Annales Univ. Sci. Budapest., Sect. Comp. 50 (2020) 199–217

COMPARING TWO QUANTUM ORACLES USING THE DEUTSCH–JOZSA ALGORITHM

Attila Kiss and Krisztián Varga

(Budapest, Hungary)

Dedicated to the memory of Professor Gisbert Stoyan

Communicated by András Benczúr

(Received January 27, 2020; accepted June 15, 2020)

Abstract. There are a lot of articles about quantum algorithms, which involve querying a black box function, often called an oracle. This paper proposes a way to combine two oracles and to determine how similar they are using the Deutsch–Jozsa algorithm. In this case being similar means, that the sets of marked states have just a few ones outside of their intersection. The new quantum algorithm is compared to its classical counterparts and a detailed analysis shows when it will outperform them.

1. Introduction

1.1. Deutsch–Jozsa algorithm

With new achivements every month regarding quantum supremacy, it seems like a commercial quantum computer might come sooner than we expect. Despite these advancements and decades of research we have only a few quantum algorithms which provide speedup over their classical counterparts, because as it turns out, these are not easy to find and there is no standardized method

Key words and phrases: Quantum computing, query, black box function, algorithms.
2010 Mathematics Subject Classification: 81P68, 68Q05, 68Q12.
The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).
https://doi.org/10.71352/ac.50.199

for creating one. The first such algorithm was the Deutsch–Jozsa algorithm [1], which provided exponential speedup, and it served as an inspiration for other famous algorithms such as Shor's algorithm [5] and Grover's algorithm [4]. The original problem of Deutsch was to decide if a Boolean function on one bit is constant or not. A classical algorithm for this has to check the two possible values of the input but the quantum algorithm solves it by just one query. For a generalization we call a Boolean function on n bits balanced if it is zero at half of the time. Assuming the function is constant or balanced the Deutsch–Jozsa algorithm decides which is the case using again only one query.

1.2. Combining quantum oracles

One frequently used unit of measurement for efficiency is the number of queries the algorithm has to make to a blackbox function, otherwise known as an oracle. For our purposes a quantum oracle is a function which operates on $n+1$ qubits, the first n qubits store the input of the function and it either flips the $n+1$st ancilla qubit or not, this way it can mark specific states. The marked states define a subset of all the possible 2^n states.

Two oracles can be combined to get new oracles in several ways. It has been shown that an intersection oracle [6] can be created using two extra helper qubits for the calculation and this simple method will be useful for us for improving our results later for our own comparison algorithm. The most interesting combination for us is when we execute two oracles after each other, because combining two oracles this way we get their symmetric difference.

Lemma 1.1. *Executing two oracles after each other on the same qubits marks only those states that can be found in their symmetric difference set.*

Proof. There are three different scenarios for a state:

1. Neither oracle marks it, this way the ancilla qubit remains zero.
2. Only one oracle marks it, this way the ancilla qubit is flipped only once.
3. Both oracles mark it, this way the ancilla qubit is flipped twice, hence it remains zero.

It can be seen that using this method only those states are marked that are in one of the oracles and not in both of them, which is the definition of the symmetric difference set. ■

The symmetric difference does not need any additional qubits, and because it tells the difference between the two functions it is perfect for our case. Since we can create the symmetric difference and the intersection set of two functions, it is straightforward that if these combined oracles are executed after each other, then the result will be a union oracle, because the symmetric difference of two sets with no intersection (in this case the symmetric difference and the intersection set of the two original oracles) is the same as their union. All the different combinations of oracles can be used for anything as any other normal oracle could be, for example to combine them even more or to use them in any oracle based algorithms such as the Grover search.

1.3. Analysing random Boolean functions

The Deutsch–Jozsa algorithm can determine if a function is constant or balanced, but this method only works with 100% probability, when these are the only two options. It has been analysed how effective this algorithm can be at deciding if a random Boolean function is constant or not by querying the oracle k times [3]. If the function is constant, then the algorithm will give the right answer, but it can easily classify close to constant functions as constant. At the worst case scenario, when only one input gives a different bit from all the others, the probability of measuring the state and getting constant as an answer tends to 1, which makes the Deutsch–Jozsa algorithm not fit for this problem.

2. Comparing oracles

In this section we propose a way to compare two different oracles. To be more precise, we are going to compare the subsets they are defining. First of all we are going to combine the oracles by querying them after each other (the order is not important). This way we are going to get their symmetric difference oracle. If we apply the Deutsch–Jozsa algorithm to this combined oracle, then we can determine if their symmetric difference is close to the empty set or not.

In case the oracles define the same subset, the combined function will be a constant zero, however it is important to note, that even if they do not have an intersection, that does not necessarily mean that it will be a constant 1. In order to get constant 1 the subsets must not have an intersection and their union must be equal to the set of all elements. It is obvious that with some knowledge about the functions beforehand we can create more accurate comparisons. For example if we know that each of the two functions can only cover a quarter of the whole problem space, and in case there is no intersection, then the measurement will determine that the function is balanced with a probability of 1. When we do not know anything about the functions, then we can just add an additional qubit, essentially doubling the problem space, so even in the worst case scenario, the symmetric difference of the functions can only cover at most half of the problem space. This way our measurements will be on a scale between constant zero and balanced. The almost constant functions are often classified as constant as mentioned earlier, which is good for us, because this means the functions are similar.

