# Solving Battleship Puzzles using Grover's Algorithm

Jan Adriaan van den Bos[1], Timo Hoogenbosch[1] and Borislav Semerdzhiev[2] *

[1]Faculty of Applied Sciences, Delft University of Technology
[2]Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology

**Abstract**

*Quantum computing can in theory be used to solve NP-hard problems more efficiently than any classical computer possibly can. This paper focuses on implementing the quantum algorithm known as Grover's algorithm to solve the famous puzzle called 'battleships'. Grover's algorithm is implemented by creating a so called oracle function that checks if a trial solution satisfies all criteria of a correct solution. The algorithm was found to be working on a simulator. However, to be able to execute it on a real quantum computer a high quantum volume is needed. At the time of writing this report there is no quantum computer available with the required quantum volume. This problem could be overcome in the future by introducing quantum error correction and/or higher fidelity gates. In order to mitigate the shortcomings of the current available hardware for running simulations, we have additionally introduced a few optimizations that aim to reduce the number of used qubits, and hence, bring forth an exponential speed up.*

## Introduction

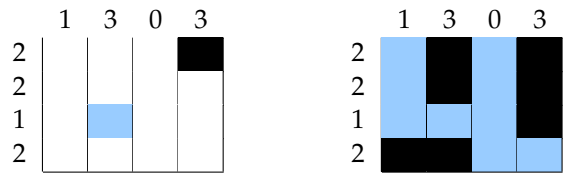
Quantum computers have been theorized to offer a large speed advantage over classical computers for certain algorithms [1][2][3]. Quantum algorithms, such as the well known Shor's algorithm [4] could change the world drastically. They can be of great use in for example cyber security, materials and pharmaceuticals research, banking and finance, and advanced manufacturing [5]. However, putting this quantum advantage to use has been difficult due to the shortcomings of existing hardware.

For a future universal quantum computer to be useful for practical applications, classical problems need to be able to be transformed so that they become solvable by a quantum algorithm. This paper aims to show an example of how a classical problem can be mapped onto a quantum algorithm that, in theory, could beat a classical algorithm if run on a sufficiently powerful quantum computer. The problem explored in this paper is finding solutions to a puzzle called *Battleships*. Multiple versions of the puzzle exist, and the most common version of this puzzle is described on *https://www.puzzle-battleships.com/*. However, in this paper, a simplified version was used. The rules of the puzzle are illustrated by Figure 1 and are as follows:

1. All cells should be filled with water or a (part of a) ship.
2. The numbers on the sides indicate the number of ship tiles in their respective row/column.
3. Ships may not touch, not even diagonally.

*Battleships* is a game that is easy to understand but can be hard to solve. Finding a solution is a so-called NP-complete problem, meaning that a classical computer cannot solve it efficiently, but a quantum computer could offer significant speed-ups. In this paper, we have attempted to write and execute an algorithm that solves the puzzle as described above. A simplified version was chosen to make the quantum algorithm easier to implement. It was also chosen to work with a smaller grid due to hardware limitations and to limit simulation times.

The reader is assumed to have basic knowledge of quantum computing, boolean algebra, and graph theory, and to be familiar with big-O notation.




**(a)** *Starting position.* **(b)** *Solution.*

**Figure 1:** *Simplified version of Battleships: A starting position and the solution.*

*

| Name | Student number |
|---|---|
| Jan Adriaan van den Bos | 5172128 |
| Timo Hoogenbosch | 5528135 |
| Borislav Semerdzhiev | 5578043 |

**Published:** January 31, 2024

# Theory

## Grover's Algorithm

Many computational problems can be classified as a so-called *search problem*, where, to solve it, one needs to find a solution in the set of all possible candidates. For some problems clever algorithms exist, and for others, one has no choice but to check all possible candidates. This strategy is called *brute-force search*. One subset of such problems is the set of NP-complete problems, of which *Battleships* is a member [6]. One property of NP-complete problems is the ability to check whether a solution is correct in polynomial time. This can be seen intuitively for *Battleships*: by checking the game's rules, one can check whether a solution is correct. How long this takes depends on the number of rows/columns $n$ like $O(n^2)$, which is polynomial.

Classical computers can easily perform brute-force search, but the time it takes to complete scales with the number of candidates $N_c$ to check, so the computational complexity is $O(N_c)$. Quantum computers provide a way to perform such a search much faster, using an algorithm discovered by Lev Grover [7]. The time needed to complete Grover's algorithm scales with the square of the number of candidates, so $O(\sqrt{N_c})$, which is significantly better for a large $N_c$. In the following section, we provide a review of Grover's algorithm. For a more complete explanation, please refer to other sources, such as [8, Ch.8].

The algorithm accomplishes its speed-up by operating on a superposition of all candidate solutions [8]. $N_c$ solutions are encoded by $n = \log_2 N_c$ qubits, which are placed in a superposition of all candidate solutions by applying Hadamard gates to all:

$$H^{\otimes n} \left|0 \overset{n}{\cdots} 0\right\rangle = \sum_{x \in \{0,1\}^n} \frac{1}{\sqrt{N_c}} |x\rangle , \tag{1}$$

where $|x\rangle$ is used to denote a candidate solution. The algorithm then requires a special gate $U_f$, called the oracle, which flips the phase of correct solutions and leaves incorrect solutions unaffected. Mathematically [8] this can be expressed as

$$U_f |x\rangle |a\rangle = |x\rangle |a \oplus f(x)\rangle , \tag{2}$$

where $|a\rangle$ is a required ancillary qubit whose function we will come back to, and where the function $f(x)$ checks whether a solution is correct: $f(x) = 0$ for incorrect solutions and $f(x) = 1$ for correct solutions. The desired phase flip of the correct solution can be achieved by setting $|a\rangle = |-\rangle$:

$$|x\rangle \frac{|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle}{\sqrt{2}} = (-1)^{f(x)} |x\rangle |-\rangle \tag{3}$$

