Universiteit
Leiden
The Netherlands

Optimizing quantum space using spooky pebble games

Quist, A.; Laarman, A.W.; Kutrib, M.; Meyer, U.

Citation
Quist, A., & Laarman, A. W. (2023). Optimizing quantum space using spooky pebble games. *Lecture Notes In Engineering And Computer Science*, 134-149. doi:10.1007/978-3-031-38100-3_10

Version:	Publisher's Version
License:	Licensed under Article 25fa Copyright Act/Law (Amendment Taverne)
Downloaded from:	https://hdl.handle.net/1887/3673406

Note: To cite this publication please use the final published version (if applicable).

Optimizing Quantum Space Using Spooky Pebble Games

Arend-Jan Quist(✉) and Alfons Laarman

Leiden Institute of Advanced Computer Science, Leiden University, Leiden, The Netherlands
{a.quist,a.w.laarman}@liacs.leidenuniv.nl

Abstract. Pebble games are usually used to study space/time trade-offs. Recently, spooky pebble games were introduced to study classical space/quantum space/time trade-offs for simulation of classical circuits on quantum computers. In this paper, the spooky pebble game framework is applied for the first time to general circuits. Using this framework we prove an upper bound for quantum space in the spooky pebble game. Moreover, we present a solver for the spooky pebble game based on a SAT solver. This spooky pebble game solver is empirically evaluated by calculating optimal classical space/quantum space/time trade-offs. Within limited runtime, the solver could find a strategy reducing quantum space when classical space is taken into account, showing that the spooky pebble model is useful to reduce quantum space.

Keywords: Pebble game · Spooky pebble game · Quantum computing · Satisfiability

1 Introduction

For a long time, scientists have been thinking how specific problems could be calculated within certain computational constraints. For example, the size of the memory and the runtime of the calculation are usually constrained. It could be useful to know whether for example some specific computational problems can be calculated on a machine with certain limited memory. To study the memory usage and runtime for a calculation, pebble games were introduced. Pebble games model the use of space and time in the run of a given circuit. The trade-off between space and time can also be easily studied by a pebble game by studying the maximum number of pebbles and time used.

A pebble game for a given circuit is played on a graph. This graph is the natural underlying graph of the circuit. Every gate in the circuit is a node in the graph. The direct input/output connection between gates is represented by a directed edge from source gate to target gate. Note that such a graph is a directed acyclic graph, as there could not be any cycle in a circuit.

The pebble game is played as follows. Every node can be pebbled and unpebbled. Placing a pebble on a node means that the result/output of the corresponding gate is calculated and stored in memory. Unpebbling a node means that the

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
M. Kutrib and U. Meyer (Eds.): RC 2023, LNCS 13960, pp. 134–149, 2023.
https://doi.org/10.1007/978-3-031-38100-3_10

calculated value is removed from memory. Thus a node can only be pebbled if its inputs are pebbled already, as the gate depends on its inputs which should be in current memory. The goal of the pebble game is to pebble the final outputs of the circuit, which means that the output values of the circuit are stored in memory.

The memory usage and runtime of a circuit can now be easily studied. The memory is modelled by pebbles, where every pebble represents one memory unit. Thus the space used by a circuit is the maximal number of pebbles used at any time. The runtime is the number of pebblings and unpebblings to play the pebble game.

There exist variations of the pebble game, see for example [5,15] for a (non complete) overview. In this paper, we will consider three types of pebble games: irreversible pebble games, reversible pebble games and the spooky pebble game. For the *irreversible pebble game* all inputs of a node must be pebbled to pebble that node, but there are no constraints for unpebbling a node. This is a model for classical computing. For the *reversible pebble game* all inputs of a node must be pebbled to either pebble or unpebble that node. This is a model for reversible computing in general and, more specific, for quantum computing on a circuit with irreversible gates [13]. This is useful for quantumly simulating a classical circuit. This pebble game was introduced by Bennett [2].

The *spooky pebble game* is an extension of the reversible pebble game for quantum computing which was introduced recently by Gidney [8] and more extensive worked out by Kornerup et al [12]. This game weakens the constraint for unpebbling a node in reversible pebbling at the cost of doing a quantum measurement. In the spooky pebble game, this is represented by a spook, which replaces a pebble and classically stores the measurement outcome. The (classical) value of the measurement outcome is used later by the quantum computer to restore the state. Thus for the spooky pebble game we can study the trade-off between classical memory, quantum memory and time. As quantumly accessible classical space (represented by spooks) is assumed to be much cheaper that quantum space (represented by pebbles) (see e.g. [1,16,17]), studying such a trade-off could be very useful for memory reduction in quantum computing.

The spooky pebble game, similar to reversible pebbling, models quantum computing for an irreversible (classical) circuit. Therefore, it is a useful model for a quantum oracle calculation of a classical function, i.e. quantum calculations of the type

$$\sum_{x\in\{0,\ldots,N-1\}} \alpha_x|x\rangle|j_x\rangle \rightarrow \sum_{x\in\{0,\ldots,N-1\}} \alpha_x|x\rangle|j_x \oplus f(x)\rangle$$

for some classical function $f : \{0,\ldots,N-1\} \rightarrow \{0,\ldots,M-1\}$. In many famous quantum algorithms like Shor's [19] and Grover's [10] such oracle calculations are essential. Thus, studying the spooky pebble game can be useful for quantum memory management for near term quantum computing on machines with constrained space and time.

