We wish to implement the following reversible map U_{lookup},

$$U_{lookup} : |e\rangle|R_Q(S)\rangle|b\rangle \mapsto |e\rangle|R_Q(S)\rangle|b \oplus (e \in S)\rangle. \tag{6}$$

We do it as follows. First note that, by Definition 6,

$$|R_Q(S)\rangle = \frac{1}{\sqrt{N_S}} \sum_\tau |R_\tau(S)\rangle |P(F_\tau)\rangle.$$

We will traverse $R_\tau(S)$ with the help of some auxiliary variables. Starting at the root node, we find the edge labeled with a prefix of e. If no such label is found then e is not present in $R_\tau(S)$. Otherwise, we traverse to the child reached by following the edge labeled by a prefix of e. Let us denote the label by L. If the child is a leaf node then terminate the process, stating that e is present in $R_\tau(S)$, else, recurse the process on e' and the tree rooted at that child node. Here e' is the binary string after removing L from e. When at some point we have determined whether $e \in S$ or not, we flip the bit b, or not. Eventually, we may conclude that $e \notin S$ before traversing the entire tree, at which point we skip the remaining logic for traversing $R_\tau(S)$ downwards (by using a control qubit). After we have traversed $R_\tau(S)$ downwards and determined whether $e \in S$, we need to undo our traversal, which we do by following the p_1 pointers (to the parent nodes) until the root is again reached, and the auxiliary variables are again set to 0.

Each comparison with the edge labels, at each traversed node, takes $O(\gamma)$ time. Hence, the entire procedure takes $O(\gamma \cdot \log m)$ time.

Toggle

Let U_{toggle} denote the following map,

$$U_{toggle} : |e\rangle|R_Q(S)\rangle \rightarrow \begin{cases} |e\rangle|R_Q(S \setminus \{e\})\rangle, & \text{if } e \in S, \\ |e\rangle|R_Q(S \cup \{e\})\rangle, & \text{if } e \notin S \end{cases} \tag{7}$$

The toggle operation primarily consists of two main parts: The memory allocation or de-allocation, followed by insertion or deletion, respectively.

We again traverse $R_\tau(S)$ with the help of some auxiliary variables. We start with the root node of $R_\tau(S)$, and traverse the tree downwards until we know, as above, whether $e \in S$ or not. If $e \notin S$, we will know where we need to insert nodes into $R_\tau(S)$, in order to transform it into $R_\tau(S \cup \{e\})$. Below, we will explain in detail how such an insertion must proceed. It turns out that we may need to insert either one node, or two, but never more. We may use the work qubits to compute the contents of the memory blocks that will hold this new node (or new nodes). These contents are obtained by XORing the appropriate bits of e and the appropriate parent/child pointers of the nodes we are currently traversing in the tree.

We may then use the U_{alloc} gate (once or twice) to obtain the indices of the blocks that will hold the new node(s). We then use RAG gates to swap in the contents of these blocks into memory. A fundamental and crucial detail must now be observed: the index of the memory blocks into where we inserted the new nodes is now left as part of the work qubits. This cannot be and must be dealt with, because every work bit must be again set to zero at the end of the procedure. However, a copy of this index now appears as the child pointer (p_2 or p_3) of the parents of the nodes we just created, and these pointers can thus be used to zero out the index. It is then possible to traverse the tree upwards in order to undo the various changes we did to the auxiliary variables.

If $e \in S$, on the other hand, we then do the inverse procedure. We will then know which nodes need to be removed from $R_\tau(S)$ (it will be either one or two nodes). By construction, these nodes will belong to blocks not in F_τ. We begin by setting these blocks to zero by swapping the blocks into the workspace (using the RAG gate), XORing the appropriate bits of e and the appropriate child/parent pointers so the blocks are now zero, and swapping them back. These blocks will then be set to zero, and we are left with a state akin to the right-hand side of (3). We then use the U_{free} gate to *free* the blocks, i.e., add their indices to F_τ once again. At this point we can traverse the tree upwards once more, in order to reset the auxiliary variables to zero, as required.

