Inverse Problems, Constraint Satisfaction, Reversible Logic, Invertible Logic and Grover Quantum Oracles for Practical Problems

Marek Perkowski

Department of Electrical and Computer Engineering, Portland State University, Portland, OR 97207, USA
mperkows@ee.pdx.edu

Abstract. It is well-known that the “Unsorted Database” quantum algorithm by Grover gives quadratic speedup to several important combinatorial and enumerative problems, such as: SAT, Graph Coloring, Maximum Cliques, Traveling Salesman and many others. Recently, quantum programming languages such as Quipper start to be used to design, verify and simulate practical quantum algorithms for important problems in Quantum Machine Learning. So far, however, no methodologies have been created to program Grover Oracles for particular classes of problems. In contrast, such methodologies have been already created for classical Constraint Satisfaction Problems. The goal of this invited talk is to show results of some initial research towards creating systematic methodologies to program quantum computers that solve search problems in Artificial Intelligence, Logic Design and Machine Learning. Our methods are based on unified oracle blocks for such problem representations as set partition algebra, cube calculus and optimal mappings. For instance, several important problems in CAD and Machine Learning can be solved using only two basic operations on set partitions; $P_1/C_2 P_2$ and $P_1/C_1 P_2$. Moreover, building oracles is the fundamental concept in the new approach to solve CSP proposed here and based on Invertible Logic introduced recently by Supriyo Datta and his team.

Keywords: Inverse problems · Oracles · Grover algorithm · Invertible logic

1 Introduction

There are two important and large classes of problems: Constraint Satisfaction Problems (CSP) and optimization problems. CSP problems are specified just by a set of constraints that the solution has to satisfy. To solve a CSP problem we want to find a solution that satisfies all the constraints (like for a robot passing a labyrinth without bouncing any wall). In optimization problems we want to find the solution that optimizes some cost function (like a robot driving from room A to room B in the labyrinth using minimum energy from its battery). Mathematical problem formulations such as Graph Coloring or Shortest Path are abstractions of many problems from real life. Mathematical optimization problems can be reduced to repeated applications of constraint satisfaction problems. Every next problem in the sequence is solved with added and modified constraints.
For instance, the Node Coloring Problem in a graph can be formulated as a problem of coloring nodes with some restricted number of colors such that every two nodes \( n_i \) and \( n_j \) that share the same edge \( e_{ij} = (n_i, n_j) \) obtain different colors. The chromatic number of the graph is the minimum number of colors used to color this graph. When we want to find the chromatic number of the graph (and the respective actual coloring) we can proceed as follows. We set some number \( K \) for which a correct coloring exists, this can be the number of nodes in the worst case. Next we exactly solve the constraint satisfaction problem for this graph, assuming that the graph is \( K \)-colorable. When we are able to color the graph with \( K \) colors, we guess that we can color the graph with \( K-1 \) colors. If we were able to do this, then we repeat coloring again as a new constraint satisfaction problem with \( K-2 \) colors, and so on. If we were not able to color the graph with \( K-2 \) colors but we were able to color it with \( K-1 \) colors then \( K-1 \) is the chromatic number of the graph. This simple principle of “reducing optimization problem to the repeated constraint satisfaction problem with some changed constraints” is applicable to very many, if not all, optimization problems. The key idea here is to formulate the problem as the Constraint Satisfaction Problem and next to generate the Oracle for this problem. If this oracle is realized in hardware as a circuit built especially for every problem instance, the solution can be very fast. Oracle can be realized as a quantum reversible circuit and used in Grover Algorithm. Here we will show that the oracle can be also built in standard binary logic or in the recently proposed Invertible Logic [5, 17]. To exercise Oracle in classical and quantum logic we need also a Generator that creates combinations to be verified by the Oracle. We call that the Oracle is exercised by the Generator.

This paper presents the universality and power of systematic creation of oracles that are next mapped to one of the three types of oracles: classical, quantum and invertible. Section 2 formulates the general idea of formulating oracles for CSP problems, used to solve both the decision and optimization problems. Section 3 presents how to solve this class of problems using classical Boolean logic. The oracle model introduced in this section is universal for quantum and Invertible Logic circuits. Section 4 explains briefly Grover Algorithm and reversible oracles for it. We illustrate how to modify the oracle in a succession of Grover Algorithm runs. Section 5 discusses how to modify or design the oracles realized in Invertible logic for this class of problems. In Sect. 6 we present how the logic programming language Prolog is used to design and verify oracles and especially the Invertible Logic oracles. Section 7 gives conclusions on this new approach to solve a large class of practical problems.

2 Classical Grover Oracles Versus Invertible Logic Oracles

Because optimization problems are reducible to CSP, let us concentrate on the Constraint Satisfaction Problem, which has by itself very many applications in logic design, logic, cryptography, robotics, Machine Learning and control. There is also some evidence that animals, and even bacteria, solve the constraint satisfaction problems by blind probabilistic mechanisms just to survive [29]. Let us present some simple CSP examples.
Graph coloring is a simple constraint satisfaction problem formulated as follows: Given is graph \( G \) with \( N \) nodes and \( E \) edges, edges being pairs of nodes \( e_{ij} = (n_i, n_j) \). Nodes are colored with function \( \text{COLOR}: N \rightarrow C \) where \( C \) is a set with \( K \) elements called colors. \( \text{COLOR}(n_1) = \text{red} \) means coloring node \( n_1 \) with color red. For every edge \( e_{ij} = (n_i, n_j) \) the constraint \( \text{COLOR}(n_i) \neq \text{COLOR}(n_j) \) which means that every two adjacent (neighbor) nodes have different colors. Correct coloring is one that the constraint is satisfied for every edge.

Find a solution to the following problem:

Is it possible to correctly color graph \( G \) with \( K \) colors? If yes, graph \( G \) is \( K \)-colorable, its chromatic number is \( K \) or less. The CSP algorithm constructively demonstrates the correct coloring and the user can easily verify the correctness of the coloring found. Note, that the above formulation is a decision version of graph coloring, not the optimization version. The answer is of type Yes/No.

Another CSP problem is Satisfiability or SAT Problem. Given is a formula \( F \) in some logic (possibly Boolean logic) and we ask “is this formula satisfiable?” Which means, “can we find a specific value for every variable that the formula \( F = 1 \) ?”. For instance, formula \( F(a, b) = (a + b) * (a' + b) * (a + b') * (a' + b') \) in Boolean logic is not satisfiable. But this formula is 3-satisfiable, which means that when we remove any single one of the four OR-terms from the product \( F(a, b) \) above, the formula would be satisfiable. For instance, formula \( F_1(a, b) = (a + b) * (a' + b) * (a + b') = ab \) for \( a = 1 \) and \( b = 1 \) the formula is satisfied. This fundamental problem has hundreds applications in real life engineering problems that can be reduced to it. Problems such as optimizing digital designs, and also practical life problems such as finding the best escape routes from some territory after Nuclear Plant disaster.

SAT Problem can be also reduced to CSP. Similarly, CSP problems can be reduced to SAT. One more problem of CSP type is a crypto arithmetic problem: \( \text{SEND} + \text{MORE} = \text{MONEY} \) in which we have to substitute digits 0–9 for letters S, E, N, D, M, O, R, Y in an unique mapping (a mapping function) such that the symbolic equation above is converted to a valid arithmetic addition on digits. This toy problem is a simplification of similar problems in cryptography, a research area with huge military and security impacts. The same way as the SAT Problem, graph coloring, or any CSP problem, this problem can be solved using an oracle, in our case a hardware oracle. For this kind of problems, the oracle is built from logic gates \( \text{AND}, \text{OR}, \text{NOT} \), logic blocks such as predicates \( (A = B) \) or \( (A > C) \) and arithmetic blocks such as adders and multipliers.

Building oracles in reversible logic is the fundament of quantum Grover algorithm. The oracle is problem-dependent and even problem-instance dependent. Quantum algorithms are circuits from quantum gates. A quantum oracle is the heart of Grover circuit, other gates in this circuit are always the same, easy and well-known. Practically, applying Grover Algorithm to a new problem means to design an oracle for the constraint satisfaction problem to be solved. When we know how to build the oracle in Boolean Logic and using standard arithmetic, we can convert the Boolean circuit of the oracle to a quantum oracle in reversible logic just by a conversion of Boolean gates and blocks to equivalent reversible logic gates such as Toffoli, Fredkin and Feynman and “reversible blocks” such as a quantum adder.
Reversible functions are mathematical functions which are one-to-one mappings of input vectors to output vectors and vice versa. If a function to be mapped is not reversible, which is frequently the case in oracles, it can be mapped to reversible gates and blocks but it requires ancilla qubits. Most Boolean Functions (single or multiple-output) are not reversible, but building quantum oracles we want to realize them with quantum gates that are reversible. This means that we need to add ancilla qubits initialized to 0 or 1, to be able to perform this mapping. Therefore, what we call “reversible gates” and “reversible blocks” are not reversible functions in mathematical sense because many of them require ancilla qubits. Thus they may correspond to mathematical Boolean functions that are not reversible, but allow the designer familiar with classical digital design of combinational circuit to use immediately his knowledge to build optimized and tricky quantum oracles [11, 15].

Let us make a strong point here, that the very idea of hardware oracle is much more than a Grover Oracle and quantum circuits. One can build a Boolean circuit of the oracle in FPGA [2, 26, 44] and find as the solution to this hardware oracle the input vectors which lead to satisfaction $F = 1$ observed on oracle’s output. Oracles are thus hardware devices to solve the inverse function problem that is known in many areas of mathematics and practical applications. Very little is published about using oracles in hardware with logic circuits that are different than quantum reversible oracles. Grover Algorithm [9] uses oracles built from various quantum reversible gates, plus blocks such as quantum adders or quantum comparators built from quantum gates. Grover algorithm gives a quadratic speedup over classical exhaustive search algorithms for the same problem. Although other quantum algorithms such as the Shor Algorithm [19] give exponential speedup, Grover Algorithm is very important because very many problems of big practical importance can be reformulated for Grover, and not that many problems can be reduced to Shor Algorithm. Several publications of various authors found solutions to important problems based on Grover Algorithm but the authors usually do not build these oracles from gates so that they cannot evaluate their practical complexity. They just show that this can be done by formulating for instance circuits as unitary matrices. In contrast, work of our PSU team designs the oracles in detail, bottom-up and using practical and experimentally verified “truly quantum gates” [23] or “quantum reversible” gates and blocks [3, 8, 11, 12, 15, 16, 21–24, 29, 32, 45].

