Course Project Final Report

Title: Spooky Tiq Taq Toe

Submitted By

Asad Tariq[1](at05439)

Sudais Baig[1] (sb05588)

CS/PHY 314/300

Quantum Computing

Section

L1

Instructor

Abdullah Khalid, Ph.D.

From the department of Integrated Sciences and Mathematics (iSciM)

Dhanani School of Science and Engineering

Habib University

6th December, 2021

[1]BS Computer Science 2023

Contents

1 Introduction

Quantum Tic Tac Toe is a unique spin on the classic tic tac toe game. It differs from the classic version of the game courtesy of the **superposition** rule which is essentially the crux of the game. Rather than having a discrete position for a particular **X** or **O**, each of the two can be in two different squares simultaneously until the outcome of the game is measured; at which point the superposition collapses and one of the two chosen squares hosts the symbol. Each state in the game can either be an entangled state or a non-entangled one [1] [2].

Several implementations of the Quantum Tic Tac Toe have been made in recent years, both web based and on mobile platforms, that have a graphical user interface for the players to play the game on. One such game, made by Evert van Nieuwenburg [3], is the one we will be drawing inspiration from and hoping to implement in a similar manner.

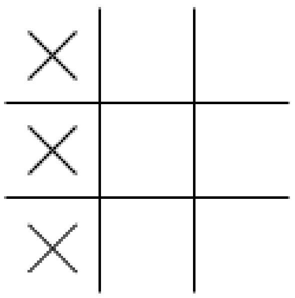
Our primary aim for this project will be to successfully build a working Quantum Tic Tac Toe game in Python with at least the basic superposition implemented. This corresponds to the user being allowed to make a superposition choice of placing his/her symbol on the board rather than placing it in a discrete manner. Furthermore, we will also aim to try to provide a measure of the superiority, if there exists at all, of a quantum player as compared to a classical one [4]. This will be done mathematically by taking into consideration all the possible moves a quantum player can make as compared to a classical player. In essence we will attempt to gauge the probability that whether or not a player playing with the quantum rules can always win against a player playing with the classical rules.

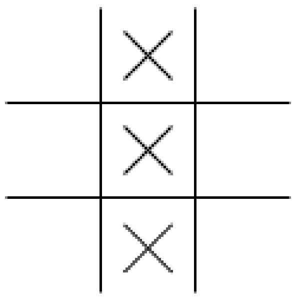
The secondary aim of our endeavour will include quantum enhancements to our base version of the game such as adding the possibility of converting a previously occupied square back into a state of superposition with the help of a Hadamard gate for example. This will result in the game becoming more complex and possibly taking longer to complete [4].

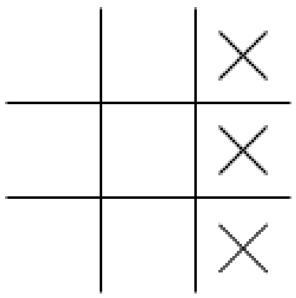
2 Background

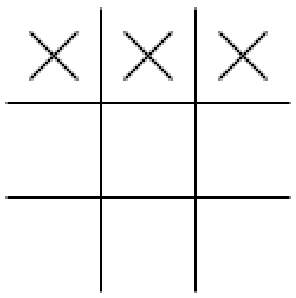
2.1 Classical Tic Tac Toe

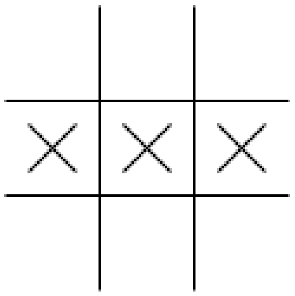
Classical Tic Tac Toe, a game that we must all have played in the past, has discrete positions upon which players can place their symbols. Therefore, we also have a finite set of sequences that can be made by either player to win the game. Those **eight** combinations are as shown below:

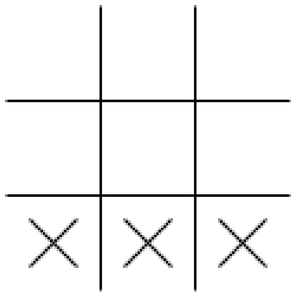

(a) Sequence 1 of 8

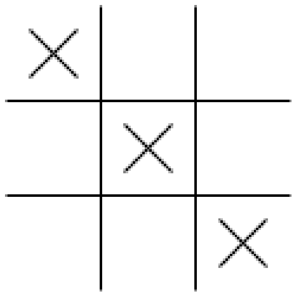

(b) Sequence 2 of 8


(c) Sequence 3 of 8


(d) Sequence 4 of 8


(e) Sequence 5 of 8


(f) Sequence 6 of 8


(g) Sequence 7 of 8

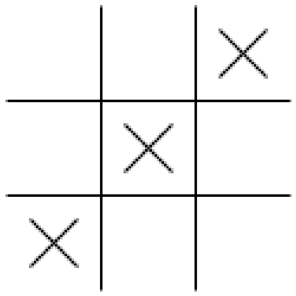

(h) Sequence 8 of 8

Figure 1: All possible winning sequences in Classical Tic Tac Toe

There can only be one winner in the classical version of this game, or it will result in a draw where no one wins. A certain set of moves will always enable a player to win, hence there is always a possibility of the game being biased towards a particular player.