2.1. Differences between applying one and two oracles

The Deutsch–Jozsa and our comparison algorithm can be described by showing how the states of the qubits change over time:

We start with the $n+1$ qubit state, where the first n qubits are 0s and the last one is 1.

$$|0\rangle^{\otimes n} |1\rangle$$

Then we apply a Hadamard transform to each qubit to get the state:

$$\frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle \left(|0\rangle - |1\rangle\right)$$

After applying the quantum oracle we obtain the state:

$$\frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle \left(|f(x)\rangle - |1 \oplus f(x)\rangle\right)$$

Since $f(x)$ can only be 0 or 1, we can see that this state can be expressed as follows:

$$\frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle \left(|0\rangle - |1\rangle\right)$$

Finally we apply a Hadamard transform a second time for the first n qubits (the last one can be ignored):

$$\frac{1}{2^n} \sum_{x=0}^{2^n-1} (-1)^{f(x)} \left[\sum_{y=0}^{2^n-1} (-1)^{\langle x,y\rangle} |y\rangle\right] = \frac{1}{2^n} \sum_{y=0}^{2^n-1} \left[\sum_{x=0}^{2^n-1} (-1)^{f(x)} (-1)^{\langle x,y\rangle}) |y\rangle\right]$$

When we measure the state we obtain $|0\rangle^{\otimes n}$ with probability:

$$\left| \frac{1}{2^n} \sum_{x=0}^{2^n-1} (-1)^{f(x)} \right|^2$$

Our comparison algorithm is almost the same, but after applying the first oracle, we apply a second one too:

$$\frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}(-1)^{f_1(x)}\left|x\right\rangle\left(\left|0\right\rangle-\left|1\right\rangle\right)$$

$$\frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}(-1)^{f_1(x)}\left|x\right\rangle\left(\left|f_2(x)\right\rangle-\left|1\oplus f_2(x)\right\rangle\right)$$

$$\frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}(-1)^{f_1(x)}(-1)^{f_2(x)}\left|x\right\rangle\left(\left|0\right\rangle-\left|1\right\rangle\right)$$

$$\frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}(-1)^{f_1(x)\oplus f_2(x)}\left|x\right\rangle\left(\left|0\right\rangle-\left|1\right\rangle\right)$$

After applying the Hadamard gates a second time, the state can be measured. The probability of obtaining $\left|0\right\rangle^{\otimes n}\left|1\right\rangle$:

$$\left|\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{f_1(x)\oplus f_2(x)}\right|^2$$

To show the connection to the symmetric difference, this is the same as writing:

$$\left|1-\frac{|x:f_1(x)\neq f_2(x)|}{2^{n-1}}\right|^2$$

2.2. Analysis of the repeated execution of the comparison algorithm

Here we present the analysis which answers these questions:

- How many times should we execute the process, to get accurate results?
- How does this algorithm compare to its classical counterparts?
- When should we use it?

From an analytic point of view it does not matter that we are talking about two functions combined. We can think of this combined function as a simple Boolean function. The similarity can be interpreted as a threshold for how many times this function gives 1 as a result. For our analysis the following notations will be used:

- n is the number of input bits for the function f,
- $f \in \{0,1\}^n \times \{0,1\}$ for the classical cases and $f \in \{0,1\}^{n+1} \times \{0,1\}^{n+1}$ for the quantum case,
- $m = |x : f(x) = 1|$ for the classical cases and $m = |x : f(x) \neq |0\rangle|$ for the quantum case,
- s is the threshold for similarity, if $m \leq s$ then f is accepted (In other words the two functions that were combined to create f are similar),
- k is the amount of times the querying process is executed.

Here we define the quantum, and two classical algorithms for accepting a function:

Definition 2.1. The quantum algorithm: Execute the Deutsch–Jozsa algorithm with the function f k times. Let w be the number of times, the result was not the state $|0\rangle$. If $\frac{w}{k} \leq \frac{s}{2^n}$ then the function is accepted. "*Q1*" from now on.

Definition 2.2. The classical algorithm: Execute the function k times for random inputs from f's domain (the same input can be chosen multiple times). Let w be the number of times, the result was 1. If $\frac{w}{k} \leq \frac{s}{2^n}$ then the function is accepted. "*C1*" from now on.

Definition 2.3. The improved classical algorithm: Execute the function for all elements in a k sized subset of f's domain (the same input can not be chosen multiple times). Let w be the number of times, the result was 1. If $\frac{w}{k} \leq \frac{s}{2^n}$ then the function is accepted. "*C2*" from now on.

Next we calculate the probability of accepting a function for all three algorithms. "*C1*" is a classic example of "sampling with replacement" which follows the binomial distribution. Let w be the number of times, the result was 1. The probability of getting 1 as the output is $\frac{m}{2^n}$. Using the formula, the probability of $w = a$:

$$P_{C1}(w = a) = \binom{k}{a} \left(\frac{m}{2^n}\right)^a \left(1 - \frac{m}{2^n}\right)^{k-a}.$$

To get the probability where $w \leq a$ we have to sum all probabilities, where $w = i$ and $i \in [0..a]$:

$$P_{C1}(w \leq a) = \sum_{i=0}^{a} P_{C1}(w = i) = \sum_{i=0}^{a} \binom{k}{i} \left(\frac{m}{2^n}\right)^i \left(1 - \frac{m}{2^n}\right)^{k-i}.$$

The condition for acceptance is $\frac{w}{k} \leq \frac{s}{2^n}$, so the probability we are looking for is $P(w \leq \lfloor \frac{ks}{2^n} \rfloor)$.