When one realizes that the fundamental, most important idea in Grover Algorithm is to find the inverse function, one can realize that quantum is not the only technology in which we can build efficient oracles and obtain speedup of algorithms when compared to classical circuits, classical algorithms or parallel algorithms. One naive method would be to build the oracle in classical Boolean logic with standard Boolean gates and logical/arithmetical blocks. Next this oracle could be exercised (exhaustively or randomly) from input to output. Whenever the output equal value 1 is found, the state of the binary input vector gives the solution to the constraint satisfaction problem described by the oracle. Although this method was used in several designs [2, 26, 44], it is harmed by the very slow speed when solving problems of practical size, like those SAT problems encountered in industrial CAD applications. This is because for the Boolean function of $n$ variables in the worst case the oracle needs to be evaluated $2^n$ times (for each of its minterms representing the Boolean function that corresponds to this oracle). Minterms for a Boolean function of n variables are products of all n literals
of this function. True minterms are those for which value of the function is 1. Circuits with quantum oracles are better that classical Boolean circuits because they operate using quantum parallelism and quantum superposition - on all vectors being potential solutions at the same time (all minterms). Thus quantum oracle iterated sufficiently many times as part of the “Grover Loop” highly increases the probability of finding one of the solutions in a single measurement of all input qubits together with the output qubit of the Boolean function realized in the oracle. Grover Algorithm implemented in quantum circuits gives a quadratic speedup when compared to exhaustive classical circuit (algorithm) for the same problem.

Recently a new and very powerful concept was found in the area of recursive Deep Neural Networks (DNN) and this is the idea of Invertible Logic [5, 6, 17, 18, 20, 27]. This logic can be realized with magnetic spins [6], but it can be also emulated using FPGAs (sacrificing speed, FPGAs used mainly for verification). So far very few applications of this idea were published and they are reduced to adders, multipliers, number factorization [14] and one class of neural networks [10]. In contrast, here we present a unified methodology to solve CSP and optimization problems by building and exercising oracles that can be realized very similarly in Quantum and Invertible Logic.

Let us compare three types of logic realized in circuits:
1. Classical Boolean combinational logic
2. Reversible (permutative) logic as used in quantum oracles
3. Invertible Logic of Supriyo Datta et al.

Classical Boolean logic propagates signals from inputs to outputs. In inverse problems we want to find the input vectors with the given output value (like \( F = 1 \) here). Thus if we want to find solutions to non-trivial inverse problems about which we have no additional information (the proverbial “Unsorted Database” in Grover case) we have to go through all or many input combinations before we find the input vector for which \( F = 1 \). Creating the binary input vectors, the solution candidates for this procedure can be done with a sequential counter or with a random number generator (RNG) such as LFSR on the input to the oracle circuit.

For a motivating example, let us assume that the oracle circuit of function \( F_2 \) is just a tree of two-input AND gates with 64 inputs in total, but the function (or its circuit) is not known to the observer. To find this function, the oracle in classical logic would be evaluated \( 2^{64} \) number of times, which is an astronomical number. Grover Algorithm would evaluate the oracle “only” \( 2^{(64/2)} = 2^{32} \) times which may be also not practical. However, in Invertible logic in which one propagates the signals from output to inputs, the Invertible Logic method would need only one evaluation of the oracle realized with invertible gates. The main principle is that the Invertible Logic Gates can propagate signals in any direction. In Invertible Logic a two-input \( C = \text{AND}(A, B) \) gate with output fixed to \( C = 1 \) creates the input value “1” on both its inputs: \( A = 1, B = 1 \). Therefore in we start from output \( F_2 = 1 \) in the above 64-input tree of AND gates, value 1 will be propagated from the function \( F_2 \) output to previous levels of the tree and ultimately will create values 1 on all 64 inputs without backtracking and in one value propagation process. If one gives value 0 to the output of two-input AND invertible gate, on inputs the sequence \( (A, B) = (0, 0), (0, 1), (1, 0) \) appears in random order, because the values \( (A, B) = (0, 0), (0, 1), (1, 0) \) all create a 0 on the
output of classical AND gate. Thus if one gives value 0 on the function $F_2$ output in this example, the circuit realized in Invertible Logic will create sequentially (randomly and with repetitions) all $2^{64} - 1$ primary input that produce $F_2 = 0$. Concluding, Invertible Logic allows to find solutions to all kinds of CSP problems and for some of them it can find a solution faster than the quantum algorithm of Grover. Active research is on technologies in which this logic can be practically realized [1, 4–6, 17, 20, 25]. Next section will relate classical Boolean Oracles, Quantum and Invertible Logic oracles, and how to design them in an uniform way, methodologically and efficiently.

3 Classical Boolean Oracles

3.1 Cryptoarithmetic Example and Discussion of TWO + TWO = FOUR Oracle

Our first larger example is to build classical Boolean Logic Oracle [2, 44]. To evaluate the standard FPGA realization of a classical Boolean Logic Oracle we used the crypto-arithmetic problem TWO + TWO = FOUR. Each letter of this symbolic equation can be substituted with values 0–9. The purpose of our oracle is to find all possible solutions to this problem. In other words, we look for all binary input combinations to this oracle that satisfy all partial constraints. The research problems are: (1) how to build such oracle, (2) how to exercise this oracle from a Generator, (3) how to build the Generator, (4) How to select the best overall design of the Generator-Verifier system that would minimize the cost and maximize the speed.

Let us discuss few strategies how to solve this problem. Mathematically this problem can be formulated as follows. Given are sets $L = \{T, W, O, F, U, R\}$ and $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Find a mapping $L \rightarrow D$ such that $\forall l_1, l_2 \in L [\text{MAP}(l_1) \neq \text{MAP}(l_2)]$ and all arithmetic constraints of addition TWO + TWO = FOUR are satisfied. The naive solution dictated directly by the problem formulation is this. We create a tree [49] with all 10 combinations for node with letter T in first tree level, next for every node of the second level we create all 10 combinations for letter W, next the same is done for letter O and so on. The tree will have 6 levels and the branching factor for every level is 10. Thus we will have to check $10^6 = 1,000,000$ cases for arithmetic correctness in the Oracle. A better idea is to observe that in standard notation for addition formulas the value of F cannot be a 0. Moreover, it is obvious that it must be a 1. Therefore we have now not 6 letters but only 5 letters T, W, O, F, U, R for substitutions. We also know that F = 1, so now our tree will have 5 levels, with branching factor 9 where we select for every letter one of {0, 2, 3, 4, 5, 6, 7, 8, 9}. This leads to 59,049 cases of leafs of the tree, they are the mappings to be verified with use of arithmetic constraints in the Oracle. A better idea is to observe that in standard notation for addition formulas the value of F cannot be a 0. Moreover, it is obvious that it must be a 1. Therefore we have now not 6 letters but only 5 letters T, W, O, F, U, R for substitutions. We also know that F = 1, so now our tree will have 5 levels, with branching factor 9 where we select for every letter one of {0, 2, 3, 4, 5, 6, 7, 8, 9}. This leads to 59,049 cases of leafs of the tree, they are the mappings to be verified with use of arithmetic constraints in the Oracle. This simple common sense trick gave us a big gain in the size of the solution space, from $= 1,000,000$ to 59,049. Next observation is that while in the first level we branch for 9 letters, in the next level we should branch for 8, next for 7, etc. So in total the number of leaves of the tree will be $9\cdot8\cdot7\cdot6\cdot5\cdot4\cdot3\cdot2 = 15,120$. Further reduction. In this case the trick was trivial, but in general it can be much more sophisticated. Our design goal for any type of oracle is to reformulate the problem in order to build a better generator and a better Oracle to
decrease the size of the solution space as much as possible. After this redesign, the tree would still generate solutions with more than one digit assignment for letters, a large redundancy. Thus, instead of generating Cartesian Product \( \{0, 2, 3, 4, 5, 6, 7, 8, 9\}^5 \times \{0, 2, 3, 4, 5, 6, 7, 8, 9\} \times \{0, 2, 3, 4, 5, 6, 7, 8, 9\} \times \{0, 2, 3, 4, 5, 6, 7, 8, 9\} \times \{0, 2, 3, 4, 5, 6, 7, 8, 9\} \times \{0, 2, 3, 4, 5, 6, 7, 8, 9\} \) we generate combinations without repetitions of five out of nine, which is \( \frac{9!}{(9 - 5)!} = 15,120 \). A more sophisticated reasoning can lead in this particular problem to finding a solution without much search that can be done "by hand". This reasoning leads for instance to solution \( 734 + 734 = 1468 \), which means \( T = 7, O = 4, W = 6, R = 8 \). If we want however to create a general method for solving ALL this kind of puzzles, we reason differently. Our goal is to create a generator for the binary oracle that will create systematically all possible "combinations without repetitions of five out of nine" for letters \( T, W, O, U, R \). (Pay attention that \( F \) is already mapped to 1 so it does not appear in the set of letters). As usually, the entire problem is solved by the Generator and the Oracle (Verifier). In this case the Selector and the Permuter (plus generator of carries \( C_1 \) and \( C_2 \)) are the Generator Part and the Arithmetic Checker is the Oracle (combinational circuit). Observe that in general the boundary between generation and verification is not fixed and some parts can be shifted from generator to verifier, the trick that is used in quantum oracles and Invertible Logic Oracles.