The amplitude of the phase-flipped solution can now be amplified using the so-called *diffusion operator* [8]

$$D = H U_{0^\perp} H, \tag{4}$$

where

$$U_{0^\perp} : \begin{cases} |x\rangle \mapsto -|x\rangle , & x \neq 0 \\ |0\rangle \mapsto \quad |0\rangle \end{cases} . \tag{5}$$

Each repetition of the oracle operator followed by the diffusion operator increases the amplitude of the correct solutions, up until an optimum. The ideal number of repetitions $t$ is given by

$$t = \left\lfloor \frac{\pi}{4 \arcsin(1/\sqrt{N_c/N_s})} - \frac{1}{2} \right\rfloor \overset{N_c \to \infty}{\approx} \left\lfloor \frac{\pi}{4} \sqrt{N_c/N_s} \right\rfloor \tag{6}$$

where $N_c$ is the number of candidate solutions, $N_s$ is the number of correct solutions, and where taking the limit of $N_c \to \infty$ produces the promised time-complexity [8]. It should be noted that iterating more than the ideal number of times causes the amplitude of the correct solutions to start decreasing again.

## Quantum Volume

The performance of a quantum computer and its usability for real-world applications depends on many factors, such as the number of qubits, gate fidelity, and coherence times. Because of this, it proves difficult to readily compare quantum computers. One way to numerically compare the performance of two quantum computers is using quantum volume. The most common definition in use today was proposed by [9], of which we will provide a rough explanation.

Suppose we have a quantum computer that can reliably execute circuits with a depth (i.e. the longest path) of up to $n$ gates on $n$ qubits, so-called square circuits. The quantum volume is then given by

$$V_Q = 2^n. \tag{7}$$

The higher the quantum volume, the harder it is to simulate the circuit on a classical computer, and the more useful that quantum computer is at outperforming its classical counterpart [10].

# Methodologies

## Simulation tools

To build quantum circuits, the Qiskit software library was used, and simulations were performed using the Qiskit Aer simulators [11], specifically the *matrix product state* simulation backend. This way of simulating quantum states is an efficient way of simulating many-qubit systems with few entangled qubits [12].

## Checking algorithm correctness

The puzzle shown in Figure 2 was chosen as a test for the algorithm because it is the smallest board that can demonstrate the ability to filter out solutions where ships are touching.

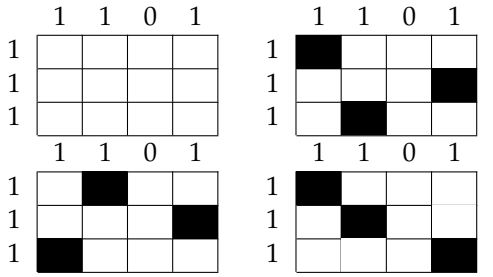



**Figure 2:** ***Top left*** *Starting position.* ***Top right & bottom left*** *Two correct solutions.* ***Bottom right*** *Wrong solution: matches numbers on the side, but ships are touching.*

## Benchmarking against quantum hardware

It was attempted to run the algorithm against the $2 \times 2$ puzzle from Figure 3 on a real quantum computer. The computer used for this was the 127-qubit IBM quantum computer, which is based on IBM's Eagle architecture. Its specifications are listed in the Appendix.


**(a)** *Starting position.* **(b)** *Solution.*

**Figure 3:** *The starting position and solution of the puzzle that was used to compare the performance of the simulator against the quantum computer.*

# Results

This section will first explain the workings of the algorithm, after which it will explain which optimizations were applied. It will also show an example of the algorithm working correctly on the simulator, and finally, it will show the result of trying to run the algorithm on real quantum hardware.

An implementation of the algorithm can be found on `https://t.ly/SpBP5`.

## Explanation of the Algorithm

The algorithm is explained in a top-down fashion, meaning that we will start by looking at the algorithm as a whole and gradually introduce more details. The general structure of the algorithm is illustrated by

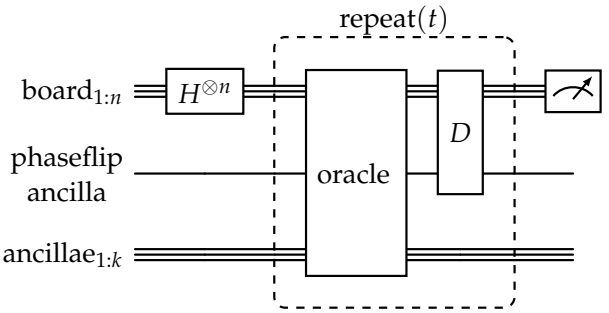



**Figure 4:** *An overview of the algorithm. The board is placed in a superposition of all candidate solutions using the $H^{\otimes n}$ operator. Then the oracle and diffusion (D) operators are repeated t times, where t is given by Equation 6. Lastly, the state of the board is measured. The number of ancillae k is given by Equation 13.*

Figure 4. The state of an $N \times M$ board is represented by an $n$-qubit register, where

$$n = N \cdot M \tag{8}$$

and where each square is represented by a qubit that is either in state $|1\rangle$, meaning it is filled by a ship, or $|0\rangle$, meaning it is filled by water. A single qubit, called the phaseflip ancilla, plays the role of the $|a\rangle$-qubit from Equation 2. A register of $k$ general ancillae is used as helper qubits during calculation. Here $k$ is defined further down by Equation 13 in the section *Optimization considerations*, since explaining how this equation is derived requires knowing how the algorithm works.