”*Q1*” is similar to ”*C1*”, the only difference is the probability of not getting $|0\rangle$ as the outcome. From the analysis of [3], the probability of getting the state $|0\rangle$ after executing the Deutsch–Jozsa algorithm with a function that gives 1 m times:

$$P(|0\rangle) = \left(1 - \frac{m}{2^{n-1}}\right)^2$$

$$P(\neg\, |0\rangle) = 1 - \left(1 - \frac{m}{2^{n-1}}\right)^2$$

Inserting these probabilities into the formula:

$$P_{Q1}(w \le a) = \sum_{i=0}^{a} P_{Q1}(w = i) =$$

$$= \sum_{i=0}^{a} \binom{k}{i} \left(1 - \left(1 - \frac{m}{2^{n-1}}\right)^2\right)^i \left(\left(1 - \frac{m}{2^{n-1}}\right)^2\right)^{k-i}$$

”*C2*” is different from the other two, because it is analogous to the problem of ”sampling without replacement” which follows a hypergeometric distribution. Let w be the number of times, the result was 1. Using the formula, the probability of $w = a$:

$$P_{C2}(w = a) = \frac{\binom{m}{a}\binom{2^n - m}{k-a}}{\binom{2^n}{k}}.$$

Similarly to ”*C1*”, to get the probability where $w \le a$ we have to sum all probabilities, where $w = i$ and $i \in [0..a]$:

$$P_{C2}(w \le a) = \sum_{i=0}^{a} P_{C2}(w = i) = \sum_{i=0}^{a} \frac{\binom{m}{i}\binom{2^n - m}{k-i}}{\binom{2^n}{k}}.$$

Not getting the state $|0\rangle$ is analogous to getting 1 as the output in the classical versions, but the probabilities differ. Let p equal the probability in the classical algorithms ($p = \frac{m}{2^n}$). The quantum version has the probability $1 - \left(1 - \frac{m}{2^{n-1}}\right)^2$ which can be expressed with p: $4p(1-p)$. *Figure* 1 shows these probabilities for all possible m values. Since we will only use functions, where $m \le 2^{n-1}$ only the left side of the graph is relevant for us. The reason for this is that the problem space will always be doubled, in order to force all results to fall between constant 0 and balanced. The graph shows that the quantum version has a higher probability for getting non zero outputs and this means that the probability of acceptance when using ”*Q1*” will tend to be smaller than when using ”*C1*”. The reason behind this is that they use the same binomial distribution formula. The effect of this probability difference can be seen in *Figure* 2. It clearly shows, that ”*Q1*” can not be used to determine if

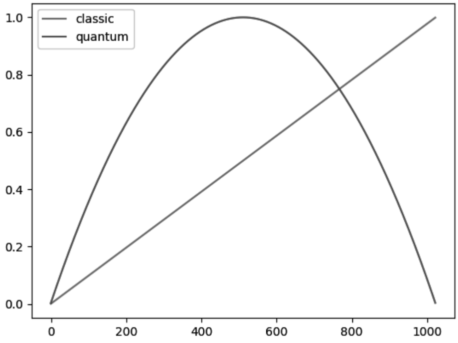
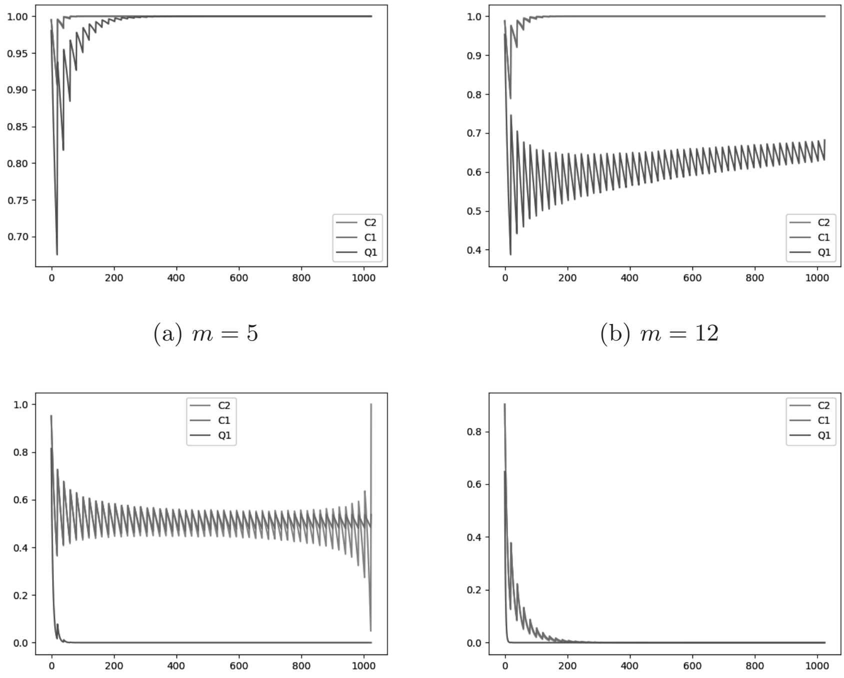


Figure 1: Different probabilities for obtaining a non zero output for all possible m values on the x axis ($n = 10$)



(a) $m = 5$

(b) $m = 12$

(c) $m = 50$

(d) $m = 100$

Figure 2: For "*C1*", "*C2*" and "*Q1*" the probability of accepting a function with all possible values for k on the x axis. ($n = 10$ and $s = 50$)

a function should be accepted or not. If it should be accepted, then the other classical algorithms are converging to 1 a lot faster and in some cases "*Q1*" converges to a value less then 1 (that can even be 0) as k gets bigger. Although when the function should be rejected, it converges faster to 0 then the others, but without a solution to the other half of the problem "*Q1*" is not suitable for us.