We now give further detail on how one must update $R_\tau(S)$ in order to insert a new element e into S. We must create a node $N := (z, p_1, p_2, p_3)$ corresponding to the element e stored at the memory location assigned by U_{alloc} procedure. Let us denote the address by k. Start with the root node of $R_\tau(S)$. If e has no common prefix with any of the labels of the root's outgoing edges, which can only happen if the root has one child, then set z to e, p_1 pointing to the root node, and, p_2 and p_3 set to 0. Moreover, set the value of the root's p_2 pointer to k if node N ends up as the left child to the root, else set root's p_3 pointer to k. In the case when e has a common prefix with one of the labels of the root's outgoing edges, let us denote the label by L and the child node by C, then further two scenarios arise: Either label L is completely contained in e, which if is the case then we traverse the tree down and run the insertion procedure recursively on e' (which is e after removing the prefix L) with the new root set C. In the case where label L is not completely contained in e, we create an internal node N' with its z variable set to the longest common prefix of e and L (which we denote by L'), p_1 pointing to root, p_2 pointing to C and p_3 pointing to N (or vice versa depending on whether node N gets to be the right or the left child). We run the U_{alloc} procedure again to get a memory location to store N'. Having done that, we now change the z value of node C to be the prefix of L after L', and the p_1 value of node C to be the memory location of N'. Additionally, we also set z of node N to be e', the suffix of e after L', and we let p_1, p_2, p_3 to be, respectively, a pointer to N', 0 and 0.

Each step in the traversal takes time $O(\gamma)$, for a total time of $O(\gamma \cdot \log m)$.

The procedure to update $R_\tau(S)$ in order to delete an element e from S is analogous to the insertion procedure mentioned above, which also can be implemented in $O(\gamma \cdot \log m)$ time.

Swap

Let U_{swap} denote the following map,

$$U_{swap} : |e\rangle|R_Q(S)\rangle|b\rangle \mapsto \begin{cases} |e\rangle|R_Q(S \cup \{e\})\rangle|0\rangle, & \text{if } e \notin S \text{ and } b = 1, \\ |e\rangle|R_Q(S \setminus \{e\})\rangle|1\rangle, & \text{if } e \in S \text{ and } b = 0, \\ |e\rangle|R_Q(S)\rangle|b\rangle, & \text{otherwise.} \end{cases}$$

To implement U_{swap}, we first run the U_{lookup} on the registers $|e\rangle$, $|R_Q(S)\rangle$ and $|b\rangle$. Conditional on the value of register $|b\rangle$ (i.e., when $b = 1$), we run U_{toggle} on the rest of the registers. We then run U_{lookup} again to attain the desired state. To summarize, the unitary $U_{swap} = U_{lookup} \cdot C_{toggle} \cdot U_{lookup}$, where C_{toggle} is controlled version of U_{toggle} (as per Lemma 3). Thus, the `swap` procedure takes a total time of $O(\gamma \cdot \log m)$. ◀

An error-less, efficient implementation of the unitary $U_{\text{superpose}}$ is impossible by using only the usual sets of basic gates. Furthermore, it is unreasonable to expect to have an error-free $U_{\text{superpose}}$ at our disposal. However, as we explained in page 9, there is a procedure to implement $U_{\text{superpose}}$ using gates from the gate set $\mathcal{B} = \{\mathsf{CNOT}, H, S, T\}$ up to spectral distance ε, using only $O(\log \frac{m}{\epsilon})$ gates.

▶ **Corollary 8.** *Let $|R_Q(S)\rangle = |R_Q^{\ell,m}(S)\rangle$ denote a quantum radix tree storing a set $S \subseteq \{0,1\}^\ell$ of size at most m. The data structure operations `look-up`, `toggle` and `swap`, as defined in the statement of Lemma 7 can be implemented in $O(\gamma \cdot \log \frac{m}{\epsilon})$ time and ϵ probability of error using gates from the gate set $\mathcal{Q}$. Here γ is the number of gates required from set $\mathcal{Q}$ to do various basic operations on a logarithmic number of qubits.*

3.4 The simulation

Recall from Section 2.3 that we take γ to be the number of gates required to do various basic operations on a logarithmic number of qubits. In our use below, it never exceeds $O(\log M)$.

▶ **Theorem 9.** *Let T, W, $m < M = 2^\ell$ be natural numbers, with M and m both powers of 2, and let $\varepsilon \in [0, 1/2)$. Suppose we are given an m-sparse QRAM algorithm using time T, W work qubits and M memory qubits, that computes a Boolean relation F with error ε.*

Then we can construct a QRAM algorithm which computes F with error $\varepsilon' > \varepsilon$, and runs in time $O(T \cdot \log(\frac{T}{\varepsilon'-\varepsilon}) \cdot \gamma)$, using $W + O(\log M)$ work qubits and $O(m \log M)$ memory qubits.