The board is placed in a superposition of all candidate solutions using the operation from Equation 1. Then the following gates are repeated $t$ times: first, the oracle operator flips the phase of correct solutions, and then the diffusion operator amplifies the amplitude of these solutions. Afterwards, the state of the board register is measured.

The implementation of the diffusion operator is illustrated by Figure 5. The implementation of the oracle operator is illustrated by Figure 6 and is explained in more detail in the coming sections. A recurring theme in the implementation of the oracle is that operators acting on the ancillae need to be reversible and that they always need to be followed by their transpose to reset to ancillae back to $|0\rangle$. Simply applying a reset operation does not work, since general reset operations involve a measurement and would collapse the superposition state.

The *check board* operator from the oracle is illustrated by Figure 7 and implements two checks to validate whether a solution is correct. These two checks are 1. checking whether the number of filled cells in a row/column matches the numbers at the edge of the board, using the *count check* operator, and 2. checking whether ships are touching or not, using

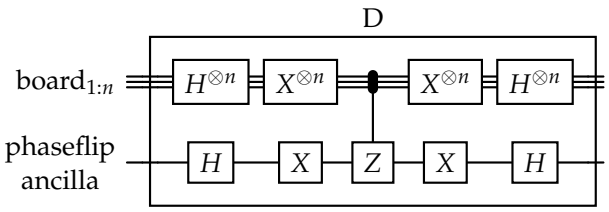



**Figure 5:** *The diffusion operator D [8] amplifies the amplitudes of states with a negative phase relative to the other states.*

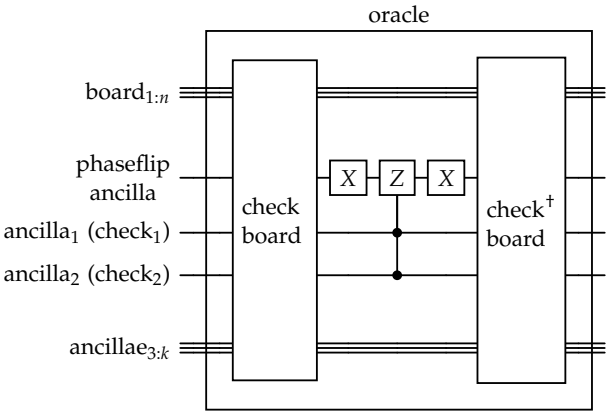



**Figure 6:** *High-level overview of the oracle operator. Two of the ancillae are separated and are used to store the result of two checks, performed by the* check board *operator, which state whether a solution is correct. The other $k-3$ ancillae are used as helper bits during calculation. Here k is given by Equation 13. At the end of the oracle, the transpose of the* check board *operator is applied to reset the state of all ancillae back to $|0\rangle$. Between the two* check board *operators, the phase flip of correct solutions is performed like explained in Equations 2 and 3. Here $f = check_1 \wedge check_2$.*

the *ship touch check* operator. The implementations of these two checks are explained in the coming two sections.

**Checking number of ship cells in a row or column** This criterion was checked by counting the number of 1s (representing ship squares) for each row and column and comparing the result to the number on the edge of the grid. This check is implemented by the *count check* operator, which is illustrated by Figure 8. To store the counter, we need to allocate

$$l_c = \lceil \log_2 \min(N, M) \rceil + 1 \tag{9}$$

qubits, and to store for every row/column whether the number on the side matches, we need to allocate

$$l_r = N + M \tag{10}$$

qubits.

**Checking proximity of ships** The condition that no ships can be touching can be simplified to a straightforward check of whether or not two ships are located next to each other, either on some of the main diagonals or on some of the anti-diagonals. To perform this check we separately look at the elements of every diagonal and check whether or not two consecutive qubits are in state $|1\rangle$, which is illustrated by Figure 11. To achieve this, we store the AND between every two consecutive qubits on a diagonal in a *diagonal neighbor* register for each diagonal. To store this register we need to allocate

$$p_n = \min(N-1, M-1) \tag{11}$$

qubits. After negating every element in this register, we perform a multiple controlled X gate using all those values to check if all of them are in the state $|1\rangle$ and set the corresponding elements of a *diagonals correct* register to $|1\rangle$ if true and $|0\rangle$ otherwise. To store the *diagonals correct* register we need to allocate

$$p_c = 2N + 2M - 6 \tag{12}$$

qubits, one for each diagonal: both main diagonals and anti-diagonals. Finally, performing a multiple-controlled X-gate from all the values in the *diagonals correct* register, we set the $check_2$ qubit to $|1\rangle$ if all the values are in state $|1\rangle$. The complexity of this part of the algorithm is $O(N \cdot M)$ and uses $2N + 2M + \min(N-1, M-1) - 6$ additional qubits.

## Optimization considerations

While writing the algorithm, we optimized for the number of qubits. This is because the computational time and space complexity of the simulation of quantum systems scales with the number of qubits. By keeping the number of qubits as low as possible, we were able to simulate larger boards.

**Reusing qubits in the oracle** To reduce the number of used qubits and to decrease the dimensionality of the state vector when running simulations, we make use of the fact that the qubits used for the *count check* can be reused for the *ship touch check*. In this way, provided we have unentangled them and reset them to the $|0\rangle$ state, we can significantly reduce the number of total qubits. In order to check if the number of ships is correct for each row and column we need a total of $\log_2 \max(N, M) + N + M$ qubits, and in order to check if ships are touching we need $2N + 2M + \min(N-1, M-1) - 6$ qubits. Therefore, by taking the maximum out of those two numbers we can solve both problems without introducing overhead. Additionally, we need 2 qubits to store the outcome of each check. The only downside of using this approach is that we need to effectively run the algorithms in reverse (run the operator transpose) to reset the qubit registers. However, while running simulations we concluded that using more qubits would affect the performance more detrimentally compared to introducing more operations.