To achieve similar or better performance than the classical algorithms, we have to modify our quantum algorithm. Since n, m and s are given, the probability of getting non 0 output can not be adjusted. The only parameter that can be changed to benefit us is s. If we increase s, then the probability $P(w \leq \lfloor \frac{ks}{2^n} \rfloor)$ will increase too. To get the quantum algorithm to behave the same way as "*C1*", we have to examine the special case, when $m = s$. In this case the limes of $P_{Q1}(w <= s)$ should be equal to $\frac{1}{2}$ when k tends to $+\infty$. Let y be the coefficient when increasing s. If we want to know exactly the relation between y and all the other parameters, then we have to solve a complicated equation:

$$\lim_{k \to +\infty} \sum_{i=0}^{\lfloor \frac{ksy}{2^n} \rfloor} \binom{k}{i} \left(1 - \left(1 - \frac{m}{2^{n-1}}\right)^2\right)^i \left(\left(1 - \frac{m}{2^{n-1}}\right)^2\right)^{k-i} = \frac{1}{2}$$

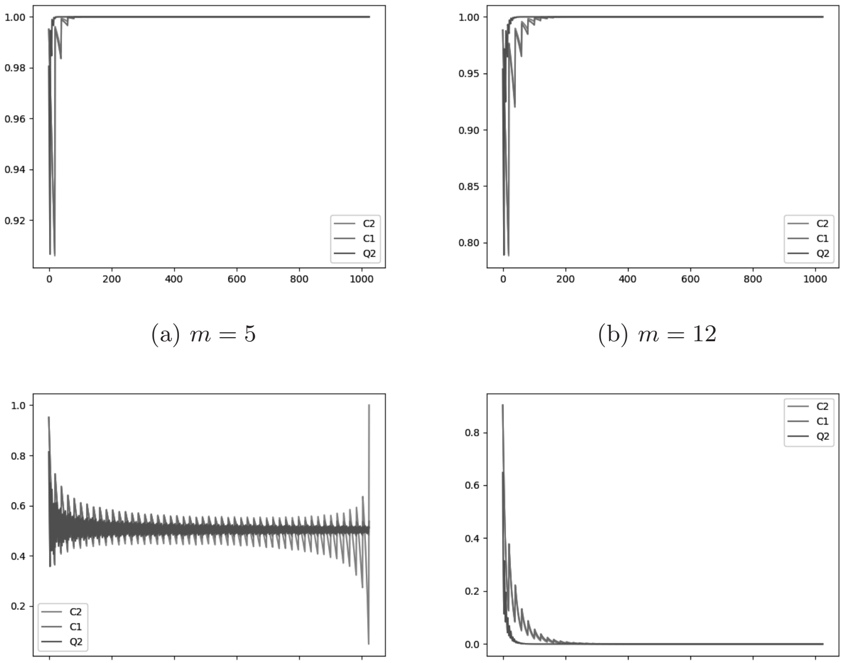
Instead of calculating this, we looked for an alternative solution. For given n and $m = s$ we set different y values manually until $\lim_{k \to 2^n} P_{Q1}(w \leq \lfloor \frac{ksy}{2^n} \rfloor)$ increased to $\frac{1}{2}$. The y values we get with this method have an interesting property. If a y value that works with given n' and $s' = m'$, then the same y value will work for any n and $s = m$ where $\frac{n}{s} = \frac{n'}{s'}$. This means that y should depend only on the value of $\frac{n}{s}$. We choose n to be 10, then we manually calculated y for different $s = m$ values. After we have the y values, we can plot the (s, y) pairs. Connecting these points resulted in a linear line. A line could be fitted to the points that gave this equation:

$$y = -\frac{s}{2^{n-2}} + 4$$

If this y coefficient is used in "*Q1*", then the results will match "*C1*" when $s = m$. The results can be connected to $P(\neg |00..0\rangle)$ that can be seen in *Figure* 1. If we plot $\frac{sy}{2^n}$ for all possible s values, then we get the same curve.

Definition 2.4. The improved quantum algorithm: Execute the Deutsch–Jozsa algorithm with the function f k times. Let w be the number of times, the result was not the state $|0\rangle$. If $\frac{w}{k} \leq \frac{sy}{2^n}$, where $y = -\frac{s}{2^{n-2}} + 4$, then the function is accepted. "*Q2*" from now on.

The new results for ”*Q2*” can be seen in *Figure* 3. The improved quantum algorithm behaves the same way as ”*C1*”, but it converges faster to the right answer. This faster convergence can be expressed with the amount of time it takes to have the probability to be higher or lower than a t threshold. *Figure* 4 and *Figure* 5 show the different k values, where the probability becomes higher than $\frac{99}{100}$. Each of these k values are the smallest such value ($k \in [1..2^n]$) that the average probability of accepting after k and $k-1$ steps is higher than $\frac{99}{100}$. The average is used, because as seen in *Figure* 3, the probability is not a smooth line. The results show that when m is close to s ”*C2*” performs better, but when m (approximately) $\leq 0,95s-6$ then ”*Q2*” takes the lead. The three algorithms tend to produce k values close to each other when m gets smaller.