Proof. Let $\mathcal{C} = (n, T, W, M, C_1, \ldots, C_T)$ be the circuit of the given m-sparse QRAM algorithm computing a relation F with error ε and, let the state of the algorithm at every time-step t, when written in the computational basis be

$$|\psi_t\rangle = \sum_{u \in \{0,1\}^w} \sum_{v \in \binom{[M]}{\leq m}} \alpha_{u,v}^{(t)} \cdot \underbrace{|u\rangle}_{\text{W qubits}} \otimes \underbrace{|v\rangle}_{\text{M qubits}} \tag{8}$$

where the set $\binom{[M]}{\leq m}$ denotes all vectors $v \in \{0,1\}^M$ such that $|v| \leq m$. Using the description of $\mathcal{C}$ and the fact that this algorithm is m-sparse we will now construct another QRAM algorithm $\mathcal{C}'$ with the promised bounds. The algorithm C' will have $w' = W + O(\log M)$ work bits, and $O(m \log M)$ memory bits. The memory is to be interpreted as an instance $|R_Q(S)\rangle$ of the quantum radix tree described above. Then $|v\rangle$ will be represented by the quantum radix tree $|R_Q(S_v)\rangle$, where $S_v = \{i \in [M] \mid v_i = 1\}$ is the set of positions where $v_i = 1$, so that each position $i \in [M]$ is encoded using a binary string of length ℓ.

The simulation is now simple to describe. First, the quantum radix tree is initialized. Then, each non-RAG instruction $C_i \in \mathcal{C}$ operating on the work qubits of $\mathcal{C}$ is applied in the same way in $\mathcal{C}'$ to same qubits among the first W qubits of $\mathcal{C}'$. Each RAG instruction, on the other hand, is replaced with the U_{swap} operation, applied to the the quantum radix tree. The extra work qubits of $\mathcal{C}'$ are used as anciliary for these operations, and we note that they are always returned to zero.

If we assume that the U_{swap} operation can be implemented without error, we then have a linear-space isomorphism between the two algorithms' memory space, which maps the state $|\psi_t\rangle$ of $\mathcal{C}$ at each time step t to the state $|\phi_t\rangle$ of $\mathcal{C}'$ after t simulated steps:

$$|\phi_t\rangle = \sum_{u,v} \alpha_{u,v}^{(t)} \cdot \underbrace{|u\rangle}_{W} \otimes \underbrace{|0\rangle}_{O(\log M)} \otimes \underbrace{|R_Q(S_v)\rangle}_{O(m \log M)}.$$

Thus, if U_{swap} could be implemented without error, we could have simulated $\mathcal{C}$ without additional error. Otherwise, as per Corollary 8, we may implement the U_{swap} unitary with an error parameter $\Omega(\frac{\varepsilon'-\varepsilon}{T})$, resulting in a total increase in error of $\varepsilon' - \varepsilon$, and an additional time cost of $O(T \log \frac{T}{\varepsilon'-\varepsilon})$. ◀

4 Simplifications of previous work

It is possible to use our main theorem to simplify the presentation of the following three results: Ambainis' Quantum Walk algorithm for solving the k-Element Distinctness problem [2], Aaronson et al's Quantum algorithms for the Closest Pair problem (CP), and the authors' previous paper on Fine-Grained Complexity via Quantum Walks [4].

All these results use quantum walk together with complicated, space-efficient, history-independent data structures. As we will see, it is possible to replace these complicated data structures with simple variants of the prefix-sum tree (Section 3.2), where the memory use is sparse, and then invoke the main theorem of our paper.

The proofs are omitted in the main body due to space constraints are instead included in the appendix.

References

1 Scott Aaronson, Nai-Hui Chia, Han-Hsuan Lin, Chunhao Wang, and Ruizhe Zhang. On the quantum complexity of closest pair and related problems. In *Proceedings of the 35th Computational Complexity Conference*, CCC '20, Dagstuhl, DEU, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CCC.2020.16.

2 A. Ambainis. Quantum walk algorithm for element distinctness. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 22–31, 2004. doi:10.1109/FOCS.2004.54.