2.2 Superposition

Quantum circuits, unlike classical circuits, can exist in multiple states in a particular instance of time. This particular feature of Quantum circuits is what is known as **superposition**. In actuality it is mathematically complex concept and can be difficult for people to grasp [1].

Quantum Tiq Taq Toe enables each player to mark two positions, instead of just a single one, upon which they can place their respective symbols if they desire to do so. Therefore, the board also exists in multiple states at the same moment. An example of such a move is as shown below in **Figure 2**:

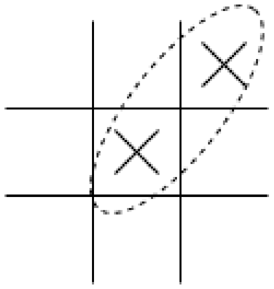
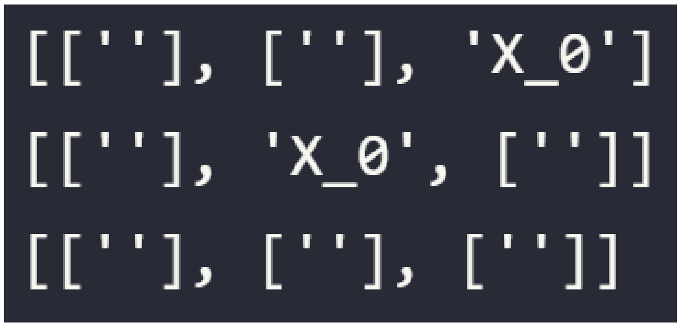


Figure 2: An example of a superposition move shown on a classical tic tac toe game board.

Superposition moves are the crux of the whole game and are the distinguishing feature of this version of the game that differentiate it from the classical one. These moves are represented on the tic tac toe board in our game as shown in **Figure 3** below.



```
[[''], [''], 'X_0']
[[''], 'X_0', ['']]
[[''], [''], ['']]
```

Figure 3: An example of how a superposition move would be represented on the tic tac toe game board.

2.2.1 How is a superposition achieved?

The easiest method of converting a regular qubit state into a superposition state is to apply the Hadamard gate on it (pictorially shown below in **Figure 4**) . This ensures that each possibility in the superposition has an equal probability of occurring when it is measured at any instance.

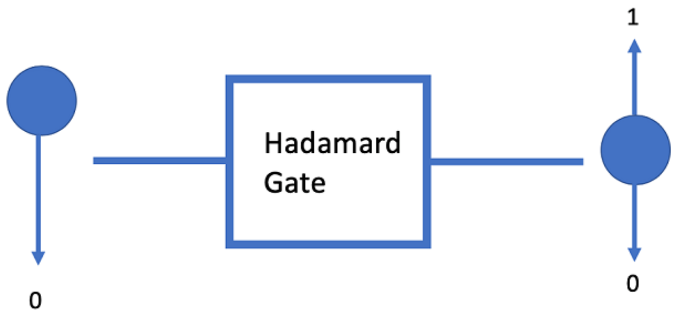


Figure 4: The Hadamard gate converts a regular qubit state into a superposition state [5]

This results in our game providing the possibility of letting the player choose two indexes upon which they want to place their respective symbols. It should also be noted that the classical rules of the game also apply to this version of the game, and thus the game can still result in a stalemate.

3 The Rules

3.1 Classical Tic Tac Toe

The rules of this version are fairly simple and straightforward and are ones that we are all well acquainted with.

1. This is a two player game and each player takes turns to place their symbol (either an **X** or an **O**) on a 3 by 3 game board.
2. Each square of the game board can only house a single symbol and once a symbol has been placed in a particular square it remains there until the end of the game.
3. In order to win, a player must achieve a sequence of 3 symbols in a row, either vertically, horizontally or diagonally as shown in **Figure 1, sub-figures (a) - (h)**.

3.2 Quantum Tiq Taq Toe

The rules for this version of the game are mostly the same except for one major difference which is with regards to the way a move can be made on the game board.

1. A two player game in which each player takes turns to place their symbol (either an **X** or an **O**) on either one square (classical) or a combination of any two squares (superposition) on a 3 by 3 game board.
2. Each square of the game board can still only house a single symbol. In the case that the move made was classical, the symbol stays there till the end of the game. However, if the move made was a superposition move then one of the indexes chosen to host the symbol would result in being blank after the measurements have been taken.
3. As before, in order to win, a player is supposed to achieve a sequence of 3 symbols in a row, either vertically, horizontally or diagonally as **Figure 1, sub-figures (a) - (h)**.