The purpose of the oracle is to select a value for every letter from set \( \{T, W, O, U, R\} \). In numerical terms, this means we verify all candidates that are "five out of nine combinations without repetitions" of digits \( 0, 2, 3, 4, 5, 6, 7, 8, 9 \) assigned to vector \(<T, W, O, U, R>\). All candidates for mapping are like this: \(<T, W, O, U, R> = <0, 2, 3, 4, 5>, <T, W, O, U, R> = <0, 2, 3, 5, 4>, ..., <T, W, O, U, R> = <5, 6, 7, 8, 9>\). The generation of all candidate solutions for the Oracle can be done by selecting subset of five letters (binary codes of letters) out of nine and then doing all their permutations. There is still a problem remaining how to generate the set of all solutions based on combinations of five out of nine without repetitions. This must be done in the most efficient way, possibly with one solution candidate at every clock cycle of the counter (Generator), a requirement of optimality formulated in [26]. Schematic of our oracle is shown in Fig. 1. There are three major blocks in our oracle: the Selector, the

![Fig. 1.](image-url)

(a) Schematics of the TWO + TWO = FOUR Generator (Selector and Permuter) and Oracle (Arithmetic Checker). Additional blocks Memory and HexDisplay are also shown from FPGA realization. The answers for the problem are displayed on a LCD display. (b) The representation of the TWO + TWO = FOUR problem that helps to create equations for the oracle of this problem.
**Permuter**, and the Arithmetic Checker (Oracle). In the practical FPGA design, there are also smaller circuits in order to store and display the combinations: the RAM (memory) module, the RAM counter, and the LCD display module. This generator/oracle combination is an improved, optimized version of our previous oracle-based designs for this and similar problems (SEND + MORE = MONEY). Optimization is the result of a redesign aimed at reducing the size of the space of all potential solutions. This limits the search, as explained in our “methodology” in Sect. 3.2. The space reduction explained here is equally applicable to quantum and Invertible Logic oracles.

To follow the above presented idea, the **Selector** from Fig. 1 is decomposed to two blocks: the (sequential) **Counter** and the (combinational) **Multiplexers**. The counter gives its outputs to the multiplexers (see Fig. 4). The Counter block of the Selector simply counts up in the given modulo. The Counter block is decomposed to five **Small Counters** (shown schematically as rectangles at the top of Fig. 4). Each of the small counters has three outputs. In the problem of TWO + TWO = FOUR, since there are five unique letters, the counter counts modulo 5 and the largest vector for every Small Counter is 4 (selection states are 000, 001, 010, 011, and 100). Figure 2 shows the beginning of the sample count sequence generated by the Counter to be used by the Multiplexers inside the Selector block.

![Fig. 2. Sample count sequence for the selector counter. In the actual circuit, the output signal is 3 bits, so that a value of 2 (in T = 6) is actually 010. Thus in time T = 6 the sequence generated on output of selector has 5 * 3 = 15 bits and is 010 000 000 000 000.](image)

The outputs of each **Small Counter** (Fig. 5) are then directly connected to control the corresponding multiplexer located at the bottom of Fig. 4. Each multiplexer has only five data inputs coming from top of the counter. The particular inputs to every multiplexer are selected for it in different way. The multiplexers are controlled with three bits each, coming from the left of MUX schematics. These signals come from respective Small Counters from the top. Figure 3 shows initial steps of the resulting output sequence generated by the entire Selector circuit.
The purpose of the Permuter (Fig. 6) is to permute in all possible ways the values given by the Selector from Fig. 4. This allows for all possible number to letter combinations to be tested (as the selector does not repeat any set of numbers). Much like we discussed for the Selector, in Permuter there are the Counter1 and Multiplexer1 blocks. (This follows a general principle of designing advanced Generators). The Counter1 consists of 5 normal counters which select which of the inputs to use. The Multiplexer1 contains the inputs of each of the numbers selected (from Selector). It then goes through a \( _4C_5, _3C_4, \text{ and } _2C_3 \) to gradually lessen the input for each further multiplexer. Figure 7 explains how signals C2, C1, C0 that control MUXes to select input data in Fig. 6 are generated. Recall that C3 = F so C3 is not used as input variable in the oracle. Figure 8 shows part of the sequence generated by the Permuter.

**Fig. 3.** Sample beginning of the sequence of combinations of five out of nine without repetitions, generated by the Selector circuit. Each digit has 4 bits. For instance, 9 = 1001, 5 = 0101, 0 = 0000. Thus the output of the Selector has \( 5 \times 4 = 20 \) bits, as shown on bottom of Fig. 4.

The purpose of the Permuter (Fig. 6) is to permute in all possible ways the values given by the Selector from Fig. 4. This allows for all possible number to letter combinations to be tested (as the selector does not repeat any set of numbers). Much like we discussed for the Selector, in Permuter there are the Counter1 and Multiplexer1 blocks. (This follows a general principle of designing advanced Generators). The Counter1 consists of 5 normal counters which select which of the inputs to use. The Multiplexer1 contains the inputs of each of the numbers selected (from Selector). It then goes through a \( _4C_5, _3C_4, \text{ and } _2C_3 \) to gradually lessen the input for each further multiplexer. Figure 7 explains how signals C2, C1, C0 that control MUXes to select input data in Fig. 6 are generated. Recall that C3 = F so C3 is not used as input variable in the oracle. Figure 8 shows part of the sequence generated by the Permuter.

**Fig. 4.** Schematics of Selector composed of the Counter and Multiplexers circuits. The first from left MUX selects between values 5, 4, 3, 2, 0. The second from left MUX selects between values 6, 5, 4, 3, 2. The third from left MUX selects between values 7, 6, 5, 4, 3. The fourth from left MUX selects between values 8, 7, 6, 5, 4. The right MUX selects between values 9, 8, 7, 6, 5. The binary sequence on output of the selector has \( 5 \times 4 = 20 \) bits. Above the sequence 0100 0110 0111 1000 1001 = 4 6 7 8 9 is shown.
Fig. 5. A single Small Counter out of five shown on top of Fig. 4. It counts in code 000, 001, 010, 011, 100. Its three-bit output out3 goes to respective multiplexer in block Multiplexers.

Fig. 6. Internals of the Permuter block. This circuit generates all permutations of data received on inputs. The inputs on top come from the Selector. The outputs at the bottom go to the Arithmetic Checker block from Fig. 10. Each of the five MUXes has 5 data inputs, 4 bits each. These MUXes are controlled but the 3-bit control C2, C1, C0. On bottom we see 5 small rectangles which represent counters, 4 bits each. At the left of Fig. 6 we see the encoded controls C2, C1, C0 and the corresponding actions of this Permuter that is the so-called “Generalized Register” type of “Micro-controlled Processor”. Above we show the case of controlling MUXes with C2C1C0 = 000.
The purpose of the Arithmetic Checker is to check the arithmetic validity of the solution proposed by the Generator using the outputs of the Permuter. The arithmetic checker was designed just by converting arithmetic equations derived from the TWO + TWO = FOUR problem. Figure 9 shows the derived equations.

Fig. 7. Explanation of the Finite State Machine Master to generate control variables C2, C1, C0. C1 and C0 are also used as inputs to Oracle.

Fig. 8. Part of the sequence of all permutations of set \{1, 2, 3, 4, 5\} generated by the Permuter.

The purpose of the Arithmetic Checker is to check the arithmetic validity of the solution proposed by the Generator using the outputs of the Permuter. The arithmetic checker was designed just by converting arithmetic equations derived from the TWO + TWO = FOUR problem. Figure 9 shows the derived equations.

\[
\begin{align*}
2(O) &= 10(C_1) + R \\
C_1 + 2(W) &= 10(C_2) + U \\
C_2 + 2(T) &= 10(C_3) + O \\
C_3 &= F
\end{align*}
\]

Fig. 9. Equations derived from the TWO + TWO = FOUR problem.
The equations shown in Fig. 9 come from each column for the TWO + TWO = FOUR problem (Fig. 1b). In this problem, it is assumed that F has to equal 1, not 0. This is because if the value was 0, there is no need for a letter. It is also noticed that carry positions were added, similar to the handwritten method of addition. The Arithmetic Checker (Fig. 10) strictly follows the equations, by replacing the ‘+’, ‘*’, and ‘=’ operators with their respective arithmetic and predicate blocks.

Fig. 10. Complete Arithmetic Checker Oracle circuit. The inputs are binary encodings of letters T, W, O, F, U, R, the constants 1, 2 and 10 and intermediate binary variables C1 and C2 (not discussed in more detail). This oracle is far from minimal logical design but it explains the general method to create oracles from arithmetic/Boolean equations. It can be optimized using methods from logic synthesis and digital design.

The RAM Module. Although when the circuits described above can make a complete oracle, there is no way for a user to know what the successful combinations are. In order to show what the combinations are, there first needs to be any type of memory to store the successful combinations. This is done on the FPGA by utilizing its RAM (Random Access Memory). The circuit used for the RAM of the FPGA is based on the concept of vectors. In a vector, information can be inserted, as well as removed according to a reference number. The circuit first stores the successful combinations (combinations with an Arithmetic Checker output of 1) into the vector. In the post-processing stage, the combinations in the vector can be displayed to the user by the RAM Counter.

Fig. 11. Concept of a vector used for memory in the TWO + TWO = FOUR oracle.
The RAM Counter. In order to cycle through all of the solutions to the problem, a counter is required. The counter used to do the task is very similar to a normal counter, incrementing its value by 1, but it has an adjustable limit. This limit is imposed on the counter such that the counter won’t go to empty cells in the vector, leaving the counter only counting the solutions.

The LCD Display. The LCD Display is required in order to display the solutions. Without it, the user will not be able to see what the calculated answers are. The circuit for the LCD is fairly simple; the output from the RAM is displayed on the LCD. Only the initialization of special ports (to turn on the backlight and other functions) is required.

