Simulating quantum algorithms is a hard problem on classical computers, it usually needs exponential time and space. Viamontes et al. proposed a new data structure the Quantum Information Decision Diagram (QuIDD) to overcome this problem and implemented it in the QuIDDPro software. Using this structure several algorithms can be simulated on classical computers with polynomial time and space. In this paper we suggest further improvement and analyse in detail its behavior on Grover’s search algorithm.

Keywords
quantum algorithms, QuIDD, Grover’s algorithm
problems of this representation is that it needs $2^n$ leaves to store a function with $n$ variables. The Reduced Order BDD (ROBDD) introduces two reduction rules to the following effect:

1. There are no nodes $v$ and $v'$ such that the subgraphs rooted at $v$ and $v'$ are isomorphic.
2. There are no internal nodes with both its edges pointing to the same node.

$$f(x_1, x_2, x_3) = (x_1 \lor x_2 \lor \neg x_3) \land
\neg (x_1 \lor \neg x_2 \lor x_1) \land
\neg (x_1 \lor x_2 \lor x_3)$$

with two variable orderings. As it can be seen, the ROBDD can use significantly less nodes if the function and the ordering is right. Also, the size of the diagram depends on the ordering, as it can be seen in Fig. 2.

### 2.2 Quantum Information Decision Diagram

The QuIDD is a variant of Algebraic Decision Diagram (ADD) [1], which is based on the ROBDD. It is used to store a matrix with complex elements. The leaves in the QuIDD are pointers to an array, which stores the values of nodes. These values can be complex numbers.

The variables of QuIDD are the rows and columns of the represented matrix. More precisely the bits of the binary representation of rows and columns. The variable $R_i$ represents the $i$-th bit of the row, and $C_j$ represents the $j$-th bit of the column. The numbering starts with 0, where the zeroth bit is the most significant one. The ordering of the variables is rows and columns interleaved. This ordering is helpful when the matrices have some block structure, usually seen when they are constructed from smaller matrices by tensor products.

$$\begin{pmatrix}
\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\
\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \\
\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\
\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2}
\end{pmatrix}$$

Fig. 1 The BDD of a function using orderings $x_1, x_2, x_3$ and $x_3, x_2, x_1$.

Solid lines represent 1 edges, dotted lines 0 edges

Fig. 2 The ROBDD of the same function using the same orderings.

To illustrate the difference between the two structures, Fig. 1 and 2 show the BDD and ROBDD representations of the function.

Fig. 3 The Hadamard matrix acting on two qubits, and it’s QuIDD representation.
For example let us see Fig. 3. If we want to check the value of an element in the first row (represented as \( R_0 = 0, R_1 = 0 \)), we start traversing the tree at its root. The first node is \( R_0 \), so we choose the 0 child, which is represented by the dashed line. The next node in this traversal is \( R_1 \), where we again choose the dashed line, and arrive at a leaf. The pointer in the leaf is 1, so we check the value in the array at the 1 place. It is 0.5, so the value in question in the matrix is 0.5. Observe, that we did not use any column variable, because all the elements in the row are 0.5.

Let us check the value of the fourth element in the third row. Because the numbering starts with 0, the binary representation of the row is 10 and the column is 11. We begin the traversal at the root. At the \( R_0 \) node, we check the zeroth bit of the row, which is 1, so we follow the solid line. The next node is \( C_0 \). The binary form of the column is 11, so again, we follow the solid line. The next node is \( R_1 \), and the second bit of the row is 0, so we follow the dashed line to the leaf containing the pointer 0, which means the value is -0.5.

### 2.3 Important examples

The identity matrix is a good example, because it can easily be written as tensor product (also known as Kronecker product) of smaller identity matrices: \( I_n = I_1 \otimes I_{n-1} \), where \( I_1 \) is the \( 2^1 \times 2^1 \) identity matrix operating on \( n \) qubits. In matrix form:

\[
I_n = I_1 \otimes I_{n-1} = I_1 \otimes I_1 \otimes I_{n-2} =
\begin{bmatrix}
I_{n-1} & 0 \\
0 & I_{n-1}
\end{bmatrix}
\begin{bmatrix}
I_{n-2} & 0 & 0 & 0 \\
0 & I_{n-2} & 0 & 0 \\
0 & 0 & I_{n-2} & 0 \\
0 & 0 & 0 & I_{n-2}
\end{bmatrix}
\]

In the QuIDD representation this means that the identity operator on \( n \) qubits can be built by three nodes and two “leaves”. One leaf is the number 0, the other represents an identity operator on \( n - 1 \) qubits. Unfolding \( I_{n-1} \), we obtain a similar structure, but the “leaf” will be an \( n - 2 \) qubit identity operator, as it can be seen in Fig. 4. Repeating this process, we obtain the QuIDD representation of the \( n \) qubit identity matrix. Each new level of recursion adds three new nodes, therefore it has \( 3n \) inner nodes and two leaves. Thus the next proposition follows.