4 Mathematical Proofs

The probability that a game of classical tic tac toe always results in the first player winning even when the moves are chosen randomly is more than **50%**! In order to show the probabilities of player one always winning, player two always winning or the game always ending up in a draw, mathematically, we need to first define the number of all possible board combinations and the number of configurations that can end up in a win in 5, 6, 7, 8 and 9 moves. [6] [7]

$$\text{Total number of possible board configurations} = 9! = 362,880 \tag{1}$$

Since there are 3 vertical, 3 horizontal and 2 diagonal lines, we have a total of 8 lines of three squares. Therefore the number of games ending on the 5^{th} move in a win are:

$$\text{P(5 moves)} = 8 * 3! * 6 * 5 = 1440 \tag{2}$$

Similarly we move on to finding the probabilities for the rest:

$$\text{P(6 moves)} = (8*3!*6*5*4) - (6*3!*2*3!) = 5760 - 432 = 5328 \tag{3}$$

$$\text{P(7 moves)} = (8*3*6*3!*5*4*3) - (6*3*6*3!*3!) = 51840 - 3888 = 47952 \tag{4}$$

$$\text{P(8 moves)} = (8*3*6*3!*5*4*3*2) - (6*3*6*3!*2*4!) = 103680 - 31104 = 72576 \tag{5}$$

$$\text{P(9 moves)} = (2*3*8*4!*4!) + (6*3*4*4!*4!) + (22*4!*4!) = 27648 + 41472 + 12672 = 81792 \tag{6}$$

Using the results from **(1) - (6)** we now calculate the probability of the first player always winning:

$$\text{P}(1^{st}\text{ wins}) = \frac{(1440\times 4!) + (47952\times 2!) + (81792\times 0!)}{9!} \approx 0.585$$

The probability of the second player always winning would be:

$$\text{P}(2^{nd}\text{ wins}) = \frac{(5328\times 3!) + (72576\times 1!)}{9!} \approx 0.288$$

In order to determine the probability of a draw we would just subtract the probability of player one always winning and player two always winning from 1:

$$\text{P(draw)} = 1 - 0.585 - 0.288 = 0.127$$

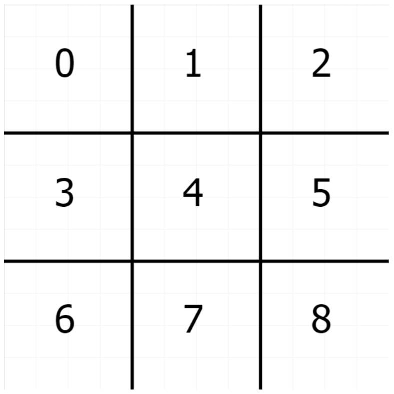
5 Implementation

The game is played as follows, each player marks either a classical or a superposition move. In case of a classical move, a single index is taken as input from the player. In case of a superposition move, two indexes are taken as input from the player. The python code marks an **X** or and **O** in the given indexes. When the board is full, the quantum circuit collapses the qubit and measures the outcomes of the superpositions. If any empty unmarked boxes remain on the board, the game continues, else a winner is decided **(see section 2.1)**.

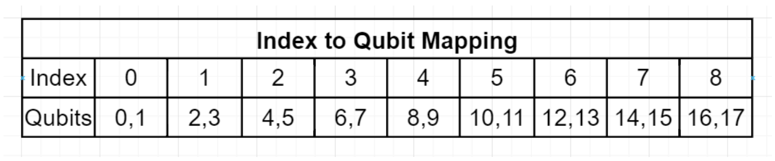
In order to mark a move on the game board, we required a mapping of the symbols to the qubits which will be reflected in the original 18 qubit state (if the move is a classical one), ψ, or in a temporary 18 qubit state (if the move is a superposition one). This mapping is as shown in the table below:

Qubits	Symbol
$\|00\rangle$	Empty
$\|01\rangle$	Unused
$\|11\rangle$	X
$\|10\rangle$	O

Since we have an 18 qubit state for the whole of the board, each of the 9 squares of the board will be represented by 2 qubits that will determine whether it is empty or if it is occupied, then what symbol does it contain. The mapping of each index of the game board with the corresponding qubits is shown in **Figure 5**.



(a) The indexing of the board



Index to Qubit Mapping									
Index	0	1	2	3	4	5	6	7	8
Qubits	0,1	2,3	4,5	6,7	8,9	10,11	12,13	14,15	16,17

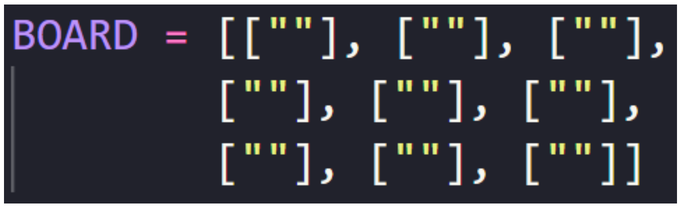
(b) The mapping of indexes to qubits

Figure 5: Indexing and mapping of our Tiq Taq Toe game board

We have implemented our Spooky Tiq Taq Toe in Python using the qiskit [8] library. Our implementation made use of both classical and quantum data structures to store and process information during the game.

The classical data structure is used to store the game state in the form of a python list, that represented the 3×3 board, where each index of the board was represented by a separate sub-list within the main list as shown in **Figure 6**. The quantum representation of the board is a 18 qubit state, ψ. At the beginning of the game, the classical Python list and all sub-lists within it were initialized to be empty, and 18 qubit state ψ was initialised to zero.