3.2 Experimental Results for the TWO + TWO = FOUR Problem

The experiment of the “TWO + TWO = FOUR Problem” is a comparison between two methods: a hardware optimized implementation of the oracle, and a software implementation of an arbitrary oracle. The hardware implementation is run on a Terasic DE2-115 board, while the software implementation is run on a laptop with an Intel Core i5 CPU clocked at 2.40 GHz. These methods are compared for speed, or in this case, the time it takes to find all solutions for the same given problem. Counters are implemented in both the hardware and software oracles, yielding for more accurate results. The answers of the hardware program are verified by the software program. The solutions are: $938 + 938 = 1876$, $928 + 928 = 1856$, $867 + 867 = 1734$, $846 + 846 = 1692$, $836 + 836 = 1672$, $765 + 765 = 1530$, $734 + 734 = 1468$. Since the Selector, Permuter, and Arithmetic Checker operate under one clock pulse, theoretically stating, without glitches in the circuit, it would take $n$ clock pulses to go through $n$ combinations. The circuit has to go through $9C5$ combinations, and from that number of combinations, it has to go through $5P5$ permutations. Solving it would figure out how many clock pulses it would take in order to complete. Let us derive the number of clock pulses required to solve the problem: $(9C5)(5P5) = (126)(120) = 15,120$. The number of clock pulses is then converted into time. 1 MHz is equivalent to 1000 kHz. Since the clock on the DE2-115 is 50 MHz, it is equivalent to 50(1000) = 50,000 kHz. There is a formula of $1/(KHz)$ which converts Kilohertz to milliseconds. Substituting 50,000 into the equation, $1/50,000$ is the result, the number of milliseconds per cycle. The number of cycles, 15120, is multiplied to the factor, receiving 0.3024 as an answer, stating that the computational time is $0.3024$ ms. Theoretical predictions were perfectly verified in experiments [2].

|      | Hardware (predicated) in milliseconds | Hardware (actual) in milliseconds | Software in milliseconds |
|------|---------------------------------------|----------------------------------|------------------------|
| Trial 1 | 0.3024                               | 0.3024                           | 1105886               |
| Trial 2 | 0.3024                               | 0.3024                           | 1136322               |
| Trial 3 | 0.3024                               | 0.3024                           | 1136897               |
| Average | 0.3024                               | 0.3024                           | 1126368               |
From the data shown in Table 1, it can be concluded that the software computational speed was significantly slower than that of both hardware times. Also, the hardware computational times were equivalent, therefore showing no signs of error in calculation. In the average computational time comparison, the software took 1126368 ms to calculate, while the hardware only took a small fraction of 0.3024 ms. Thus, from the data, the hardware has around 3724762 speed up time.

In conclusion of the TWO + TWO = FOUR experiment, the hardware implementation of the oracle performed significantly faster than the software implementation in terms of computational time. For all trials tested, the hardware performed over three million times faster than the software equivalent. The results of the predicted hardware time and the actual hardware time were the same. Each trial always took exactly 0.3024 ms to calculate. This is because the circuits implemented in the FPGA operate on every clock pulse, and it is very rare for the clock pulse to have a glitch. Thus, the computational time of the FPGA can be proven by hand. There were no sources of error while taking data, as the timers were implemented on the respective systems. However, there can be lots of improvements to the experiment that could be made. First of all, the software tested is for an arbitrary amount of problems, meaning that it is very inefficient. Therefore, in order to have more accurate data, a specific oracle for the TWO + TWO = FOUR problem needs to be made. Also, the clocks of both systems are different (50 MHz on the FPGA versus 2.4 GHz on the computer). If one were to measure the ratio of FPGA speed-up, the clocks would have to be equivalent.

We can observe that this problem can be solved differently without creating carry variables C₁, C₂, C₃ in Fig. 9. In the second variant the decimal adder can be designed with binary-encoded digits and with internal binary carry signals that are not considered as input variables to the oracle. The oracle is not created as in Fig. 10 based on equations from Fig. 9, but is just the adder with two inputs (T₁, W₁, O₁) and (T₂, W₂, O₂) and outputs (F, O₃, U, R). This adder realizes equation (T₁, W₁, O₁) + (T₂, W₂, O₂) = (F, O₃, U, R). But now in this second variant additional constraints are needed: T₁ = T₂, W₁ = W₂, O₁ = O₂ = O₃. This is a common tradeoff when designing Generator-Oracle systems. While we call the first variant of oracle as shown here the “Oracle with Control of Intermediate Signals”, the second type of the oracle we call the “Oracle without Control of Intermediate Signals”, in this case carries C₁, C₂ and C₃ are these intermediate signals. Observe that in this variant we do not use sequential generator. All knowledge is in combinational oracle, which makes it a good candidate for Grover oracle and for Invertible Logic Oracle. Both oracle types can have some advantages and disadvantages, depending on the problem. These two design types of oracles illustrate again the tradeoff that we deal with when designing efficient oracles.

3.3 Concluding on Boolean Oracles and Their Relation to Quantum Oracles and Invertible Logic Oracles

Boolean Oracles realized in FPGAs are useful for problems for which the designer has no information and no heuristics to solve the problem. There exists also no other than exhaustive algorithm for these types of problems (in some problems dynamic programming may be better, so there is in these cases no reason to create our type of “unsorted database” approach based on generators and oracles). Under these conditions
classical hardware oracles are better than software programs for small problems. However, in case of large problems these oracles cannot be used because of the size of hardware to exercise systematically and exhaustively all possible input combinations. In software some efficient search algorithm can be used that will execute cuts in tree branches early and can backtrack efficiently. For problems that have many solutions, good design of the oracle plus good design of the generator help to solve problems [2] that are difficult to solve otherwise, but still not too large in size. Tricks as those illustrated in Sect. 3.1 are therefore used. Other good solution that may be helpful for some problems is to design generators as special counters that count in advanced codes that correspond to depth-first or breadth-first searches. Any sequences of binary vectors can be simply created by adding large ROMs at outputs of standard counters. Concluding, there is some combination of the problem size and problem type for which hardware oracles based on classical logic are practical, realistic solutions. They may be realized in ASIC, FPGA or any new nano-technology such as memristors.

Please note that designing classical generator/oracle systems is fundamental to our general methodology. This is because the oracle concept is the base of our “CSP methodology of problem-solving in hardware”. It is only a technical aspect to translate Boolean oracles to reversible logic used in quantum computing or to invertible logic used in magnetic spins. These are the two technologies that will prove useful to solve these problems in future when the number of reliable qubits in quantum computers will increase, and when larger Invertible Logic circuits with magnetic spins will be built. At this time the ideas presented in this paper will become useful not only to better solve toy problems as used for illustration here, but also to solve quickly practical problems of large size.

The theoretical base to create quantum and Invertible Logic oracles already exists. Actually, the literature presents many solutions to reversible/quantum arithmetic blocks such as adders, multipliers, shifters, code converters, comparators and others. Thus converting classical logic/arithmetic blocks to reversible blocks for Grover or other Quantum Algorithm can be automated. General Boolean functions that can appear in some oracles are more difficult to convert because in quantum the EXOR-based circuits are better while in classical design the OR-based circuits are better. However, there are many methods to convert SOP-based logic to EXOR-based logic such as Exclusive-Or-Sum-of-Products (ESOP) [13]. Observe also that converting combinational Boolean circuit to Invertible Logic is theoretically trivial because Hamiltonians are known for every two-input binary gate ([4] and many papers by Biamonte), and Hamiltonians are also known for many other gates and blocks such as adders and multipliers. There exist also methods to realize arbitrary Hamiltonians built from smaller Hamiltonians.

3.4 General Methodology for Oracle Design

While designing a binary oracle the designer has to ask himself first – “what is the problem type that I want to solve?” Knowledge of these problem types and blocks used to solve them is very helpful. Let us explain the essence of this question. For any realization of the oracle, especially quantum and classical, we need some generator that would create a set of input vectors to exercise the oracle. In Grover Algorithm the vector of Hadamard gates serves as this generator of all possible binary strings being
We want to reduce the size of the input vectors in order to reduce (often dramatically) the size of the entire solution space and the cost and operation time of the oracle. Knowing the type of the problem helps to find good encoding of data and as the result helps to reduce the cost and increase the speed. Below we discuss this problem.

Observe that starting from $|0\rangle$ the Hadamard gate creates superposition of $|0\rangle$ and $|1\rangle$. Two Hadamard gates working in parallel create a superposition of $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$, in another variant of Dirac’s notation $|0\rangle$, $|1\rangle$, $|2\rangle$ and $|3\rangle$. Therefore $n$ Hadamard gates working in parallel generate all binary numbers from 0 to $2^{n-1}$. We can see that the parallel vector of Hadamard gates is a “quantum generator” of all numbers from 0 to $2^{n-1}$. But also, assuming that the individual bits of these numbers represent presence or absence of an item in the set of $n$ elements, these numbers represent all possible subsets of the set. For two-element set, like this: $|00\rangle = \emptyset$, $|10\rangle = \{a\}$, $|01\rangle = \{b\}$, $|11\rangle = \{a, b\}$. Therefore, as also known from Grover and quantum algorithms, the vector of Hadamard gates is a quantum generator of all subsets of a set. This is used in Grover Oracles and also in other quantum algorithms such as Bernstein–Vazirani. This property of Hadamard operator leads to natural design of oracles in which we look for a solution being a subset of the set $\{2, 49\}$. For instance, when the designer wants to find the best partition of a set to two subsets $X$ and $Y$, every binary vector represents a subset of bits with value 1 as subset $X$ and a subset of elements with value 0 as subset $Y$. The oracle evaluates if this separation to two subsets satisfies all constraints. The same method can be used to design oracles which look for a mapping from a set to a set. Several algebraic systems, such as the “Partition Calculus” [12, 15, 16, 47] and “Cube Calculus” [46] are based hierarchically on set operations and relations. Therefore these algebraic systems can use the natural Hadamard-based encoding of solution candidates that are verified by constraints [21, 22, 30]. For instance, several important problems in CAD and Machine Learning can be solved using only two basic operations on set partitions from Partition Calculus; operator $\Pi_1 \cdot \Pi_2$ which finds the product of partitions, and relation (predicate) $\Pi_1 \leq \Pi_2$. These problems include Ashenhurst-Curtis Decomposition [12], Bi-Decomposition, State Minimization, Concurrent State Minimization and Encoding, ROM-based design and other.

