Tight Bounds on the Spooky Pebble Game: Recycling Qubits with Measurements

Niels Kornerup*[1], Jonathan Sadun, and David Soloveichik*[2]

[1]Department of Computer Science, The University of Texas at Austin, United States of America
[2]Department of Electrical and Computer Engineering, The University of Texas at Austin, United States of America

Pebble games are popular models for analyzing time-space trade-offs. In particular, reversible pebble game strategies are frequently applied in quantum algorithms like Grover's search to efficiently simulate classical computation on inputs in superposition, as unitary operations are fundamentally reversible. However, the reversible pebble game cannot harness the additional computational power granted by intermediate measurements, which are irreversible. The spooky pebble game, which models interleaved Hadamard basis measurements and adaptive phase corrections, reduces the number of qubits beyond what purely reversible approaches can achieve. While the spooky pebble game does not reduce the total space (bits plus qubits) complexity of the simulation, it reduces the amount of space that must be stored in qubits. We prove asymptotically tight trade-offs for the spooky pebble game on a line with any pebble bound. This in turn gives a tight time-qubit tradeoff for simulating arbitrary classical sequential computation when using the spooky pebble game. For example, for all $\varepsilon \in (0, 1]$, any classical computation requiring time T and space S can be implemented on a quantum computer using only $O(T/\varepsilon)$ gates and $O(T^{\varepsilon}S^{1-\varepsilon})$ qubits. This improves on the best known bound for the reversible pebble game with that number of qubits, which uses $O(2^{1/\varepsilon}T)$ gates. For smaller space bounds, we show that the spooky pebble game can simulate arbitrary computation with $O(T^{1+\varepsilon}S^{-\varepsilon}/\varepsilon)$ gates and $O(S/\varepsilon)$ qubits whereas any simulation via the reversible pebble game requires $\Omega(S \cdot (1 + \log(T/S)))$ qubits.

We also consider the spooky pebble game on more general directed acyclic graphs (DAGs), capturing fine-grained data dependency in computation. We show that for an arbitrary DAG even approximating the number of required pebbles in the spooky pebble game is PSPACE-hard. Despite this, we are able to construct a time-efficient strategy for pebbling binary trees that uses the minimum number of pebbles.

1 Introduction

Pebble games provide a convenient abstraction for reasoning about space and time usage in computation. Pebble games were first used in [1] to determine optimal register allocation for computing straight line programs. In [2] the authors applied the irreversible pebble game to show that any computation running on a Turing machine in time $T(n)$ can be executed on another

*Research supported by Schmidt Science Polymath award

Turing machine with space $T(n)/\log T(n)$. Since then it has found uses in establishing time-space trade-offs for computing functions including matrix vector products ([3]) and memory hard functions based on hashing (eg. [4, 5, 6, 7, 8]). In [9], Bennett used a reversible pebble game to give a general mechanism for reversibly simulating irreversible computation. As an example of more recent work, [10] applied the reversible pebble game to analyze the post-quantum security of these memory hard functions.

In [11] Gidney introduced a new *spooky pebble game* to study time-qubit trade-offs in quantum simulation of classical computation on inputs in superposition. Many quantum algorithms use simulation of classical computation in superposition as a subroutine. For example, applying Grover's search [12] to find an x such that $f(x) = 1$, requires a quantum circuit that implements the following mapping:

$$\sum_x \alpha_x |x\rangle |j_x\rangle \to \sum_x \alpha_x |x\rangle |j_x \oplus f(x)\rangle$$

which represents the evaluation of f on inputs in superposition. Since quantum gates are unitary (and therefore reversible) operations, $f(x)$ has traditionally been simulated with a reversible circuit. This comes with a time-space overhead that is often overlooked. The spooky pebble game uses intermediate measurements to make such a simulation more efficient than would be possible using reversible circuits. We expand upon this pebble game and show tight time-space trade-offs for how efficiently it can simulate classical computation.

While the spooky pebble game can reduce the number of qubits needed to perform a computation, it is worth noting that it introduces new classical ancillary bits and does not reduce the total memory (qubits plus classical bits). Nonetheless, qubits are a much more limited resource than classical bits [13]. As such, we believe that this trade-off makes the spooky pebble game pragmatic for designing algorithms that run on quantum computers.

Previous work For any $\varepsilon \in (0, 1]$ the reversible pebble game can be used to reversibly simulate any irreversible computation that runs in time T with S space using $O(T^{1+\varepsilon}S^{-\varepsilon})$ steps and $O(S(1+\log(T/S)))$ space [9, 14]. However, the asymptotic notation above hides a constant factor cost of approximately $\varepsilon 2^{1/\varepsilon}$ in the space term [14].

It was also shown that, using the reversible pebble game, space $O(T^{\varepsilon}S^{1-\varepsilon})$ is sufficient to simulate irreversible computation in linear time, but similarly, this result features a steep but constant $2^{1/\varepsilon}$ cost in the time of the simulation [15]. Král'ovič also showed that any simulation via reversible pebbling the line with $O(S \cdot (1 + \log T/S))$ qubits must use $\Omega(T \cdot (1 + \log T/S))$ steps—a lower bound that is not achieved by any known reversible pebbling algorithm [15]. In [10] the authors used a parallel version of the reversible pebbling game to show (parallel) time-space efficient algorithms for computing cryptographic objects known as data-independent memory hard functions.

Other works have tried to directly improve the qubit efficiency of running classical subroutines on a quantum computer without going through reversible simulation. In [16] the authors showed how a classically controlled quantum Turing machine can simulate a classical Turing machine with no loss in time or space complexity. In [17, 18] the authors showed that one qubit is sufficient to simulate NC^1 in polynomial time.

In [11], Gidney introduced the idea of measurement-based uncomputing with his spooky pebble game. The spooky pebble game extends the reversible pebble game by allowing intermediate measurements that enable irreversible behavior. The pebble game is called spooky because these measurements have a chance to produce undesired phases (or ghosts) that need to be removed before completing the computation; otherwise they will obstruct the desired interference patterns generated by subsequent gates. Gidney showed how the spooky pebble game can be used to simulate any classical computation with only a constant factor blowup in space and a

Source	Game	Pebbles	Steps
	irreversible	2	$O(n)$
[11]	spooky	3	$O(n^2)$
Corollary 3.8	spooky	$2 + 1/\varepsilon$	$O(n^{1+\varepsilon}/\varepsilon)$
[9]	reversible	$O(\varepsilon 2^{1/\varepsilon} \log n)$ [14]	$O(n^{1+\varepsilon})$
[11], Corollary 3.9	spooky	$O(\log n)$	$O(n \log n)$
[15]	reversible	$O(n^{\varepsilon})$	$O(2^{1/\varepsilon} n)$
Corollary 3.7	spooky	$O(n^{\varepsilon})^{\dagger}$	$O(n/\varepsilon)$
Lemma 3.5	spooky	s	$O(mn)^{\ddagger}$

† When $(2/\varepsilon)^{2/\varepsilon} \leq n$
‡ Where $\binom{m+s-2}{s-2} \geq n$

Figure 1: Summary of results on pebbling the line of length n. All asymptotics hold simultaneously for $n \to \infty$ and $\varepsilon \to 0$.

quadratic blowup in the running time, which is impossible for reversible computation [15]. The spooky pebble game was recently extended to work on arbitrary DAGs in [19, 20], similar to the reversible pebble game. These works developed a SAT solver that can compute the minimum runtime necessary to solve the spooky pebble game on an arbitrary DAG. In [20] the authors recently proved that deciding if a DAG can be pebbled with s pebbles is a PSPACE-complete problem, although their hard case requires a DAG with maximum in-degree $s - 1$.

Our results We build on the spooky pebble game framework introduced by Gidney ([11]) and prove asymptotically tight upper and lower bounds on spooky pebbling the line. We do this by proving that—for any pebble bound—there always exists an optimal strategy that has a specific form, constructing such an algorithm, and analyzing a recurrence relation to lower bound the runtime of such algorithms.

We delineate the entire achievable frontier of the spooky pebble game and our tight trade-off bounds can be applied in many different regimes. For example, for any $\varepsilon \in (0, 1]$, any classical computation that runs in T time with S space can be simulated on a quantum computer using only $O(T/\varepsilon)$ steps and $O(T^{1+\varepsilon} S^{1-\varepsilon})$ qubits. This is an exponential improvement in ε over the reversible bound in [15]. We also show that any computation can be simulated in $O(T^{1+\varepsilon} S^{-\varepsilon}/\varepsilon)$ steps with only $O(S/\varepsilon)$ qubits, which is fewer qubit than would be possible (independent of the time bound) with reversible pebbling [21, 22, 15]. Interestingly, when $\varepsilon = 1/\log(T/S)$, this matches the (unobtained) lower bound for reversible pebbling proved in [15].

We then show that any irreversible pebbling can be converted to a spooky pebbling using only one additional pebble. This gives us that, in general, the number of pebbles needed for the spooky pebble game is PSPACE-hard to approximate. Finally we discuss the spooky pebble game on directed acyclic graphs (DAGs) where we show that it is possible to spooky pebble the complete binary tree of height h (i.e., $n = 2^h - 1$ nodes) using $h + 1$ pebbles and $O(n \log n)$ steps. This is less pebbles than is possible in the reversible pebble game addressing an open question posed in [20] regarding efficient algorithms for the spooky pebble game on trees.

2 Preliminaries

Reversible Computation We say that a computation is *reversible* if, after every time step, there is a unique predecessor state for the computation; a Turing machine that moves from left to right inverting the bits on its tape is reversible while one that sets its tape to zero is

not. When a series of quantum transformations are applied to a quantum system, it results in a unitary transformation U being performed on the system. So long as no measurements are taken, the original state of the quantum system can be restored from the final state by applying the inverse $U^\dagger$ [23]. Thus measurement-free quantum computation must be reversible. This means that implementing classical algorithms using a quantum computer often implicitly involves the additional step of making the algorithm reversible.

Reversible computation is also important when considering energy-efficient classical computation. Thermodynamics gives computation an energy lower bound of $kT \ln 2$ per irreversible step—a barrier that can be circumvented by making computation reversible [24, 25]. Unfortunately, all known general techniques for converting irreversible algorithms to reversible ones feature an asymptotic time or space overhead [9, 26, 27, 28]. In fact, relative to random oracles there is a provable separation between time-space trade-offs for reversible and irreversible computation [29].

Of the strategies for making classical computation reversible, the most widely used method is the reversible pebble game [9]. In the reversible pebble game, steps of an irreversible algorithm are simulated reversibly by placing and removing pebbles on a line, or more generally, any directed acyclic graph (DAG). The largest number of pebbles placed at any time corresponds to the space needed for the simulation and the number of steps corresponds to the time of the simulation.

Uncomputation Recycling space is key to space efficiency. But while irreversible algorithms can simply erase values whenever they are no longer needed, such values require more care to dispose of in reversible and quantum computation. In order to erase a value in a reversible manner, it is necessary to end up in a state where it would be possible to efficiently recompute the deleted value. Thus *uncomputation*, the reversible analog of deletion, requires access to the same information needed to compute the deleted value. The importance of uncomputation in quantum circuits it twofold: (1) qubits are an expensive resource for quantum algorithms so it is important to design them so that they can run on as few qubits as possible and (2) failing to uncompute values in a quantum circuit results in entangled garbage that prevents desired interference patterns [30].