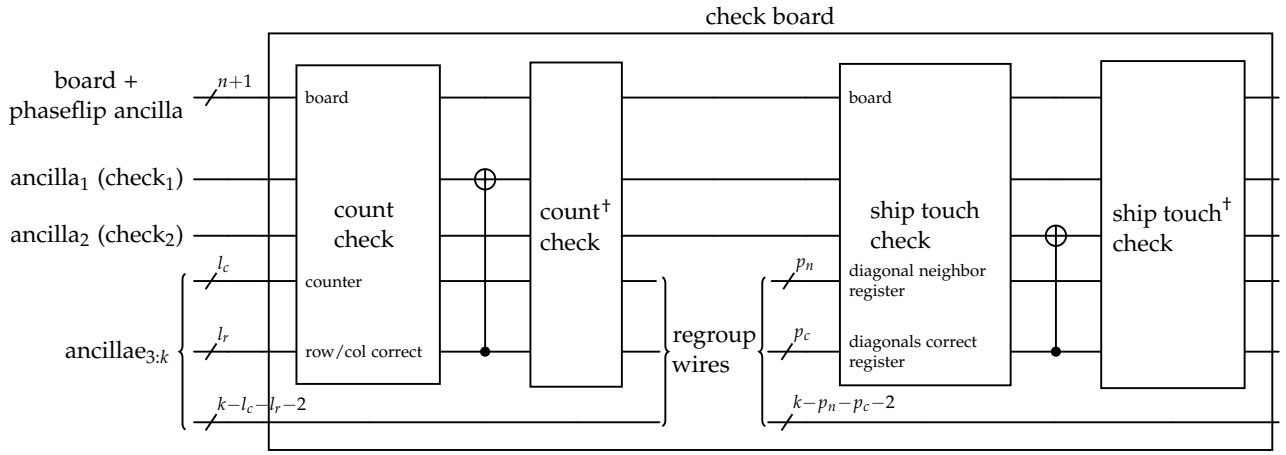



**Figure 7:** *Implementation of the* check board *operator. The operator sets the value of the two check qubits to* $|1\rangle$ *if a solution passes two checks: 1. if the number of ship tiles in a row/column matches the numbers on the side of the board, implemented by the* count check *operator, and 2. if ships are touching, implemented by the* ship touch check *operator. The ancillae are grouped in a way to illustrate their respective functions within the two check operators. In between the two checks, the wires are regrouped because the two checks require ancilla registers of different sizes. The function of the various ancilla registers is explained in the check operators' relative sections. The numbers of wires in the busses (notated using a "/") are given by Equations 9 through 13.*

In conclusion, the total number of ancillary qubits the oracle uses is:

$$\begin{aligned} k = \max( & \\ & \log_2 \min(N, M) + N + M, \\ & 2N + 2M + \min(N-1, M-1) - 6 \\ & ) + 2 \end{aligned} \tag{13}$$

**Preliminary reduction of the number of board qubits using a classical algorithm** It turns out that if we ignore the rule for ships touching, the problem is not NP-complete anymore and can be solved in polynomial time by representing the array of row and column numbers as a graph, as illustrated by Figure 12, and finding the minimum cut. To build this graph connect the source to nodes 1 to N, representing the rows of the board, and the sink to nodes 1 to M representing the columns of the board. Those edges should have capacity equal to row[i] and column[j] respectively for each row i and column j. Finally, for each cell (i, j) on the board, create a new node and create an edge from row[i] to this node with weight 1, and an edge from this node to column[j] also with weight 1. To find the minimum cut we use Dinitz's maximum flow algorithm [13], which also gives us the minimum cut in the graph, and if the flow that passes through the network is equal to the sum of the ship counts in the row/column arrays, then there is at least one solution that satisfies the row/column number requirement. Furthermore, we can also check if a cell has the same value for all valid solutions. To do this we can perform two checks. If we want to know if the value for a given cell$(i, j)$ is always $|0\rangle$, then we can reduce row$[i]$ and col$[j]$ by 1 and set the edge between $R_i$ and $R_iC_j$ to 0. Now we run Dinitz's algorithm again and check if the current maximum flow $+ 1$ is not equal to the previous answer. If it is not, then the value for the current cell is always 0. To check if cell$(i, j)$ is always 1, we simply set the edge between $R_i$ and $R_iC_j$ to 0 and run the maximum flow algorithm on the modified graph. If the new maximum flow is not equal to the original flow, then the value of the cell is always 1.

Dinitz's algorithm for finding the maximum flow in a graph runs in $O(|V| + |E|^2)$ where V is the set of nodes and E is the set of edges. Hence, this pre-computation runs in $O(N \cdot M \cdot (|V| + |E|^2))$ since we need to test every cell on the board. Due to the way we built our graph, |V| is equal to $N \cdot M + N + M$ and |E| is equal to $2 \cdot N \cdot M + N + M$. Therefore for a board with N=M, we get total runtime of $O(N^6)$.

To test how many qubits we would save due to this pre-computation, we simulated all possible $4 \times 4$ board configurations complying with the row/column number condition. From Figure 13 we can conclude that doing this pre-computation saves us, on average, about 70% of the qubits we would normally need. And for quite a few configurations of the board, we would find the only possible solution straight away.