The remainder of this paper is organized as follows. In Sect. 2, we will explain the theory about the spooky pebble game. Note that we are the first that study

the spooky pebble game on general DAGs instead of line graphs as in [12]. In Sect. 3, we will prove a theorem about the maximum memory cost for a quantum oracle computation with respect to the equivalent classical computation when there is also classical memory available. Section 4 describes a solver of the spooky pebble game developed by us. This solver encodes the game as a satisfiability problem and tries to solve it. To our knowledge this is the first solver for the spooky pebble game. In Sect. 5 the details of our open source solver implementation are presented. This section also describes experiments to create optimal quantum space, classical space and time trade-offs with the solver.

2 Background: Spooky Pebble Game

In this section, we will first explain measurement based uncomputation for quantum computing, which is the basis for the spooky pebble game. Then we will formally introduce (spooky) pebble games. Most of this section was introduced by [8,12].

2.1 Measurement-Based Uncomputation

In quantum computing, the only operations one can apply are reversible operations and measurements. Thus, to apply irreversible (classical) operations without using additional space, a measurement is needed. The disadvantage of such measurements is that superpositions of inputs, which usually form the strength of quantum computing, collapse unintendedly. Measurement-based uncomputation is a technique to do an irreversible uncomputing operation using a measurement such that, under certain conditions, the uncollapsed state can be restored using the outcome of the measurement. This technique can be applied for quantum oracles where the quantum input remains in memory during the entire computation and no interference gates are applied in between the measurement-based uncomputation and the restoring of the state. These conditions are satisfied when we quantumly simulate a classical oracle computation, which we will focus on in this paper.

Figure 1 depicts the idea of measurement-based uncomputation in a circuit. Assume that an irreversible function f is computed on some quantum superposition input $\sum_x \alpha_x|x\rangle$. This is done by the reversible operation U_f which writes the outcome in a new register, resulting in the state $\sum_x \alpha_x|x\rangle|f(x)\rangle$. Now, (a part of) the input is changed by another complicated computation, while keeping the function output in memory. If we want to remove the function output from the memory, we cannot simply uncompute the function output by applying U_f again as the input has gone. Instead the function output is measured after applying a Hadamard gate to it. On measurement outcome b, a phase $(-1)^{bf(x)}$ is added to the state. After computing the semi-classical CNOT dependent on the measurement outcome, the memory of the function output is free and can be freely used as an $|0\rangle$ ancilla by other computations. If we want to correct for the phase $(-1)^{bf(x)}$, we need the state with the function outputs in memory

again. To get this, we need the input $\sum_x \alpha_x|x\rangle$ again and compute the function f. Now we can apply a Z-gate on the function output register, dependent on the measurement outcome b. Now, the function output can be uncomputed by applying U_f again, and we are back in the state with input $\sum_x \alpha_x|x\rangle$.

Note that this trick of temporary uncomputing the function output of an irreversible function f when the inputs are not available is not possible in usual reversible computing. Hence, measurement-based uncomputation can be seen as quantum semi-reversible computation.

For a more formal and extensive description of measurement-based uncomputation, we refer to [12].

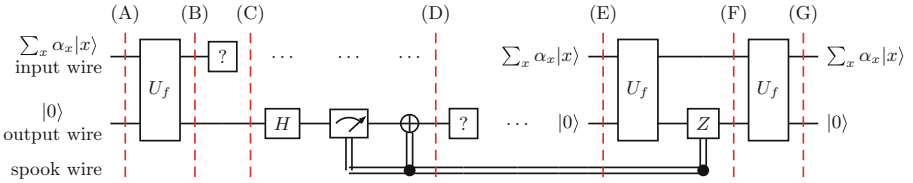

Fig. 1. This circuit describes measurement-based uncomputation. The three wires depict the input and output of the function f and the spook wire to classically store the measurement outcome. The labels (A), (B), etc. correspond to the corresponding spooky pebbling configurations as shown in Fig. 2.

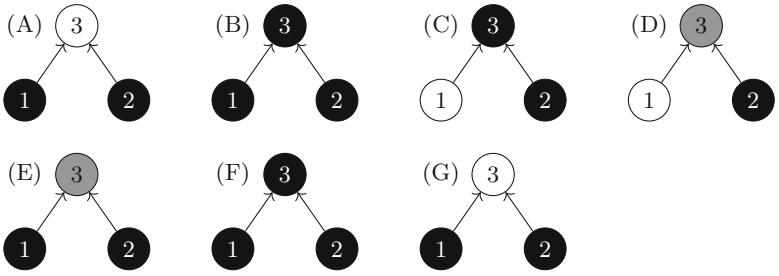

Fig. 2. Spooky pebbling configurations for the labels in the circuit of Fig. 1. In Definition 2, the spooky pebble game is introduced. The function inputs in vertices (1) and (2) are used to compute a function f and store the output in vertex (3). In other words: $f((1),(2)) = (3)$. A white vertex is unpebbled, a black one is pebbled and a grey vertex is spooked. Note that in this example not the entire input is changed: vertex (2) is pebbled (i.e. hold in memory) all over the computation and only vertex (1) is unpebbled (i.e. removed from memory).

2.2 Pebble Games

In this section, we will formally define three types of pebble games: irreversible, reversible and spooky.

We will first define the irreversible and reversible pebble game. The irreversible pebble game is a natural model for classically computing a circuit. The reversible pebble game is a natural model for reversible computation on irreversible circuits, e.g. quantumly computing a classical circuit.