3 Daniel J. Bernstein, Stacey Jeffery, Tanja Lange, and Alexander Meurer. Quantum algorithms for the subset-sum problem. In Philippe Gaborit, editor, *Post-Quantum Cryptography*, pages 16–33, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

4 Harry Buhrman, Bruno Loff, Subhasree Patro, and Florian Speelman. Limits of quantum speed-ups for computational geometry and other problems: Fine-grained complexity via quantum walks, 2021. arXiv:2106.02005.

5 Stacey Jeffery. *Frameworks for Quantum Algorithms*. PhD thesis, University of Waterloo, 2014.

6 Greg Kuperberg. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM J. Comput.*, 35(1):170–188, July 2005. doi:10.1137/S0097539703436345.

7 María Naya-Plasencia and André Schrottenloher. Optimal merging in quantum -xor and -xor-sum algorithms. In *Advances in Cryptology – EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II*, pages 311–340, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-45724-2_11.

A Simplifications of previous work (the proofs)

A.1 Ambainis' Walk Algorithm for Element Distinctness

Ambainis' description and analysis of his data structure is complicated, and roughly 6 pages long, whereas a presentation of his results with a simple data structure and an appeal to our theorem requires less than 2 pages, as we will now see. Also, the presentation of the algorithm is considerably muddled by the various difficulties and requirements pertaining to the more complicated data structure. In a presentation of his results that would then appeal to Theorem 1, we have a very clear separation of concerns.

Ambainis' algorithm is a $\widetilde{O}(n^{\frac{k}{k+1}})$-time solution to the following problem:

▶ **Definition 10** (k-Element Distinctness). *Given a list L of n integers in Σ are there k elements $x_{i_1}, \dots, x_{i_k} \in L$ such that $x_{i_1} = \cdots = x_{i_k}$.*

Ambainis' algorithm for k-Element Distinctness [2] is quantum walk algorithm on a Johnson graph $J(n, r)$ with $r = n^{k/k+1}$ and runs in $\widetilde{O}(n^{k/k+1})$ time. The crucial ingredient in making the algorithm time efficient is the construction of data-structure which can store a set $S \subseteq [n] \times \Sigma$ of elements of size r, under efficient insertions and removals, so that one may efficiently query at any given time whether there exist k elements $(i_1, x_1), \ldots, (i_k, x_k)$ in S with distinct indices $i_1, \ldots, i_k$ but equal labels $x_1 = \cdots = x_k$. Ambainis makes use of skip-lists and hash tables, ensuring that all operations run in $O(\log^4(n + |\Sigma|))$ time. However, if one does not care about space-efficiency, there is a much simpler data structure that serves the same purpose. The following definition is illustrated in Figure 2.

▶ **Definition 11.** *Let $S \subseteq [n] \times \Sigma$, with $|S| = r$ and $|\Sigma| = n^{O(1)}$ a power of 2, and such that every $i \in [n]$ appears in at most one pair $(i, x) \in S$. The k-element-distinctness tree that represents S, denoted $T_k(S)$, is a complete rooted binary tree with $|\Sigma|$ leaves. Each leaf node $x \in \Sigma$ is labeled by a bit vector $\mathtt{B}_x \in \{0,1\}^n$ and a number $\mathtt{count}_x \in \{0, \ldots, n\}$, so that $\mathtt{B}_x[i] = 1$ iff $(i, x) \in S$, and the $\mathtt{count}_x$ is the Hamming weight of $\mathtt{B}_x$. Each internal node w is labeled by a bit $\mathtt{flag}_w \in \{0,1\}$ which indicates whether there exists a leaf x, descendent of w, with $\mathtt{count}_x \geq k$.*

Memory Representation

A k-element-distinctness tree is represented in the memory by an array of $|\Sigma| - 1$ bits of memory holding the flags of the internal nodes, followed by $|\Sigma|$ blocks of $n + \lceil \log n \rceil$ bits of memory each, holding the labels of the leaf nodes. The blocks appear in the same order as a breadth-first traversal of $T_k(S)$. Consequently, for every $S \subseteq [n] \times \Sigma$ there is a corresponding binary string of length $|\Sigma| - 1 + (n + \lceil \log n \rceil)|\Sigma|$ that uniquely encodes $T_k(S)$. Crucially, if $|S| = r$, then at most $O(r(\log \Sigma + \log n))$ of these bits are 1. So for $|\Sigma| = \mathsf{poly}(n)$, the encoding is $\widetilde{O}(r)$-sparse.