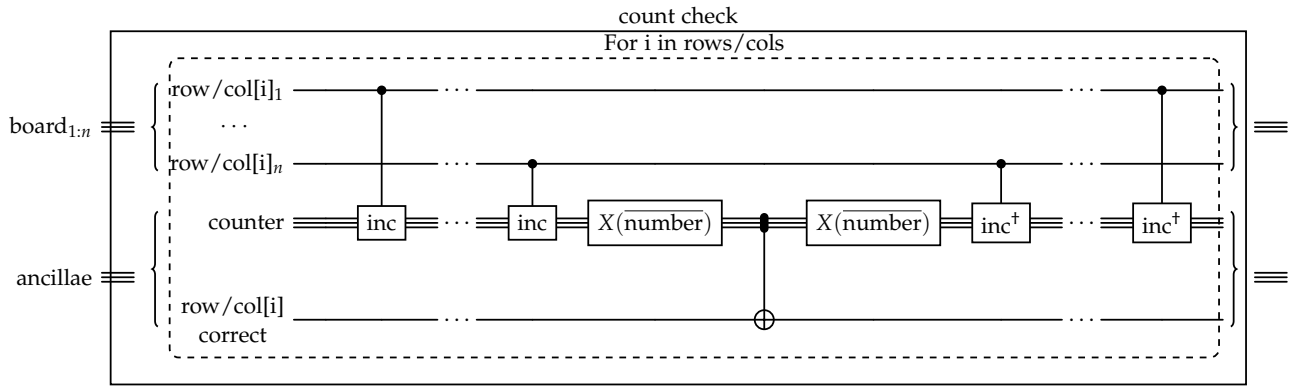



**Figure 8:** *Implementation of the* count check *operator. For each row/column, the number of ship tiles is counted by applying controlled increment gates to the* counter *register for every qubit in that row/column. Then the* $X(\overline{\text{number}})$ *operator is applied to the* counter*. Here* number *refers to the binary representation of the number on the side of the board for that row/column, and the overline represents the fact that the binary number needs to be inverted (a NOT operation applied). If the number of ship tiles in the row/column is equal to the number of the side of the board, then the* counter *register contains only 1s. Based on this the ith element of the* row/col correct *register is set. The* row/col correct *is returned to the outside of the operator. The definitions of the* inc *and* $X(number)$ *operators are given in Figure 9 and 10.*

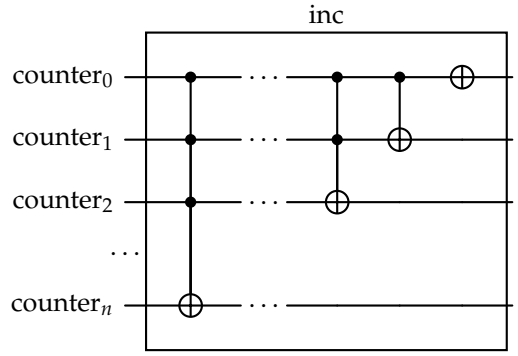



**Figure 9:** *The increment operator increments the binary representation of the register it is applied to.*

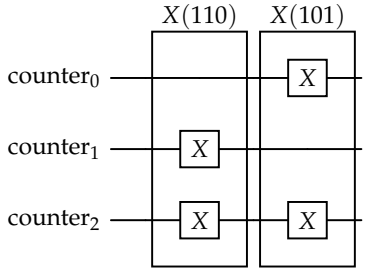



**Figure 10:** *The* $X(\text{number})$ *gate applies X-gates conditionally on the qubits of the supplied register, based on the value of the corresponding bit in the binary representation of the number.*

## Testing the algorithm using the simulator

The test whether the algorithm was working correctly, it was run on the problem from Figure 2. The result is illustrated by Figure 14. The two correct solutions show clear peaks, while all incorrect solutions, including the one where the only error is touching ships, have very small peaks.

## Running on quantum hardware

The algorithm was run both on a simulator and a real IBM quantum computer for the problem from Figure 3. The results are displayed in Figure 16. It is clear from the results that the algorithm works on the simulator, but fails on the real hardware. This is likely due to the quantum volume of the computer. The quantum volume of the computer used in this experiment is $2^7 = 128$, meaning it cannot execute a circuit larger than 7 by 7 [9]. Our algorithm requires a much larger quantum volume. This shows that it is not the amount of qubits, but the quantum volume that limits the performance of the quantum computer in this case. Further research could focus on quantum error correction and better gate fidelities to make this algorithm executable on a quantum computer.

# Discussion

## Possible further optimizations

**Reducing the number of board qubits using the ships touching condition** If we decide to filter out solutions that do not comply with the third rule beforehand, we can greatly reduce the number of qubits needed to represent the board. However, difficulty presents itself in how to do the mapping of the state to a valid solution. We came up with the following way to achieve that. First, we decide to only eliminate solutions that have ships touching diagonally only in one direction(so between cells $(i, j)$ and $(i + 1, j - 1)$). Now, we would like to have separate qubits representing a valid sequence of 0s and 1s for every diagonal (no two consecutive 1s). For ex-