In addition to algebraic models, it is important how the variables (symbols) are encoded in various binary codes. Some problems may require other than standard methods of information encoding, for instance using “one-hot encoding” or “thermometer encoding” of numbers (data). Therefore, the first question is this: “what is my problem type and how should I encode the data”. As an example, when we solve the graph theory problems we may encode nodes while treating edges as pairs of nodes. Or we can encode edges as bits in a set-theoretical representation with as many qubits as edges in our graph. We can also encode a graph as a binary incidence matrix. Permutation problems such as Traveling Salesman or Generalized Traveling Salesman would require to deal with permutations. Few standard ways of dealing with permutations have been created [8, 15, 31]. One method is to treat the binary input vector of numbers as a single permutation of natural numbers with $k$ successive bits for a number and to use constraints that will not allow the repeated numbers to be considered as solution candidates [31]. Other method to solve permutation problems with oracles is to
create inside the oracle, just at its input, a large encoder that converts the set of all subsets to the set of all permutations [15]. As done usually in AI and ML, the problem is solved by a combination of a “generator” and “verifier”. A simple generator creates too many candidates so that most of them are next disqualified by the “verifier” (all the constraints in the oracle). A more advanced generator creates only reasonable candidates that would require less constraints checking, but creating such a generator may be more difficult. The most natural generator in quantum is the “generator of all subsets of a set”. Observe that in the case of quantum oracles the “initial counter” coming from vector of Hadamards generates always a set of subsets, which may be converted by a special circuit to another type of combinational objects similarly as it was done in Sect. 3.1 [49].

The proposed methodology for designing oracles and especially Grover Oracles is the following:

1. We ask ourselves – “what kind of problem are we solving?” Is this a “subset of set” problem, a mapping problem, a “combination with repetitions” problem, a permutation problem, a spectral transform problem [11], etc.? The answers to this problem and the respective data encoding have huge influence on the final oracle design (as illustrated in cryptoarithmetic puzzles).

2. Next question – how the data for this problem should be encoded? Can we use one of the well-known encoding methods like natural number encoding, one-hot-encoding, thermometer encoding, Gray Code Encoding, etc.? Answers to question 1 and question 2 are related.

3. Next question, assuming the type of problem and the type of encoding, we ask “what are the blocks to be designed for the oracle and how they are combined?”. There can be standard blocks such as shifters, arithmetic operators or predicates. Comparators and combinational counters, such as variants of “counter of ones” are often used. Some blocks must be designed from scratch and then in case of quantum oracles the methods of logic synthesis for reversible circuits should be used [13]. In case of quantum circuits these blocks can be designed on the level of binary reversible logic [13] or, better, on the level of truly quantum primitives such as Control-V and Control-V+, Pauli rotations, etc. [23, 39, 41].

4. The blocks are taken from library of standard or quantum blocks. In case of non-standard reversible blocks they are designed using methods from “reversible logic synthesis”. Next the blocks are combined to larger blocks and subsystems. The final output AND-gate (Multi-input Toffoli gate in quantum case) is added to combine together binary answers from all partial constraints.

5. If an optimization problem is solved (see Sect. 4) we have to create a part of the oracle that is modified in every repeated CSP problem solved by calling the successive Grover algorithm. The same is true for our Invertible Logic oracle-based method of problem-solving.
4 Grover Quantum Oracles

4.1 Minimum Set of Support Problem

As an example of bottom-up systematic design of Grover Oracle for optimization problem, we present here a new approach to find the minimum set of support for binary switching functions. Next, the essential part of this algorithm, “POS $\rightarrow$ SOP conversion for unate functions” is sped up by Grover quantum search algorithm that brings a quadratic speedup. We present below in detail how to build the Grover oracle. Our quantum algorithm can be easily adapted to solve other important but similar problems. There is a large body of literature on the Minimum Set of Support Problem, because this problem finds many important applications such as: minimization of Boolean functions (SOP, POS, PLA, FPGA, etc.) [34, 37], Ashenhurst-Curtis Decomposition and other functional decomposition methods for binary and multi-valued functions [35] and information systems [33], cryptography, data mining and Machine Learning, large databases, rule and expert systems [36], index generation functions [38], complementation of Boolean Functions, rough set problems, minimizing Petrick functions (prime covering in SOP), etc. This problem is also known as the “attribute reduction problem” and “unate covering problem”.

A. What is the problem?

Given a Boolean function $f: \{0, 1\}^n \rightarrow \{0, 1, x\}$ of $n$ variables (the function is completely specified, or mostly likely, incompletely specified in case of ML problems) the minimum set of support is the minimum number of variables required to express the function as an equation or set of rules. This method is useful to reduce the unnecessarily large representation of functions (for instance using ON and OFF sets for binary functions) to all representations that include minimum numbers of variables. Using the minimum set of support variables simplifies the logic synthesis or Machine Learning methods. Several methods have been proposed for reduction of features (variables, attributes) in Machine Learning and knowing all the minimal sets of attributes has an importance in those applications where learned rules on less features are easier to understand and do not lead to overfitting.

B. Classical approach to find Minimum Set of Support for Boolean function

The new procedure for finding Minimum set of support is as follows:

1. List all the OFF-minterms row-wise and ON-minterms column-wise in a rectangular table and perform bitwise Exclusive-OR operation between all possible pairs of binary strings of OFF-minterms with ON-minterms. Write the resultant binary strings as the Boolean OR of the corresponding variables. In every intersection of row and column we have the sum of variables which separate the given ON-minterm from the column and the respective OFF-minterm from the row.
2. After completing the entire table, create a POS formula being the AND of all OR terms from the intersection cells in point-1 above.
3. Convert the above POS formula to an equivalent minimal SOP formula. This can be done, for instance, by a step-by-step Boolean Multiplication together with
absorption of products created in every step. In the final SOP formula each product corresponds to a minimum set of support of variables. This step is solved using Grover Algorithm.

Comment 1. There are several methods how to convert POS to SOP. The case here is simpler as our POS is a unate function. All these methods can be used at this step to create a quantum oracle, but our interest in this paper is restricted to unate functions.

Comment 2. The user may verify the correctness of results of this algorithm by folding Karnaugh maps with respect to all combinations of variables that are not present in every minimum set of support. For instance, for a function of five variables \( f(a, b, c, d, e) \) the correctness of the minimum set of support \{a, b\} is verified by sequentially folding according to variables \( c, d \) and \( e \), without encountering contradictions and thus creating function \( f_1(a, b) = f(a, b, c, d, e) \).

A useful pre-processing method is to simplify initial POS (in point 2) by using repeatedly the Boolean laws: \( A \cdot \overline{A} = A \); \( A \cdot (A + B) = A \) and \( (A + B) \cdot (A + B + C) = A + B \). Therefore, we start the simplification from the shortest OR terms and we remove the OR terms containing them. The minimum set of support problem is an NP problem [34], thus all exact algorithms for classical computers can be applied only to relatively small problems. For larger problems, the classical algorithms are heuristic and take large time and space complexity to find only some minima or approximate minima. In this section, we illustrate our methodology by presenting a hybrid classical/quantum algorithm to solve exactly the minimum set of support problem for \( k \)-valued switching functions. It provides a quadratic speedup with respective to its classical counterpart algorithm for the stage of POS to SOP exact conversion of a unate Boolean function.

### 4.2 Example of Finding the Minimum Set of Support for a Binary Function

We illustrate our algorithm from Sect. 4.1.B step by step.

Consider a Boolean function represented in a Karnaugh map as shown in Fig. 12.

| \( ab \) | \( cd \) | \( 00 \) | \( 01 \) | \( 11 \) | \( 10 \) |
|---|---|---|---|---|---|
| \( 00 \) | x | x | 1 | x |
| \( 01 \) | 1 | x | x | 1 |
| \( 11 \) | x | x | 0 | x |
| \( 10 \) | 0 | x | x | 0 |

Fig. 12. The Karnaugh map of an incompletely specified Boolean function.
**Step 1.** The possible values of the Boolean function are 0 (OFF) and 1 (ON). The cells on the intersection of OFF-minterms and ON-minterms are created by bitwise EXOR. For instance, bitwise EXORing of OFF-minterm 1111 with ON-minterm 0011 the binary string 1100 is obtained which is written as \( a + b \).

| OFF\ON | 0011 | 0100 | 0110 |
|--------|------|------|------|
| 1111   | \( a+b \) | \( a+c+d \) | \( a+d \) |
| 1000   | \( a+c+d \) | \( a+b \) | \( a+b+c \) |
| 1010   | \( a+d \) | \( a+b+c \) | \( a+b \) |

**Fig. 13.** Illustration of the method to find all separating variables for every pair of true and false minterms of a Boolean function from Fig. 9. True minterm 1111 and false minterm 0011 are separated with variable a or variable b, this is the formula \( a + b \) seen at the intersection of row 1111 and column 0011.

**Step 2.** A POS formula is created being the AND of all the terms from cells in Fig. 13, which becomes the product of sums as represented as below, \( \equiv (a + b) \cdot (a + c + d) \cdot (a + d) \cdot (a + c + d) \cdot (a + b) \cdot (a + b + c) \cdot (a + d) \cdot (a + b + c) \cdot (a + b) \).