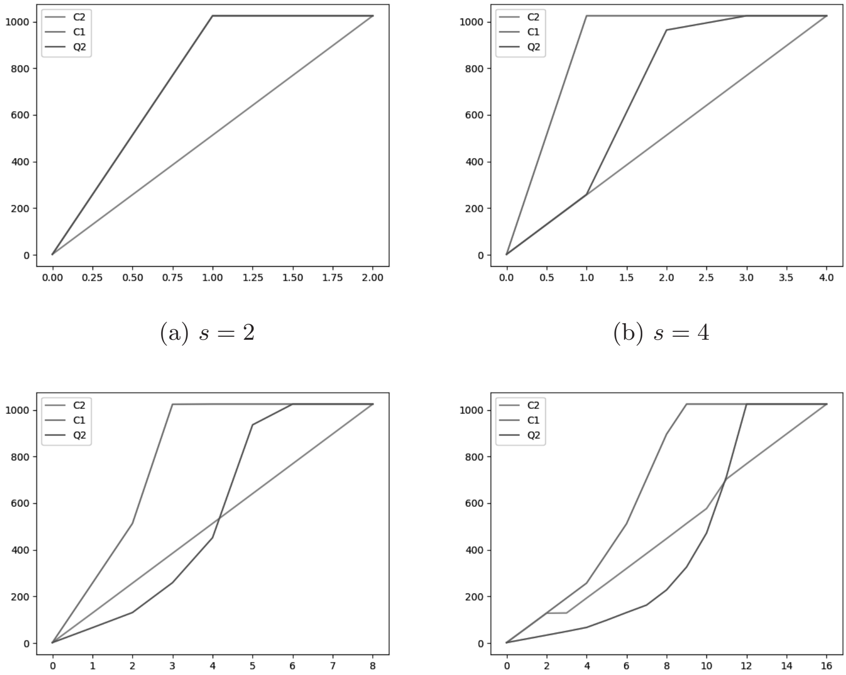
(a) $m = 5$

(b) $m = 12$

(c) $m = 50$

(d) $m = 100$

Figure 3: For ”*C1*”, ”*C2*” and ”*Q2*” the probability of accepting a function with all possible values for k on the x axis. ($n = 10$ and $s = 50$)



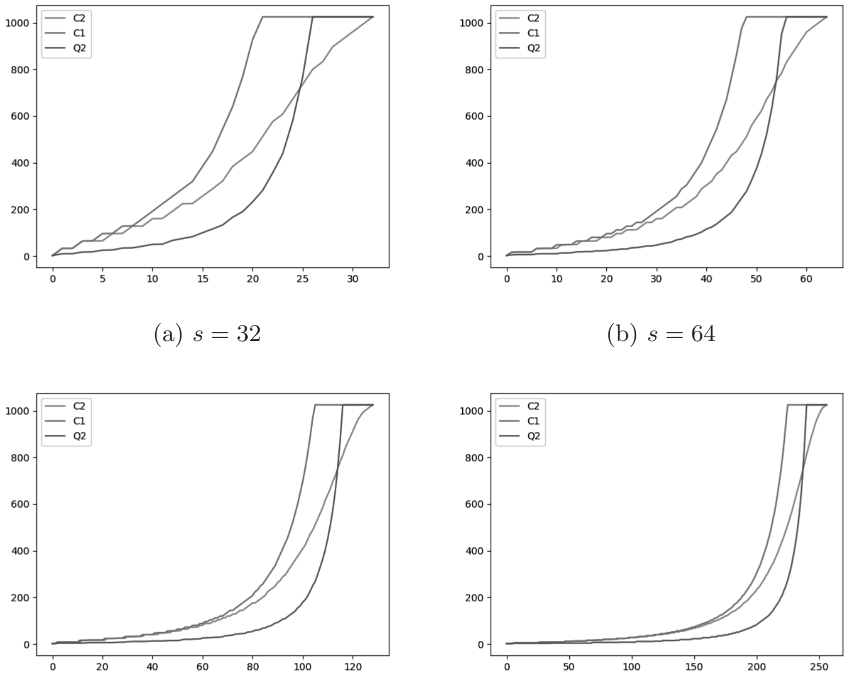
(a) $s = 2$

(b) $s = 4$

(c) $s = 8$

(d) $s = 16$

Figure 4: For "*C1*", "*C2*" and "*Q2*" the different k values for all possible m values (on the x axis) when s is fixed, where k represents the smallest $k \in [1..2^n]$, for which the average probability of accepting a function after k and $k-1$ query to the algorithms is higher than $\frac{99}{100}$. ($n = 10$, $s \in \{2, 4, 8, 16\}$, $m \in [0..s]$)