We say a quantum circuit *uncomputes* a function f if it can perform the following mapping on any quantum state where the x_i are all distinct:

$$\sum_i \alpha_i |x_i\rangle |f(x_i)\rangle \to \sum_i \alpha_i |x_i\rangle |0\rangle$$

Note that the standard unitary U_f for computing f, which maps $U_f |x\rangle |b\rangle = |x\rangle |b \oplus f(x)\rangle$ is also a unitary that uncomputes f. Carefully balancing computation and uncomputation is vital for designing space and time efficient quantum algorithms.

Ghosting Gidney points out that full uncomputation is not always necessary for recycling qubits when simulating classical algorithms in superposition. Instead of full uncomputation, he developed a clever scheme using intermediate measurements to "compress" qubits into classical bits [11]. We say that a quantum circuit $\mathcal{C}$ *ghosts* a register if it can perform the following mapping on any quantum state where the x_i are all distinct:

$$\sum_i \alpha_i |x_i\rangle |y_i\rangle \xrightarrow{\mathcal{C}} \sum_i \alpha_i (-1)^{b \cdot y_i} |x_i\rangle |0\rangle$$

and b is a classical bitstring returned by the circuit. In [11] Gidney calls this $(-1)^{b \cdot y_i}$ phase a *ghost* of y, as the logical value of this register has been erased and replaced with a phase.

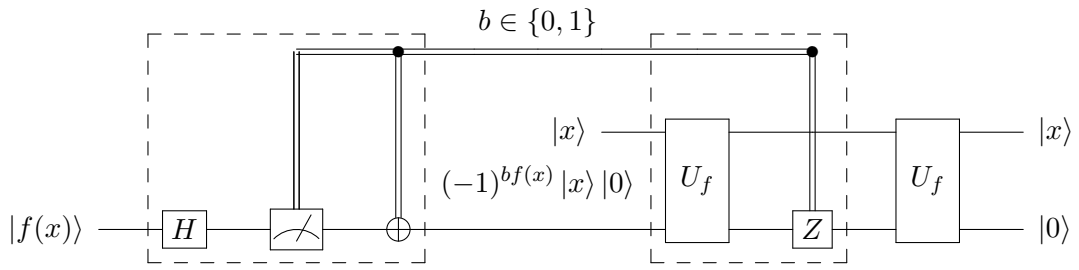

Figure 2: Circuit to ghost $f(x)$ and later correct the phase in order to overall uncompute $f(x)$ using measurement-based uncomputing, as described in [11].

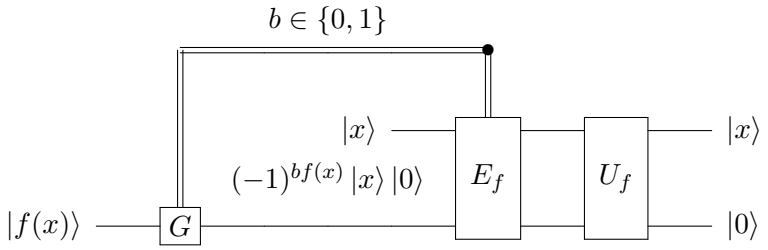

Figure 3: The circuit from Figure 2 using G and E_f circuit macros. Note that the • on the classical wire above the E_f gate represents a controlled application of the component Z gate rather than a controlled application of an entire module.

Since the logical value of the register has been zeroed out, it can be used as if it were a fresh ancilla register for any subsequent computation acting only in the standard basis. However for a quantum circuit to behave correctly, this ghost must be removed before acting on this register in another basis. By recreating the state of the y register as it was before ghosting and recalling the classical bitstring b, the ghost can be removed by applying a b-controlled Z gate to the register. Finally uncomputation can be used to properly uncompute the register. While this might seem like uncomputation with extra steps, we will see that *reusing registers before we could uncompute their values allows for more qubit and gate efficient quantum algorithms.*

Ghosting is not a unitary operation and thus requires intermediate measurements: for example, if you tried to ghost the second qubit in the state $(|00\rangle + |01\rangle)/\sqrt{2}$ and receive $b = 0$, then you would end up with the non-normalized state $(|00\rangle + |00\rangle)/\sqrt{2} = \sqrt{2}\,|00\rangle$. Importantly, note that ghosting is only defined to perform this mapping when the x_i are distinct, as that is sufficient to prevent interference between basis states and make it a norm-preserving operation.

In [11] Gidney shows that ghosting can be performed by measuring in the Hadamard basis, recording the result as classical output b, and then using b to apply a controlled X-gate to the register. Figure 2 shows what this circuit looks like, as the gates on the left "ghost" the $f(x)$ register and the gates on the right recompute $f(x)$ before removing the ghost and uncomputing $f(x)$. Importantly, the gates that ghost the $f(x)$ register do not use the x register, so these operations can be performed even when the x register is holding a different value. However, removing the ghost then requires the presence of x.

For compactness, we compress the highlighted gates on the left and the right of Figure 2 into single gates G and E_f as shown in Figure 3. When performing a classical subroutine on a quantum computer, the G gate can be applied to any register at any time to reset the logical value of its qubits (ghosting). However, whenever this is done, the E_f gate must be reapplied later in the circuit when it is possible to recompute the original value in the erased register and remove the phase (unghosting).

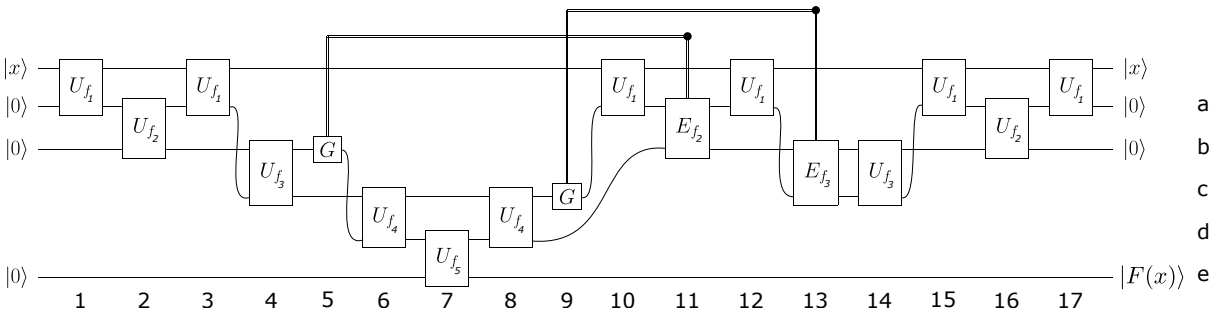

Figure 4: Qubit efficient computation of $F(x) = f_5 \circ f_4 \circ f_3 \circ f_2 \circ f_1(x)$ using ghosting. Again the • on the classical wire above the E_{f_i} gates represents a controlled application of the component Z gate rather than a controlled application of an entire module.

Figure 4 shows how these G and E_f gates can be applied to compute a function with less qubits than would be possible using reversible simulation. This application is general: we can think of a function f as mapping a computer from its current logical configuration to its configuration at the next time step. Since the measured value b must be stored from the time a value is ghosted until the time when its ghost is removed, we are not able to reduce the overall space complexity of the simulation; we are only able to reduce the number of required qubits.

Gidney's original formulation of the spooky pebble game only produced a ghost when the returned value $b \neq 0$ since when $b = 0$ no phase is added [11]. However, we will apply the G gate on groups (registers) of qubits and the probability that all measured qubits yield zero will be negligibly small. As such, we say that a ghost is always created when we apply the G gate.

Pebble games The concept of interweaving computation, uncomputation, and ghosting to efficiently simulate classical computation can be perfectly encapsulated by an appropriate pebble game. In a pebble game we are given a DAG where the nodes represent the variables involved in a computation and the edges show the dependencies for computing these variables. For example the computation $c = a + b$ can be represented with three nodes for a, b, c and edges $a \to c$ and $b \to c$. A pebble can be placed on a node to indicate that the variable is currently stored in a register for the computation. Thus a pebble can be placed on node c only when there are already pebbles on a and b. Additional restrictions on when pebbles can be placed or removed create different pebble games that can be used to simulate irreversible, reversible, or quantum computation with ghosting.

Such graph representations naturally characterize computations such as straight line programs, circuits, and data-independent cryptographic functions acting on words (registers) of size w. In this setting, pebble games directly model time-space trade-offs—as a pebbling of the DAG for a function f with s pebbles placed concurrently and τ steps corresponds to a way to compute f with $O(ws)$ bits in $O(\tau)$ steps.

While pebble games are defined for general DAGs, it is possible to think of each pebble as representing a snapshot of the entire state of an arbitrary sequential computation and then represent the computation as a pebble game on a line. If you can construct a pebbling $\mathcal{P}_n$ that works on the line of length n in τ_n steps and uses at most s_n concurrent pebbles, this leads to a naive way to simulate a computation that requires T time and S space in $O(S \cdot \tau_T)$ steps[1] and $O(S \cdot s_T)$ space (or qubits). A more clever way to apply pebbling to simulation involves assigning S states to each pebble, which lets us pebble a shorter line without increasing the asymptotic space needed per pebble or the time needed per step.

[1]There is a multiplicative term of S here because each state must be copied before simulating the next step.

Proposition 2.1 (Implicit in [9])**.** *Let M be an irreversible sequential machine that computes a function f with T steps using S bits. Let $\mathcal{P}_{T/S}$ be a (spooky) pebbling strategy for the line of length T/S that runs in $\tau_{T/S}$ steps and uses at most $s_{T/S}$ concurrent pebbles. Then there exists a quantum circuit that can compute f with $O(S \cdot \tau_{T/S})$ gates and $O(S \cdot s_{T/S})$ qubits.*

We now more formally define the irreversible, reversible, and spooky pebble games that we will use throughout this paper. Here the $\triangle$ operator denotes the symmetric difference.

Definition 2.2. *The irreversible pebble game is a one player game on a DAG $G = (V, E)$ where the goal is to place a pebble on exactly the nodes $T \subseteq V$ called* targets *with out-degree zero. A pebbling (strategy) is a list of subsets of V. $\mathcal{P} = [\mathcal{P}_0, \ldots, \mathcal{P}_\tau]$ where $\mathcal{P}_0 = \emptyset$ and $\mathcal{P}_\tau = T$. A strategy is valid as long as*

- $\|\mathcal{P}_i \triangle \mathcal{P}_{i+1}\| = 1$, *and*
- *If $v \in \mathcal{P}_{i+1} \setminus \mathcal{P}_i$, then $parents(v) \subseteq \mathcal{P}_i$.*

We can analyze a pebbling by looking at the number of pebbles and steps it requires.