**Step 3.** Using laws from the algorithm, the expression gets reduced to \( \equiv (a + b) \cdot (a + d) \). Transforming this POS to a minimum SOP is in general a difficult problem. In this particular didactic case, the transformation is trivial and based on Boolean algebra \( (a + b) \cdot (a + d) \equiv a + ab + ad + bd \equiv a + bd \).

We found in the last formula that the Boolean function from Fig. 12 depends on only a single variable \( \{a\} \) or it depends on two variables \( \{b, d\} \). The function has exactly two minimum sets of support – \( \{a\} \) and \( \{b, d\} \). For large functions the above “POS to minimum SOP” transformation is a complicated NP problem and thus we apply Grover Algorithm to solve this step.

### 4.3 Grover Algorithm

Grover algorithm is one of the few most famous quantum algorithms. Grover algorithm performs searching on a “black box,” an unsorted database, in order to find an element that satisfies the oracle [9]. The oracle is built specifically for the given problem instance. The idea of Grover’s algorithm is to place the qubits representing entire search space of size \( N \) in a superposition state. Then the phase of the states marked by oracle is inverted, followed by an inversion about the mean operation, which is also known as the diffusion operation. Diffusion operation amplifies the amplitude of the marked states to increase probability that this state will be a result of measurement performed on a vector of input qubits. Oracle followed by diffusion is called the Grover Loop. After \( O \sqrt{N} \) iterations of Grover Loop, the probability of measuring the target solution approaches to 1 (in case of a single solution) [9, 39, 40].
4.4 Oracle for Minimum Set of Support

A. Oracle circuit for Grover Algorithm

To build a Grover’s algorithm to find the minimum set of support for function from Fig. 12 an Oracle for this specific problem is built in such a way that it checks if the input satisfies the following three constraints: (A) that the POS is satisfied, (B) that it is a new solution and not one found previously, (C) that the number of variables in solution is equal to a constant Threshold given on the input. The first thing to build the Grover oracle is to encode the input and output of the problem in binary. For our oracle, the search space is a collection of all potential solutions. Potential solutions are all subsets of the set variables \{a, b, c, d\}. Some of these subsets are the searched Minimum Sets of Support for this function.

The block diagram of the proposed quantum oracle is shown in Fig. 14. It consists of three major blocks; (A), (B) and (C). The inverse circuit corresponds to blocks (A), (B) and (C) in reverse order must be designed respectively, to restore all the modified qubits to their original values. This is done in the “Mirror Part” of the oracle composed from mirror of C, followed by mirror of B and followed by mirror of A. This mirror part is not shown in Fig. 14 due to space limitations. Qubit |k\rangle is the solution of the oracle. It marks that all constraints of the oracle (A), (B) and (C) are satisfied.

Fig. 14. Part of Oracle to find all Minimum Sets of Support for the binary function from Fig. 12 with the minimum set of support \{\{a\}, \{b, d\}\}. POS = (a + b) \cdot (a + d) = 1 \oplus (a'b') \cdot (a'd') is realized in Block (A). The Hadamard gates at top left do not belong to the oracle and are explained in point 2 of the algorithm. They are drawn here for didactic reasons as they serve as a generator of all subsets of set \{a, b, c, d\} by creating the superposition of all numbers from 0 to 15.
Block (A) is a Boolean satisfiability function in POS form. Qubit $|i\rangle$ is initialized to $|1\rangle$ to realize $\text{POS}(a + b) \cdot (a + d)$ based on DeMorgan theorem from Toffoli gates and inverters. This qubit recognizes every solution of this POS from the superposition of inputs. It is given as one of three AND-ed inputs to the far right Toffoli gate that creates the solution to the entire oracle. Block (B) does not exist in the first run of Grover and will be discussed later on. This block represents modification of constraints in subsequent runs of Grover Algorithm typical for the optimization problems, which we already discussed.

Block (C) contains three counters and the equality comparator. The three counters count together the number of input variables required to satisfy the POS function. Each counter adds a 1 if the corresponding variable controlling it has value 1. The Comparator $X = Y$ compares the output from the counters with the threshold value given as a constant values $|n1\rangle$ and $|n2\rangle$ of Threshold on input to Grover Algorithm. Then a Toffoli gate is applied on ancilla qubit $|k\rangle$ controlled by $|i\rangle$, $|1\rangle$, and $|j\rangle$. If the conditions are met, ancilla bit k will be flipped. It changes the solution phase so that the solutions that satisfy both conditions can be marked.

**B. Finding all solutions to the minimum set of support problem**

In this section we will explain how the Grover Algorithm is used multiple times with modified oracles to find all exact minimum solutions to our problem. The first run of the Grover uses the oracle without block (B). Qubit $|i\rangle$ gives all solutions that satisfy $\text{POS}(a + b) \cdot (a + d)$. In general, we start from the lowest bound of the solution cost and we go up. In this case, we optimistically assume that there exists a single variable that satisfies the POS. Thus the value of Threshold is set to 1 and all solutions with single input variables are checked with counters and the comparator. The solution $a$ is found. Now block (B) is compiled to the oracle by the pre-processing standard computer that controls the quantum computer. Block (B) includes now the representation of the first solution $a$ which is subtracted (inhibition operator $X \cdot Y'$ realized as part of the large Toffoli gate at right). Therefore, all possible solution sets that include variable $a$ are being excluded. These are all products of literals included in cube $a$. No solution with one variable is found by subsequent runs of Grover with this modified oracle. Now we look for solutions with two variables. We set the value of Threshold to value 2. A new solution $bd$ is found. It is next added to the ESOP realized in the output qubit from block (B), marking the solution $bd$ as already used. So now the block (B) is $a \oplus bd$ as shown in Fig. 10. (Let us remind that in our encoding solution $a$ is represented as 1000 and solution $bd$ as 0101, so that solutions are disjoint minterms for which OR-ing is the same as EXOR-ing, based on the rule $A + B = A \oplus B \oplus AB$. Therefore, the logic sum of all previous solutions in block B can be stored as their EXOR). This method of creating a sequence of oracles is general and we applied it to design sequences of Grover oracles for various problems.

With the full oracle as in Fig. 11 the Grover algorithm finds no more solutions. But it is still unclear if there are solutions with three or four variables? In this case, because of properties of unate symmetric functions all solutions with three variables are cancelled by solution $a$ (1101, 1110, 1011) or cancelled by solution $bd$ (0111, 1101). So there are no other solutions. In general, one has to keep increasing the value of threshold if he attempts to find all minimum sets of support. To confirm that there are no any solutions
besides those that we already found, the oracle can be also run by Grover with blocks (A) and (B) but without (C). Our hybrid Algorithm finds that there is no solution to this function, thus no more solutions exist at all. Every binary vector \(|a, b, c, d\rangle\) of a solution can be verified on block (A) of the oracle run outside of the Grover Algorithm. In one more variant of our approach, the number of remaining solutions can be found using the Quantum Counting Algorithm that returns the number of values 1 in the function.

4.5 Remarks on This Design

As an example of the methodology proposed in Sects. 2, 3 and 4 of this paper we presented a hybrid algorithm in which pre-processing, i.e. creating the POS formula, is done in a standard computer. Only solving of the NP problem of \("\text{POS} \rightarrow \text{SOP} conversion for a positive unate function and with finding all prime implicants}\) is solved by our hybrid algorithm based on a sequence of calls of Grover with modified oracles. We presented the detailed design of the oracle from reversible gates and blocks. More details of the blocks used here and a discussion can be found in [12]. Other typical blocks are presented in other cited here papers of the PSU team. This Sect. 4 illustrated some elements of our methodology: (1) selection of encoding, (2) problem representation, (3) combining of constraints, (4) hybrid design and role of the standard computer that supervises the quantum computer, (5) repetition of Grover with modified oracles to solve the optimization problem.

Very similar Grover oracles can be built for other fundamental CAD problems: function complementation, binate covering, unate covering, and prime implicant generation for SOP minimization. Moreover, these problems can be solved by our approach for multiple-valued functions as well. The method presented in this section is very similar when applied to binary or multiple-valued functions from ML [30], and their quantum component is exactly the same. Similarly to the oracle example from Sect. 3, this oracle can be also converted to Invertible Logic Oracle and solved with Invertible Logic methods explained in Sect. 5. All these oracles can be simulated with the logic programming CSP software outlined in Sect. 6.

5 Invertible Logic Oracles

First, to convince the reader about power of invertible logic let us give few more simple examples.

1. A circuit is a tree of three AND gates as in Fig. 15. Because for the 2-input-AND gate for the value 1 on output is the combination (1, 1) on inputs, the backward (output to input) propagation of signals is represented as in Fig. 15 and the oracle requires only one evaluation. Classical circuit would require in the worst case \(2^4 = 16\) evaluations and the quantum Grover oracle would require \((16)^{1/2} = 4\) evaluations because of its quadratic speedup.
2. A circuit is a tree of Boolean Gates $F_2 = (ab) \oplus (cd)$ as in Fig. 16. The snapshots show the propagation of signals backward with fast finding of one solution. EXOR is a better combining gate than the OR gate, because for output 1 it has only two not three input combinations $(0, 1)$ and $(1, 0)$.

3. Suppose we have a graph with three nodes as in Fig. 17. In Fig. 17a the graph is 1-colorable, the graphs from Figs. 17b and 17c are 2-colorable and the graph from Fig. 17d has a chromatic number of 3. The oracle for the graph from Fig. 17d in ternary-input binary-output logic is shown in Fig. 17e. Here outputs of all three inequality comparators are equal 1, which means that all partial constraints are satisfied. Let us analyze how Invertible Logic works. Let us say the top comparator is satisfied by coloring node 1 with color $a$ and node 2 with color $b$. Now color $b$ is propagated to upper input of the middle comparator. If this comparator selects color $a$ on its second input, it will propagate 1 to output. But the bottom comparator will be not satisfied as it will have color $a$ on both inputs. Thus if the middle comparator will select color $c$ on lower input it will produce output 1 but also the bottom comparator will be satisfied as it will get various colors $a$ and $c$ on its inputs. The coloring with only colors $a$ and $b$ is not possible, which is illustrated for one coloring case in Fig. 17d. The inequality comparator for edge $(1, 2)$ produces a zero on output, but inequality comparators for edges $(1, 3)$ and $(2, 3)$ produce a 1 on their outputs so the circuit is not satisfiable (not 3-satisfiable) but the circuit is 2-satisfiable. This example shows the close relation between the graph coloring, SAT and MAX-SAT problems. Similarly, every oracle can be in theory transformed to a SAT problem, because oracle is a Boolean function and every Boolean function can be realized in a POS (CNF) form.