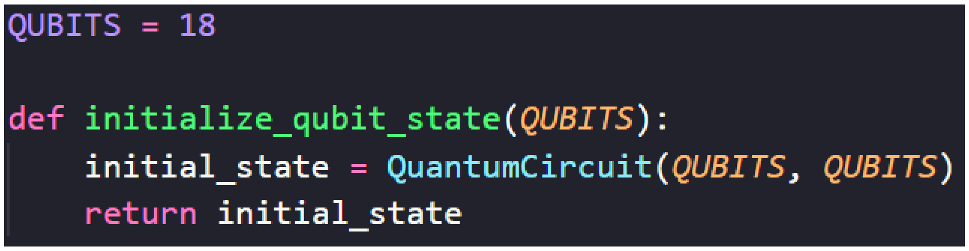
$$\psi = |000000000000000000\rangle \tag{7}$$



```
BOARD = [[""], [""], [""],
         [""], [""], [""],
         [""], [""], [""]]
```

Figure 6: The initialization of the list used to represent the classic tic tac toe game board

The `initialize_qubit_state(QUBITS)` function, in **Figure 7**, takes the number of qubits as its parameter and initializes a qubit state, ψ, as shown above in **(7)**.

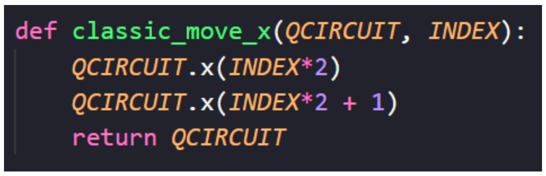


```
QUBITS = 18

def initialize_qubit_state(QUBITS):
    initial_state = QuantumCircuit(QUBITS, QUBITS)
    return initial_state
```

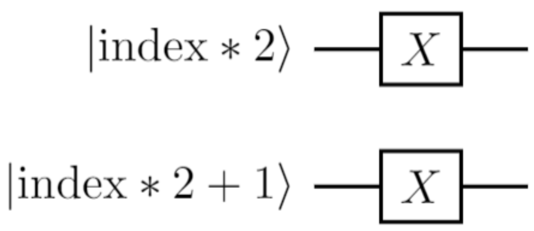
Figure 7: The `initialize_qubit_state` function that initializes a QuantumCircuit with the input number of quantum and classical bits.

Once the mapping and the necessary function to initialize the 18 qubit state had been established, we needed to implement functions that would allow us to alter the state of the game board according to the player's moves. We first began with implementing the functions required to make the classical moves. For marking an **X** we needed to flip two qubits. The first one was at location $2 * index$ and the other one was at $2 * index + 1$, where index was the square number upon which the player had desired to place the symbol. The implementation of this both via code and via a quantum circuit is as shown below in **Figure 8**.



```
def classic_move_x(QCIRCUIT, INDEX):
    QCIRCUIT.x(INDEX*2)
    QCIRCUIT.x(INDEX*2 + 1)
    return QCIRCUIT
```

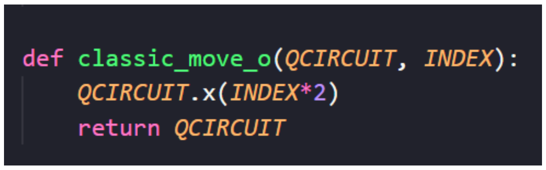
(a) Function to mark a classical **X**



(b) The corresponding circuit to mark a classical **X**

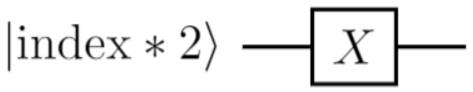
Figure 8: Code and circuit implementations of marking a classical **X** on the game board

For marking an **O** we needed to a single qubit at location $2 * index$, where index was the square number upon which the player had desired to place the symbol. The implementation of this both via code and via a quantum circuit is as shown below in **Figure 9**.



```
def classic_move_o(QCIRCUIT, INDEX):
    QCIRCUIT.x(INDEX*2)
    return QCIRCUIT
```

(a) Function to mark a classical **O**

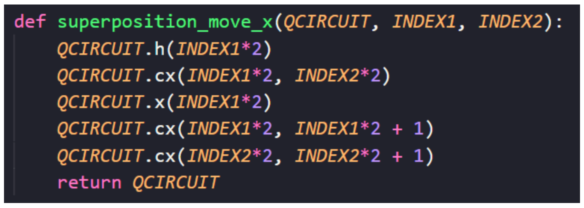


(b) The corresponding circuit to mark a classical **O**

Figure 9: Code and circuit implementations of marking a classical **O** on the game board

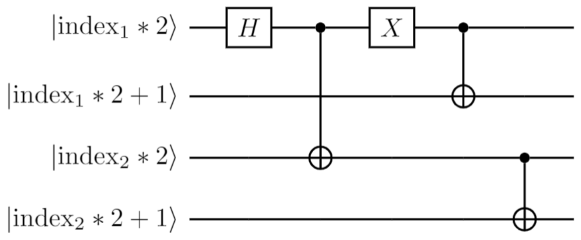
Having figured out the way to implement classical moves, we next moved on to the superposition moves. For these we decided to use an entirely new 18 qubit temporary circuit that would be recording all the superpositions. The need for this temporary circuit was due to the fact that we could not mark the superposition moves on the original 18 qubit state of the board, ψ as when the time came to collapse and measure it, we would loose all the original moves that had been made. Therefore, only the classical moves were being directly reflected on the original 18 qubit state, ψ, while the superposition moves were being implemented on the temporary 18 qubit state.

