Plagiarism Declaration

"I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write."

Team : Prapti Setty

Patrick Jennings

Isaac Walker

David Phillips

JC Zhang

Daniel Cosgrove

Eoin Dowling

Signed and submitted by : _____

(on behalf of team 37)

Second Year - Development Report

1. Introduction

Background and Problem Statement

The project that group 37 is doing is the Microsoft Quantum Computing Project. Microsoft set us the challenge of learning about Quantum Computing, and testing their recently released Quantum Computing Development Kit. This development kit includes what a developer would need to get started in the field: a Q# language and compiler along with a Q# library, a local quantum computing simulator, as a Visual Studio extension.

The main objective is to decide whether the tool released adequately simulates how a quantum computer would work (as true quantum computers are still in their infancy).

After brainstorming for the first week, we had to choose one idea to settle on and pursue. The team chose to pursue coding an algorithm in Q#. The chosen algorithm is the triangle finding problem. Since we had previously studied the graph theory briefly, we felt we could take on the task of solving this algorithm while simultaneously learning new concepts of not only quantum, but also Q#.

Technical Approach (how the work was done)

The team consists of 7 people, 4 second years and 3 third years. The third years acted as the managers and oversaw all the development. The second years, with assistance from the third years developed the solution. The developing team focused on two parts, the quantum implementation and the classical implementation. The classical implementations were completed much faster than the quantum implementation due to an abundance of resources being available for the classical algorithms. Once the classical implementations were completed, the team began to also discuss a possible method to show 'Classic vs Quantum' ad eventually settled on a graph.

2. Requirements

Functional and non-functional requirements for the Project

The main functional requirement set to us by Microsoft is to test the software and the best method for this is to learn Q#, research an algorithm and then implement it. The chosen algorithm is the Quantum algorithm for finding triangles in a sparse graph.

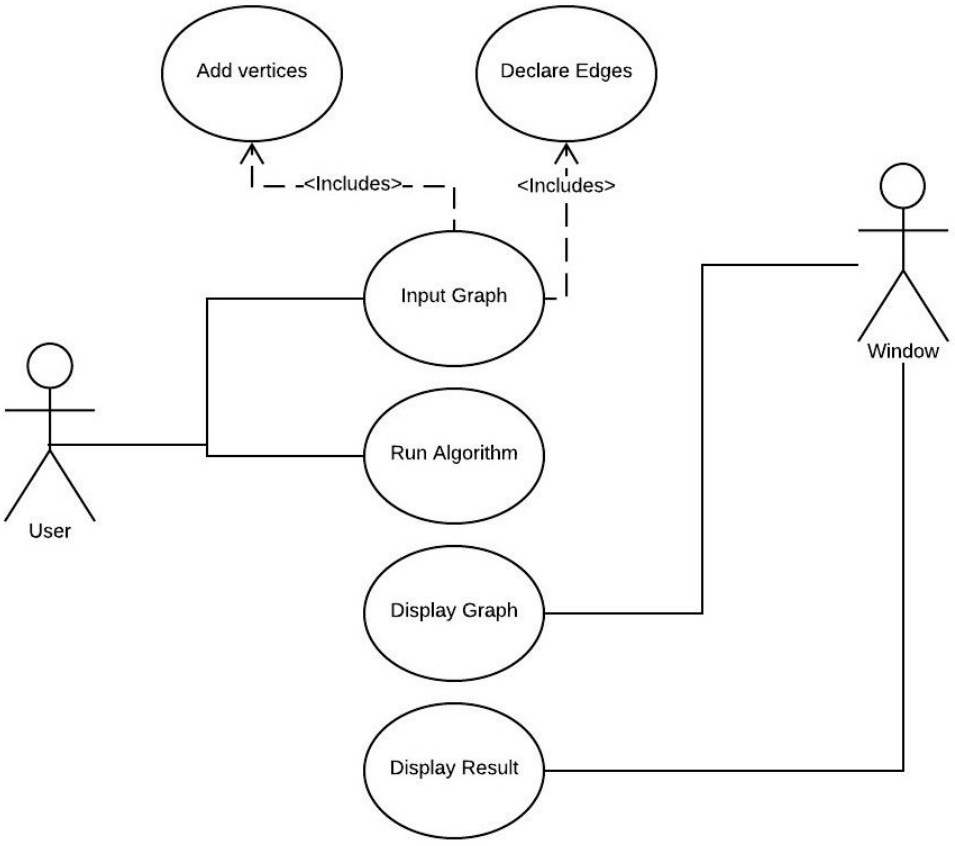
The input for this algorithm is the adjacency matrix for the graph, the calculations required will be calculated as the algorithm is implemented. The output will be the three vertices of the triangle found. Our project will allow for the user to enter an undirected unweighted graph and assign edges between vertices. An adjacency matrix of the graph will be generated and inputted into Q#. Q# will then find a