(a) $s = 32$

(b) $s = 64$

(c) $s = 128$

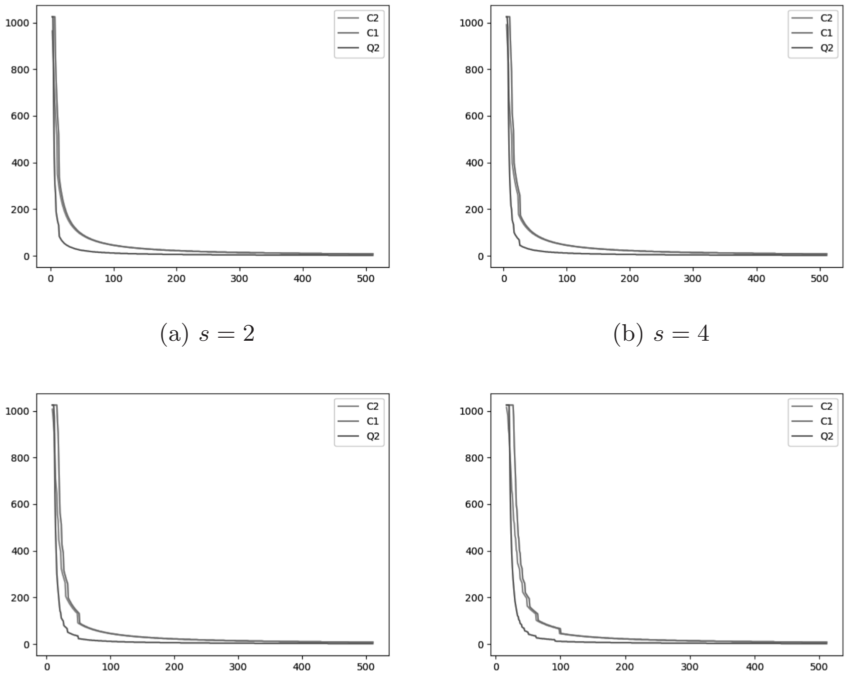
(d) $s = 256$

Figure 5: For ”*C1*”, ”*C2*” and ”*Q2*” the different k values for all possible m values (on the x axis) when s is fixed, where k represents the smallest $k \in [1..2^n]$, for which the average probability of accepting a function after k and $k-1$ query to the algorithms is higher than $\frac{99}{100}$. ($n = 10$, $s \in \{32, 64, 128, 256\}$, $m \in [0..s]$)

A similar result can be observed in *Figure* 6 and *Figure* 7, when the function should not be accepted. Here each of the k values are the smallest such value ($k \in [1..2^n]$) that the average probability of not accepting after k and $k-1$ steps is lower than $\frac{1}{100}$. When m is close to s, then ”*C2*” performs better, but when m (approximately) $\geq 1,05s + 6$ then ”*Q2*” becomes the fastest one to converge to 0. The graphs in *Figures* 4, 5, 6 and 7 can be used to estimate, how many k iterations are needed in order to achieve the correct answer with at least 99% accuracy. Keep in mind, that when m is close to s, then ”*Q2*” will never reach this level of accuracy.

An algorithm is considered to be better at a given m value then the other, if it converges faster to the correct answer. Assuming a uniform distribution of m values, then ”*Q2*” is better than the others at accepting, when $\frac{s}{2} \leq 0,95s - 6$.

Rearranging this inequality gives $13,333 < 14 \leq s$. Assuming the same for not accepting, "*Q2*" is better, when $\frac{2^n - s}{2} \geq 1,05s + 6$. This can be simplified to $s \leq \frac{2^n - 12}{3,1}$. In summary, "*Q2*" should be used, when $14 \leq s \leq \frac{2^n - 12}{3,1}$ and it will be the most efficient, if the m values are clustered at two points, one less than and other greater than s, and these clusters are as far from s as possible.



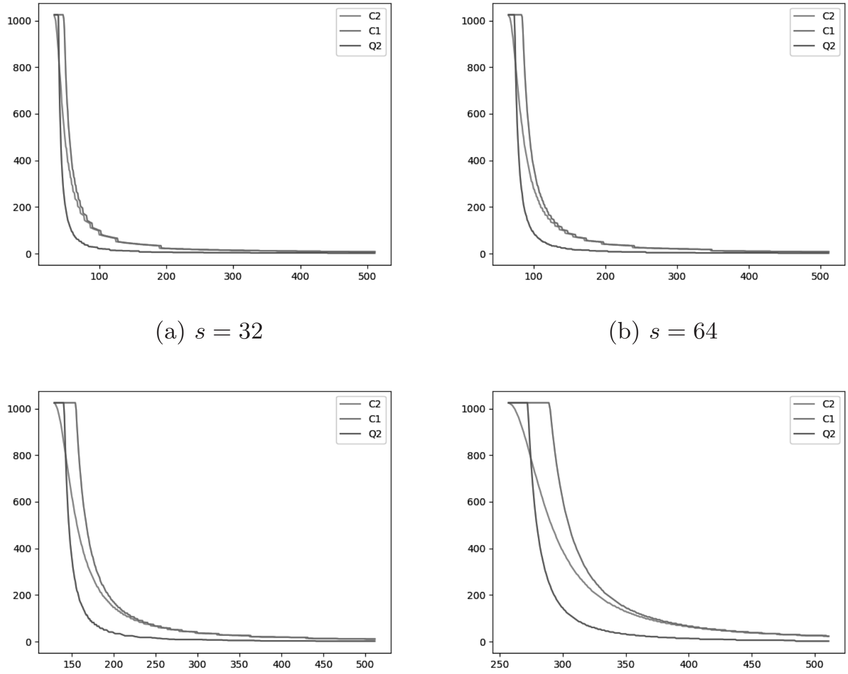
(a) $s = 2$

(b) $s = 4$

(c) $s = 8$

(d) $s = 16$

Figure 6: For "*C1*", "*C2*" and "*Q2*" the different k values for all possible m values (on the x axis) when s is fixed, where k represents the smallest $k \in [1..2^n]$, for which the average probability of not accepting a function after k and $k-1$ query to the algorithms is lower than $\frac{1}{100}$. ($n = 10$, $s \in \{2, 4, 8, 16\}$, $m \in [(s+1)..2^{n-1}]$)



(a) $s = 32$

(b) $s = 64$

(c) $s = 128$

(d) $s = 256$

Figure 7: For "*C1*", "*C2*" and "*Q2*" the different k values for all possible m values (on the x axis) when s is fixed, where k represents the smallest $k \in [1..2^n]$, for which the average probability of not accepting a function after k and $k-1$ query to the algorithms is lower than $\frac{1}{100}$. ($n = 10$, $s \in \{32, 64, 128, 256\}$, $m \in [(s+1)..2^{n-1}]$)

3. Experiments

Now that we are familiar with the theoretical background, we should see it in practice. As of today, there are not a lot of quantum computers for the average user, but there are quantum simulators, and quantum cloud computing services that can be used for testing. Our algorithm was implemented using quantum gates on the IBMQ Quantum Experience quantum computing service. The experiments mostly use 2 and 3 qubit oracles, because circuits with small number of qubits are easy to understand and create. An ancilla qubit, sometimes extra qubits are also needed for computing either an oracle or the combination of two oracles. The previously mentioned additional qubit can be implemented too to double the the number of possible inputs and to normalize the results. All of the examples were executed 1000 times, in order to show that they will give correct results.