Definition 1 ((Ir)reversible pebbling). *Let $G = (V, E)$ be a directed acyclic graph (DAG). The set of* roots *R is defined as $R = \{v \in V \mid \textit{out-degree}(v) = 0\}$. A* (ir)reversible pebbling strategy *on G is a sequence of sets of pebbled vertices $P_0, P_1, \ldots, P_T \subseteq V$ such that the following conditions hold:*

1. *$P_0 = \emptyset$;*
2. *$P_T = R$;*
3. *for every $t \in \{1, 2, \ldots, T\}$ there exists one $v \in V$ such that one of the following holds:*
   - pebble(v): *$P_t = P_{t-1} \cup \{v\}$ and all direct predecessors (i.e. in-nodes) of v are in P_{t-1};*
   - unpebble(v):
     For irreversible *pebbling: $P_t = P_{t-1} \backslash \{v\}$;*
     For reversible *pebbling: $P_t = P_{t-1} \backslash \{v\}$ and all direct predecessors (i.e. in-nodes) of v are in P_{t-1}.*

Now, we will define the spooky pebble game, a natural model for quantumly computing a classical circuit using measurement-based uncomputation. The definition for this game is mainly adapted from [12]. The spooky pebble game can be viewed as an extended generalization of the (ir)reversible pebble game. The main difference between spooky pebbling and (ir)reversible pebbling is the addition of spooks. To regulate the spooks, the actions *spook* and *unspook* are added, and for the actions *pebble* and *unpebble* the requirement that no spook is changed is added.

Definition 2 (Spooky pebbling). *Let $G = (V, E)$ be a directed acyclic graph (DAG). The set of* roots *R is defined as $R = \{v \in V \mid \textit{out-degree}(v) = 0\}$. A* spooky pebbling strategy *on G is a sequence of pairs of pebbles and spook pebbles $(P_0, S_0), (P_1, S_1), \ldots, (P_T, S_T) \subseteq V \times V$ such that the following conditions hold:*

1. *$P_0 = \emptyset$ and $S_0 = \emptyset$;*
2. *$P_T = R$ and $S_T = \emptyset$;*
3. *for every $t \in \{1, 2, \ldots, T\}$ there exists one $v \in V$ such that one of the following holds:*
   - pebble(v): *$P_t = P_{t-1} \cup \{v\}$ and $S_t = S_{t-1}$ and all direct predecessors (i.e. in-nodes) of v are in P_{t-1};*
   - unpebble(v): *$P_t = P_{t-1} \backslash \{v\}$ and $S_t = S_{t-1}$ and all direct predecessors (i.e. in-nodes) of v are in P_{t-1};*
   - spook(v): *$P_t = P_{t-1} \backslash \{v\}$ and $S_t = S_{t-1} \cup \{v\}$;*

– unspook(v): $P_t = P_{t-1} \cup \{v\}$ *and* $S_t = S_{t-1} \backslash \{v\}$ *and all direct predecessors (i.e. in-nodes) of* v *are in* P_{t-1}. [1]

For a pebbling strategy on a DAG G, the following three quantities are useful. The *pebbling time* is the integer T. The *pebbling cost* is $\max_{t \in [T]} |P_t|$ and (for the spooky pebbling game) the *spook cost* is $\max_{t \in [T]} |S_t|$.

In Fig. 3 we see some examples of moves for the different types of pebble games. For the configuration on the top, the move *unpebble(5)* can be applied for all irreversible, reversible and spooky pebbling. The move *unpebble(4)* can only be applied in the irreversible pebble game, as its input vertex 1 is not pebbled. The move *spook(4)* can only be applied in the spooky pebble game, as the (ir)reversible pebble game don't have spooks.

In Figs. 1 and 2, the relation between the spooky pebble game and measurement-based uncomputation is depicted. Quantum space is represented by pebbles and spooks represent classical space. Like in the reversible pebble game, pebbling and unpebbling are just applying classical (irreversible) gates on a quantum computer, storing the gate-outputs in additional space. Placing a spook can be interpreted as applying measurement-based uncomputation and storing the measurement outcome in memory. Unspooking can be interpreted as restoring the uncomputated state by recomputing the function.

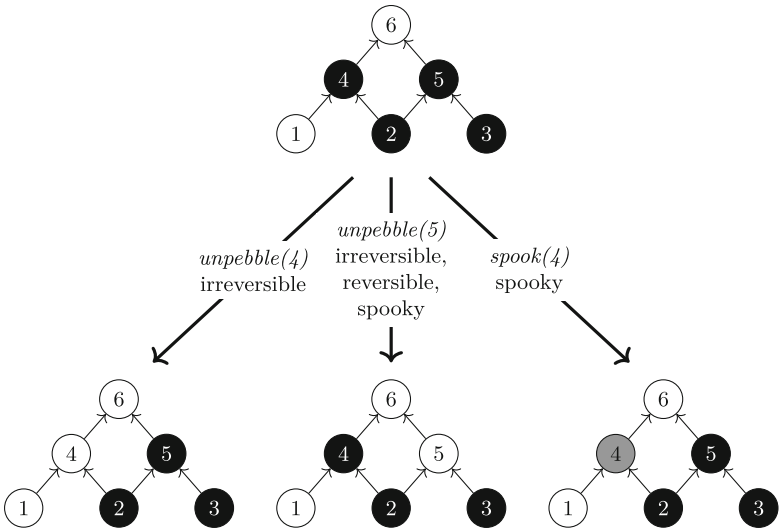

Fig. 3. Three examples of steps for different types of pebble games. A white node represents an empty vertex, a black vertex is a pebbled vertex and a grey vertex is a spooked vertex.

[1] Note that a pebble is placed on v when v is unspooked.

3 Theorem: Spooky Pebble Cost Equals Irreversible Pebble Cost

In this section, we present some results to relate pebble cost and price of different types of pebble games presented above. We will show first that if a DAG with m roots (outputs) can be irreversible pebbled with C pebbles, then it can be spooky pebbled with at most $C+m$ pebbles. Intuitively, this result holds because the irreversible pebble game can be simulated by the spooky pebble game with additional pebbles at all roots. Gidney [8] already proved a similar result for line graphs where an optimal irreversible pebble strategy with $C = 2$ pebbles is obvious, but we generalize this to arbitrary DAGs and arbitrary irreversible pebbling strategies. The statement can be formalized as follows.