triangle if present and return it's three vertices. The result of the call to Q# will then be displayed visually in C#.

The non-functional requirements of this project are sparse. The method used to judge the operation of the system will be whether we are able to implement the algorithm or not. The team here in Trinity will provide a short simple report of their explorations of the algorithm and outlining whether the quantum algorithm implementation is better than the traditional implementation.

User Interaction Scenarios including Use Case diagrams

The Use Case shows how actors behave in the system. The two actors in our system are the user; who specifies the graph that the algorithm will be called upon, and the Window; who displays the graph and editing tools visually, and outputs the result of the algorithm to the user.

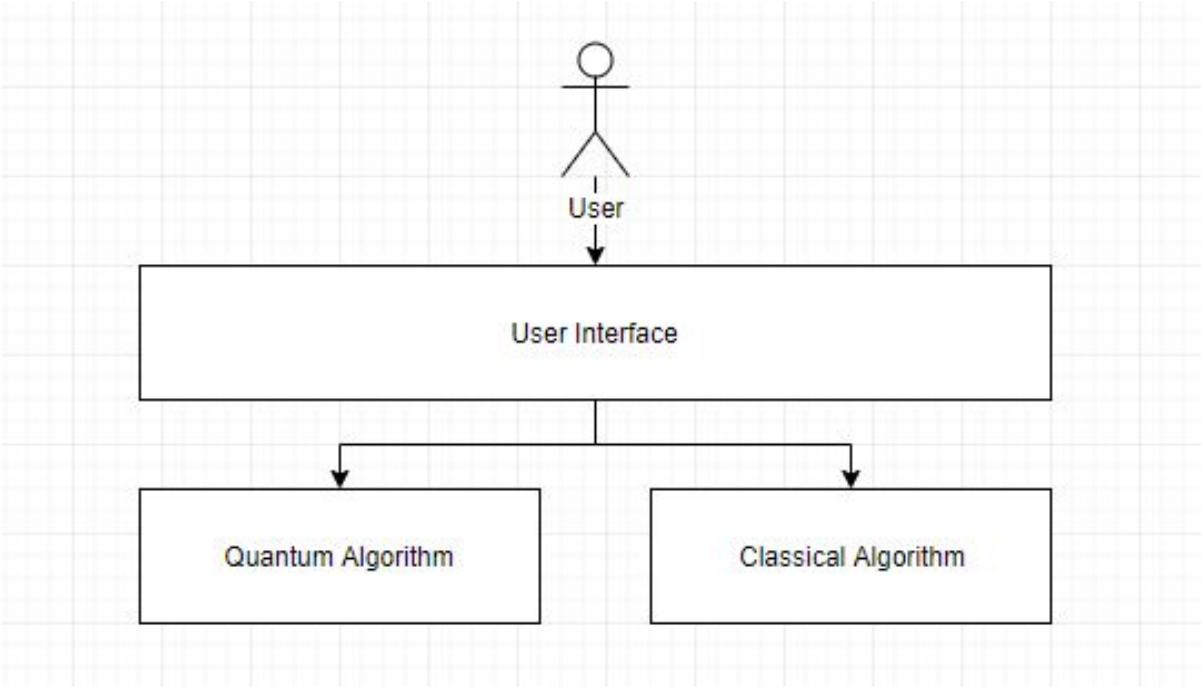


3. Design

Design used including Architecture diagram

"A **system architecture** or **systems architecture** is the conceptual model that defines the structure, behavior, and more views of a **system**. An **architecture** description is a formal description and representation of a **system**, organized in a way that supports reasoning about the structures and behaviors of the **system**." - https://en.wikipedia.org/wiki/Systems_architecture

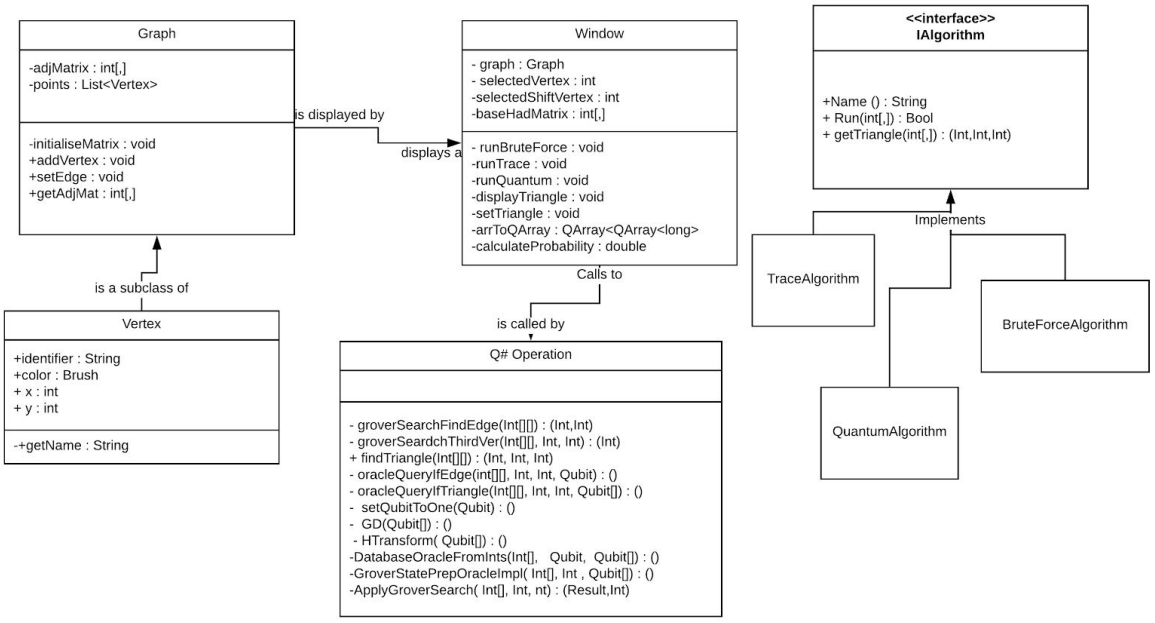
The main language used was Q# as the spec of the project was to test the provided tool and our method of evaluation is to implement an algorithm in the quantum language (Q#). For the interface and the classical algorithm, as mentioned previously, C# will be used.