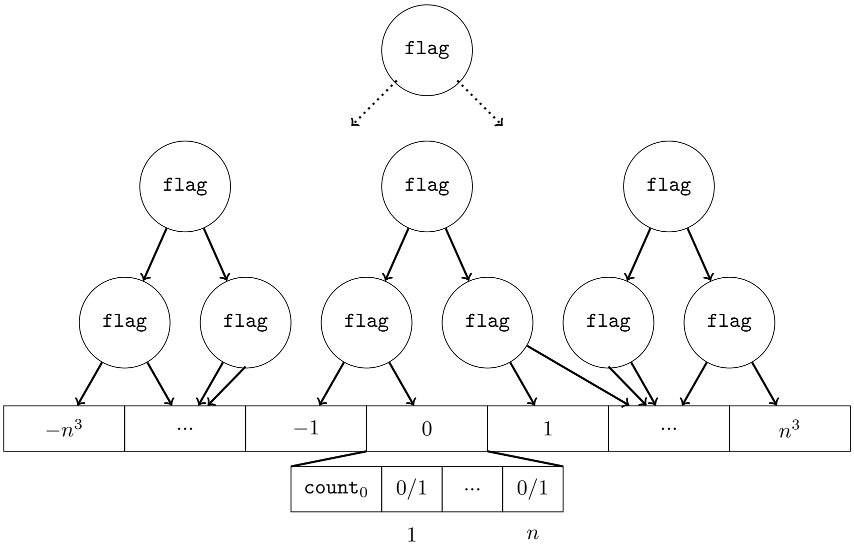

Figure 2 Data structure for the k-Element Distinctness problem.

Implemention of data structure operations

It is clear from the definition of k-element-distinctness tree and its memory representation that a tree $T_k(S)$ represents a set $S \subseteq [n] \times \Sigma$ in a history-independent way. We will now argue that all the required data structure operations take $O(\log n)$ time in the worst case. Let (i, x) denote an element in $[n] \times \Sigma$.

- **Insertion.** To insert (i, x) in the tree, first increase the value of the `count` variable of the leaf x, and set $\mathtt{B}_x[i] = 1$. Then, if $\mathtt{count} \geq k$, set $\mathtt{flag}_w = 1$ for all w on the root-to-x path. This update requires $O(\log n)$ time as $|\Sigma| = \mathsf{poly}(n)$.
- **Deletion.** The procedure to delete is similar to the insertion procedure. To delete (i, x) in the tree, first decrease the value of the $\mathtt{count}_x$ and set $\mathtt{B}[i] = 0$. If $\mathtt{count} < k$, then, for all w on the root-to-x path which do not have both children w_0, w_1 with $\mathtt{flag}_{w_0} = \mathtt{flag}_{w_1} = 1$, set $\mathtt{flag}_w = 0$. This requires $O(\log n)$ time.
- **Query.** To check if the tree has k distinct indices with the same x, we need only check if $\mathtt{flag}_{\mathrm{root}} = 1$, which takes $O(1)$ time.

Runtime, error and memory usage

Using the above data-structure, the runtime of Ambainis' algorithm is now $\widetilde{O}(n^{\frac{k}{k+1}})$ time. The total memory used is $O(n|\Sigma|)$ bits. However, note that at any point of time in any branch of computation Ambainis' walk algorithm stores sets of size $r = O(n^{\frac{k}{k+1}})$. Hence their algorithm with this data structure is a $\widetilde{O}(n^{\frac{k}{k+1}})$-sparse algorithm. Thus, invoking Theorem 1 we conclude the following.

▶ **Corollary 12.** *There is a bounded-error* QRAM *algorithm that computes k-Element Distinctness in $\widetilde{O}(n^{k/k+1})$ time using $\widetilde{O}(n^{k/k+1})$ memory qubits.*

A.2 Quantum Algorithms for Closest-Pair and related Problems

The paper of Aaronson et al [1] provides quantum algorithms and conditional lower-bounds for several variants of the Closest Pair problem (CP).

Let $\Delta(a, b) = \|a - b\|$ denote the Euclidean distance. We then describe the Closest Pair problem under Euclidean distance Δ, but we could have chosen any other metric Δ in d-dimensional space which is strongly-equivalent to the Euclidean distance (such as ℓ_p distance, Manhattan distance, ℓ_∞, *etc*).