Theorem 1. *Let $G = (V, E)$ be a DAG with m roots. Assume there exists a irreversible pebbling strategy to pebble G with irreversible pebble cost C. Then there exists a spooky pebbling strategy to pebble G with spooky pebble cost at most $C + m$.*

Proof. Let $P_0, P_1, \ldots, P_T$ be a irreversible pebbling strategy with irreversible pebble cost C. Now we can simulate this strategy by a spooky pebble game. This simulation consists of two stages.

In the *first stage*, the irreversible pebbling strategy is followed with the following modifications. If a vertex is pebbled in the irreversible game, then so in the spooky game. If a pebble is removed from a vertex in the irreversible game, then that pebble is spooked in the spooky game. Thus at the end of this stage, all roots (outputs) are pebbled and all other vertices are spooked. Note that this first stage takes C pebbles, because the irreversible pebbling strategy needs C pebbles.

Now we turn to the *second stage*. In this stage, we will unspook all spooked pebbles. Note that every vertex $v \in V$ can be unspooked in the spooky pebble game using at most C additional pebbles, apart from the m pebbles at the roots. This unspooking (which is in fact pebbling and unpebbling vertex v) can be done by repeating the irreversible pebbling strategy from the first stage only to the vertices in the subDAG with root v. In other words, vertex v can always be unspooked by a projection of the simulation of the irreversible pebbling strategy $P_0, P_1, \ldots, P_T$ on subDAG $\hat{G}(v)$, where $\hat{G}(v)$ is defined as the largest subDAG of G with root v. This costs at most C additional pebbles because the irreversible pebbling strategy can cost at most C pebbles. As all roots are pebbled during the second stage, unspooking a vertex $v \in V$ costs at most $C + m$ pebbles.

We can unspook the vertices in the reverse order as the order we pebbled them (for the first time t) in the irreversible pebbling strategy. This is to ensure that unspooking a vertex v can never be done when there is still a spook on a successor w of v. Unspooking such w could then give a spook on v. So if all vertices are unspooked in this reverse of placement order, all spooks will be removed at the end of the second stage. Thus we end up with the DAG in the final state without spooks and with pebbles on all roots. □

Now we will analyse the spooky pebble time and the spook cost of the strategy described in the above theorem.

First we analyse the spook cost. We observe that at most $|V| - m$ spooks are needed in the first stage, because every non-root vertex is spooked. Note that this bound is an upper bound: in most cases there is a strategy such that less spooks are needed. One could spare spooks by unpebbling a vertex v if possible instead of always spooking it in the first stage. This in general reduces the number of spooks.

For the spooky pebble time we can do the following observations. If the irreversible pebble time is T, then the first stage takes also time T. In the second stage, for every vertex $v \in V \backslash R$ a projection of the first stage strategy to $\hat{G}(v)$ is applied to unspook vertex v. Thus the time to unspook any vertex $v \in V \backslash R$ can be at most T. So the spooky pebble time of the strategy of Theorem 1 is at most $T(|V| - m + 1)$, as every of the $|V| - m$ non-root vertices $v \in V \backslash R$ needs to be unspooked.

4 Spooky Pebble Game Solver

In this section, we present our solver algorithm for the spooky pebble game. This solver tries to find a solution to the pebble game with a constrained number of pebbles and spooks. For the irreversible [9] and reversible [6] pebble game, optimizing the number of pebbles is shown to be PSPACE-complete. For the spooky pebble game no such result exist, but presumably it is a difficult problem too. Thus, the spooky pebble game is encoded as a satisfiablilty problem, similar to the method of [14] for the reversible pebble game.

To encode the spooky pebble game into a satisfiability problem, we use the boolean variables $p_{v,i}$ and $s_{v,i}$ with v the vertex and i the time. The variable $p_{v,i}$ indicates whether vertex v is pebbled at time i, and similar for spooking with variable $s_{v,i}$. We use the shorthand $x_i = \bigcup_{v \in V} p_{v,i} \cup \bigcup_{v \in V} s_{v,i}$ to describe all variables at time i.

We use the clauses as stated below to describe the game. If there is an assignment of variables that satisfies all clauses, i.e. evaluates to true, this assignment describes a solution of the game. The clauses below are for the game with T timesteps.

Initial clauses: For V the set of vertices in DAG G we have initial clauses:

$$I(x_0) = \bigwedge_{v \in V} \neg p_{v,0} \wedge \neg s_{v,0} \tag{1}$$

Final clauses: For R the set of outputs (roots) in DAG G we have final clauses:

$$F(x_T) = \bigwedge_{v \in R} p_{v,T} \wedge \bigwedge_{v \notin R} \neg p_{v,T} \wedge \bigwedge_{v \in V} \neg s_{v,T} \tag{2}$$

Move clauses: There are the following move clauses for edgeset E of DAG G, for every time $i = 0, \ldots, T-1$:

To add a pebble all predecessors must be pebbled and a spook on this vertex should be removed[2]:

$$M_1(x_i, x_{i+1}) = \bigwedge_{(v,w)\in E} [(\neg p_{v,i} \wedge p_{v,i+1}) \implies (p_{w,i} \wedge p_{w,i+1})] \wedge \bigwedge_{v\in V} [(\neg p_{v,i} \wedge p_{v,i+1}) \implies \neg s_{v,i+1}] \quad (3)$$