In order to implement the `superposition_move_x` function we required the temporary 18 qubit state and the two indexes on which to apply the superposition on as parameters. Within the function, three quantum gates - **Hadamard, Controlled-NOT (X)** and **X** - were used to convert the input state into a superposition which was essentially a linear combination of all the different possibilities one could get when the state collapses and measurement is taken. The way the superposition was achieved is shown below in **Figure 10** in terms of both the code and the circuit.



```
def superposition_move_x(QCIRCUIT, INDEX1, INDEX2):
    QCIRCUIT.h(INDEX1*2)
    QCIRCUIT.cx(INDEX1*2, INDEX2*2)
    QCIRCUIT.x(INDEX1*2)
    QCIRCUIT.cx(INDEX1*2, INDEX1*2 + 1)
    QCIRCUIT.cx(INDEX2*2, INDEX2*2 + 1)
    return QCIRCUIT
```

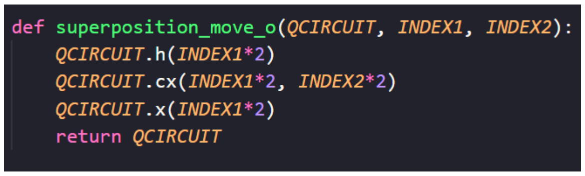
(a) Function to mark a superposition **X**


$|\text{index}_1 * 2\rangle$
$|\text{index}_1 * 2 + 1\rangle$
$|\text{index}_2 * 2\rangle$
$|\text{index}_2 * 2 + 1\rangle$
H
X

(b) The corresponding circuit to mark a superposition **X**

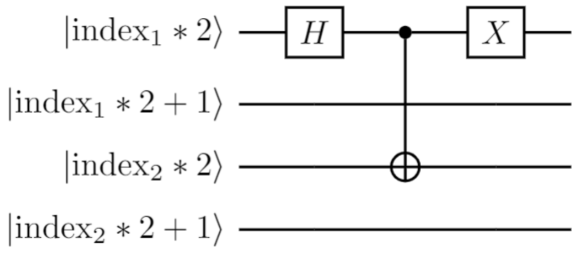
Figure 10: Code and circuit implementations of marking a superposition **X** on the game board

The implementation of the `superposition_move_o` function was similar to that of the `superposition_move_x` function. It also took the temporary 18 qubit state and the two indexes on which to apply the superposition on as parameters. All three previous gates mentioned were also used here to convert the input state into a superposition; a linear combination of all possibilities each of which could be the result when the state is measured after a collapse. The implementation of the function in our code as well as in the form of a circuit is shown below in **Figure 11**.



```
def superposition_move_o(QCIRCUIT, INDEX1, INDEX2):
    QCIRCUIT.h(INDEX1*2)
    QCIRCUIT.cx(INDEX1*2, INDEX2*2)
    QCIRCUIT.x(INDEX1*2)
    return QCIRCUIT
```

(a) Function to mark a superposition **O**

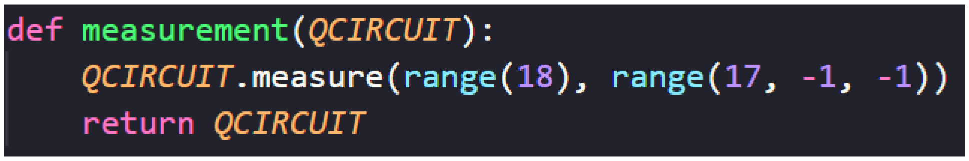

$|\text{index}_1 * 2\rangle$
$|\text{index}_1 * 2 + 1\rangle$
$|\text{index}_2 * 2\rangle$
$|\text{index}_2 * 2 + 1\rangle$
H
X

(b) The corresponding circuit to mark a superposition **O**

Figure 11: Code and circuit implementations of marking a superposition **O** on the game board

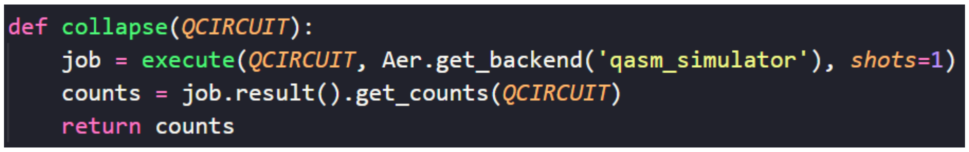
Once the board has been fully occupied, we measure the temporary 18 qubit state that we had created to record all the superposition moves. At the time of collapsing, it provided us with **one** out of all the possible states that the board could have been in. According to the outcome obtained after collapsing, we then alter our original 18 qubit state of the board to reflect the current state of the game board.