![Fig. 4](image)

**Proposition 1.** The QuIDD representation of the \( n \) qubit identity matrix uses \( 3n + 2 \) nodes. □

One of the most used operators in quantum algorithms is the Hadamard operator. The one qubit version of its matrix is

\[
H_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}
\]

It is easy to create Hadamard matrices that operate on more qubits, because \( H_n = H_1 \otimes H_{n - 1} \). The Hadamard matrix is a good fit to the QuIDD structure because of this property. We can write

![Fig. 5](image)
The matrix and QuIDD representation of the product $H_n R_n H_n$ can be stored like the identity matrix, only with different values in the leaves. Therefore this product also can be stored with $3n + 2$ nodes.

**Proposition 4.** The QuIDD representation of the $H_n R_n H_n$ product uses $3n + 2$ nodes. □

In quantum algorithms one typical way to represent a boolean function $f$ is by the operator $V_f$ that multiplies the vector $|i\rangle$ by $(-1)^{f(i)}$. Its matrix differs from the identity in that it has $(-1)$ instead of 1 where $f(i) = 1$. Call an $i$ marked if $f(i) = 1$. In the case of one marked element, we can construct the QuIDD representation like in the case of $R_n$. As we can see in Fig. 8, the place of the $(-1)$ does not change the structure of the QuIDD, only the edges between the nodes. So if there is only one marked element, the size of $V_f$ is the same as the size of $R_n$. 

**Fig. 7** The matrix and QuIDD representation of the product $H_n R_n H_n$.

Another important operator is $H_n R_n H_n$. It is easy to see that $H_n R_n H_n = I_n - 2P_n$, where $P_n$ is a $2^n \times 2^n$ projection matrix whose elements are $1/2^n$. So the $H_n R_n H_n$ product can be stored like the identity matrix, only with different values in the leaves. Therefore this product also can be stored with $3n + 2$ nodes.

**Proposition 3.** The $R_n$ matrix can be stored using $5n + 1$ nodes. □

**Fig. 6** The construction of matrix $R_n$.

Another important operator is $H_n R_n H_n$. It is easy to see that $H_n R_n H_n = I_n - 2P_n$, where $P_n$ is a $2^n \times 2^n$ projection matrix whose elements are $1/2^n$. So the $H_n R_n H_n$ product can be stored like the identity matrix, only with different values in the leaves. Therefore this product also can be stored with $3n + 2$ nodes.

**Proposition 4.** The QuIDD representation of the $H_n R_n H_n$ product uses $3n + 2$ nodes. □

In quantum algorithms one typical way to represent a boolean function $f$ is by the operator $V_f$ that multiplies the vector $|i\rangle$ by $(-1)^{f(i)}$. Its matrix differs from the identity in that it has $(-1)$ instead of 1 where $f(i) = 1$. Call an $i$ marked if $f(i) = 1$. In the case of one marked element, we can construct the QuIDD representation like in the case of $R_n$. As we can see in Fig. 8, the place of the $(-1)$ does not change the structure of the QuIDD, only the edges between the nodes. So if there is only one marked element, the size of $V_f$ is the same as the size of $R_n$. 

**Fig. 7** The matrix and QuIDD representation of the product $H_n R_n H_n$. 

Another important operator is $H_n R_n H_n$. It is easy to see that $H_n R_n H_n = I_n - 2P_n$, where $P_n$ is a $2^n \times 2^n$ projection matrix whose elements are $1/2^n$. So the $H_n R_n H_n$ product can be stored like the identity matrix, only with different values in the leaves. Therefore this product also can be stored with $3n + 2$ nodes.