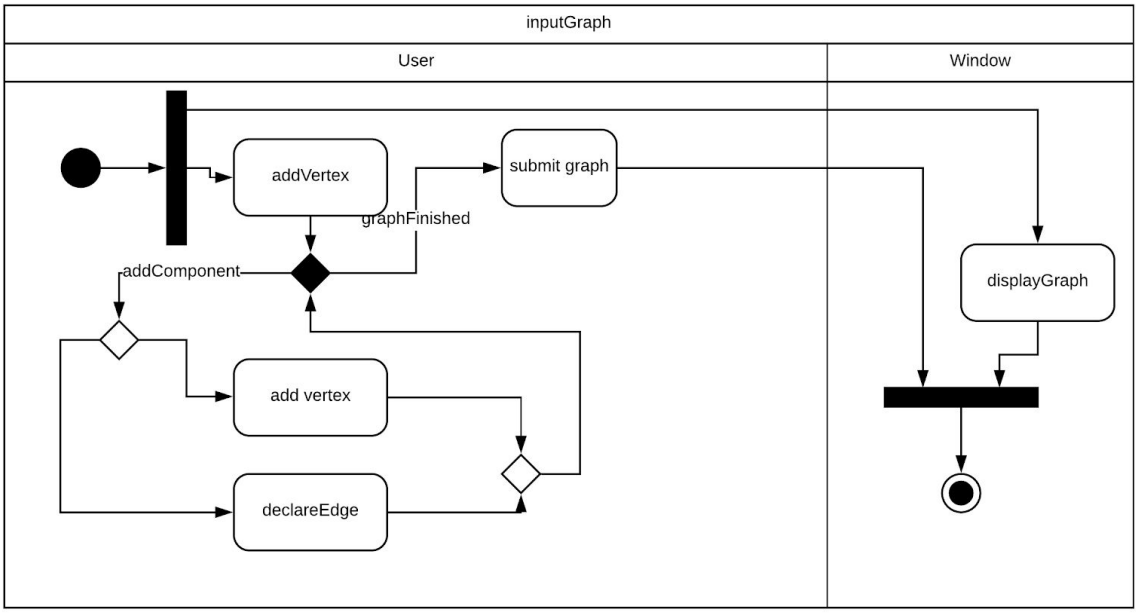
-

UML Class and Sequence diagrams

Our base class for the project was the Window class; building our design around the user-interface. This class also holds an instance of the graph class, as well as instances of each of three algorithms. The three algorithms; Quantum, Trace and BruteForce, implement from an algorithm interface allowing their performance to be easily measured and for more structured code.



The code for the quantum algorithm is contained in the Operation file. The Q# language does not allow for global variables, only permitting “operations” and “functions”. We have broken down Grover's algorithm into operations based on the quantum circuit, and the classical and trivial steps are implemented using functions.