▶ **Definition 13** (Closest Pair ($\mathsf{CP}(n, d)$) problem)**.** *In the $\mathsf{CP}(n, d)$ problem, we are given a list P of n distinct points in $\mathcal{R}^d$, and wish to output a pair $a, b \in P$ with the smallest $\Delta(a, b)$.*

We may also define a threshold version of CP.

▶ **Definition 14.** *In the TCP(n, d) problem, we are given a set $P = \{p_1, \ldots, p_n\}$ of n points in $\mathbb{R}^d$ and a threshold $\varepsilon \geq 0$, and we wish to find a pair of points $a, b \in P$ such that $\Delta(a, b) \leq \varepsilon$, if such a pair exists.*

For simplicity, so we may disregard issues of representation of the points, we assume that all points are specified using $O(\log n)$ bits of precision. By translation, we can assume that all the points lie in in the integer hypercube $[L]^d$ for some $L = \mathsf{poly}(n)$, and that $\delta \in [L]$, also.

It is then possible to solve CP by running a binary search over the (at most n^2) different values of $\delta \in \{\Delta(p_i, p_j) \mid i, j \in [n]\}$ and running the corresponding algorithm for TCP. This will add an additional $O(\log n)$ factor to the running time.

The TCP(n, d) problem is a query problem with certificate complexity 2. If one is familiar with quantum walks, it should be clear that we may do a quantum walk on the Johnson graph over n vertices, to find a pair with the desired property, by doing $O(n^{2/3})$ queries to the input. Again, if one is familiar with quantum walks, one will realize that, in order to implement this walk efficiently, we must dynamically maintain a set $S \subseteq [n]$, and at each step in the quantum walk, we must be able to add or remove an element i to S, and answer a query of the form: does there exist a pair $i, j \in S$ with $\Delta(p_i, p_j) \leq \varepsilon$?

The only difficulty, now, is to implement an efficient data structure that can dynamically maintain S in this way, and answer the desired queries, while being time and space efficient. Aaronson et al construct a data-structure which can store a set $S \subseteq [n] \times [L]^d$ of points of size r, under efficient insertions and removals, so that one may query at any given time whether there exist two points in S which are ε-close. They do so by first discretizing $[L]^d$ into a hypergrid of width $\varepsilon/\sqrt{d}$, as explained below, and then use a hash table, skip list, and a radix tree to maintain the locations of the points in the hypergrid.

The presentation of the data structure in the paper is roughly 6 pages long, and one must refer to Ambainis' paper for the error analysis, which is absent from the paper. As we will see, a simple, sparse data structure for the same purpose can be described in less than 2 pages, and then an appeal to Theorem 1 gives us the same result up to log factors.

Discretization

We discretize the cube $[L]^d$ into a hypergrid of width $w = \frac{\varepsilon}{\sqrt{d}}$, and let $\mathrm{id}(p)$ denote the box containing p in this grid. I.e., we define a function $\mathrm{id}(p) : [L]^d \to \{0,1\}^{\lceil d \log(L/\varepsilon) \rceil}$ by

$$\mathrm{id}(p) = (\lfloor p(1)/w \rfloor, \dots, \lfloor p(d)/w \rfloor) \qquad \text{(represented in binary).}$$

Let $\Sigma = \{0,1\}^{\lceil d \log(L/\varepsilon) \rceil}$ denote the set of all possible boxes. We say that two boxes $g, g' \in \Sigma$ are neighbours if

$$\sqrt{\sum_{i=1}^{d} \|g(i) - g'(i)\|^2} \leq \sqrt{d}.$$

A loose estimate will show there can be at most $(2\sqrt{d}+1)^d$ neighbours for any box. This method of discretization ensures the following crucial property:

▶ **Observation 15** (Observation 45 [1]). *Let p, q be any two distinct points in $[0, L]^d$, then*
1. *if $\mathrm{id}(p) = \mathrm{id}(q)$, then $\Delta(p,q) \leq \varepsilon$, and*
2. *if $\Delta(p,q) \leq \varepsilon$, then $\mathrm{id}(p)$ and $\mathrm{id}(q)$ are neighbours.*

From Observation 15, it follows that $i, j \in [n]$ exist with $\Delta(p_i, p_j) \leq \varepsilon$, if and only if we have one of the following two cases:
- Either there is such a pair i, j with $\mathrm{id}(p_i) = \mathrm{id}(p_j)$.
- Or there is no such pair, and then there must exist two neighbouring boxes $\mathrm{id}(i)$ and $\mathrm{id}(j)$, each containing a single point, with $\Delta(p_i, p_j) \leq \varepsilon$.