Definition 2.3. *The number of steps $T(\mathcal{P})$ in a pebbling strategy $\mathcal{P} = [\mathcal{P}_0, \ldots, \mathcal{P}_\tau]$ is τ.*

Definition 2.4. *The number of pebbles $S(\mathcal{P})$ in a pebbling strategy $\mathcal{P} = [\mathcal{P}_0, \ldots, \mathcal{P}_\tau]$ is* $\max_{i \in [\tau]} \|\mathcal{P}_i\|$.

When restricted to reversible computation, we can modify Definition 2.2 to get a pebble game that naturally models the restrictions imposed by reversibility.

Definition 2.5. *The reversible pebble game has the same setup as the irreversible pebble game in Definition 2.2. A strategy is valid as long as*

- $\|\mathcal{P}_i \triangle \mathcal{P}_{i+1}\| = 1$,
- *If $v \in \mathcal{P}_{i+1} \setminus \mathcal{P}_i$, then $parents(v) \subseteq \mathcal{P}_i$, and*
- *If $v \in \mathcal{P}_i \setminus \mathcal{P}_{i+1}$, then $parents(v) \subseteq \mathcal{P}_i$.*

The pebble game capturing ghosting admits better pebbling strategies than are possible in the reversible pebble game, assuming we are concerned with the number of qubits rather than the total space complexity. Given an irreversible pebbling, we can exactly characterize when it produces a ghost.

Definition 2.6. *The ghosting sequence $\mathcal{G}(\mathcal{P}) = [\mathcal{G}_0, \ldots, \mathcal{G}_{T(\mathcal{P})}]$ is defined recursively by $\mathcal{G}_0 = \emptyset$ and $\mathcal{G}_{i+1} = (\mathcal{G}_i \cup \{v \in \mathcal{P}_i | parents(v) \not\subseteq \mathcal{P}_i\}) \setminus \mathcal{P}_{i+1}$.*

The ghosting sequence cumulatively tracks the locations where pebbles were removed without access to their parents. These are exactly the steps that prevent a strategy from being a valid strategy for the reversible game; a valid reversible pebble game strategy is exactly a strategy with a ghosting sequence consisting only of empty sets. The spooky pebble game relaxes this condition to only requiring the *last* ghost set to be empty. Note that ghosts can be removed by placing a pebble over them.

Definition 2.7. *The spooky pebble game has the same setup as the irreversible pebble game in Definition 2.2. A strategy is valid as long as*

- $\|\mathcal{P}_i \triangle \mathcal{P}_{i+1}\| = 1$,
- *If $v \in \mathcal{P}_{i+1} \setminus \mathcal{P}_i$, then $parents(v) \subseteq \mathcal{P}_i$, and*
- *In the ghosting sequence $\mathcal{G}(\mathcal{P})$, the final ghost set $\mathcal{G}_\tau = \emptyset$.*

Note that by construction, the spooky pebble game always lies somewhere between the reversible and irreversible pebble games. Therefore it is natural to compare it to these other pebble game and investigate when its behavior is closer to that of reversible or irreversible pebbling.

Figure 5 gives an example of a spooky pebbling on the line of length 5 using only 3 pebbles. Note that this models the same computation performed in Figure 4. Each numbered step in Figure 5 corresponds to the state of the quantum circuit in Figure 4 after the corresponding numbered gate is applied; a wire on row c in Figure 4 occurs exactly at the same numbered steps as a pebble at column c in Figure 5 and corresponds to storing the value $f_3 \circ f_2 \circ f_1(x)$ in quantum memory within both models. In the reversible pebbling game, this task requires 4 pebbles. The ability to ghost pebbles and remove the ghosts at later steps lets us use one fewer pebble, showing that the spooky pebble game can be used to save qubits.

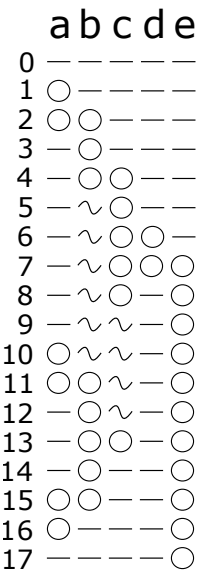

Figure 5: A spooky pebbling of the line. Here ∘ indicates a pebble and ∼ indicates a ghost.

Rather than describe a pebbling as a list of pebbled nodes, we will sometimes describe algorithms that generate these lists. These algorithms feature the functions "place" and "remove". The t'th call to these functions defines the value of $\mathcal{P}_t$ from $\mathcal{P}_{t-1}$ as follows:

- place(v_i): $(\mathcal{P}_t) = (\mathcal{P}_{t-1} \cup \{v_i\})$
- remove(v_i): $(\mathcal{P}_t) = (\mathcal{P}_{t-1} \setminus \{v_i\})$

Intuitively, the place instruction creates a pebble on a node while the remove instruction destroys that pebble. For the spooky pebble game, ghosts are implicitly created by remove instructions according to Definition 2.6.

3 Spooky pebbling the line

Before constructing spooky pebbling algorithms, we first define two useful related pebbling tasks.

Definition 3.1. *An* unpebbling *is a sequence of place and remove instructions that, assuming the graph starts with pebbles exactly on all nodes with out-degree zero (and no ghosts), results in the graph having neither pebbles nor ghosts on any of its nodes.*

An unghosting *is a sequence of place and remove instructions that, assuming the graph starts with ghosts exactly on all target nodes (nodes with out-degree zero) and no pebbles, results in the graph having neither pebbles nor ghosts on any of its nodes.*

While an unghosting assumes that the graph only starts with ghosts on nodes with out-degree zero, these sequence of place and remove instructions will remove any additional ghosts that start elsewhere on the DAG, as pebbles must be placed on all nodes in the graph to remove ghosts on the out-degree zero nodes. In the rest of this section we restrict our attention to the line graph $G = (\{v_1, \ldots, v_n\}, \{(v_1, v_2), \ldots, (v_{n-1}, v_n)\})$. We will canonically refer to $\mathcal{P}$ as a pebbling algorithm, $\mathcal{U}$ as an unpebbling subroutine, and $\mathcal{U}_g$ as an unghosting subroutine.

The spooky pebbling algorithms we discuss in this paper all have a similar recursive form. They start by using a sequence of alternating place and remove instructions to place a pebble on some vertex v_k, leave a pebble on this vertex to recursively pebble the rest of the line, and use an unpebbling subroutine to remove the pebble placed at v_k. In Section 4 we prove the existence of

time-optimal pebbling strategies that employ this structure. The algorithms we discuss in this section are most naturally expressed in the language of unghosting subroutines (where v_n starts as a ghost). These subroutines can be converted into pebbling algorithms using the following lemma:

Lemma 3.2. *Let $\mathcal{U}_g$ be an unghosting algorithm for the line of length n that uses s pebbles and takes τ steps. Then there exists a pebbling algorithm on the line of length $n+1$ that uses $(s+1)$ pebbles and takes at most $(\tau+1)$ steps.*

Proof. We take $\mathcal{U}_g$ and modify it by adding a pebble instruction on v_{n+1} immediately after the first instruction acting on v_n—which must be a pebble instruction. Note that since v_n starts with a ghost, $\mathcal{U}_g$ must place a pebble at v_n during some step. The resulting algorithm is a valid pebbling that uses at most $\tau+1$ steps and $s+1$ pebbles. □

We now show a simple spooky pebbling algorithm that uses a constant number of pebbles to pebble the line.

Proposition 3.3 (from [11])**.** *There exists a spooky unghosting algorithm that can unghost the line graph of length n using $O(n^2)$ steps and only two pebbles.*

Proof. Consider the following algorithm:

$\mathcal{U}_g^{n^2}(\{v_1,\ldots,v_n\})$:
    `place(`v_1`)`
    `if(`$n=1$`):`
        `remove(`v_1`)`
    `else:`
        `for` $i \in [2, n-1]$`:`
            `place(`v_i`)`
            `remove(`v_{i-1}`)`
        `place(`v_n`)`
        `remove(`v_n`)`
        `remove(`v_{n-1}`)`
        `run` $\mathcal{U}_g^{n^2}(\{v_1,\ldots v_{n-1}\})$

This algorithm removes the ghost on v_n, leaving a ghost on v_{n-1}. This process is repeated until the remaining ghost is on v_1, at which point we can remove the pebble without creating a ghost. The number of steps in this pebbling algorithm is given by $T(1)=2$ and $T(k)=2k+2+T(k-1)$. Unrolling this recursion yields $T(n)=n^2+3n-2$ which is $O(n^2)$. □

We can convert $\mathcal{U}$ into a pebbling algorithm for the line of length n with 3 pebbles using Lemma 3.2.

Corollary 3.4 (from [11])**.** *There exists a spooky pebbling algorithm that can pebble the line graph of length n using $O(n^2)$ steps and only three pebbles.*

With the use of additional pebbles, we can create asymptotic improvements in the number of steps used by pebbling algorithms. One additional pebble produces a pebbling that requires only $O(n^{3/2})$ steps. This extra pebble lets us leave a "checkpoint" in our computation that makes removing ghosts possible with fewer steps. However this checkpoint must then be removed in an unpebbling subroutine. This idea can be extended to create a general tradeoff between pebbles and steps.

Lemma 3.5. *For any $s, m \in \mathbb{N}$ such that $n \leq \binom{m+s-1}{s-1}$, there exists an unghosting algorithm for the line of length n that uses at most s pebbles and $O(mn)$ steps.*

Proof. We consider the following unghosting algorithm that uses s pebbles for when the line has length exactly $n = \binom{m+s-1}{s-1}$ for some value of m:

```
U_g({v_1, ..., v_n}, s): //Assume that n = C(m+s-1, s-1) for some value of m.
    if (n ≤ s):
        for i ∈ [1, n]:
            place(v_i)
        for i ∈ [0, n − 1]:
            remove(v_{n−i})
    else:
        let k = C(m+s-2, s-1)
        place(v_1)
        for i ∈ [2, k]:
            place(v_i)
            remove(v_{i−1})
        run U_g({v_{k+1}, ... v_n}, s − 1) // Line of length n − k = C(m+s-2, s-2).
        remove(v_k)
        run U_g({v_1, ..., v_k}, s) // Line of length k = C(m+s-2, s-1).
```

Note that if $m' = m - 1$ then $k = \binom{m'+s-1}{s-1}$, so our assumption on n holds in the recursive cases. While the recursive calls to $\mathcal{U}_g$ may run on lines that contain ghosts on nodes other than the target, removing the ghost at the target will require placing pebbles on all other nodes. This will remove any additional ghosts present at the start of the recursive call. To upper bound the number of steps required to run $\mathcal{U}_g$ on a line of length $n = \binom{m+s-1}{s-1}$ with s pebbles, we will upper bound the number of steps that act on any node and multiply that by the total number of nodes. Let $T_i(m, s)$ be the number of times $\mathcal{U}_g$ acts on node v_i and $T^*(m, s) = \max_i T_i(m, s)$. Then directly from the construction of $\mathcal{U}_g$ we have:

$$T^*(m,s) \leq \begin{cases} 2 & s \geq \binom{m+s-1}{s-1} \\ \max\left(2 + T^*(m-1,s), T^*(m,s-1)\right) & s < \binom{m+s-1}{s-1}. \end{cases}$$

In other words, when $s \geq n$, each node has a pebble placed and removed at most once. Otherwise, when $s < n$, the node visited the most times either is before v_{k+1} and gets a pebble placed and removed and is part of the call $\mathcal{U}_g(\{v_1, \ldots, v_k\}, s)$ or is after k and is part of the call $\mathcal{U}_g(\{v_{k+1}, \ldots v_n\}, s-1)$.