---

**Fig. 15.** Propagation from output to inputs in a single oracle evaluation for function $a \cdot b \cdot c \cdot d$

**Fig. 16.** Propagation from output to inputs in a single oracle evaluation for function $a \cdot b \oplus c \cdot d$ (only some solutions shown)
Now that we appreciate the power of Invertible Logic, let us observe that every oracle is basically a kind of SAT-solver that solves a non-standard type of SAT using Boolean gates and blocks, and not only the classical POS-SAT formula. Thus every classical logic oracle \[2, 26\] or every reversible logic oracle \[3, 8, 11, 12, 15, 16, 21, 22, 24, 28, 30, 31\] can be easily converted to the Invertible Logic oracle and simulated in software, emulated and verified using an FPGA. Most importantly, Invertible Logic Oracle can be realized using the real hardware nano-technology such as magnetic spins. The only question that remains when we want to use standard FPGA or one of nano-technologies is this: “how to design the inside of logic gates and blocks that externally are standard Boolean circuits?” Several methods have been proposed \[1, 4–7, 10, 14, 17, 18, 20, 43\].

Any hardware realization of an Oracle is in essence a Boolean Function (it can be extended to multi-valued functions), so the theory of such functions, as well as their synthesis methods, can and should be used to build oracles. For instance, oracles from \[8\] make use of the theory of symmetric Boolean problems, because of symmetries found in problem data such as 3 Missionaires and 3 Cannibals from the known puzzle.

In addition, internally the blocks of Invertible Logic oracles are like neural nets or other systems based on Hamiltonian Dynamics. Therefore, when the entire circuit is built from blocks, the blocks are externally Boolean (or multi-valued) but their internal design is based on methods typical for Neural Networks design and Adiabatic Quantum Computing design. Theoretically, every oracle can be built from AND gates and Inverter gates, or from some larger set of gates and blocks, but a better method is to create new types of blocks corresponding to relations that can be satisfied (1) or not satisfied (0), based on their characteristic functions. For instance, in case of the graph coloring problem we can create a network of comparators of inequality, one comparator for every edge of the graph. The inputs to these comparators are encoded data for colors of their corresponding nodes. Finally, a large AND gate is used to combine answers from partial constraints on all edges of the graph (the inequality comparators). However, another approach would be to create a relation for every node of the graph that

---

![Fig. 17. Graph Coloring in network of constraints realized in Invertible Logic, (a) one-colorable graph, (b) two-colorable graph, (c) two-colorable graph, (d) three-colorable graph, (e) good coloring of graph from d with colors a, b and c, (f) bad coloring of graph from d with two colors, constraint for nodes 1 and 2 is not satisfied because these neighbor nodes have the same color a.](image-url)
would operate on this principle – “If my color is different from all colors of my neighbors I will keep my color. If my color is in a disagreement with any color of my neighbor, I will change my color randomly”. A Hamiltonian can be calculated for such rules. Thus the number of blocks in the oracle is the number of nodes. When all nodes are happy with colors of their neighbor nodes the minimum energy is reached and thus the solution to the graph coloring problem with K colors is found. In case that we want to solve the optimization variant of graph coloring, the successful oracle for K1 colors is repeated with a smaller number of colors K1 < K (see Sect. 2 and [12, 31]). This means in a special case that the oracle blocks for constraint COLOR(n_i) ≤ K are replaced in the repeated oracle with blocks COLOR(n_i) ≤ K − 1.

How to design the internals of the blocks? In case of Invertible Logic analog circuit design methods are used when building gates with nano-magnets. Hamiltonian design methods and stochastic system based methods are used when emulating with FPGAs. A standard approach is to follow this line of transformations:

\[ \text{Circuit} \rightarrow \text{TruthTable} \rightarrow \text{CharacteristicFunction} \rightarrow \text{Linear Programming} \]
\[ \rightarrow \text{Hamiltonian} \rightarrow \text{Invertible Logic Block} \]

Many problems require arithmetic, so how are the arithmetic operators implemented? Even the AND gate implemented as a Hopfield Network internally needs arithmetic – adders and constant multipliers - as well as some non-linear threshold or similar operators typically used in Neural Nets. Various number systems can be used to design the Hamiltonian-based internals of relations that specify the problem. For example such number systems as SNR – Stochastic Number Systems [1, 20] or Frequency Number Representation [43].

6 Prolog Oracle Simulator

Currently, simulating a correctness of a quantum circuit and in particular a Grover-based algorithm with a complex oracle is difficult because of the small number of qubits that can be used by the contemporary quantum circuit simulators [12, 41]. However, the designer of an oracle of any kind (Boolean, Quantum or Invertible) wants usually to verify correctness of his design and the density of solutions in the solution space. In quantum case this is because the entire Grover Algorithm is known to be correct and the non-oracle components of Grover Algorithm hardware are easy. This is also true for other quantum algorithms such as Shor, Bernstein-Vazirani or Quantum Walk. Let us observe that we do not need a quantum simulator to achieve this “quantum circuit built from permutative gates” verification goal. Instead of simulating the oracle being a part of Grover Loop inside the iterated Grover Algorithm it is often sufficient just to simulate the oracle itself as an invertible logic circuit, even before converting to reversible logic with ancillae qubits (reversible gates are described by permutative matrices). Simulation of oracles can be done in hardware description languages such as VHDL or System Verilog, but it is better to use a logic programming languages such as Prolog, because of their natural capability to simulate CSP systems.
based on mechanisms such as backtracking and matching. While Verilog cannot simulate directly the Invertible Logic circuits, Prolog and CSP Languages can do this.

A program in Prolog [42] can easily simulate an oracle just by defining all gates and blocks as their truth tables (characteristic functions) or as blocks (sub-systems) composed hierarchically from lower level gates and blocks. Next, oracles built from these blocks can be exercised from inputs to outputs, from outputs to inputs or in any possible direction by fixing subsets of inputs and outputs and finding values of all remaining signals such that all constraints (relations) are satisfied. The designer can create variants such as the “Oracle with Control of Intermediate Signals”, and the “Oracle without Control of Intermediate Signals” and set intermediate signals to some heuristically assumed values.

The Prolog simulator is composed from two types of blocks: the Oracle and the Generator [42]. Generator creates inputs to Oracle in case of classical input-to-output simulation. In case of output-to-input simulation the “hidden generator” creates combinations of inputs/outputs for individual gates but the simulations start from outputs assigned to constant values. For instance, an adder with two-bit arguments input1 and input2 and result output is defined as adder[input1, input2, output]. It works as follows: adder[1, 2, X] produces X = 3. But also: adder[X, 2, 3] produces X = 1. In addition, adder[X, Y, 3] will give: X = 0, Y = 3, next will produce X = 1, Y = 2, next X = 2, Y = 1, next X = 3, Y = 0. The order of generating the pairs of inputs X and Y will depend on the “Generator”.

Therefore, if the user just describes the oracle as a composition of any type of gates, classical or invertible, or reversible, or Hamiltonian gates and blocks, our simulator in Prolog would be able to verify correctness of his/her design. Moreover, various directions of simulating gates, blocks or subsystems are possible that can help the designer when he considers various architectures for complex oracles. Also, in some problems we do not know how many solutions a given problem instance has. In case that we want to find ALL solutions to our problem we use the Quantum Counting algorithm [28] in addition to Grover Algorithm. If we want to verify the correctness of our entire quantum algorithm composed of Quantum Counting Algorithm and Grover Algorithm, it is useful to know how many solutions our problem instance has (how many “true minterms” or “minterms with value 1” the oracle function has). Again, the Prolog simulator helps in this respect. Therefore, the Prolog simulator becomes a universal simulator of oracles, regardless if we want to use these oracles for classical Boolean logic as in Sect. 3, for reversible logic in Grover Algorithm as in Sect. 4, or for Invertible Logic as in Sect. 5. Prolog Simulator can be also used to verify any reversible logic circuits (with Ancillae) used in quantum algorithms such as Shor Algorithm and similar, as well as several Quantum Random Walk Algorithms.

7 Conclusions

Solving optimization problems can be reduced to the repeated solving of decision problems such as CSP with oracles modified at every CSP round. CSP problems can be solved by exercising oracles. These oracles can be exercised from inputs to output sequentially as in classical Boolean logic. In quantum they are exercised in parallel...
thanks to superposition and quantum parallelism used in the Grover Algorithm. Based on an observation that both these formerly known approaches solve the inverse function and that the Invertible Logic can be used to solve the inverse functions, in this paper we propose a general methodology to solve optimization and CSP problems based on designing oracles bottom–up from a hierarchy of blocks. These oracles can be exercised in Prolog from outputs to inputs. Next, as the first method, these oracles can be converted to reversible logic and used in Grover Algorithm (with adding ancillae qubits). As the second method, these classical logic oracles can be converted to Invertible Logic [5, 6, 17, 18, 20, 27] and realized with nano-magnetic spins. As the third method, Invertible Logic oracles can be emulated using standard FPGAs using stochastic number representations. Various number systems, radices and operators can be used to operate on numbers and encoded symbolic data in these oracles [1, 7]. Please observe that there are many systems to represent numbers and arithmetic operators executed on them [1, 2, 5, 20]. Some of these number systems may be better when used inside the gates and blocks of invertible logic. This is an area of current research. (It is very likely that some day FPGAs based on magnetic spin technology will be invented and fabricated). As the fourth method, the oracles can be exercised sequentially (but only from input to output) in standard Boolean Logic and realized in standard FPGAs. Finally, as the fifth method a complete quantum circuit can be built for a quantum neural network built as a composition of small quantum networks. The top network is an oracle, the small networks are gates and blocks.