To remove a pebble all predecessors must be pebbled or the pebble must be changed into a spook:

$$M_2(x_i, x_{i+1}) = \bigwedge_{(v,w)\in E} [(p_{v,i} \wedge \neg p_{v,i+1}) \implies ((p_{w,i} \wedge p_{w,i+1}) \vee s_{v,i+1})] \quad (4)$$

To add a spook the vertex must be unpebbled:

$$M_3(x_i, x_{i+1}) = \bigwedge_{v\in V} [(\neg s_{v,i} \wedge s_{v,i+1}) \implies (p_{v,i} \wedge \neg p_{v,i+1})] \quad (5)$$

To remove a spook all predecessors must be pebbled and the spook must be changed into a pebble:

$$M_4(x_i, x_{i+1}) = \bigwedge_{(v,w)\in E} [(s_{v,i} \wedge \neg s_{v,i+1}) \implies ((p_{w,i} \wedge p_{w,i+1}) \wedge p_{v,i+1})] \quad (6)$$

Cardinality clauses: For P the maximal number of pebbles and S the maximal number of spooks we have for every $i = 0, 1, \ldots, T$ the clauses:

$$C(x_i) = \left(\sum_{v\in V} p_{v,i} \leq P\right) \wedge \left(\sum_{v\in V} s_{v,i} \leq S\right) \quad (7)$$

Using these clauses, we actually encode our problem as in Bounded Model Checking format (BMC) [3, Chapter 18]. Using this formula the problem can be easily defined for arbitrary times T. The following BMC formula is used:

$$I(x_0) \wedge C(x_0) \wedge M(x_0, x_1) \wedge C(x_1) \wedge \cdots \wedge M(x_{T-1}, x_T) \wedge C(x_T) \wedge F(x_T). \quad (8)$$

Here, M is the conjunction $M = M_1 \wedge M_2 \wedge M_3 \wedge M_4$ of the move clauses from Eq. (3)–(6). With this BMC formula, the time T can be gradually unrolled until a solution is found. This formula can be given to a SAT solver to find a solution for the spooky pebble game problem.

In Algorithm 1 we state our heuristic for the spooky pebble game solver. The solver starts with time $T = 0$. If the formula is proven to be *unsatisfiable* (i.e. the solver shows that there is no solution for the given SAT formula), T is incremented by 1. If the SAT solver cannot find a solution in runtime T_{wait}

[2] Hereby we prevent a vertex to be pebbled and spooked at the same time.

Algorithm 1. Heuristic for spooky pebble game solver (runtime timeout T_{max})

```
T := 0
formula = setup_SAT_formula(T)
while result ≠ sat do
    result := run_SAT_solver(formula)        ▷ returns timeout after T_wait seconds
    if result = timeout then
        T := T + T_skip
    if result = unsat then
        T := T + 1
    formula = unroll_SAT_formula(T)          ▷ update formula upto time T
```

Algorithm 2. Sequential time optimizer heuristic
(T is parallel time of the game solution)

```
for t_0 = T, T − 1, ..., 1 do
    for v in V do
        if v is unpebbled at time t_0 then
            for t = t_0, t_0 − 1, ..., 0 do
                if v was used to (un)pebble or unspook a succesor at time t then
                    break
                if v was unspooked at time t then
                    break
                if v is pebbled at time t then
                    remove the pebble from v between time t and t_0
                    break
```

but the formula cannot be proven to be unsatisfiable, we have a *timeout* and T is incremented with T_{skip}[3]. If the SAT solver finds a *satisfiable* solution, the program is stopped as a strategy for the spooky pebble game is found. This entire heuristic has a timeout of T_{max}.

Using push and pop statements for the final clauses, incremental SAT solving can be used, which updates the SAT formula for every new time T. This incremental extension of the formula speeds up the SAT solver in next runs.

Note that our implementation of clauses uses parallel time. In other words, multiple pebbles and spooks can be added or removed at the same timestep. For example, in the pebble configuration at the top of Fig. 3, the steps *unpebble(2)*, *unpebble(3)* and *pebble(6)* can be applied in one parallel timestep. In general, the parallel pebbling time for a game is much smaller than the sequential pebbling time as multiple sequential steps can be applied in one parallel step. This highly reduces the number of BMC enrollings to find a solution. As every BMC enrolling duplicates the move and cardinality clauses as well as the variables all the time, the size of the SAT formula can be highly reduced too. This reduction of clauses and variables in the SAT formula speeds up the SAT solver.

[3] We define T_{skip} (which is possibly greater than 1) because every call to the SAT solver takes some 'useless' time to optimize the SAT formula. Thus the heuristic usually can find a spooky pebble game solutions faster when $T_{skip} > 1$.

We built a tool that converts a parallel solution to a sequential solution such that only one pebble or spook is added or removed at the same time. This tool also detects useless pebbling and unpebblings. More precisely, if a pebble is placed but not used to pebble any successor, this pebbling and corresponding unpebbling is detected and removed. The precise procedure is stated in Algorithm 2. With this algorithm, the sequential time and the number of used pebbles of the parallel solution provided by the solver can be reduced. These sequential times are usually reported for pebbling strategies, see e.g. [14].

5 Implementation and Experiment

We implement our spooky game solver algorithm from Sect. 4 in Python. Our code is open source can be found in [18]. The open source Z3 solver for satisfiability is used as SAT solver [7].

To test the performance of our implementation, we use an Intel Core i7-6700 CPU @ 3.40GHz × 8 processor. The benchmark circuits we use are the well known ISCAS85 benchmarks [4]. Using the open source tool mockturtle, the circuits are transformed to XOR-majority graphs (DAGs), which can be easily transformed to circuits with common quantum gates [20]. Table 1 shows the information on the DAGs of the ISCAS85 benchmarks.