When m or s is 1, we observe that $s \geq \binom{m+s-1}{s-1}$, and so the base case $T^*(1, s) = T^*(m, 1) = 2$. Since every recursive step at most increases T^* by 2 every time that m increases by 1, it follows that $T^*(m, s) \leq 2m$. This lets us conclude that when the line has length $n = \binom{m+s-1}{s-1}$, we can unghost it using $O(mn)$ steps.

When $n < \binom{m+s-1}{s-1}$ we observe that the above unghosting algorithm can be truncated to the shorter line of length n by ignoring steps on nodes after v_n. Again the largest number of steps on any node is at most $2m$ so we can conclude that the unpebbling algorithm takes at most $2mn$ steps. □

We can convert this to a pebbling algorithm with Lemma 3.2.

Theorem 3.6. *For any $s, m \in \mathbb{N}$ such that $n \leq \binom{m+s-2}{s-2} + 1$, there exists a pebbling algorithm for the line of length n that uses at most s pebbles and $O(mn)$ steps.*

Our Theorem 4.13 will show that the above algorithm for spooky pebbling the line is asymptotically optimal for all pebble bounds.

Corollary 3.7. *For any constant $\varepsilon \in (0, 1]$ there exists a spooky pebbling algorithm on any line of length $n \geq \lceil 2/\varepsilon \rceil^{\lceil 2/\varepsilon \rceil}$ that uses $\lceil n^\varepsilon \rceil$ pebbles and $O(n/\varepsilon)$ steps.*

Proof. Let m be the smallest integer such that $n \leq \binom{m+\lceil n^\varepsilon \rceil - 2}{\lceil n^\varepsilon \rceil - 2} + 1$. Applying the algorithm in Theorem 3.6 to the line of length n lets us pebble it with $O(mn)$ steps and $\lceil n^\varepsilon \rceil$ pebbles. We now want to show that when $m' = \lceil 2/\varepsilon \rceil$ and $n \geq \lceil 2/\varepsilon \rceil^{\lceil 2/\varepsilon \rceil}$, $n \leq \binom{m'+\lceil n^\varepsilon \rceil - 2}{\lceil n^\varepsilon \rceil - 2}$. Observe that:

$$\binom{m' + \lceil n^\varepsilon \rceil - 2}{\lceil n^\varepsilon \rceil - 2} \geq \left(\frac{m' + n^\varepsilon - 2}{m'}\right)^{m'} \geq n^2 / \lceil (2/\varepsilon) \rceil^{\lceil 2/\varepsilon \rceil} \geq n$$

By our choice of m we can conclude that $m \leq m'$ and therefore the number of steps for the pebbling algorithm is $O(n/\varepsilon)$. □

Corollary 3.8. *For any $\varepsilon \in (0, 1]$, there exists a spooky pebbling algorithm on the line of length n that uses $O(1/\varepsilon)$ pebbles and $O(n^{1+\varepsilon}/\varepsilon)$ steps.*

Proof. Let $m = \lceil (2n^\varepsilon - 1)/\varepsilon \rceil$ and $s = \lceil 1/\varepsilon \rceil + 2$.

$$\binom{m+s-2}{s-2} + 1 \geq \left(\frac{m+s-2}{s-2}\right)^{s-2} = \left(\frac{\lceil (2n^\varepsilon - 1)/\varepsilon \rceil + \lceil 1/\varepsilon \rceil}{\lceil 1/\varepsilon \rceil}\right)^{\lceil 1/\varepsilon \rceil} \geq \left(\frac{2n^\varepsilon/\varepsilon}{2/\varepsilon}\right)^{\lceil 1/\varepsilon \rceil} \geq n$$

Therefore, by Theorem 3.6, there exists a pebbling algorithm on the line using $s = O(1/\varepsilon)$ pebbles and $O(mn)$ or $O(n^{1+\varepsilon}/\varepsilon)$ steps. □

Corollary 3.9. *There exists a spooky pebbling algorithm on the lines of length n that uses $O(\log n)$ pebbles and $O(n \log n)$ steps.*

Proof. By Corollary 3.8 with $\varepsilon = 1/\log n$. □

These results are summarized in Figure 1.

4 Lower bounds on spooky pebbling the line

We use a divide and conquer approach to describe the general structure of time-optimal spooky pebbling algorithms for a given pebble bound. We then analyze the runtime of such an algorithm to obtain an asymptotically tight lower bound on spooky pebbling the line.

Optimal structured spooky pebbling

Let $\mathcal{P} = [\mathcal{P}_0, \ldots, \mathcal{P}_\tau]$ be an arbitrary spooky pebbling of the line with nodes $v_1, \ldots v_n$. We will show that $\mathcal{P}$ uses at least as many steps as a spooky pebbling algorithm matching $\mathcal{P}'$ in Lemma 4.1.

Lemma 4.1. *Let $\mathcal{P}$ be an arbitrary spooky pebbling of the line using $T(\mathcal{P}) = \tau$ steps and $S(\mathcal{P}) = s$ pebbles. Then there exists k, pebbling algorithm $\mathcal{P}^*$, and unpebbling algorithm $\mathcal{U}^*$ such that the pebbling algorithm $\mathcal{P}'$ described below has $T(\mathcal{P}') \leq \tau$ and $S(\mathcal{P}') \leq s$.*

```
1  P'({v_1, ..., v_n}):
2      place(v_1)
3      for i ∈ [2, k]:
4          place(v_i)
5          remove(v_{i-1})
6      run P*({v_{k+1}, ... v_n})
7      run U*({v_1, ..., v_k})
```

In other words, $\mathcal{P}'$ is the same algorithm as in Lemma 3.5, but with the choice of k determined by the values of n and s in some arbitrary way. We prove Lemma 4.1 over the course of Section 4.

Given a spooky pebbling algorithm and a node v_i, we define a notation for the first and last time that this node contains a pebble:

Definition 4.2. *Let* $\mathit{First}(v_i, \mathcal{P}) = \min(\{t | v_i \in \mathcal{P}_t\})$ *and* $\mathit{Last}(v_i, \mathcal{P}) = \max(\{t | v_i \in \mathcal{P}_t\})$.

Lemma 4.3. $\forall i \in [2, n-1]$, $\mathit{Last}(v_i, \mathcal{P}) < \mathit{Last}(v_{i-1}, \mathcal{P})$.

Proof. Let $t = \text{Last}(v_i, \mathcal{P})$, which implies that there is never a pebble placed on v_i after $\mathcal{P}_t$. By the definition of a spooky pebbling we require that $\mathcal{G}_\tau = \emptyset$, so $v_i \notin \mathcal{G}_{t+1}$ as we never place a pebble on v_i after step t. Since this is a valid pebbling, we can conclude that $v_{i-1} \in \mathcal{P}_t$ and thus $\text{Last}(v_i, \mathcal{P}) < \text{Last}(v_{i-1}, \mathcal{P})$. □

Definition 4.4. *Let* $\mathit{First}(v_n, \mathcal{P}) = t$. *We call* $\mathcal{P}$ sweep-first *if a pebble is placed at* v_n *only once and each* v_i *has at most one pebble placed on it before step* t.

Lemma 4.5. *Let* $G = (V, E)$ *be a line and* $S \subseteq V$ *be such that* $|S| = s$. *Then there exists a valid sequence of pebbling instructions that places pebbles on exactly* S *that is time-optimal and uses at most* $s + 1$ *pebbles.*

Proof. Consider the following pebbling sequence:

```
1  I({v_1, ..., v_n}, S):
2      k = argmax_i v_i ∈ S
3      place(v_1)
4      for i ∈ [2, k]:
5          place(v_i)
6          if(v_{i-1} ∉ S):
7              remove(v_{i-1})
```

The above sequence uses at most $s + 1$ pebbles and takes exactly $2k - s$ steps. Since placing a pebble on v_k requires placing pebbles on each node before it and we want to only end with pebbles on S, any pebbling sequence that completes this task must place at least k pebbles and remove $k - s$ pebbles. Therefore the above pebbling sequence uses the time-optimal number of steps. □

Note that $\mathcal{I}$ only uses s pebbles when $v_{k-1} \in S$. Using the above lemma, we can turn any pebbling into one that is sweep first.

Lemma 4.6. *Let* $\mathcal{P}$ *be any spooky pebbling algorithm where* $T(\mathcal{P}) = \tau$ *and* $S(\mathcal{P}) = s$. *Then there exists a sweep-first pebbling algorithm* $\mathcal{P}'$ *where* $T(\mathcal{P}') \leq \tau$ *and* $S(\mathcal{P}') \leq s$.

Proof. Let t be the last step where $\mathcal{P}$ places a pebble on v_n. We know that $t < \text{Last}(v_{n-1}, \mathcal{P})$. Thus by Lemma 4.3 we know that $t < \text{Last}(v_i, \mathcal{P})$ for all $i < n$. $\mathcal{P}'$ will place pebbles on exactly $\mathcal{P}_t$ in $t' = 2n - \|\mathcal{P}_t\|$ steps by running $\mathcal{I}(\{v_1, \ldots v_n\}, \mathcal{P}_t)$. We know that $t' \leq t$ since each v_i has a

pebble placed and each node not in $\mathcal{P}_t$ must have had a pebble removed before step t in $\mathcal{P}$. We define $\mathcal{P}' = [\mathcal{P}_0, \mathcal{P}'_1, \ldots, \mathcal{P}'_{t'} = \mathcal{P}_t, \mathcal{P}_{t+1}, \ldots, \mathcal{P}_\tau]$. Since $t' \leq t$ we know that $T(\mathcal{P}') \leq \tau$. We can additionally conclude that $S(\mathcal{P}') \leq s$ since $\mathcal{P}_t$ must contain a pebble at v_{n-1} and so $\mathcal{I}$ only uses s pebbles. Note that while $\mathcal{P}$ and $\mathcal{P}'$ may have different ghosting sequences, $\mathcal{P}'$ is still a valid spooky pebbling. This is because $\mathcal{P}$ must place a pebble (and subsequently remove it without creating a ghost) at every location where $\mathcal{P}'$ had a ghost during step t' due to Lemma 4.3. □

Now we are ready to prove Lemma 4.1.