**Proposition 4.** The QuIDD representation of the $H_n R_n H_n$ product uses $3n + 2$ nodes. □

In quantum algorithms one typical way to represent a boolean function $f$ is by the operator $V_f$ that multiplies the vector $|i\rangle$ by $(-1)^{f(i)}$. Its matrix differs from the identity in that it has $(-1)$ instead of 1 where $f(i) = 1$. Call an $i$ marked if $f(i) = 1$. In the case of one marked element, we can construct the QuIDD representation like in the case of $R_n$. As we can see in Fig. 8, the place of the $(-1)$ does not change the structure of the QuIDD, only the edges between the nodes. So if there is only one marked element, the size of $V_f$ is the same as the size of $R_n$. 

**Fig. 7** The matrix and QuIDD representation of the product $H_n R_n H_n$. 

Another important operator is $H_n R_n H_n$. It is easy to see that $H_n R_n H_n = I_n - 2P_n$, where $P_n$ is a $2^n \times 2^n$ projection matrix whose elements are $1/2^n$. So the $H_n R_n H_n$ product can be stored like the identity matrix, only with different values in the leaves. Therefore this product also can be stored with $3n + 2$ nodes. 

**Proposition 4.** The QuIDD representation of the $H_n R_n H_n$ product uses $3n + 2$ nodes. □

In quantum algorithms one typical way to represent a boolean function $f$ is by the operator $V_f$ that multiplies the vector $|i\rangle$ by $(-1)^{f(i)}$. Its matrix differs from the identity in that it has $(-1)$ instead of 1 where $f(i) = 1$. Call an $i$ marked if $f(i) = 1$. In the case of one marked element, we can construct the QuIDD representation like in the case of $R_n$. As we can see in Fig. 8, the place of the $(-1)$ does not change the structure of the QuIDD, only the edges between the nodes. So if there is only one marked element, the size of $V_f$ is the same as the size of $R_n$.
Proposition 5. The $V_j$ matrix operating on $n$ qubits can be stored using $5n + 1$ nodes, if there is one marked element. □

![Figure 8](image-url) An example for a two qubit $V_j$, where the third element is marked.

### 2.4 Basic operations

The main advantage of QuIDD is that is can use the operations of the Algebraic Decision Diagram for adding, multiplying or calculating the tensor product of two matrices. The basic operation is the Apply algorithm, which calculates the element-wise operation $op$ of two QuIDDs [2].

The Apply algorithm takes two nodes and an operator as parameters, and traverses the two trees, recursively calling itself in the process. When it reaches leaf nodes in both trees, it calculates $(leaf_1)op(leaf_2)$. Let $v_i$ and $v_j$ be two nodes, and $x_i$ and $x_j$ their variables, respectively. Let $T(v)$ and $E(v)$ be the two child of $v$, representing the 1 and 0 choices of its variable. Then $\text{Apply}(v_i, v_j, op)$ first checks $x_i$ and $x_j$ in the ordering.

If $x_i$ is before $x_j$ then it calls $\text{Apply}(T(v_j), v_j, op)$ and $\text{Apply}(E(v_j), v_j, op)$.

If $x_j$ is before $x_i$ then $\text{Apply}(v_i, T(v_j), op)$ and $\text{Apply}(v_i, E(v_j), op)$.

And if $x_i = x_j$ then $\text{Apply}(T(v_i), v_i, op)$ and $\text{Apply}(E(v_i), E(v_i), op)$.

This can be used for addition. It takes $O(ab)$ time and results in a structure of size $O(ab)$, where $a$ and $b$ are the number of nodes in the two operands.

The tensor product $A \otimes B$ of two $N \times N$ matrices is an $N^2 \times N^2$ matrix, which is a block matrix built from matrices $a_{ij}B_j$. To calculate this, we first shift all the variables of $B$ to be after the variables of $A$. Then we use the Apply algorithm. Because of the shift, now all of $B$’s elements are multiplied by $A$’s elements. The shifting of the variables have a complexity of $O(b)$, and because we use the standard Apply, its complexity still $O(ab)$.

So the tensor product can be computed in $O(ab)$ steps [11].

### 3 Ideas for Improvement

If we have a long chain of multiplications, as it is typical in case of quantum algorithms, one can pose the question what is the best way to calculate the product. Any bound on the time complexity of QuIDD’s multiplication implies an upper bound on the size of the product. In further multiplications one can calculate with this size bound to estimate the complexity.

Using the associativity of matrix multiplication, by a standard dynamical programming approach one can find the optimal regrouping of the chain in polynomial time (see for example Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein [3]).