The functions to measure the qubit state and to collapse it are shown below in **Figures 12** and **13** respectively.



```python
def measurement(QCIRCUIT):
    QCIRCUIT.measure(range(18), range(17, -1, -1))
    return QCIRCUIT
```

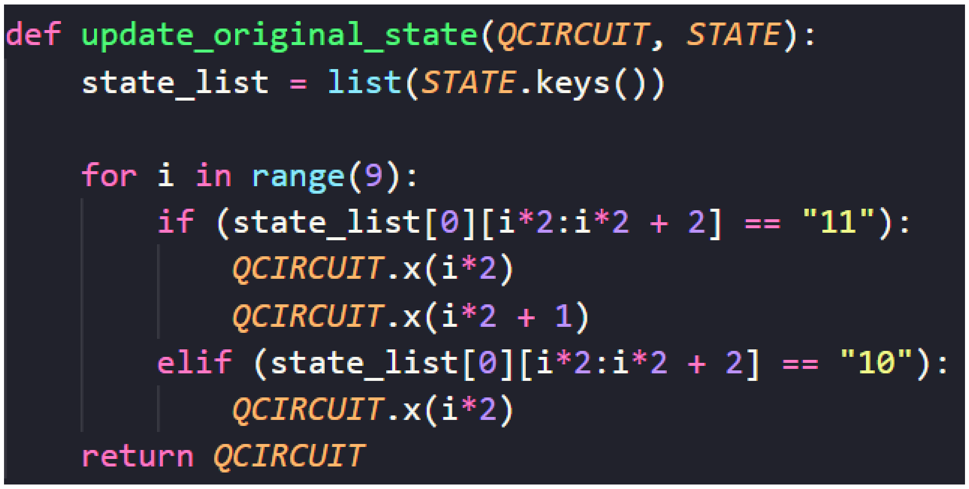
Figure 12: The measurement function that is used to obtain the output from the collapsed qubit state.



```python
def collapse(QCIRCUIT):
    job = execute(QCIRCUIT, Aer.get_backend('qasm_simulator'), shots=1)
    counts = job.result().get_counts(QCIRCUIT)
    return counts
```

Figure 13: The collapse function that is used to reduce the temporary state to a single possibility.

Once the temporary qubit has been collapsed, we alter the original state accordingly with the help of the function shown in **Figure 14**. It taken in as parameters: a quantum circuit and a state list which contains the outcome of the collapse of the 18 qubit temporary state. This measurement outcome is copied onto the original state using a series of **NOT-gates** as shown in the figure.



```python
def update_original_state(QCIRCUIT, STATE):
    state_list = list(STATE.keys())

    for i in range(9):
        if (state_list[0][i*2:i*2 + 2] == "11"):
            QCIRCUIT.x(i*2)
            QCIRCUIT.x(i*2 + 1)
        elif (state_list[0][i*2:i*2 + 2] == "10"):
            QCIRCUIT.x(i*2)
    return QCIRCUIT
```

Figure 14: The function used to update the original state

After having updated the original 18 qubit state of the board, the board will result in having some unoccupied spaces again. If no winning sequence (as shown in **Figure 1, sub-figures (a) - (h)**) is formed, the game continues onward until a winning sequence is made or the game results in a stalemate.

6 Analysis

In order to analyze the **worst-case** complexity of our version of the Quantum Tiq Taq Toe, we first need to determine the complexity of each of its main components, namely:

- Classical move O
- Classical move X
- Superposition move O
- Superposition move X

From **Figure 8(b)** we can see that for the classical move X:

1. Width = 2
2. Depth = 1

From **Figure 9(b)** we can see that for the classical move O:

1. Width = 1
2. Depth = 1

From **Figure 10(b)** we can see that for the superposition move X:

1. Width = 4
2. Depth = 4

From **Figure 11(b)** we can see that for the superposition move O:

1. Width = 3 (**Note: The fourth wire is just shown for completeness and uniformity**)
2. Depth = 3

6.1 Iteration 1

In the first iteration of the game, that is the first time the whole board gets filled, the **maximum** number of superposition moves that each player can make is **2**. Therefore, we will have two **superposition X** moves and two **superposition O** moves at the most in the first iteration. This would fill up 8 out of the 9 possible locations, hence there will always be at least one classical move in the first iteration of the game. This might be either for the symbol **X** or for the symbol **O** depending on which player chooses to make the classical move. Therefore, we have:

$$\text{Moves in iteration 1} = (2 \times \text{Superposition X}) + (2 \times \text{Superposition O}) + (1 \times \text{Classical X or O})$$

The Width Complexity of the first iteration can therefore be calculated as follows (**Note: the cases of the classical moves have been taken into consideration separately and the complexity has been calculated for each of those cases**):

$$\text{Iteration 1 Width Complexity (worst-case if Classical X)} = (2 \times 4) + (2 \times 3) + (2) = 8 + 6 + 2 = 16$$

$$\text{Iteration 1 Width Complexity (worst-case if Classical O)} = (2 \times 4) + (2 \times 3) + (1) = 8 + 6 + 1 = 15$$

The Depth Complexity of the first iteration would just be the maximum number of gates that are applied on any of the wires in the whole circuit. Therefore, it would always be equivalent to the Depth Complexity of the superposition X move as it has the maximum value of the Depth Complexity as compared to all the other components (**Note: here the cases of the classical moves do not impact the calculations and therefore have not been considered separately**):

$$\text{Iteration 1 Depth Complexity (worst-case)} = 4$$

6.2 Iteration 2

Moving onto the second iteration of the game, we will be left with **4** empty locations and hence, in the worst-case scenario each player can choose to make one superposition move each. Therefore, we have:

$$\text{Moves in iteration 2} = (1 \times \text{Superposition X}) + (1 \times \text{Superposition O})$$