Proof of Lemma 4.1. By Lemma 4.6 we can assume that $\mathcal{P}$ is sweep-first. Without loss of generality, let $\mathcal{P}$ be the algorithm with $T(\mathcal{P}) = \tau$ and $S(\mathcal{P}) = s$ that maximizes over all algorithms the least value of k where $v_k \in \mathcal{P}_{\text{First}(v_n,\mathcal{P})}$. Let $t = \text{First}(v_n, \mathcal{P})$. By our choice of $\mathcal{P}$ and k, for all $i < k$, $\mathcal{P}$ must place and remove a pebble at v_i; thus we can rearrange these steps from $\mathcal{P}$ to match lines 2 through 5 of $\mathcal{P}'$. Since $\mathcal{P}$ is a sweep-first pebbling we know that these are the only instructions acting on $v_1, \ldots, v_k$ before step t. We now partition the remaining instructions of $\mathcal{P}$ into $\mathcal{P}^*$, the set of all instructions acting on nodes $v_{k+1}, \ldots, v_n$, and $\mathcal{U}^*$, the set of all instructions acting on $v_1, \ldots, v_k$ — while maintaining their relative order within $\mathcal{P}$.

We will now show that $\mathcal{P}^*$ is a valid pebbling of $\{v_{k+1}, \ldots, v_n\}$ that uses at most $s-1$ pebbles. Since (i) $\mathcal{P}$ is a valid pebbling, (ii) $\mathcal{P}'$ as constructed above maintains a pebble at v_k during the execution of $\mathcal{P}^*$, and (iii) instructions acting on vertices $v_1, \ldots, v_{k-1}$ cannot change the validity of an instruction on $v_{k+1}, \ldots, v_n$, it follows that $\mathcal{P}^*$ is a valid pebbling. Now assume for the sake of contradiction that $\mathcal{P}^*$ placed s pebbles during some step. Then since $\mathcal{P}$ uses only s pebbles, there must be some first step t' in $\mathcal{P}$ where $\{v_1, \ldots, v_k\} \cap \mathcal{P}_{t'} = \emptyset$ before the s'th pebble is placed on vertices $\{v_{k+1}, \ldots, v_n\}$. Let ℓ be the least value where $v_\ell \in \mathcal{P}_{t'}$. We will construct another sweep-first pebbling algorithm $\mathcal{P}^\perp$ that has $\ell > k$ as the least value where $v_\ell \in \mathcal{P}^\perp_{\text{First}(v_n,\mathcal{P}^\perp)}$ while maintaining $T(\mathcal{P}^\perp) \leq \tau$ and $S(\mathcal{P}^\perp) \leq s$, contradicting our choice of $\mathcal{P}$. Intuitively $\mathcal{P}^\perp$ will behave like $\mathcal{P}$ except that it has a pebble at v_ℓ instead of v_k during step t. $\mathcal{P}^\perp$ has $\mathcal{P}^\perp_{\text{First}(v_n,\mathcal{P}^\perp)} = (\mathcal{P}_t \cup \{v_\ell\}) \setminus \{v_k\}$ and reaches this state by following the first t instructions of $\mathcal{P}$, except that $\mathcal{P}^\perp$ ghosts v_k after a pebble is placed on v_{k+1} and ignores any instruction to remove v_ℓ before placing a pebble on v_n.

After this, $\mathcal{P}^\perp$ continues following the instructions of $\mathcal{P}$ on all vertices excluding $v_k, \ldots v_\ell$ until step t' of $\mathcal{P}$. At this point we know that $\mathcal{P}_{t'}$ has empty intersection with $v_k, \ldots, v_{\ell-1}$ and therefore $\mathcal{P}$ and $\mathcal{P}^\perp$ have pebbles in the same positions and ghosts on the same positions excluding $v_k, \ldots, v_{\ell-1}$. However $\mathcal{P}_{t'}$ has a pebble at v_ℓ and therefore must place pebbles on all of $v_k, \ldots, v_{\ell-1}$. So if $\mathcal{P}^\perp$ copies all steps of $\mathcal{P}$ after step t', it will also remove any ghosts it had in $v_k, \ldots v_{\ell-1}$. Thus $\mathcal{P}^\perp$ is a valid spooky pebbling that uses at most s pebbles and τ steps and violates our choice of $\mathcal{P}$ since $\ell > k$.

We know that $\mathcal{U}^*$ must be a valid unpebbling by the same reasoning as $\mathcal{P}^*$. All that remains is to show that $\mathcal{U}^*$ uses at most $s-1$ pebbles. As $\mathcal{P}$ is a sweep-first algorithm that uses at most s pebbles, we know that for all steps $t' > t$, $\|\mathcal{P}_{t'} \cap \{v_1, \ldots, v_k\}\| \leq s-1$. Thus executing the same instructions in $\mathcal{U}^*$ cannot use more than $s-1$ pebbles and $\mathcal{P}'$ is a valid spooky pebbling that uses no more pebbles or steps than $\mathcal{P}$. □

By Lemma 4.1 we can assume that for any fixed pebble count, there is a minimum step algorithm that matches the structure of $\mathcal{P}'$. By recursively applying this argument, we get a recurrence relation for the minimum number of steps needed to pebble the line of length n using at most s pebbles. In general this recurrence has a tree like structure based on the choice of k at each level. However, the value of k that minimizes the expression is the optimal choice.

Corollary 4.7. *Let $T_P(n,s)$ ($T_{UP}(n,s)$) be the minimum number of steps needed to pebble (unpebble) a line of length n using at most s pebbles. Then:*

$$T_P(n,s) = \begin{cases} 1 & s \geq 1 \text{ and } n = 1 \\ \infty & s < 3 \text{ and } s < n \\ \min_{k \in [1,n)} 2k - 1 + T_{UP}(k, s-1) + T_P(n-k, s-1) & o.w. \end{cases}$$

The ∞ comes from observing that there is no way to spooky pebble a line longer than s with s pebbles when $s < 3$. Unfortunately the above recurrence features a call to T_{UP} and the non-obvious choice of k makes it hard to analyze. Now we establish how the time complexities of pebbling, unpebbling, and unghosting relate so that we can convert Corollary 4.7 into a form that is easier to work with.

Relationships between Pebbling, Unpebbling, and Unghosting

In Section 3, our constructions were more naturally expressed using unghosting rather than pebbling. In order to prove the tightness of our upper bounds, it is convenient to also express our lower bounds in terms of unghosting. For such a bound to be useful, we require a tight relationship between pebbling and unghosting. We have already proven a way to convert an unghosting to a pebbling.

Lemma 3.2. *Let $\mathcal{U}_g$ be an unghosting algorithm for the line of length n that uses s pebbles and takes τ steps. Then there exists a pebbling algorithm on the line of length $n+1$ that uses $(s+1)$ pebbles and takes at most $(\tau+1)$ steps.*

Now we will show how to convert a pebbling into an unghosting.

Lemma 4.8. *Let $\mathcal{P}$ be a pebbling algorithm for the line of length $n+1$ that uses $s+1$ pebbles and takes $\tau+1$ steps. Then there exists an unghosting algorithm on the line of length n that uses s pebbles and takes at most τ steps.*

Proof. Let t be the the last step of $\mathcal{P}$ operating on v_{n+1}. This must be a pebbling step, so both v_n and v_{n+1} are in $\mathcal{P}_t$. Divide $\mathcal{P}$ into two components around this point, $\mathcal{P}_{\text{pre}}$ for the first t steps and $\mathcal{P}_{\text{post}}$ for the remaining steps. Since $\mathcal{P}_{\text{pre}}$ must have spent at least one step on pebbling each of v_n and v_{n+1}, there must be some algorithm $\mathcal{P}^*_{\text{pre}}$ running in at most $t-2$ steps that ends in $\mathcal{P}_t \setminus \{v_n, v_{n+1}\}$. Moreover, by Lemma 4.5, $\mathcal{P}^*_{\text{pre}}$ may be constructed to require at most $|\mathcal{P}_t \setminus \{v_n, v_{n+1}\}| + 1 \leq s$ pebbles.

Let $\mathcal{P}^*_{\text{post}}$ be a sequence of pebbling instructions derived from $\mathcal{P}_{\text{post}}$ by removing any instructions operating on v_n before $\text{Last}(v_n, \mathcal{P}_{\text{post}})$, then inserting an instruction placing a pebble on v_n immediately before that last removal of v_n. This is valid, as the new placement occurs immediately before a removal (so the predecessor of v_n is pebbled), and no operations occur on v_{n+1} that would be affected by changing whether a pebble is present on v_n.

$\mathcal{P}^*_{\text{pre}}$ followed by $\mathcal{P}^*_{\text{post}}$ is then a valid unghosting of the line graph up to v_n, excluding v_{n+1}. Since $\mathcal{P}^*_{\text{pre}}$ is at least 2 steps shorter than $\mathcal{P}_{\text{pre}}$ and $\mathcal{P}^*_{\text{post}}$ is at most 1 step longer than $\mathcal{P}_{\text{post}}$, the composition takes at most τ steps. Moreover, the composition requires at most s pebbles, $\mathcal{P}^*_{\text{pre}}$ by Lemma 4.5 and $\mathcal{P}^*_{\text{post}}$ because it operates as in $\mathcal{P}_{\text{post}}$ but without an extra pebble sitting on v_{n+1}. □

Combining the results of Lemma 4.8 and Lemma 3.2 we show a tight relationship between unghosting and pebbling. We also characterize the relationship between unghosting and unpebbling.

Lemma 4.9. *Let $\mathcal{U}_g$ be an unghosting algorithm for the line of length n that uses s pebbles and takes τ steps. Then there exists an unpebbling algorithm for the line of length n that uses s pebbles and takes $\tau + 1$ steps.*

Proof. Define $\mathcal{U}_p$ by first ghosting v_n, then running $\mathcal{U}_g$. □

Lemma 4.10. *Let $\mathcal{U}_p$ be an unpebbling algorithm for the line of length n that uses s pebbles and takes τ steps. Then there exists an unghosting algorithm for the line of length n that uses s pebbles and takes $\tau + 1$ steps.*

Proof. Define $\mathcal{U}_g$ by running $\mathcal{U}_p$, but before $\text{Last}(v_n, \mathcal{U}_p)$, skip all preexisting operations on v_n, and immediately before $\text{Last}(v_n, \mathcal{U}_p)$, add a new step to place a pebble on v_n. $\mathcal{U}_g$ only differs from $\mathcal{U}_p$ by sometimes not having a pebble on v_n when $\mathcal{U}_p$ does, so it uses at most as many pebbles, and it inserts only a single new step. □

Theorem 4.11. $T_P(n+1, s+1) = T_{UG}(n, s) + 1$. $T_{UG}(n, s) = T_{UP}(n, s) \pm 1$.

Lower bounds on the number of steps for pebbling

Now that we have established the relationship between pebbling, unpebbling, and unghosting, we can lower bound the number of steps needed to perform these tasks. While pebbling is the most natural quantity to evaluate, it turns out that unghosting has a much cleaner analysis than pebbling.

We can now restate Corollary 4.7 in the language of unghosting.