Table 1. Properties of the DAGs of the ISCAS85 benchmarks used to benchmark the spooky pebble game solver.

name	vertices	roots	edges
c432	172	7	260
c499	177	32	246
c880	276	26	374
c1355	177	32	246
c1908	193	25	257
c2670	401	46	533
c3540	830	22	1395
c5315	1089	96	1503
c6288	979	32	1639
c7552	988	62	1584

For all these benchmarks we run a performance test. The goal is to find a front of spooky pebble game solutions with optimal parameters (sequential) time, number of pebbles and number of spooks. The procedure is described in detail in Algorithm 3. For various numbers of spooks a solution is searched with a decreasing number of pebbles. For each constraint, the solver is run 5 times with different seeds. If at least one solution is found, the number of pebbles is decreased by 5. If no solution is found, the number of pebbles is not decreased anymore and the iteration with next number spooks is started.

As runtime parameters for the solver (Algorithm 1), we use $T_{wait} = 15\,\text{s}$ and $T_{max} = 2\,\text{min}$. The solver gives quite little results for these runtimes on large benchmarks[4].

Thus, we used a longer runtime for the large benchmarks. For the four largest benchmarks, $T_{wait} = 60\,\text{s}$ and $T_{max} = 8\,\text{min}$ is used. For all benchmarks, T_{skip} is set to 5.

[4] Large benchmarks need many clauses and variables to encode one BMC iteration, and usually also take a larger pebbling time (more BMC iterations). Hence, their SAT formula is much larger, which makes it a harder job for the SAT solver to find a solution.

Algorithm 3. Optimal solutions search for benchmarks
Function spooky_pebble_game_solver(pebbles, spooks) is described in Algorithm 1

```
for spooks = vertices, vertices/5, vertices/10, vertices/20, 0 do
    spooky_pebble_game_solver(∞, spooks)          ▷ run solver 5 times
    pb := minimal #pebbles in found solutions
    for pebbles = pb, pb-5, pb-10, ... do
        spooky_pebble_game_solver(pebbles, spooks)  ▷ run solver 5 times
        if no solution is found by solver then
            break
```

In Figs. 4 and 5 the results of the runs of the spooky pebble game solver are depicted. Figure 4 shows the solutions found by the solver for each benchmark. The color of the datapoint indicates the number of spooks used by the solution. Figure 5 depicts the runtime of the solver with respect to the constraints on the number of pebbles and spooks (P resp. S in Eq. (7)). Note that the number of spooks is plotted on power scale to clarify the different colors of the datapoints. The runs with infinite pebbles (base case runs in Algorithm 3) are omitted from the plots as no constraints are needed[5]. Also the runs with a timeout (that did not find a solution) are omitted from the plots.

From the data as shown in Figs. 4 and 5 we see that for the large benchmarks with more than 800 vertices the datapoints are less coherent, despite the longer timeout times T_{wait} and T_{max} for these benchmarks. Especially for c3540, c5315 and c7552 still not so many datapoints are found, their sequential time plots don't show the same general behaviour as the others, and their runtimes are less uniform. This might be due to the large size of the benchmarks and the specific structure of their DAGs which causes instabilities in the performance of the underlying SAT engine.

From the data in Fig. 4 we observe the following. First observation is that for almost all benchmarks the solver could find a solution with less pebbles when spooks are allowed. In the case with 0 spooks, no solution can be found below a certain value of pebbles, while there are solutions with less pebbles that use spooks. In fact, within the runtime constraints of the solver, we can find a quantum/classical memory management strategy that reduces the quantum space needed to quantumly simulate a classical circuit. Thus, the spooky pebble framework is useful to reduce quantum space for a calculation.

Another direct observation is that in general the sequential time for a solution increases when the number of pebbles decreases. This is exactly what we would expect, because this is the usual space-time trade-off for calculations.

On the other hand, we observe that for solutions with relatively many pebbles the sequential time increases when the number of spooks increases. We would not expect this from first instance, as with more classical space we expect the time

[5] One could also argue that the pebble constraint equals the number of vertices. But the number of vertices is much larger than all other pebble constraints and would be far beyond the plot range.