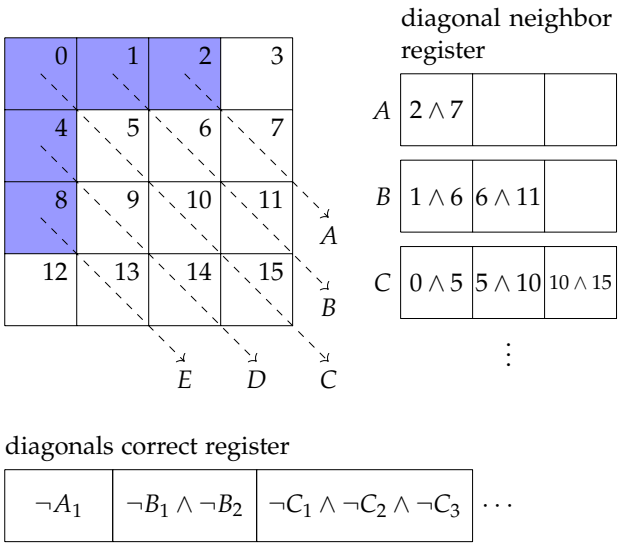



**Figure 11:** *Action of the* ship touch check *gate for the main diagonals (going top-left to bottom-right). For every main diagonal, the* diagonal neighbor register *is filled, and then, using the values stored in that register, the corresponding element of the* diagonals correct register *is filled. The procedure is repeated for the anti-diagonals (going bottom-left to top-right) and the result is appended to the* diagonals correct register*. At the end of the procedure, the* diagonals correct register *contains for every diagonal whether it is correct or not.*

ample if the current diagonal has 3 elements, there would be the following valid sequences: 000, 001, 010, 100, and 101. This way, we have reduced the number of possible sequences from $2^3 = 8$ to just 5. Finally, we concatenate those qubits representing the current diagonal, for each diagonal, in order from the 1st one to the last one. Now if we need to restore the board using those qubits, since we know which qubits correspond to each diagonal, we can do the following - map each number to a valid sequence of 0s and 1s (000 $\mapsto$ 000, 001 $\mapsto$ 001, 010 $\mapsto$ 010, 011 $\mapsto$ 100, 100 $\mapsto$ 101) and by checking if the current superposition of qubits is equal to the key in any of those mappings, we can retrieve the values (if the current cell we want to retrieve is a 1 in a given key-value element, first do a controlled X-gate between the diagonal qubits and the chosen key, after that do a multiple controlled X-gate and using the result flip the current cell to $|1\rangle$). The number of valid sequences of size $N$ corresponds to the $N+2$-th Fibonacci number (first few Fibonacci numbers are 1, 1, 2, 3, 5, Fib$[n-1]$ + Fib$[n-2]$, ...) and this sequence grows much slower than $2^N$. The number of qubits required to represent an $N \times N$ board can be calculated using $\sum_{n=1}^{N} 2 \cdot \log_2 \text{Fib}[n+2] - \log_2 \text{Fib}[N+2]$. Figure 15 graphs how this method scales up with larger boards. We can conclude that while we save virtually no qubits for small $N$, as $N$ grows larger the reduction factor converges to a 3rd of the qubits we need to use. However, since hardware limitations currently allow us to only work with boards as large as 4x4, we decided not to use this method of reducing the board state. Still, it is worth mentioning for future usage.

## Conclusions and Outlook

Grover's Algorithm can be implemented to solve the puzzle called *Battleships*. The final complexity of the algorithm comes to $O(2^{\frac{N \cdot M}{2}} \cdot N \cdot M)$ and the number of required qubits correspond to $N \cdot M + \max(\log_2 \max(N, M) + N + M, 2N + 2M + \min(N - 1, M - 1) - 6) + 3$. As we can see, as $N$ and $M$ grow, the number of qubits required to store the board state overshadows the number of qubits required for performing the oracle checks. Therefore, the additional algorithm we devised for eliminating the qubits for the cells we know the values of beforehand, can greatly reduce the number of qubits depending on the row and column configuration. Although our algorithm brings an exponential performance improvement, its usability is still limited as it can not be run on a quantum computer yet due to the limited quantum volume of the available hardware. This problem could be overcome in the future by introducing quantum error correction and/or higher fidelity gates.