Corollary 4.12. *Let $T_{UG}(n, s)$ ($T_{UP}(n, s)$) be the minimum number of steps needed to unghost (unpebble) a line of length n using at most s pebbles. Then:*

$$T_{UG}(n, s) = \begin{cases} 2 & s \geq 1 \text{ and } n = 1 \\ \infty & s < 2 \text{ and } s < n \\ \min_{k \in [1,n]} 2k - 1 + T_{UP}(k, s) + T_{UG}(n - k, s - 1) & \text{o.w.} \end{cases}$$

Note that we cannot apply Theorem 4.11 to the base case in Corollary 4.7. Instead the base case in the above recurrence can be verified by inspection. We can then use Theorem 4.11 to convert our lower bound on unghosting to pebbling. By Theorem 4.11 we know that $T_{UP}(k, s) \geq T_{UG}(k, s) - 1$. We therefore define the following recurrence relation that is a lower bound on $T_{UG}(n, s)$:

$$T(n, s) = \begin{cases} 2 & s \geq 1 \text{ and } n = 1 \\ \infty & s < 2 \text{ and } s < n \\ \min_{k \in [1,n]} 2k - 2 + T(k, s) + T(n - k, s - 1) & \text{o.w.} \end{cases} \quad (1)$$

While $T(n, s)$ is not exactly the minimum number of steps needed to unghost the line of length n with s pebbles, it is a sufficiently tight lower bound. A function similar to $T(n, s)$ appears in [31, 32] as the solution to the backtracking problem in dynamic programming. They provide efficient algorithms that can be used to compute the optimal value of k in each recursive call. Another similar recurrence appears in the problem of sequence reversing, where there is a proven lower bound similar to the one we present here [33, 34]. In the rest of this section we use this recurrence relation to prove that unghosting with at most s pebbles requires $\Omega(mn)$ steps which is sufficient to obtain the following theorem.

Theorem 4.13. *For any line of length n, any pebble bound s, and any positive integer m where $n \geq \binom{m+s-2}{s-2} + 1$, a spooky pebbling of that line with at most s pebbles uses at least $\Omega(mn)$ steps.*

One difficulty in analyzing our recurrence relation is that the choice of k that minimizes this expression is not obvious, making it difficult to directly compute this function. Instead we will describe a family of related recurrence relations for a fixed choice of n and s using trees with n leaf nodes and right-depth $s-1$ to determine the choices of k. The lowest valued recurrence in this family gives us the value of $T(n, s)$.

Definition 4.14. *Let $\mathcal{T}$ be a binary tree, $\mathcal{T}.l$ be the left subtree, and $\mathcal{T}.r$ be the right subtree. Let $|\mathcal{T}|_L$ be the number of leaves in a given binary tree. If $|\mathcal{T}|_L = n$ then we can define the recurrence relation:*

$$T_{\mathcal{T}}(n,s) = \begin{cases} 2 & s \geq 1 \text{ and } n = 1 \\ \infty & s < 2 \text{ and } s < n \\ 2|\mathcal{T}.l|_L - 2 + T_{\mathcal{T}.l}(|\mathcal{T}.l|_L, s) + T_{\mathcal{T}.r}(|\mathcal{T}.r|_L, s-1) & o.w. \end{cases} \tag{2}$$

We can think of this as being Equation (1) except that k is selected as the number of leaves in the left sub-tree rather than the value that minimizes the expression. This lets us fix n and s in order to see how different choices of k lead to different values for the recurrence. Then the minimum of $T_{\mathcal{T}}(n, s)$ over all trees must equal to $T(n, s)$. Instead of directly computing the recurrence, we can compute $T_{\mathcal{T}}(n, s)$ for a specific tree $\mathcal{T}$ by assigning a cost function to binary trees that is equal to $T_{\mathcal{T}}(n, s)$.

Definition 4.15. *Let N be any node in a tree. We let $R(N)$ denote the right-depth of N, which is equal to the number of times you must take right children to reach N from the root. We likewise define $L(N)$ as the left-depth of N.*

Definition 4.16. *For any depth bound s, the* cost *of a binary tree $C_s(\mathcal{T})$ is equal to the sum of the costs of its nodes. Each non-leaf node of a tree has cost -2. Any leaf node N where $R(N) < s$ has a cost of $2L(N) + 2$ while a leaf node where $R(N) \geq s$ has a cost of ∞.*

Lemma 4.17. *The cost of the tree $C_s(\mathcal{T})$ is equal to $T_{\mathcal{T}}(|\mathcal{T}|_L, s)$.*

Proof. We prove this by structural induction on the tree. In the base case we have a tree that is a single leaf. Then $C_s(\mathcal{T}) = T_{\mathcal{T}}(n, s) = 2$ as desired. Now consider an arbitrary tree $\mathcal{T}$. By the inductive hypothesis we can assume that $C_s(\mathcal{T}.l) = T_{\mathcal{T}.l}(|\mathcal{T}.l|_L, s)$ and $C_{s-1}(\mathcal{T}.r) = T_{\mathcal{T}.r}(|\mathcal{T}.r|_L, s-1)$. Given that the cost of a tree is the sum of the costs of its nodes, $C_s(\mathcal{T}) = 2|\mathcal{T}.l|_L - 2 + C_s(\mathcal{T}.l) + C_{s-1}(\mathcal{T}.r)$, as all nodes in $\mathcal{T}.l$ have their left depth increased by one, all nodes in $\mathcal{T}.r$ has their right depth increased by one, and the root of $\mathcal{T}$ has a cost of -2. By our inductive hypothesis, this implies that

$$C_s(\mathcal{T}) = 2|\mathcal{T}.l|_L - 2 + T_{\mathcal{T}.l}(|\mathcal{T}.l|_L, s) + T_{\mathcal{T}.r}(|\mathcal{T}.r|_L, s-1) = T_{\mathcal{T}}(|\mathcal{T}|_L, s).$$

If we have a tree with a leaf node where $R(N) \geq s$ then the tree has a cost of infinity and $T_{\mathcal{T}}(n, s)$ must reach the base case where $s \leq 1$ and $s < n$ when evaluating that node. □

Corollary 4.18. *Let $T_{(n,s)}$ be the tree with n leaf nodes that minimizes $C_s(T_{(n,s)})$. Then $T(n, s) = C_s(T_{(n,s)})$.*

It is easier to reason about $C_s(T_{(n,s)})$ than $T(n, s)$ as we can start with an arbitrary tree with n leaf nodes and show how it would be possible to mutate that tree and reduce its cost without computing the full value of the recurrence relation.

Now that we have established the equivalence between the cost of a tree $T_{(n,s)}$ and $T(n,s)$, we want to prove some things about the structure of this tree. When $s < 2$ and $s < n$ we know that $T(n,s) = \infty$, so whenever possible a tree will never have a sub-tree with such parameters as nodes. This means that any tree $T_{(n,2)}$ must have $T_{(1,1)}$ as its right child. By construction, this means that the right-depth of any node in the tree $T_{(n,s)}$ is at most $s-1$. Note that the cost of all the intermediate nodes is only a function of the total number of nodes in the tree.

What follows is a collection of technical lemmas that give the number of nodes in $T_{(n,s)}$ with specific left and right depths. Together they let us characterize the number of leaves in $T_{(n,s)}$ with each cost. This lets us construct a lower bound on the cost of $T_{(n,s)}$, which in turn is a lower bound on the total number of steps needed to unghost a line of length n using at most s pebbles.

Lemma 4.19. *Let $T_{(n,s)}$ be the tree in Corollary 4.18. Let X be a leaf of $T_{(n,s)}$ with the largest value for $L(X)$, $m = L(X)$, and X' be any other leaf where $L(X') < m-1$. Then $R(X') = s-1$.*

Proof. Assume that $R(X') < s-1$. Note that X must be a left child, as it is the node with largest left depth. Then we could replace the parent of X with its other child and replace X' with an intermediate node whose left child is X and right child is X'. Doing so must reduce the left depth of X by at least one and does not change the left depth of any other node in the tree, so the new tree has a lower cost. But this is a contradiction since $T_{(n,s)}$ is supposed to be the tree with the lowest cost. □

The above lemma tells us a surprising amount about the structure of $T_{(n,s)}$. Since all nodes X where $L(x) < m-1$ and $R(X) < s-1$ cannot be leaves, these nodes all have left and right children. We can therefore derive exactly how many leaves $T_{(n,s)}$ must have with each left depth less than m.

Lemma 4.20. *Let $T_{(n,s)}$ be the tree in Corollary 4.18 and m be the largest value of $L(N)$ for $N \in T_{(n,s)}$. If $m > 1$, then we know that the number of leaf nodes $T_{(n,s)}$ possesses with left depth $L < m$ is $\binom{L+s-2}{s-2}$.*

Proof. By Lemma 4.19 we know that no node with a left depth less than $m-1$ and a right depth less than $s-1$ can be a leaf. Thus all such nodes must have left and right children. We will identify each node with a string of l's and r's indicating which directions are taken from the root of the tree to arrive at that node. Since all nodes with right depth s or larger have cost infinity, we know that all nodes with right depth $s-1$ must be leaf nodes. Thus each leaf node with left depth less than $m-1$ must be a right child (otherwise, their sibling would have right depth s) and be identified with a string ending in an r. There are $\binom{L+s-2}{s-2}$ strings containing L copies of l and $s-1$ copies of r that end in an r. This exactly corresponds to the number of leaf nodes with left depth $L < m-1$; each of these leaf nodes must exist in the tree because none of their ancestors can be leaves by Lemma 4.19, and there can be no other leaves with this left depth unless there is a leaf with cost ∞.

Since, by Lemma 4.19, all nodes with left depth $L = m-2$ and right depth $R < s-1$ must exist in the tree and cannot be leaves, each of these nodes has a left child. There is a bijection between the number of leaves with left depth $L = m-1$ and these children, as their rightmost descendants are exactly the leaf nodes with left depth $m-1$. Since there are $\binom{m+R-2}{R}$ nodes with left depth $m-2$ and right depth $R < s-1$, there are

$$\sum_{R=0}^{s-2} \binom{m+R-2}{R} = \sum_{R=0}^{s-2} \binom{m+R-2}{m-2} = \binom{m+s-3}{m-1} = \binom{L+s-2}{s-2}$$

leaf nodes with left depth $m-1$. □

Lemma 4.21. *Let $T_{(n,s)}$ be the tree in Corollary 4.18 and m be the largest left depth of any node in $T_{(n,s)}$ and m' be the largest left-depth of any node in $T_{(n',s)}$. If $m' > m$ then $n' > n$.*

Proof. Assume that $n' \leq n$. By Lemma 4.20 for all $L < m$, $T_{(n,s)}$ and $T_{(n',s)}$ must have the same number of leaf nodes with left depth L. This means that $T_{(n,s)}$ must have at least $n - n'$ more leaves with left depth m than $T_{(n',s)}$ has with left depth at least m. We construct a new tree $T^*_{(n',s)}$ by taking $T_{(n,s)}$ and removing $n - n'$ leaves with left depth m by replacing their parent with its right child. The tree $T^*_{(n',s)}$ represents a valid binary tree and its cost is less than that of $T_{(n',s)}$ since the two trees have the same total number of nodes, the same number of leaves with any left depth less than m, and $T^*_{(n',s)}$ has no leaves with left depth larger than m. This implies that $T_{(n',s)}$ was not constructed according to Corollary 4.18, giving us a contradiction. □

Lemma 4.22. *Let $T_{(n,s)}$ be the tree in Corollary 4.18 and n be the smallest value such that $T_{(n,s)}$ contains a node with left depth $m+1$. Then $T_{(n-1,s)}$ contains exactly $\binom{m+s-2}{s-2}$ leaves with left depth m and $n = 1 + \binom{m+s-1}{s-1}$.*

Proof. Assume for the sake of contradiction that $T_{(n,s)}$ contained at least two nodes with left depth $m+1$. By our choice of n we know that $T_{(n-1,s)}$ contains no nodes with left depth $m+1$. By Lemma 4.20 we know that $T_{(n,s)}$ and $T_{(n-1,s)}$ have the same number of leaves with each left depth less than m. This means the difference in cost between $T_{(n,s)}$ and $T_{(n-1,s)}$ must be at least m.[2] But there is another tree $T^*_{(n,s)}$ that costs only $m-1$ more than $T_{(n-1,s)}$ constructed by replacing the leftmost leaf of $T_{(n-1,s)}$ with an intermediate node whose children are both leaves. Since $T^*_{(n,s)}$ has cost less than $T_{(n,s)}$, we get a contradiction.