Concluding, the very general “oracle-based” methodology for solving CSP and optimization problems outlined in this paper can be applied to many important problems from Design Automation, Logistics, Optimization, Control, Artificial Intelligence, Machine Learning and Robotics. The methodology will become even more practical with the appearance of: (1) quantum computers that will have more qubits, (2) magnetic spin technologies with higher number os gates, (3) standard FPGAs with very-high-quality hardware random number generators. It is also an open problem how well these quantum oracles will work in Noisy Intermediate Scale Quantum (NISQ) computers [48] without error correction.

References

1. Ardakani, A., Leduc-Primeau, F., Onizawa, N., Hanyu, T., Gross, W.J.: VLSI implementation of deep neural network using integral stochastic computing. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 25(10), 2588–2599 (2017)
2. Cheng, A., Tsai, E., Perkowski, M.: Methodology to create hardware oracles for solving constraint satisfaction problems. In: 22nd International Workshop on Post-Binary ULSI Systems, pp. 36–43. Toyama International Conference Center, Toyama (2013)
3. Dhawan, S., Perkowski, M.: Comparison of influence of two data-encoding methods for grover algorithm on quantum costs. In: ISMVL, pp. 176–181 (2011). https://doi.org/10.1109/ismvl.2011.29
4. Biamonte, J.D.: Non-perturbative k-body to two-body commuting conversion Hamiltonians and embedding problem instances into Ising spins. Phys. Rev. A 77, 052331 (2008)
5. Camusari, K., Faria, R., Sutton, B., Datta, S.: Stochastic p-bits for invertible logic. Phys. Rev. X 7, 031014 (2017)
6. Debashis, P., Faria, R., Camsari, K.Y., Appenzeller, J., Datta, S., Chen, Z.: Experimental demonstration of nanomagnet networks as hardware for Ising computing. In: IEEE International Electron Devices Meeting (IEDM), pp. 34.3.1–34.3.4 (2016)

7. Gaines, B.R.: Stochastic computing systems. In: Tou, J.T. (ed.) Advances in Information Systems Science. Advances in Information Systems Science, pp. 37–172. Springer, Boston (1969). https://doi.org/10.1007/978-1-4899-5841-9_2

8. Gao, P., Li, Y., Perkowski, M., Song, X.: Realization of quantum oracles using symmetries of Boolean functions. Quantum Inf. Comput. 20(5&6), 0417–0446 (2020)

9. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: 28th Annual ACM Symposium on Theory of Computing, pp. 212–219 (1996)

10. Hinton, G.E., Sejnowski, T.J., Ackley, D.H.: Boltzmann machines: constraint satisfaction networks that learn. Department of Computer Science, Carnegie-Mellon University, Technical report CMUCS-84-119 (1984)

11. Lee, B., Perkowski, M.: Quantum machine learning based on minimizing Kronecker-Reed-Muller forms and Grover search algorithm with hybrid oracles. In: 2016 Euromicro Conference on Digital System Design (DSD), pp. 413–422 (2016)

12. Li, Y., Tsai, Y., Perkowski, M., Song, X.: Grover-based Ashenhurst-Curtis decomposition using quantum language quipper. Quantum Inf. Comput. 19(1&2), 0035–0066 (2019)

13. Mishchenko, A., Perkowski, M.: Fast heuristic minimization of exclusive sums-of-products. In: RM 2001 Workshop (2001)

14. Monaco, J.V., Vindiola, M.M.: Factoring integers with a brain-inspired computer. IEEE Trans. Circuits Syst. I Regul. Pap. 65(3), 1051–1062 (2018)

15. Perkowski, M.: Methodology to design oracles for Grover algorithm, poster presentation. In: Workshop on Design Automation for Quantum Computers, IEEE 2017 International Conference On Computer Aided Design, Marriott Hotel, Irvine, CA (2017)

16. Luba, T., Selvaraj, H.: A general approach to Boolean function decomposition and its application in FPGA based synthesis. VLSI Des. 3(3–4), 289–300 (1995)

17. Pervaiz, A.Z., Ghantasala, L.A., Camsari, K., Datta, S.: Hardware emulation of stochastic p-bits for invertible logic. Sci. Rep. 7 (2017). Article No. 10994

18. Pervaiz, A.Z., Sutton, B.M., Ghantasala, L.A., Camsari, K.Y.: Weighted p-bits for FPGA implementation of probabilistic circuits. arXiv e-prints (2017)

19. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. 26, 1484–1509 (1997)

20. Smithson, S.C., Onizawa, N., Meyer, B.H., Gross, W.J., Hanyu, T.: Efficient CMOS invertible logic using stochastic computing. IEEE Trans. Circuits Syst. I Regul. Pap. 66(6), 2263–2274 (2019)

21. Tsai, E., Perkowski, M.: A quantum algorithm for automata encoding. Facta Universitatis. Ser. Electron. Energ. 33, 169–215 (2020)

22. Tsai, E., Perkowski, M.: Towards the Development of Quantum Design Automation Tools: A Methodology for Construction of Oracles to Solve Constraint Satisfaction Problems using Grover’s Algorithm (2020, Submitted)

23. Tsai, E., Perkowski, M.: Realization of Arbitrary Symmetric Functions in Quantum Logic Using Two-Qubit Gate (2020, Submitted)

24. Wang, Y., Perkowski, M.: Improved complexity of quantum oracles for ternary Grover algorithm for graph coloring. In: ISMVL, pp. 294–301 (2011). https://doi.org/10.1109/ismvl.2011.42

25. Whitfield, J.D., Faccin, M., Biamonte, J.D.: Ground-state spin logic. Europhys. Lett. 99(5), 57004 (2012)

26. Butler, J.T., Sasao, T.: Combinational computing. One object per clock. In: Reed-Muller Symposium, Toyama, Japan (2013)
27. Sutton, B., Camsari, K.Y., Behin-Aein, B., Datta, S.: Intrinsic optimization using stochastic nanomagnets. Sci. Rep. 7(1), 1–9 (2017)
28. Brassard, G., Höyer, P., Tapp, A.: Quantum counting. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 820–831. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055105
29. Venkatachalapathy, R.: Systems isomorphisms in stochastic dynamic systems. PSU, Systems Science, Ph.D. Dissertation (2019)
30. Sivakumar, S., Li, Y., Perkowski, M.: Grover Algorithm for Minimum Set of Support Problem of Multi-Valued Functions (2020, Submitted)
31. Zhang, W.: Quantum Algorithms for Two-Arm robot and generalization to Travelling Salesman Problem (2020, in Preparation)
32. Hou, W., Perkowski, M.: Quantum Algorithm for Knapsack problem (2020, Submitted)
33. Rawski, M., Jóźwiak, L., Luba, T.: Functional decomposition with an efficient input support selection for sub-functions based on information relationship measures. J. Syst. Architect. 47, 137–155 (2001)
34. Konieczny, P.A., Jóźwiak, L.: Minimal input support problem and algorithms to solve it, vol. 95-E-289. Eindhoven University of Technology Report E, Faculty of Electrical Engineering, Eindhoven, 01 January 1995
35. Mishchenko, A., Files, C., Perkowski, M., Steinbach, B., Dorotska, C.: Implicit algorithms for multi-valued input support manipulation. In: 4th International Workshop on Boolean Problems (2000)
36. Kiran, R.U., Reddy, P.K.: An improved multiple minimum support based approach to mine rare association rules. IEEE (2009). 978-1-4244-2765-9/09
37. Łuba, T., Rybnik, J.: Algorithmic approach to discernibility function with respect to attributes and objects reduction. Found. Comput. Decis. Sci. 18(3–4), 241–258 (1993)
38. Sasao, T., Fumishi, I., Iguchi, Y.: On an exact minimization of variables for incompletely specified index generation functions using SAT. Note Multiple-Valued Logic Jpn. 38, 1–8 (2015)
39. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, Cambridge (2000)
40. Boyer, M., Brassard, G., Høyer, P., Tapp, A.: Tight bounds on quantum searching. Fortschr. Phys. 46, 493 (1998)
41. Cross, A.: The IBM Q experience and QISKit open-source quantum computing software. APS March Meeting (2018). Abstract id L58.003. Bibcode 2018 APS .. MARL58003C
42. Al-Bayaty, A., Perkowski, M.: Simulating Boolean, Quantum and Invertible Logic Oracles using a Prolog-based system. Report PSU (2020, in Preparation)
43. Taha, M.M.A., Perkowski, M.: Realization of arithmetic operators based on stochastic number frequency signal representation. In: ISMVL 2018, pp. 215–220 (2018)
44. Cheng, A.: Designing FPGA Oracles for Cryptography Problems. PSU report (2013)
45. Li, Y.: Quantum Oracles for Graph Coloring and Maximum Clique. PSU report in preparation (2020)
46. Perkowski, M., Foote, D., Chen, Q., Al-Rabadi, A., Jozwiak, L.: Learning hardware using multiple-valued logic – Part 2: cube calculus and architecture. IEEE Micro Chips Syst. Softw. Appl. 22(3), 52–61 (2002)
47. Kohavi, Z.: Switching and Finite Automata Theory. McGraw-Hill, New York (1978)
48. Preskill, J.: Quantum computing in the NISQ era and beyond. arXiv:1801.00862v3 [quant-ph], 31 July 2018
49. Perkowski, M., Liu, J., Brown, J.: Quick software prototyping: CAD design of digital CAD algorithms. In: Zobrist, G. (ed.) Progress in Computer Aided VLSI Design, vol. 1, pp. 353–401. Ablex Publishing Corp, New York (1989)