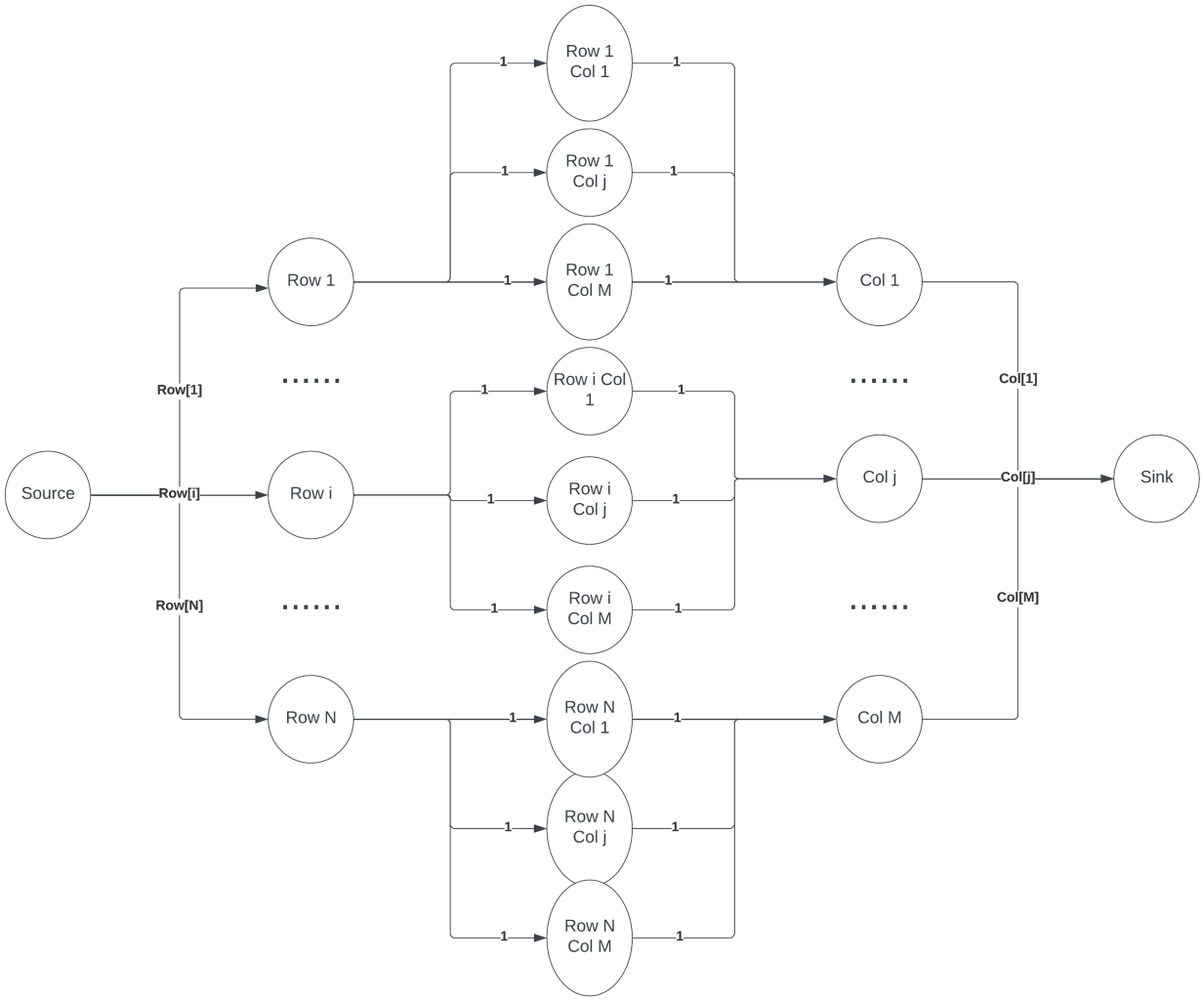



**Figure 12:** *Representation of the row and column array as a network graph in which the minimum cut represents the number of ships we can place while satisfying the rule of the numbers on the side of the board.*

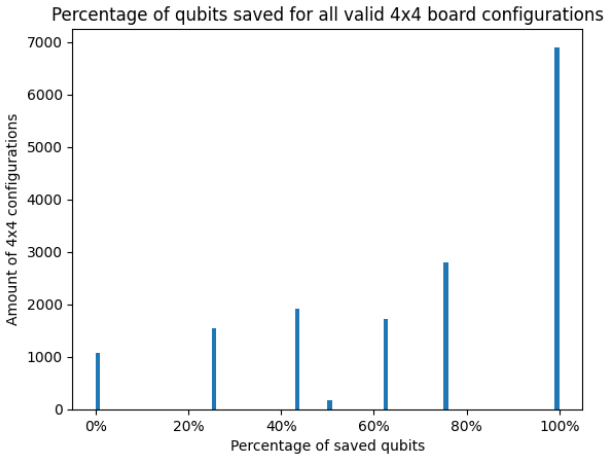



**Figure 13:** *What percentage of the qubits (which we would normally need to store the board state) we save if we were to preliminarily deduce values of cells we can be sure about for all possible* $4 \times 4$ *configurations that comply with the second rule of the puzzle.*

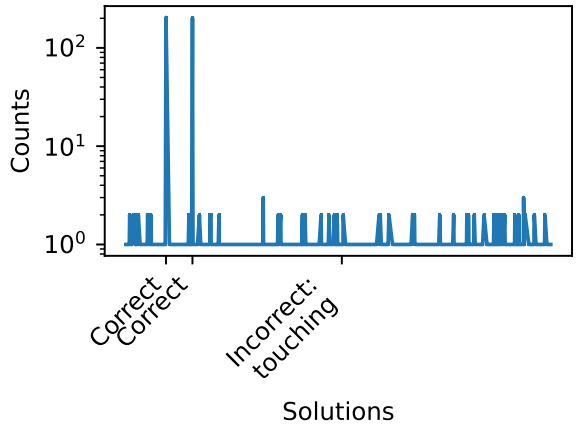



**Figure 14:** *Result of the test of the algorithm for the problem from Figure 2. The y-axis shows counts out of 1024.*

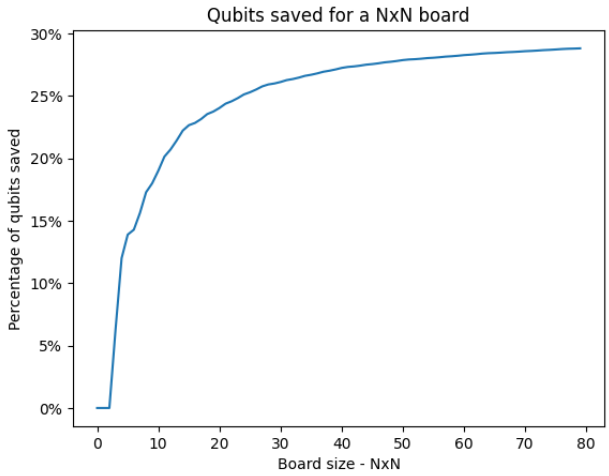



**Figure 15:** *The percentage of qubits we can save by representing the board diagonals only as correct sequences of 0s and 1s.*