The oracles were created using Toffoli gates [2]. A Toffoli gate will mark 1 or more states and it connects to 3 qubits. The 2 qubits where the gate has dots determine if the third one's value should be negated or not. By default the gate activates if both input qubits are 0, but this can be altered if a NOT gate is placed before and after on one or both input qubits. There are four different states that can be marked this way and if there are more than 3 qubits, then the gate will mark more states. In summary, it marks all the states that will negate the ancilla qubit and if a qubit is not an input to this gate, then its value does not matter and it will double the number of marked states.

The circuit in *Figure* 8 shows the generalised algorithm for three qubits, where F and G are the two functions.

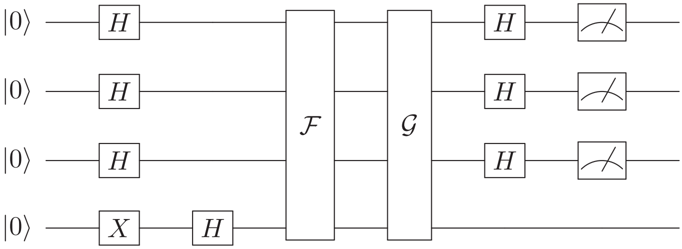


Figure 8: The proposed general algorithm

3.1. Two different oracles with intersection

Our first real example (*Figure* 9) uses four qubits. One of them is the ancilla bit, therefore the number of possible states is $2^3 = 8$. The first two Toffoli gates (and the NOT gates before and after) correspond to the first oracle F, and the second two to the second oracle G. It can be noticed that the first part of F is the same as the first part of G, this will give their intersection, the states 000 and 001. Both F and G oracles mark two additional states, 100, 110 for F and 101, 111 for G.

Both F and G marked four states initially, which would make it possible that their combination will mark more than half of the states (in case they have no intersection), but we know that this will not happen here. There are maximum eight possible states and that is twice the number of the marked states, therefore the combination of F and G (their symmetric difference function) is balanced, and every time the outcome is measured it gives the same state: 001.

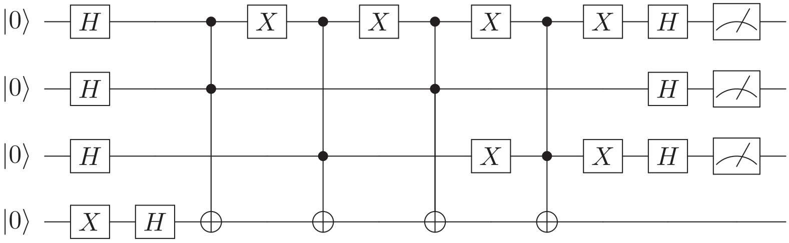


Figure 9: Two oracles combined that gives a balanced result

3.2. Two different oracles with no intersection

The next example (*Figure* 10) has two oracles (F referring to the first two Toffoli gates and G to the last two) that have no intersection and their union will mark all possible states. If we execute this circuit and measure the outcome, the result will always be 00, which means the function is constant. The problem is, that it is a ”constant 1” type of function, which means the symmetric difference is huge, but ”constant 0” type functions will produce the same outcome, and in that case the symmetric difference is small.

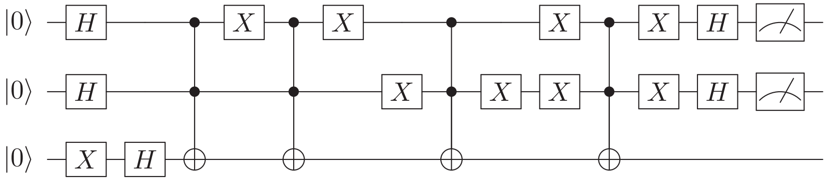


Figure 10: Oracles marking all the possible states

To counter this problem, we can introduce one more qubit which will double the number of possible states and therefore, even when the demonstrated worst case scenario happens, the result will be that it is a balanced function. This additional qubit will always be marked to be 0, this way the number of marked states will not increase. More qubits need to be added to achieve this. F and G oracles will be intersected with a newly created one. The new oracle consists only of a single CNOT gate that connects the additional qubit to one of the helper qubits. F and G will connect to the other helper qubit. Finally an intersection oracle is created with a Toffoli gate connecting the two helper qubits to the original ancilla qubit. The new oracle, F and G are executed again to negate their effects on the helper qubits. (The helper qubits are needed to perform the intersection.) *Figure* 11 illustrates the above described circuit.