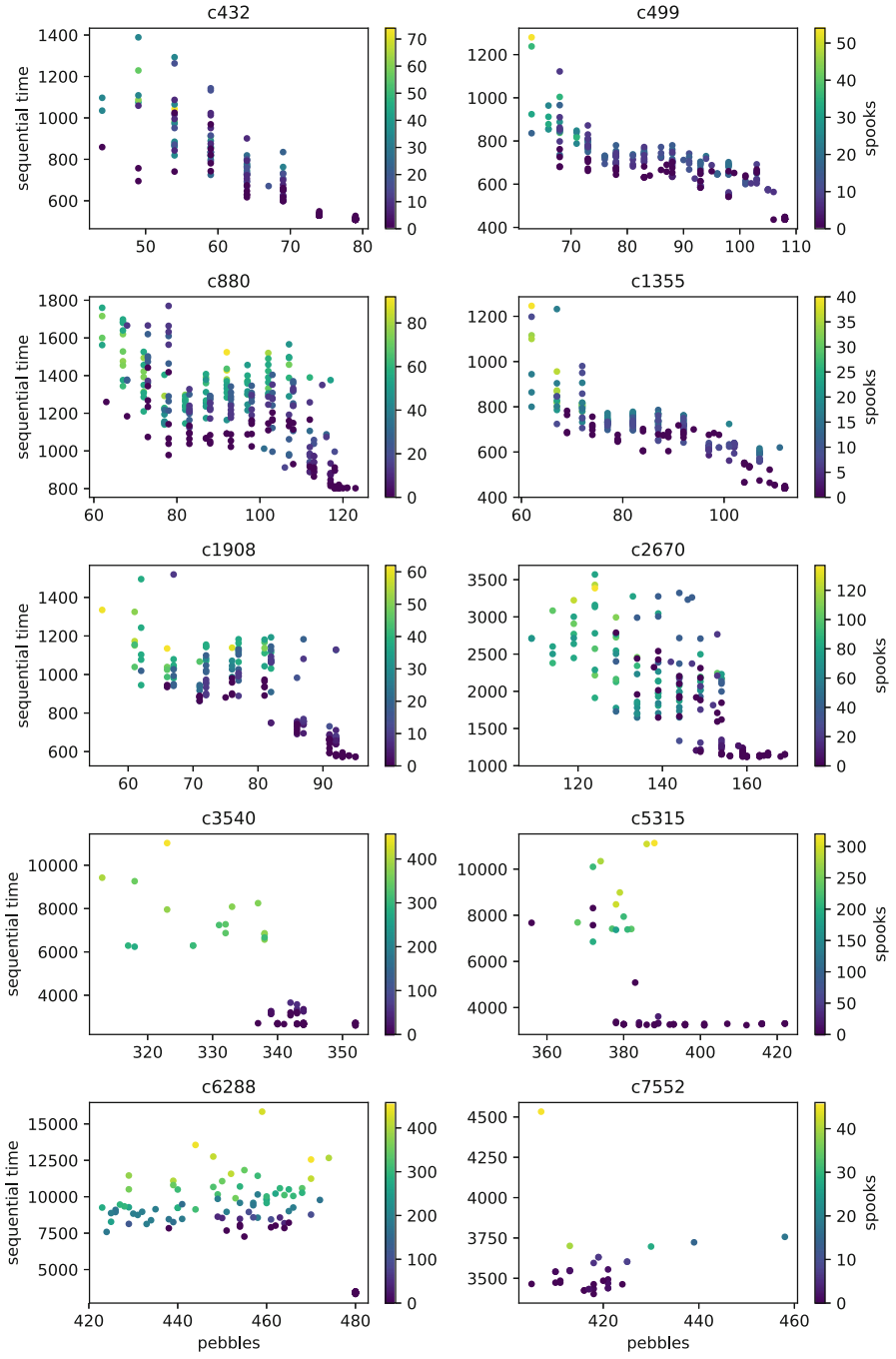

Fig. 4. Solutions of the spooky pebble game with sequential time, number of pebbles and number of spooks found by the heuristic of Algorithm 3.