We now describe the data structure itself. Let us assume without loss of generality that n is a power of 2.

▶ **Definition 16** (Data Structure for CP). *Let $S \subseteq [n] \times \Sigma$, with $|S| = r$, and such that every $i \in [n]$ appears in at most one pair $(i, x) \in S$. The closest-pair tree that represents S, denoted by $T_{CP}(S)$, is a complete rooted binary tree with $|\Sigma|$ leaves. Each leaf node $x \in \Sigma$ is labeled by a number $\texttt{external}_x \in \{0, \dots, n\}$, and a prefix-sum tree $P(S_x)$ representing the set $S_x = \{i \in [n] \mid (i, x) \in S\}$. Each internal node w is labeled by a bit $\texttt{flag}_w \in \{0,1\}$. These labels obey the following rules:*
- *If $|S_x| = 1$, then $\texttt{external}_x$ is the number of boxes $y \neq x$, which are neighbours of x, and which have $|S_y| = 1$ and $\Delta(p_i, p_j) \leq \varepsilon$ for the (unique) $j \in S_y$.*
- *If $|S_x| \geq 2$, then $\texttt{external}_x = 0$.*
- *The $\texttt{flag}_w = 1$ if any of the children x descendants to the internal node w have either $|S_x| \geq 2$ or $|S_x| = 1$ and $\texttt{external}_x \geq 1$.*

It follows from the above discussion that there exist two elements $(i,x),(j,y) \in S$ with $\Delta(p_i,p_j) \leq \varepsilon$ if and only if $\texttt{flag}_{\text{root}} = 1$ in $T_{CP}(S)$. We now show how to efficiently maintain $T_{CP}(S)$ under insertions and removals.

Memory Representation

A TCP tree is represented in the memory by an array of $|\Sigma| - 1$ bits of memory holding the flags of the internal nodes, followed by $|\Sigma|$ blocks of $n \log n + n$ bits of memory each, holding the labels of the leaf nodes. The blocks appear in the same order as a breadth-first traversal of $T_{CP}(S)$. Consequently, for every $S \subseteq [n] \times \Sigma$ there is a corresponding binary string of $|\Sigma| - 1 + (n \log n + n)|\Sigma|$ that uniquely encodes $T_{CP}(S)$. Crucially, if $|S| = r$, then at most $O(r(\log|\Sigma| + \log n))$ of these bits are 1. Since $|\Sigma| = L^{O(d)} = \mathsf{poly}(n)$ (recall $d = O(1)$), the encoding is $\widetilde{O}(r)$-sparse.

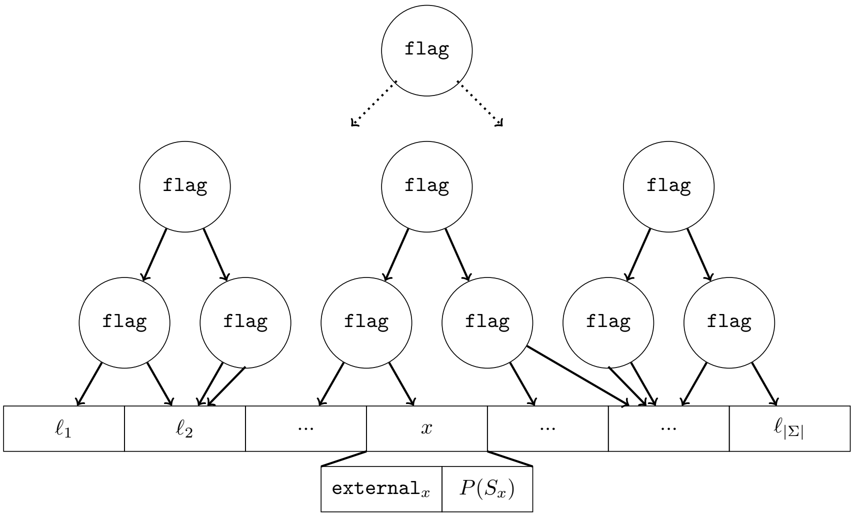

Figure 3 Data structure for the CP problem.