However, in quantum algorithms one frequently has some additional information about the structure of some subproducts. When it is clear from the algorithm that a matrix corresponding to a subproduct has a small QuIDD representation then it might be a good idea to perform the multiplications in such order that we can use this information. We show how this can work on one of the most frequently used test problems of quantum circuit simulations, Grover’s quantum search algorithm.

### 4 Grover’s search algorithm

Lov Grover’s search algorithm is the first quantum search algorithm. It uses $\Theta(\sqrt{N})$ quantum steps to find a marked element in an $N$ element set. To do this, it uses an operator $G_n$ on an $n$ qubit state vector $|x\rangle$, where $n = \log N$. This can be simulated as a simple matrix-vector multiplication, $x' = G_n \cdot x$, where $x$ is the current state vector, and $x'$ is the next. To get a marked element with a probability of at least $1/4$, this has to be done approximately $\frac{\pi \sqrt{N}}{4 k}$ times, where $k$ is the number of the marked elements.

The operator $G_n$ can be written as $G_n = HRH \cdot V_j$ [5]. In this case $V_j$ corresponds to the function

$$f(x) = \begin{cases} 1, & \text{if } x \text{ is a marked element} \\ 0, & \text{otherwise} \end{cases}$$

If there is only one marked element, $G_n$ differs from $HRHV_j$, in one column, which is multiplied by $(-1)$. For simplicity let this column be the first one. If not, the structure of the QuIDD representation does not change, only the edges between the nodes. We can build $G_n$ from four different submatrices. Let $B_n$ be the matrix where all elements are the same, $D_n$ be the matrix where the diagonal elements are the same constant, and the additional elements are another one. Let $B'_n$ be the matrix, which is like $B_n$, but with the first column multiplied by $(-1)$, and $D'_n$ obtained from $D_n$, in the same way. Observe, that $G_n = D_n'$, using these notations we can write...
and so on.

Since $D_{s} = I_{s} - 2P_{s}$, and $B_{s} = -2P_{s}$, the corresponding QuIDD structures are known (see Proposition 4).

As we can see in Fig. 10, one new level in the QuIDD requires seven new nodes, and the “leaves” remain $D_{s}$, $B'_{s}$, $D'_{s}$, and $b$. So this matrix uses $O(n)$ nodes. Using that $D'_{n} = G_{n}$, see that

**Proposition 6.** Storing $G_{n}$ using the QuIDD structure requires $O(n)$ nodes. □

### 5 Complexity of Grover’s Algorithm using QuIDD

In the paper about the QuIDD structure [11] the potential speed up was illustrated through Grover’s algorithm. The runtime complexity of the algorithm was said to be $O(A^{15}n^3)$ where $A$ is the number of nodes in the oracle, and $n$ is the number of qubits. In this section, first we give a detailed explanation of this fact for the sake of completeness.

**5.1 The original bound**

The complexity of one matrix multiplication using the QuIDD data structure is $O(ab^2)$, where $a$ and $b$ is the number of nodes in the QuIDD representation of the two matrices. The first step in the algorithm is the initialization of the state vector which uses $n$ operations. Then we apply the Hadamard operator $H_n$ to the state vector $|x\rangle$. In the QuIDD structure $H_n$ needs only $O(n)$ nodes, and a vector with only one type of element can be stored in only one node. Therefore this multiplication uses $O(n^2)$ operations. However, the new state vector can be stored in one node, because every qubit is in the same state.

The iteration part of the algorithm uses the operator $(-H_{n}R_{n}H_{n}V_f)$ on the current state vector. Let $A$ be the number of nodes in $V_f$. With this the first multiplication $(V_f |x\rangle)$ uses $O(A^2)$ operations. The resulting state vector cannot use more nodes than this, so it can be stored with $O(A^2)$ nodes. The next operation is a multiplication by $H_n$, so the complexity is $O((nA^2)^2) = O(A^4n^2)$.

The next step in the algorithm is the operator $R_x$. Because it can be stored with $O(n)$ nodes, the cost of this operation is $O(nA^4n^2) = O(A^4n^6)$. The last step is a multiplication with $H_n$ with an overall complexity of $O((nA^4n^6)^2) = O(A^{15}n^3)$.