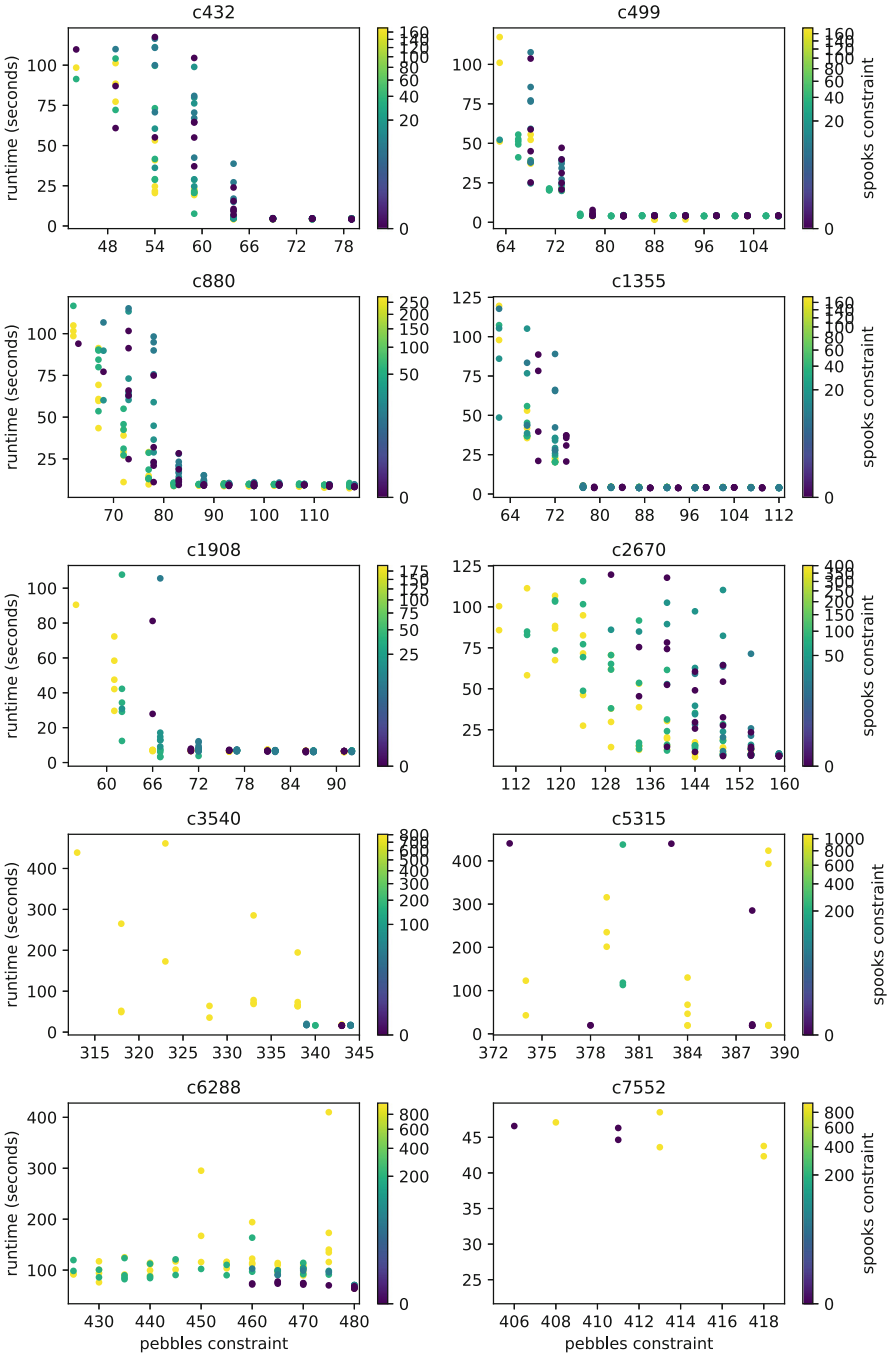

Fig. 5. Runtime of the heuristic spooky pebble game solver in Algorithm 3 for given constraints on number of pebbles and spooks. The runs with timeout (120 s resp. 480 s) are not shown in the figures.

to decrease by a similar space-time trade-off. This feature could be explained by considering that the solver uses parallel time instead of sequential time. So in a solution with small (or even minimal) parallel time there might be a lot of useless spook-unspook movements on vertices that are pebbled. Such useless spook-unspook movements increase the number of used spooks and increase the sequential time of the solution. Thus it would be a good idea for further research to design and implement an algorithm that detects and removes useless spook-unspook movements on pebbled vertices to optimize the solutions.

From Fig. 5 we see that in general the runtime of the solver increases when the constraint on the number of pebbles decreases. Similarly, the runtime increases when the constraint on the number of spooks decreases. This is also what we would expect, as stronger constraints makes it more difficult to find a solution.

6 Conclusions and Further Research

In this paper, we explored the spooky pebble game, a model for the trade-off between quantum space, classical space and time in a quantum computation. Using the spooky pebble game, we proved that a classical calculation that uses space C can be executed by a quantum computer using quantum space $C + m$ and some additional classical space. Here, m is the space of the output.

Moreover, we showed the design and implementation of a spooky pebble game solver, which gives a memory management strategy for a quantum computation with a limited amount of quantum space and classical space. Within limited runtime, this solver provides strategies with less quantum space when additional classical space is allowed. Moreover, with this solver we can find fronts for trade-offs between quantum space, classical space and time used to execute a circuit.

An idea for further research would be to optimize the spooky pebble game solver. As shown in the results, many pebbling strategies found by the solver seem to use useless spooks. The solver results can be improved by detecting such useless spooks and removing them.

Another idea for further research is searching for interesting constraints on the structure of the DAG which makes searching optimal solutions of spooky pebble game easier. For e.g. trees there might be efficient algorithms to find optimal spooky pebbling solutions, like [11] shows for reversible pebbling.

References

1. Babbush, R., et al.: Encoding electronic spectra in quantum circuits with linear t complexity. Phys. Rev. X **8**(4), 041015 (2018)
2. Bennett, C.H.: Time/space trade-offs for reversible computation. SIAM J. Comput. **18**(4), 766–776 (1989)
3. Biere, A., Heule, M., van, M.H.: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, 2nd 336. IOS Press, Amsterdam (2021)
4. Brglez, F., Fujiwara, H.: A neutral netlist of 10 combinational benchmark circuits. In: Proceedings of IEEE International Symposium Circuits and Systems, June 1985, pp. 695–698 (1985)

5. Chan, S.M.: Just a pebble game. In: 2013 IEEE Conference on Computational Complexity, pp. 133–143. IEEE (2013)
6. Chan, S.M., Lauria, M., Nordstrom, J., Vinyals, M.: Hardness of approximation in pspace and separation results for pebble games. In: 2015 IEEE 56th Annual Symposium on Foundations of Computer Science, pp. 466–485. IEEE (2015)
7. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
8. Gidney, C.: Spooky pebble games and irreversible uncomputation (2019). https://algassert.com/post/1905
9. Gilbert, J.R., Lengauer, T., Tarjan, R.E.: The pebbling problem is complete in polynomial space. In: Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing, pp. 237–248 (1979)
10. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, pp. 212–219 (1996)
11. Komarath, B., Sarma, J., Sawlani, S.: Reversible pebble game on trees. In: Xu, D., Du, D., Du, D. (eds.) COCOON 2015. LNCS, vol. 9198, pp. 83–94. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21398-9_7
12. Kornerup, N., Sadun, J., Soloveichik, D.: The spooky pebble game. arXiv preprint arXiv:2110.08973 (2021)
13. Li, M., Tromp, J., Vitányi, P.: Reversible simulation of irreversible computation. Physica D **120**(1–2), 168–176 (1998)
14. Meuli, G., Soeken, M., Roetteler, M., Bjorner, N., De Micheli, G.: Reversible pebbling game for quantum memory management. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 288–291. IEEE (2019)
15. Nordström, J.: New wine into old wineskins: a survey of some pebbling classics with supplemental results (2015). https://www.csc.kth.se/~jakobn/research/PebblingSurveyTMP.pdf
16. Park, D.K., Petruccione, F., Rhee, J.K.K.: Circuit-based quantum random access memory for classical data. Sci. Rep. **9**(1), 3949 (2019)
17. Peikert, C.: He gives C-sieves on the CSIDH. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12106, pp. 463–492. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45724-2_16
18. Quist, A.J.: Spooky pebble game. GitHub repository (2023). https://github.com/Quist-A/spooky-pebble-game
19. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science, pp. 124–134. IEEE (1994)
20. Soeken, M., Roetteler, M., Wiebe, N., De Micheli, G.: Design automation and design space exploration for quantum computers. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pp. 470–475. IEEE (2017)