Implementation of data structure operations

It is clear from the definition of TCP tree and its memory representation that a tree $T_{CP}(S)$ represents a set $S \subseteq [n] \times \Sigma$ in a history-independent way. We will now argue that all the required data structure operations take $O(\log n)$ time in the worst case. For every $(i,x) \in [n] \times [L]^d$ there is a corresponding $(i,z) \in [n] \times \Sigma$, with $z = \text{id}(x)$, stored in the data structure.

- **Insertion.** To insert (i,x) in the tree, first go to the memory location corresponding to leaf x. Begin by inserting i in the prefix-sum tree $P(S_x)$. Then three cases arise
  - If $|S_x| = 1$ then for every neighbour y of x with $|S_y| = 1$ do the following: Using the prefix-sum tree at leaf y obtain the only non-zero leaf index j of $P(S_y)$. This operation takes $\log n$ time. Then check if $\Delta(p_i,p_j) \leq \varepsilon$, if yes then increase the values of both $\texttt{external}_x$ and $\texttt{external}_y$ by 1. If this caused $\texttt{external}_y > 0$ then set $\texttt{flag}_w = 1$ for all internal nodes w on the path from leaf y to the root of $T_{CP}(S)$.
    After going over all neighbours, check if $\texttt{external}_x \geq 1$, if it is then set $\texttt{flag}_w = 1$ for all internal nodes w on the path from leaf x to the root of $T_{CP}(S)$. This process takes at most $(2\sqrt{d}+1)^d \log n$ time as there will be at most $(2\sqrt{d}+1)^d$ neighbours, which is $O(\log n)$ for $d = O(1)$.

  - If $|S_x| = 2$ using the prefix-sum tree $P(S_x)$ obtain the only other non-zero leaf index $i' \neq i$ of $P(S_x)$. Then for all neighbours y of x with $|S_y| = 1$ do the following: Using the prefix-sum tree $P(S_y)$ obtain the only non-zero index j of $P(S_y)$. Check if $\Delta(p_{i'}, p_j) \leq \varepsilon$, and if so decrease the value of $\texttt{external}_y$ by 1. If that results in making $\texttt{external}_y = 0$ then set $\texttt{flag}_w = 0$ for the parent of y, unless the other child y' of the parent of y has $|S_{y'}| \geq 2$ or $\texttt{external}_{y'} \geq 1$. Likewise, among all the internal nodes w that are on the path from the root to y's parent, update the $\texttt{flag}_w$ accordingly, i.e., set $\texttt{flag}_w = 1$ if any child u of w has $\texttt{flag}_u = 1$, and otherwise set $\texttt{flag}_w = 0$. Having done that, set $\texttt{external}_x = 0$ and set $\texttt{flag}_w = 1$ for all internal nodes w from leaf x to the root $T_{CP}(S)$. This process also takes $O(\log n)$ time (when d is a constant).
  - If $|S_x| > 2$ then do nothing.
- **Deletion.** The procedure to delete is similar to the insertion procedure.
- **Query.** To check if the tree has a pair $(i, x), (j, y) \in S$ such that $\Delta(p_i, p_j) \leq \varepsilon$, we need only check if $\texttt{flag}_{\text{root}} = 1$, which takes $O(1)$ time.

Runtime, error and memory usage

Using the above data-structure, the runtime of this TCP algorithm is now $\widetilde{O}(n^{\frac{2}{3}})$ time. The total memory used is $\widetilde{O}(n|\Sigma|)$ bits. However, note that at any point of time in any branch of computation this algorithm stores sets of size $r = O(n^{\frac{2}{3}})$. Hence their algorithm with this data structure is a $\widetilde{O}(n^{\frac{2}{3}})$-sparse algorithm. Thus, invoking Theorem 1 we conclude the following.

▶ **Corollary 17.** *There is a bounded-error* QRAM *algorithm that computes TCP in $\widetilde{O}(n^{2/3})$ time using $\widetilde{O}(n^{2/3})$ memory qubits.*

A.3 Fine-Grained Complexity via Quantum Walks

The authors' own paper [4] shows that the quantum 3SUM conjecture, which states that there exists no truly sublinear quantum algorithm for 3SUM, implies several other quantum lower-bounds. The reductions use quantum walks together with complicated space-efficient data structures. We had already realized, when writing the paper, that simple yet space-inefficient data structures could be used instead, and included this observation in the paper, so we will not repeat it here. Section 3.1, with the space inefficient sparse data structures, is 4 pages long, whereas section 3.2, with the complicated space efficient data structures, is 12 pages long.