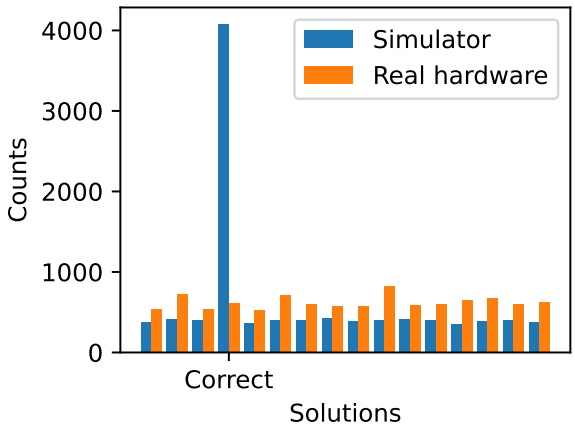



**Figure 16:** *Comparison of the performance between simulator and real hardware for the problem from Figure 3. Counts are out of 10 000.*

## References

[1] John Preskill. "Quantum computing and the entanglement frontier". In: (2012). Publisher: arXiv Version Number: 3. DOI: 10.48550/ARXIV.1203.5813. URL: https://arxiv.org/abs/1203.5813 (visited on 01/24/2024).

[2] Richard P. Feynman. "Simulating physics with computers". In: *International Journal of Theoretical Physics* 21.6 (June 1982), pp. 467–488. ISSN: 0020-7748, 1572-9575. DOI: 10.1007/BF02650179. URL: http://link.springer.com/10.1007/BF02650179 (visited on 01/24/2024).

[3] John Preskill. "Quantum Computing in the NISQ era and beyond". In: *Quantum* 2 (Aug. 6, 2018), p. 79. ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79. URL: https://quantum-journal.org/papers/q-2018-08-06-79/ (visited on 01/24/2024).

[4] Peter W Shor. "Introduction to quantum algorithms". In: *Proceedings of Symposia in Applied Mathematics*. Vol. 58. 2002, pp. 143–160.

[5] Francesco Bova, Avi Goldfarb, and Roger G Melko. "Commercial applications of quantum computing". In: *EPJ quantum technology* 8.1 (2021), p. 2.

[6] Merlijn Sevenster. "BATTLESHIPS AS A DECISION PROBLEM". In: *ICGA Journal* 27.3 (Sept. 1, 2004), pp. 142–149. ISSN: 13896911, 24682438. DOI: 10.3233/ICG-2004-27303. URL: https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/ICG-2004-27303 (visited on 01/19/2024).

[7] 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 '96*. the twenty-eighth annual ACM symposium. Philadelphia, Pennsylvania, United States: ACM Press, 1996, pp. 212–219. ISBN: 978-0-89791-785-8. DOI: 10.1145/237814.237866. URL: http://portal.acm.org/citation.cfm?doid=237814.237866 (visited on 01/19/2024).

[8] Phillip Kaye, Raymond Laflamme, and Michele Mosca. *An introduction to quantum computing*. 1. publ. Oxford: Oxford University Press, 2007. 274 pp. ISBN: 978-0-19-857000-4 978-0-19-857049-3.

[9] Andrew W. Cross et al. "Validating quantum computers using randomized model circuits". In: *Physical Review A* 100.3 (Sept. 20, 2019), p. 032328. ISSN: 2469-9926, 2469-9934. DOI: 10.1103/PhysRevA.100.032328. URL: https://link.aps.org/doi/10.1103/PhysRevA.100.032328 (visited on 01/24/2024).

[10] Olivia Di Matteo. *Quantum volume*. https://pennylane.ai/qml/demos/quantum_volume/. Date Accessed: 2024-01-24. Nov. 2020.

[11] Qiskit contributors. *Qiskit: An Open-source Framework for Quantum Computing*. 2023. DOI: 10.5281/zenodo.2573505.

[12] Guifré Vidal. "Efficient Classical Simulation of Slightly Entangled Quantum Computations". In: *Physical Review Letters* 91.14 (Oct. 2003). ISSN: 1079-7114. DOI: 10.1103/physrevlett.91.147902. URL: http://dx.doi.org/10.1103/PhysRevLett.91.147902.

[13] Yefim Dinitz. "Dinitz' Algorithm: The Original Version and Even's Version". In: Jan. 2006, pp. 218–240. ISBN: 978-3-540-32880-3. DOI: 10.1007/11685654_10.

[14] David C. McKay et al. *Benchmarking Quantum Processor Performance at Scale*. Nov. 10, 2023. arXiv: 2311.05933[quant-ph]. URL: http://arxiv.org/abs/2311.05933 (visited on 01/29/2024).

[15] Andrew Wack et al. *Quality, Speed, and Scale: three key attributes to measure the performance of near-term quantum computers*. Oct. 28, 2021. arXiv: 2110.14108[quant-ph]. URL: http://arxiv.org/abs/2110.14108 (visited on 01/29/2024).

# Appendices

## Specification of the IBM quantum computer

A summary of the specifications of the IBM quantum computer that was used is given in Table 1. A per-qubit specification can be found on https://quantum.ibm.com/services/resources?tab=systems&system=ibm_osaka.

**Table 1:** *Specifications of the IBM Osaka quantum computer. From https://quantum.ibm.com/services/resources?tab=systems&system=ibm_osaka.*

| Parameter | Value |
|---|---|
| Qubits | 127 |
| EPLG[14] | 2.8% |
| CLOPS[15] | $5 \cdot 10^3$ |
| Processor type | Eagle r3 |
| Basis gates | ECR, ID, RZ, SX, X |
| Median ECR error | $7.503 \cdot 10^{-3}$ |
| Median SX error | $2.396 \cdot 10^{-4}$ |
| Median readout error | $2.490 \cdot 10^{-2}$ |
| Median $T_1$ | 285.25 $\mu$s |
| Median $T_2$ | 152.29 $\mu$s |