The Width Complexity of the whole circuit would remain the same as the moves made in this iteration would always be on the same wires and hence no new wires would need to be added to implement the next set of moves.

$$\text{Iteration 2 Width Complexity (worst-case if Classical X in iteration 1)} = 16$$

$$\text{Iteration 2 Width Complexity (worst-case if Classical O in iteration 1)} = 15$$

However, the Depth Complexity depends upon which superposition move is made where on the board in the second iteration. If the player with the symbol **X** decides to make his/her superposition move on the any of the locations he/she had previously made a superposition move in the first iteration then the Depth Complexity would increase by 4. Otherwise it would increase by 3 if he/she decides to make a superposition move on the indexes that got free after the superposition move of symbol **O** collapsed.

$$\text{Iteration 2 Depth Complexity (same locations as of Superposition move in iteration 1)} = 4+4 = 8$$

$$\text{Iteration 2 Depth Complexity (}\textbf{not}\text{ at same locations of Superposition move in iteration 1)} = 4+3 = 7$$

6.3 Iteration 3

In iteration 3, in the worst case, a superposition **O** or a superposition **X** move can be made. Therefore:

$$\text{Moves in iteration 3} = (1 \times \text{Superposition O}) \mid (1 \times \text{Superposition X})$$

The Width Complexity of the whole circuit would remain the same as the moves made in this iteration would always be on the same wires and hence no new wires would need to be added to implement the next set of moves.

$$\text{Iteration 3 Width Complexity (worst-case if Classical X in iteration 1)} = 16$$

$$\text{Iteration 3 Width Complexity (worst-case if Classical O in iteration 1)} = 15$$

The Depth Complexity now depends upon whether a superposition **X** move is made or a superposition **O** move is made.

$$\text{Iteration 3 Depth Complexity (Superposition X } \textbf{and} \text{ iteration 2 case 1)} = 4+4+4 = 12$$

$$\text{Iteration 3 Depth Complexity (Superposition X } \textbf{and} \text{ iteration 2 case 2)} = 4+3+4 = 11$$

$$\text{Iteration 3 Depth Complexity (Superposition O } \textbf{and} \text{ iteration 2 case 1)} = 4+4+3 = 11$$

$$\text{Iteration 3 Depth Complexity (Superposition O } \textbf{and} \text{ and iteration 2 case 2)} = 4+3+3 = 10$$

6.4 Iteration 4

Lastly, we would be left with just one empty square on the game board and it will be the turn of **Player 1**. Hence:

$$\text{Moves in iteration 4} = (1 \times \text{Classical X})$$

The Width Complexity remains the same:

$$\text{Iteration 3 Width Complexity (worst-case if Classical X in iteration 1)} = 16$$

$$\text{Iteration 3 Width Complexity (worst-case if Classical O in iteration 1)} = 15$$

The Depth Complexity increases by 4 due to a classical **X** move:

$$\text{Iteration 4 Depth Complexity (Classical X } \textbf{and} \text{ iteration 3 case 1)} = 4+4+4+4 = 16$$

Iteration 4 Depth Complexity (Classical X **and** iteration 3 case 2) $= 4+3+4+4 = 15$

Iteration 4 Depth Complexity (Classical X **and** iteration 3 case 3) $= 4+4+3+4 = 15$

Iteration 4 Depth Complexity (Classical X **and** and iteration 3 case 4) $= 4+3+3+4 = 14$

6.5 Overall

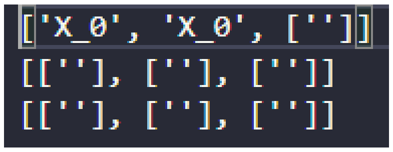
As a whole the **worst-case** complexities of the Quantum Tiq Taq Toe would be as follows:

- Width Complexity = 16 (if classical X) or 15 (if classical O)
- Depth Complexity = 16 or 15 or 14 (depending on moves as analysed above)

7 Example Game

An example Quantum Tiq Taq Toe game will be shown move by move in order to provide a visual understanding of how it works.

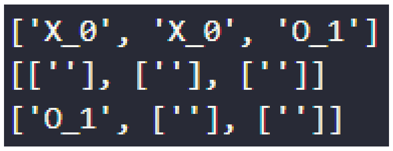
1. Move 1: Superposition X at index 0 and 1



```
['X_0', 'X_0', ['']]
[[''], [''], ['']]
[[''], [''], ['']]
```

Figure 15: The first move

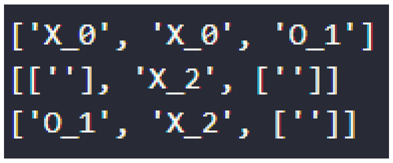
2. Move 2: Superposition O at index 2 and 6



```
['X_0', 'X_0', 'O_1']
[[''], [''], ['']]
['O_1', [''], ['']]
```

Figure 16: The second move

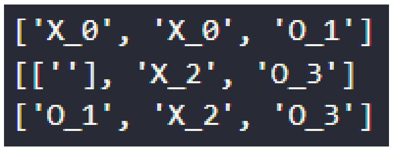
3. Move 3: Superposition X at index 4 and 7



```
['X_0', 'X_0', 'O_1']
[[''], 'X_2', ['']]
['O_1', 'X_2', ['']]
```