**Proposition 7.** One iteration of Grover’s algorithm has a runtime complexity of $O(n^{14}n^{14})$ which is polynomial in the number of qubits.
Proof. The initialization part of the algorithm has $O(1)$ complexity. The iteration part computes $(-HnRnHn|x>)$ with a complexity of $O(A^3 n^3)$ as shown before. □

If there is only one marked element, then $A = O(n)$. Thus the complexity of one iteration is $O(n^3)$.

**Proposition 8.** The number of nodes needed to store the state vector $|x>$ does not change between iterations.

**Proof.** The algorithm changes only the values of the marked and non-market elements, so the number of different coefficients stays the same. Moreover, a marked element remains marked and a non-marked remains non-marked during the algorithm. So in the QuIDD structure we only have to change the two values representing the amplitudes. □

### 5.2 The new bounds

From the test results in the original paper, it seems this upper bound is very pessimistic. So we use the associative property of the matrix multiplication to get tighter upper bounds, and possibly optimize the running time of the simulation. Because the matrices the algorithm uses do not change during the iteration, they can be computed ahead of time, and even multiplied together. We analyse different bracketings, and their effect on the complexity of the algorithm.

For this part, we assume that there is only one marked element.

If we calculate the matrix $HnRnHnVf$ at the beginning of the algorithm, the iteration only uses one matrix multiplication. The complexity of the iteration phase is $O((n \cdot 1)^2) = O(n^2)$, because the state vector can be stored with $O(1)$ nodes.

In the initialization phase, we have to calculate the matrix $HnRnHnVf$. Because of Propositions 2, 3 and 5 the complexity of the matrix multiplications is $O(((n \cdot n^2) \cdot n^2 \cdot n^2)) = O(n^7)$.

**Proposition 9.** The complexity of Grover’s algorithm using the QuIDD structure when we calculate the matrix $HnRnHnVf$ in the initialization step is $O(n^7)$ in the initialization step and $O(n^2)$ in the iteration step.

We can also examine the bracketing $(HnRnHn) \cdot (Vf) |x>$. The initialization for this is less complex, because we only have to calculate the product $HnRnHn$, which is $O(((n \cdot n^2) \cdot n^2)) = O(n^6)$.

**Proposition 10.** The vector $Vf |x>$ can be stored with $O(n)$ nodes. □

**Proof.** $|x>$ can be stored with $O(n)$ nodes, and $Vf$ changes only the sign of the marked element, so the overall structure of the vector stays the same. □

**Proposition 11.** The complexity of Grover’s algorithm with QuIDD structure when we calculate the matrix $HnRnHn$ in the initialization step is $O(n^6)$ step and $O(n^2)$ in the iteration step.

**Proof.** As we shown before the complexity of calculating $HnRnHn$ is $O(n^6)$. In the iteration part we have to use two matrix multiplications. First, we calculate $Vf |x>$ in $O(n^2)$ steps. The resulting vector will be a QuIDD with $O(n)$ nodes. Then multiplying this vector with $HnRnHn$ takes $O(n^2)$ steps. The iteration part thus has an $O(n^2)$ complexity. □

<table>
<thead>
<tr>
<th>Table 1 Complexity and measured runtime with one marked element</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>original</td>
</tr>
<tr>
<td>$HnRnHnVf</td>
</tr>
<tr>
<td>$(HnRnHn) \cdot (Vf)</td>
</tr>
</tbody>
</table>

Table 1 contains the simulation results of one iteration of Grover’s algorithm. We choose $n = 100$ qubits to simulate, in this case $N = 2^{100}$ would be the number of elements in the search. The whole algorithm would take $r \approx 8.8 \cdot 10^{14}$ iterations. The simulation was run on an Intel U7600 processor at 1.2 GHz with 2 GB RAM. Without compression, we would need to store $2^{100} \approx 10^{30}$ amplitudes only for the state vector, but with the QuIDDPro software the peak memory usage was under 13 MB in this simulation.

### 6 Conclusion

The QuIDDPro simulator helps expanding the number of qubits which can be used in testing algorithms, but still in a limited fashion. Using a simulator one can check the intermediate states of the algorithm without measurement, which can be very useful when someone tests an algorithm. We have shown that using different groupings of the operators in Grover’s algorithm, the computational complexity can be shifted from the iteration to the initialization. The runtime measurement shows the same results as the estimation.

Unfortunately, it is not clear that this works for every quantum algorithm. For example it would be interesting to see if the matrix of the Fourier transformation can be efficiently stored in some similar way.
Acknowledgement

Research is supported by OTKA-108947.

References