Now $T_{(n,s)}$ must contain exactly one node with left depth $m+1$. By Lemma 4.20 this means that it contains exactly $\binom{L+s-2}{s-2}$ leaves with left depth L, so the total number of leaves in $T_{(n,s)}$ is:

$$\begin{aligned} n &= 1 + \sum_{L=0}^{m} \binom{L+s-2}{s-2} \\ &= 1 + \binom{m+s-1}{s-1} \end{aligned}$$

This means that $T_{(n-1,s)}$ must have $\binom{m+s-1}{s-1}$ leaves. All of these leaves have left depth at most m by Lemma 4.21. So Lemma 4.20 gives us that $T_{(n-1,s)}$ has:

$$\begin{aligned} &\binom{m+s-1}{s-1} - \left(\sum_{L=0}^{m-1} \binom{L+s-2}{s-2}\right) \\ &= \binom{m+s-1}{s-1} - \binom{m+s-2}{s-1} \\ &= \binom{m+s-2}{s-2} \end{aligned}$$

leaves with left depth m. □

Lemma 4.23. *Let $T_{(n,s)}$ be the tree in Corollary 4.18 and $\binom{m+s-1}{s-1} \leq n$. Then the cost of $T_{(n,s)}$ is $\Omega(mn)$.*

[2] $T_{(n,s)}$ must have two leaves with left depth $m+1$ while $T_{(n-1,s)}$ has one fewer node that must have left depth m. Since $T_{(n,s)}$ has one more intermediate node, its cost is at least m more than $T_{(n-1,s)}$.

Proof. By Lemma 4.20, Lemma 4.21, and Lemma 4.22, we know that for all $L \in \{0, 1, 2, \ldots, m\}$ the number of leaf nodes with left depth L is $\binom{L+s-2}{s-2}$. Thus the cost of these leaf nodes is at least:

$$\sum_{L=0}^{m} (2L+2)\binom{L+s-2}{s-2}.$$

Expanding the above summation yields:

$$\frac{2(m+1)(m(s-1)+s)}{s(s-1)}\binom{m+s-1}{s-2}.$$

Using that $\binom{n}{k-1} = \frac{k}{n-k+1}\binom{n}{k}$ and $s \geq 2$, this is at least:

$$\begin{aligned}\frac{2(m(s-1)+s)}{s}\binom{m+s-1}{s-1} &= 2\left(\frac{m(s-1)}{s}+1\right)\binom{m+s-1}{s-1} \\ &\geq (m+2)\binom{m+s-1}{s-1}.\end{aligned}$$

By Lemma 4.20 and Lemma 4.22, if $n = \binom{m+s-1}{s-1} + k$ then the remaining k leaves must all have left depth larger than m. Thus the total cost of the leaves is at least:

$$(m+2)\binom{m+s-1}{s-1} + 2(m+1)k \geq (m+2)n.$$

There are $n-1$ intermediate nodes with cost -2 so the total cost of the tree must be at least:

$$(m+2)n - 2(n-1) = mn - 2.$$

Which is $\Omega(mn)$ as desired. □

By applying Lemma 4.23 to lower bound the cost of trees and Lemma 4.17 to apply this lower bound to Equation (1), we obtain the following corollary:

Corollary 4.24. *The number of steps needed to unghost a line of length $n \geq \binom{m+s-1}{s-1}$ with s pebbles is $\Omega(mn)$.*

By applying Theorem 4.11 to Corollary 4.24, we get Theorem 4.13.

5 Spooky pebbling beyond the line

In Section 3 and Section 4 we discussed how the spooky pebble game can be applied to simulate arbitrary irreversible sequential computation on a quantum computer. While this gives us very general results, additional structure on the classical computation can lead to more efficient simulation. Here we consider computation that can be expressed using a dependency graph—such as boolean circuits, straight line (or oblivious) programs, and data-independent memory hard functions (e.g. [35, 36, 37, 3, 38, 7, 39]). When considering a pebbling of a DAG, each pebble represents only a single word rather than an entire copy of the computation's state. It turns out that there is a close relationship between the number of pebbles required in the irreversible and spooky pebble games.

Spooky pebbling from irreversible pebbling

We start by showing a general strategy that can be used to convert any irreversible pebbling into a spooky pebbling that uses at most one additional pebble. This result is a special case of Theorem 1 in [19, 20] which presents a generalization of this argument for DAGs with multiple target nodes (nodes with out-degree zero). For completeness we show the proof below.

Lemma 5.1. *If a DAG G with n nodes and one target can be pebbled in the irreversible pebble game using T steps and s pebbles, then it can be spooky pebbled using $O(nT)$ steps and at most $s+1$ pebbles.*

Proof. Let $\mathcal{P}$ be an irreversible pebbling of G that uses s pebbles. We will construct the following spooky pebbling $\mathcal{P}_s$ that uses at most $s+1$ pebbles. $\mathcal{P}_s$ first simulates $\mathcal{P}$, which since we are operating in the spooky pebble game, may leave some ghosts behind. Additionally the first time that $\mathcal{P}_s$ places a pebble at any node v_i, we push v_i onto an initially empty stack. Once $\mathcal{P}_s$ has placed a pebble at the target node v_t, it removes all other pebbles (leaving a large number of ghosts) and repeats the following procedure until the stack is empty: We pop the top element off of our stack and call it v_g. If v_g does not contain a ghost, we continue to the next element in the stack. Once we find a v_g that contains a ghost, we have $\mathcal{P}_s$ simulate $\mathcal{P}$ until v_g is pebbled again, leaving ghosts on some nodes. $\mathcal{P}_s$ then removes the pebble at v_g (without creating a ghost) and then removes all pebbles other than v_t, possibly addling ghosts. By our choice of v_g, future iterations of this procedure cannot result in a ghost on a prior v_g, so this procedure terminates in less than n iterations with pebbles only on v_t and no ghosts. Since $\mathcal{P}_s$ is replaying steps from $\mathcal{P}$ with at most one additional pebble on v_t, it will use at most $s+1$ pebbles. □

Note that since T is without loss of generality $\Omega(n)$, we can say that the above procedure also runs in time $O(T^2)$. Since any spooky pebbling is also an irreversible pebbling, the above lemma is sufficient to give us a nearly tight two sided bound on the minimum number of pebbles needed in the spooky pebble game.

Corollary 5.2. *Let G be a DAG that requires s pebbles in the irreversible pebble game. The fewest pebbles needed for the spooky pebble game on G is either s or $s+1$.*

Demaine and Liu proved that determining the fewest pebbles needed in the irreversible pebble game is PSPACE-hard to approximate.

Proposition 5.3 (Theorem 1[3] in [40])**.** *The minimum number of pebbles needed in the irreversible pebble game on DAGs with n nodes, a single target, and maximum in-degree 2 is PSPACE-hard to approximate to within an additive $n^{1/3-\varepsilon}$ for any $\varepsilon > 0$.*

By combining Proposition 5.3 with Corollary 5.2, we see that finding even approximate minimum pebble spooky pebbling algorithms for arbitrary DAGs solves a PSPACE-hard problem. Thus unless P = PSPACE, there can be no polynomial time algorithm that generates minimum pebble spooky pebbling strategies on arbitrary DAGs. Quist and Laarman showed the PSPACE-completeness of this problem in [20] through a different reduction; however, going through Proposition 5.3 extends this result to a restricted subset of DAGs (those with maximum in-degree two with a single sink node) and shows that even approximating the minimum number of pebbles to an additive factor of $n^{1/3-\varepsilon}$ is PSPACE-hard.

Theorem 5.4. *The minimum number of pebbles needed in the spooky pebble game on DAGs with n nodes, a single sink, and maximum in-degree 2 is PSPACE-hard to approximate.*

[3] Although Theorem 1 as stated in [40] does not refer to the number of targets, the full construction presented in section 2.3.2 of [41] is a DAG with a single target node.

Spooky pebbling the binary tree graph

While finding the minimum number of pebbles needed in the spooky pebble game is PSPACE-hard to approximate in general, it is possible to find pebble and step efficient strategies for specific graph topologies. The binary tree with directed edges from the leaves to the root is a natural topology for algorithms like divide and conquer, where a problem is solved by combining the results of two smaller sub-problems. Concrete algorithms with this structure include floating point pairwise summation and computing Merkle trees [42, 43].

It is known that $h+1$ pebbles and $\Theta(n)$ steps are both necessary and sufficient to pebble a binary tree of height h with $n = 2^h - 1$ nodes in the irreversible pebble game [44]. In [15], Král'ovič showed that $(h + \Theta(\log^* h))$ pebbles are necessary and sufficient this task in the reversible pebble game, where $\log^*$ is the iterated logarithm function. Despite this there are no known polynomial time pebbling strategies for the binary tree using the optimal number of pebbles; the best known tradeoff is a strategy that uses $(1+\varepsilon)h$ pebbles and runs in $n^{O(\log 1/\varepsilon)}$ steps [45]. We show the existence of a spooky pebbling strategy for the binary tree that uses the same number of pebbles as the most efficient irreversible pebbling while only being a $O(\log n)$ factor slower. Thus the spooky pebble game is almost as efficient as the irreversible pebble game and is provably more efficient than the reversible pebble game on binary trees.