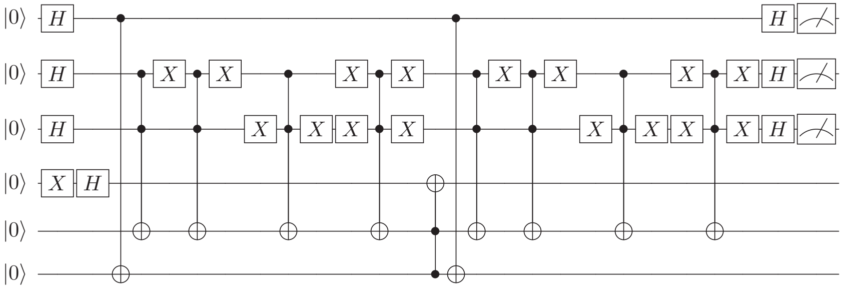


Figure 11: Example of doubling the number of possible states

The new circuit will always have the outcome state of 001, which means that it is a balanced function, which is exactly what we wanted. The schematic circuit in *Figure* 12 shows the general concept.

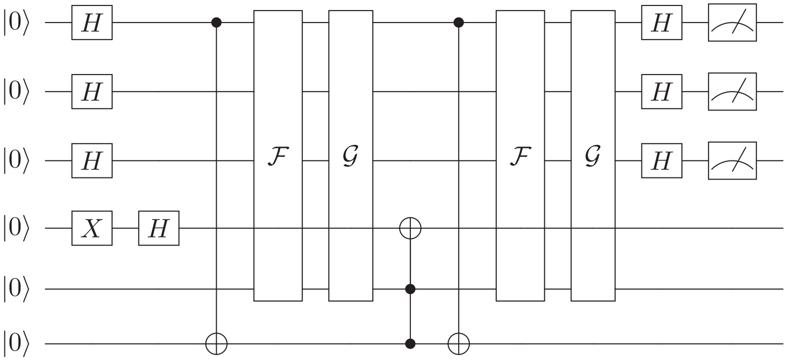


Figure 12: Generalized version of the example in *Figure* 11

3.3. Two identical oracles

There is one special case that was not discussed yet. What happens when the oracles are the same? In quantum computing every operation must be reversible and both the CNOT and Toffoli gates are inverses of themselves, therefore by applying these operations twice, the output should be the same as the input was. Our oracles consists of these two gates, and by following this logic, executing the same oracle two times is the same as the identity operation, which means no states will be marked and the output of our algorithm would always be $|0\rangle$, in other words: a constant zero function. Unfortunately we can only execute the algorithm a fixed number of times and our result will be right with a certain probability, therefore this method is not good for checking if the functions are equivalent, but it is sufficient to measure how similar they are.

3.4. Two general oracles

Figure 13 shows an average scenario, where the symmetric difference function is neither constant nor balanced. The combined oracle only marks two states out of all possible $2^n = 8$. 2 out of 8 is right in the middle of the constant-balanced scale and the corresponding results are the 001 010 011 000 states, all with a probability of 25%.

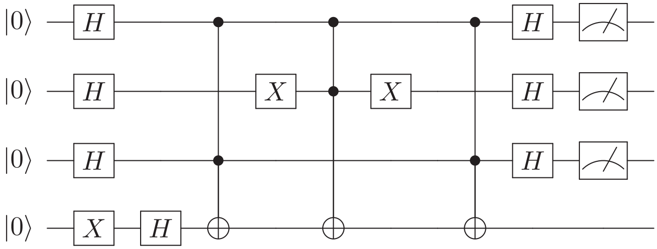


Figure 13: Average case example

4. Conclusion

We had shown that the Deutch-Jozsa algorithm can be used for comparing oracles, by executing it on the combined oracle. The results after the repeated execution of this method can be used to determine if the number of states in the symmetric difference set is below or not a predefined similarity threshold. Although the results are not correct 100% of the time, but the probability of getting the wrong answer can be minimized to a desired amount by executing the algorithm enough times. We found a way to ensure that no additional knowledge is required of the functions in order to apply the algorithm to them, by doubling the number of possible states. The proposed quantum algorithm was compared to its classical counterparts and although not always, but it often gave better results. The only times it performed worse was when the size of the symmetric difference was close to the similarity threshold. This method of comparison can have useful applications in the future, when quantum computers are more common, because comparison is a frequent task in everyday life and it is easy to find bijection between Boolean functions and comparable objects, for example pictures or long DNA sequences. This algorithm is especially good at classifying two clusters, where one is below the threshold and the other is above it. It can not be used when small differences are important, because it will never give a precise answer, but it can be used as a deciding factor in order to determine if the precise equality checking is even necessary.

References

[1] **Deutsch, D. and R. Jozsa,** Rapid solution of problems by quantum computation, *Proc. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci.*, **439** (1992), 553–558.

[2] **Toffoli, T.,** Computation and construction universality of reversible cellular automata, *J. Comput. Syst. Sci*, **15** (1977), 213–231.

[3] **Benatti, F. and L. Marinatto,** On deciding whether a Boolean function is constant or not, *International Journal of Quantum Information*, **1** (2003), 237–246.

[4] **Grover, L.K.,** A fast quantum mechanical algorithm for database search, *Symposium on the Theory of Computing, ACM*, **28** (1996), 212–219.

[5] **Shor, P.W.,** Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, *Symposium on Foundations of Computer Science, IEEE*, **35** (1994), 124–134.

[6] **Salman, T. and Y. Baram,** Quantum set intersection and its application to associative memory, *J. Mach. Learn. Res.*, **13** (2012), 3177–3206.

A. Kiss and K. Varga
Eötvös Loránd University
Faculty of Informatics
Department of Information Systems
Budapest
Hungary
kiss@inf.elte.hu
scout@inf.elte.hu