Figure 17: The third move

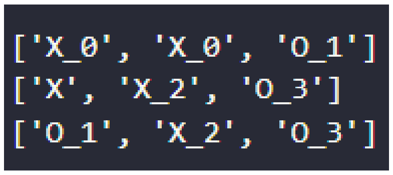
4. Move 4: Superposition O at index 5 and 8



```
['X_0', 'X_0', 'O_1']
[[''], 'X_2', 'O_3']
['O_1', 'X_2', 'O_3']
```

Figure 18: The fourth move

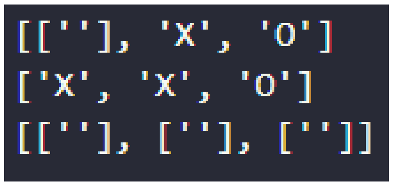
5. Move 5: Classical X at index 3



```
['X_0', 'X_0', 'O_1']
['X', 'X_2', 'O_3']
['O_1', 'X_2', 'O_3']
```

Figure 19: The fifth move

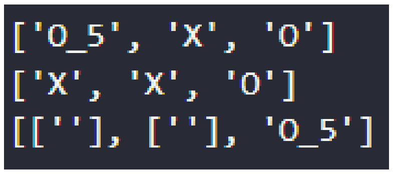
6. The first collapse



```
[[''], 'X', 'O']
['X', 'X', 'O']
[[''], [''], ['']]
```

Figure 20: The first collapse

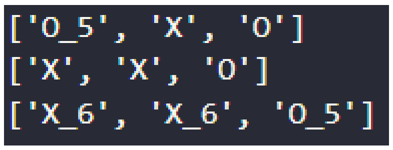
7. Move 6: Superposition O at index 0 and 8



```
['O_5', 'X', 'O']
['X', 'X', 'O']
[[''], [''], 'O_5']
```

Figure 21: The sixth move

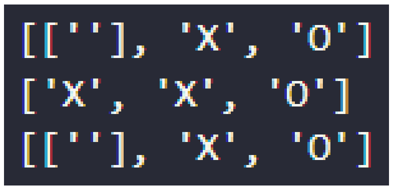
8. Move 7: Superposition X at index 6 and 7



```
['O_5', 'X', 'O']
['X', 'X', 'O']
['X_6', 'X_6', 'O_5']
```

Figure 22: The seventh move

9. The second collapse: **Both** players win!



```
[[''], 'X', 'O']
['X', 'X', 'O']
[[''], 'X', 'O']
```

Figure 23: The second collapse

As can be seen from **Figures 15 - 23** an interesting situation can occur wherein both players can result in creating a sequence at the same instance thereby resulting in a double win situation. This was previously impossible with the classical version of this game.

8 Conclusion

Quantum Tiq Taq Toe serves as a good introduction to the concepts of superposition and quantum entanglement. It aims to develop and understanding of these concepts in a fun and engaging way.

The game can be made more interesting by the inclusion of non-uniform probabilities of measurements when collapsing a state thereby making the game more challenging and unpredictable. Moreover, a interactive graphical user interface (GUI) can also be implemented to make the game more aesthetic and easy to understand as to how to play. This may be down using either Flask [9] or tkinter [10] Python library.

Furthermore, in terms of the analysis, future work can include the mathematical calculations required to determine the probability of the first and second player always winning in Quantum Tiq Taq Toe. This is something we were unfortunately unable to cover due to time constraints. Moreover, the ability to entangle more than one symbol on the same index is also an element to the game that can be explored in the future.

References

[1] A. Goff, "Quantum tic-tac-toe: A teaching metaphor for superposition in quantum mechanics," 2006, doi: 10.1119/1.2213635.

[2] A. Kumar, A. Sheng, and C. M. Huang, "Generalized Quantum Tic-Tac-Toe," 2016, Accessed: Nov. 04, 2021. [Online]. Available: https://www.semanticscholar.org/paper/Generalized-Quantum-Tic-Tac-Toe-Kumar-Sheng/c22321f3b1fd8f7481b571ab7343c8b3ace41c15

[3] E. van Nieuwenburg, "Quantum TiqTaqToe – TicTacToe with Quantum Physics," 2019. https://quantumtictactoe.com/ (accessed Nov. 04, 2021).

[4] M. Nagy and N. Nagy, "Quantum Tic-Tac-Toe: A Genuine Probabilistic Approach," vol. 2012, no. 11, pp. 1779–1786, Nov. 2012, doi: 10.4236/AM.2012.331243.

[5] Vos, J. All about Hadamard Gates - Manning. https://freecontent.manning.com/all-about-hadamard-gates/ (2019).

[6] Bottomley, H. How many Tic-Tac-Toe (noughts and crosses) games are possible? http://www.se16.info/hgb/tictactoe.htm (2001).

[7] Solving Tic Tac Toe, Making it Interesting? – Extra Polynymous. https://estebanhufstedler.com/2020/03/26/solving-tic-tac-toe-making-it-interesting/ (2020)

[8] Qiskit. https://qiskit.org/.

[9] Welcome to Flask — Flask Documentation (2.0.x). https://flask.palletsprojects.com/en/2.0.x/.

[10] tkinter — Python interface to Tcl/Tk — Python 3.10.0 documentation. https://docs.python.org/3/library/tkinter.html.