Theorem 5.5. *There exists a spooky pebbling algorithm on the complete binary tree with $n = 2^h - 1$ nodes that uses $h+1$ pebbles and $O(n \log n)$ steps.*

Proof. Before getting to the algorithm, we present the following subroutine:

```
1  fast_pebble(T, h): // T is the root and h is the height of the tree
2      if(h = 1):
3          place(T)
4      else:
5          run fast_pebble(T.l, h − 1)
6          run fast_pebble(T.r, h − 1)
7          place(T)
8          remove(T.l)
9          remove(T.r)
```

The above subroutine uses at most $h+1$ pebbles and $2^{h+1} - 1$ steps to place a pebble at the root of the tree while leaving ghosts at all other internal nodes. We will now use fast_pebble to construct a recursive unpebbling algorithm for a complete binary tree with root $\mathcal{T}$ and depth h:

```
1  U(T, h) :
2      remove(T)
3      run fast_pebble(T.l, h − 1)
4      run fast_pebble(T.r, h − 1)
5      place(T)
6      remove(T)
7      remove(T.r)
8      run U(T.l, h − 1)
9      run U(T.r, h − 1)
```

We note that $\mathcal{U}(\mathcal{T}, h)$ uses at most $h+1$ pebbles; we do not need to remove $\mathcal{T}.l$ before step 8 as the pebble on this nodes is removed at the beginning of the recursive call. The number of steps this algorithm takes follows the recurrence relation:

$$T(h) \leq 2T(h-1) + 2^{h+2} + 2$$

Unrolling the recurrence gives us that $T(h)$ is $O(h2^h)$, which in turn is $O(n \log n)$. We then note that $\mathcal{U}$ can be converted into a pebbling algorithm $\mathcal{P}$ by removing step 6. Doing this does not change the number of pebbles used and reduces the total number of steps by one, so $\mathcal{P}$ also uses at most $h + 1$ pebbles and only $O(n \log n)$ steps. □

6 Future work

As the main result of this paper, we have shown asymptotically tight upper and lower bounds for the spooky pebble game when played on the line graph. Going beyond the line graph presents significant challenges as the number of pebbles needed in the spooky pebble game to pebble an arbitrary DAG is PSPACE-hard to approximate. Despite this, for the binary tree graph with height h and $n = 2^h - 1$ nodes we were able to construct a time-efficient (i.e. $O(n \log n)$) pebbling strategy that uses the optimal number of pebbles (i.e. $h+1$). The time optimality of this algorithm is unknown and it would be interesting to construct a more time-efficient algorithm for the binary tree or prove it is impossible. On the other extreme there is a straightforward spooky pebbling algorithm that uses n pebbles and only $O(n)$ steps. Is there a general tradeoff between the number of pebbles and the number of steps for the binary tree similar to our results for the line graph? Beyond binary trees we think it would be interesting to explore strategies for the spooky pebble game on other graph topologies like butterfly graphs used in computing the DFT [3], and those used in the construction of data-independent memory-hard functions [38, 7, 39]. Finally, the spooky pebble game is but one way to use intermediate measurements to improve the time-space efficiency of quantum algorithms. Whether other methods of using intermediate measurements could yield better time and space efficiency tradeoffs remains open.

7 Acknowledgments

We would like to thank Craig Gidney for insightful discourse regarding his blog post on the spooky pebble game as well as Eddie Schoute and Patrick Rall for discussions about potential applications of the spooky pebble game. Additionally we would like to thank the anonymous reviewers of our paper for taking the time to review our manuscript. We sincerely appreciate their comments and suggestions, which improved the quality of this manuscript.

References

[1] Ravi Sethi. "Complete register allocation problems". In Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (STOC 1973). Page 182–195. Association for Computing Machinery (1973).

[2] John Hopcroft, Wolfgang Paul, and Leslie Valiant. "On time versus space". J. ACM **24**, 332–337 (1977).

[3] Martin Tompa. "Time-space tradeoffs for computing functions, using connectivity properties of their circuits". Journal of Computer and System Sciences **20**, 118–132 (1980).

[4] Wolfgang J. Paul, Robert Endre Tarjan, and James R. Celoni. "Space bounds for a game on graphs". In Proceedings of the Eighth Annual ACM Symposium on Theory of Computing (STOC 1976). Page 149–160. Association for Computing Machinery (1976).

[5] Thomas Lengauer and Robert E. Tarjan. "Asymptotically tight bounds on time-space tradeoffs in a pebble game". J. ACM **29**, 1087–1130 (1982).

[6] Cynthia Dwork, Moni Naor, and Hoeteck Wee. "Pebbling and proofs of work". In Proceedings of the Advances in Cryptology (CRYPTO 2005). Pages 37–54. Springer Berlin Heidelberg (2005).

[7] Ling Ren and Srinivas Devadas. "Proof of space from stacked expanders". In Proceedings of the 14th International Conference on Theory of Cryptography (TCC 2016). Volume 9985, page 262–285. Springer-Verlag (2016).

[8] Jeremiah Blocki and Samson Zhou. "On the depth-robustness and cumulative pebbling cost of argon2i". In Proceedings of the Theory of Cryptography (TCC 2017). Pages 445–465. Springer International Publishing (2017).

[9] Charles H. Bennett. "Time/space trade-offs for reversible computation". SIAM Journal on Computing **18**, 766–776 (1989).

[10] Jeremiah Blocki, Blake Holman, and Seunghoon Lee. "The parallel reversible pebbling game: Analyzing the post-quantum security of imhfs". In Proceedings of the Theory of Cryptography: 20th International Conference (TCC 2022). Page 52–79. Springer-Verlag (2022).

[11] Craig Gidney. "Spooky pebble games and irreversible uncomputation". `algassert.com/post/1905` (2019).

[12] Lov K. Grover. "A fast quantum mechanical algorithm for database search". In Proceedings of the twenty-eighth annual ACM Symposium on Theory of Computing (STOC 1996). Pages 212–219. (1996).

[13] Mark Oskin, Frederic T. Chong, and Isaac L. Chuang. "A practical architecture for reliable quantum computers". Computer **35**, 79–87 (2002).

[14] Robert Y. Levine and Alan T. Sherman. "A note on Bennett's time-space tradeoff for reversible computation". SIAM Journal on Computing **19**, 673–677 (1990).

[15] Richard Král'ovič. "Time and space complexity of reversible pebbling". In Proceedings of the Theory and Practice of Informatics (SOFSEM 2001). Pages 292–303. Springer Berlin Heidelberg (2001).

[16] Simon Perdrix and Philippe Jorrand. "Classically controlled quantum computation". Mathematical Structures in Computer Science **16**, 601 (2006).

[17] Farid Ablayev, Cristopher Moore, and Christopher Pollett. "Quantum and stochastic branching programs of bounded width". In Proceedings of Automata, Languages and Programming (ICALP 2002). Pages 343–354. Springer Berlin Heidelberg (2002).

[18] Alessandro Cosentino, Robin Kothari, and Adam Paetznick. "Dequantizing read-once quantum formulas". In Proceedings of the 8th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2013). Volume 22 of LIPIcs, pages 80–92. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2013).

[19] Arend-Jan Quist and Alfons Laarman. "Optimizing quantum space using spooky pebble games". In Reversible Computation (RC 2023). Pages 134–149. Springer Nature Switzerland (2023).

[20] Arend-Jan Quist and Alfons Laarman. "Trade-offs between classical and quantum space using spooky pebbling" (2024). arXiv:2401.10579.

[21] Ming Li and Paul Vitanyi. "Reversibility and adiabatic computation: Trading time and space for energy". Proceedings of the Royal Society A **452**, 769–789 (1996).

[22] Ming Li, John Tromp, and Paul Vitányi. "Reversible simulation of irreversible computation". Physica D: Nonlinear Phenomena **120**, 168–176 (1998).

[23] Eleanor Rieffel and Wolfgang Polak. "Quantum computing: A gentle introduction". The MIT Press. (2011). 1st edition.

[24] R. Landauer. "Irreversibility and heat generation in the computing process". IBM Journal of Research and Development **5**, 183–191 (1961).

[25] Charles H. Bennett. "Logical reversibility of computation". IBM Journal of Research and Development **17**, 525–532 (1973).
[26] Klaus-Jörn Lange, Pierre McKenzie, and Alain Tapp. "Reversible space equals deterministic space". Journal of Computer and System Sciences **60**, 354–367 (2000).
[27] Mehdi Saeedi and Igor L. Markov. "Synthesis and optimization of reversible circuits—a survey". ACM Computing Surveys **45**, 1–34 (2013).
[28] Scott Aaronson, Daniel Grier, and Luke Schaeffer. "The Classification of Reversible Bit Operations". In Proceedings of the 8th Innovations in Theoretical Computer Science Conference (ITCS 2017). Volume 67 of Leibniz International Proceedings in Informatics (LIPIcs), pages 23:1–23:34. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2017).
[29] Michael P. Frank and M. Josephine Ammer. "Relativized separation of reversible and irreversible space-time complexity classes" (2017). arXiv:1708.08480.
[30] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. "Unqomp: Synthesizing uncomputation in quantum circuits". In Proceedings of the 42nd International Conference on Programming Language Design and Implementation (SIGPLAN 2021). Page 222–236. PLDI 2021. Association for Computing Machinery (2021). code: eth-sri/Unqomp.
[31] Raymond Wheeler and Richard Hughey. "Optimizing reduced-space sequence analysis". Bioinformatics **16**, 1082–1090 (2000).
[32] Lee A. Newberg. "Memory-efficient dynamic programming backtrace and pairwise local sequence alignment". Bioinformatics **24**, 1772–1778 (2008).
[33] Philippe Dumas. "Reversing a finite sequence". "`https://algo.inria.fr/seminars/sem94-95/pottier.pdf`" (1995). Summary of a seminar talk given by Loïc Pottier.
[34] José Grimm, Loïc Pottier, and Nicole Rostaing-Schmidt. "Optimal Time and Minimum Space-Time Product for Reversing a Certain Class of Programs". Technical Report RR-2794. INRIA (1996). url: `https://inria.hal.science/inria-00073896`.
[35] W. J. Paul and R. E. Tarjan. "Time-space trade-offs in a pebble game". Acta Informatica **10**, 111–115 (1978).
[36] Nicholas Pippenger. "A time-space trade-off". J. ACM **25**, 509–515 (1978).
[37] Thomas Lengauer and Robert Endre Tarjan. "Upper and lower bounds on time-space trade-offs". In Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing (STOC 1979). Page 262–277. Association for Computing Machinery (1979).
[38] Joël Alwen and Vladimir Serbinenko. "High parallel complexity graphs and memory-hard functions". In Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing (STOC 2015). Page 595–603. ACM (2015).
[39] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. "Depth-robust graphs and their cumulative memory complexity". In Proceedings of the Advances in Cryptology (EUROCRYPT 2017). Pages 3–32. Springer (2017).
[40] Erik D. Demaine and Quanquan C. Liu. "Inapproximability of the standard pebble game and hard to pebble graphs". In Proceedings of Algorithms and Data Structures (WADS 2017). Pages 313–324. Springer (2017).
[41] Erik D. Demaine and Quanquan C. Liu. "Inapproximability of the standard pebble game and hard to pebble graphs" (2017). arXiv:1707.06343.
[42] D.D.M. Cracken, D.D. McCracken, W.S. Dorn, and Ltd John Wiley & Sons. "Numerical methods and fortran programming: With applications in engineering and science". Goldstine Printed Materials. Wiley. (1964).
[43] Ralph C. Merkle. "A digital signature based on a conventional encryption function". In Proceedings of Advances in Cryptology (CRYPTO 1987). Pages 369–378. Springer Berlin Heidelberg (1988).

[44] Stephen A. Cook. “An observation on time-storage trade off”. In Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (STOC 1973). Page 29–33. Association for Computing Machinery (1973).

[45] Balagopal Komarath, Jayalal Sarma, and Saurabh Sawlani. “Reversible pebble game on trees”. In Proceedings of Computing and Combinatorics (COCOON 2015). Pages 83–94. Springer International Publishing (2015).