4. Implementation

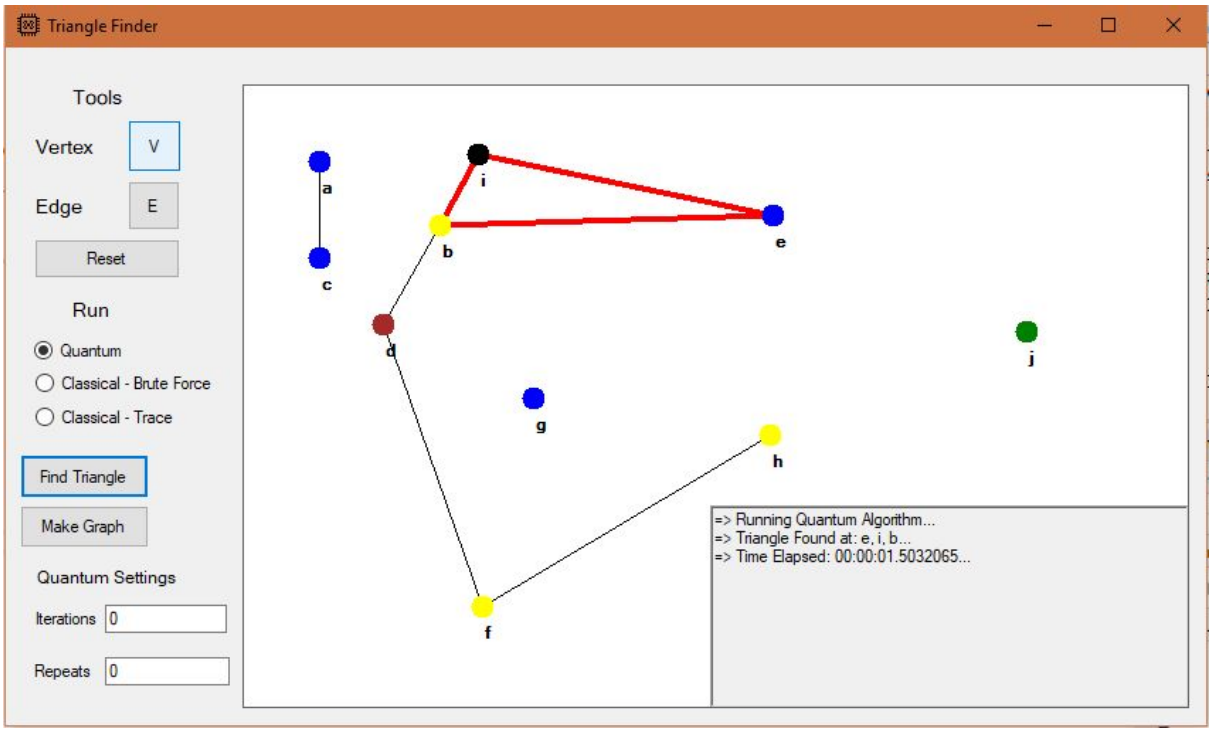
Tools/Libraries/Platforms used

As the main aim was to test whether the simulation tool is an accurate representation of how a quantum computer would work, the main language used was Q# and the quantum development kit was the main tool used. The development kit comes with a quantum simulator on which we ran our algorithm to test it.

There were a number of algorithms previously implemented that could be used as reference and were available on the github account provided as reference in the instruction set.

User interface

The user would input the graph on the user interface as shown below. To place a point, the user must click the Vertex option and then place it on the graph at the desired location. The connect 2 vertices to make an edge, the user must first select the two vertices by clicking on them while holding down the Shift button and then click on Edge or click on the E key on the Keyboard.



Implementation of algorithms described

Classical : The Classical algorithms have 2 approaches, The brute force approach and the trace matrix approach. These approaches were researched and implemented by the following steps:

Brute force:

1. Check each group of 3 vertices
2. By using 3 nested loops
3. If there is an edge between all 3 vertices
4. There is a triangle

Trace Matrix:

1. Use matrix multiplication
2. To get cube of adjacency matrix
3. Take trace of this matrix
4. If it is non-zero
5. It contains a triangle

Quantum: The approach for the quantum algorithm was significantly more complex than the classical ones and the method to implement them was:

1. Create array of all possible edges from adjacency matrix
2. Create an array of qubits, being log to the base 2 of the size of the array of edges, ceilinged, and with one extra which is the marked qubit that will contain the result
3. Create an oracle from the marked elements of the array. The oracle is a so called 'black box' which is essentially an unknown boolean function which returns one or zero

4. Run the oracle, followed by amplitude amplification for a predetermined number of steps
5. Check the marked qubit for a success or failure
6. Find the edge that the search found
7. If the marked qubit shows success, this is an edge. If successful, move to step 8, otherwise, repeat from step 2, or exit with failure if the max number of repeats has been reached
8. Similarly, find the array within the adjacency matrix that corresponds to one of the vertices, and run steps 2 to 6
9. If a success is found, this means that the vertex found makes an edge with the vertex we are checking
10. Check the second vertex of the found edge to check if an edge exists with the third vertex
11. If one does, a triangle is present, otherwise return to do steps 2 to 6 to find another vertex, or go back and find another edge if the max number of repeats has been reached

Due to the nature of quantum physics, the search algorithm implemented, known as Grover's search algorithm, is probabilistic. This means that increasing the number of iterations of the oracle and of amplitude amplification does not result in a more accurate result. We found that the square root of the number of qubits, or 1, whichever is larger, tends to give the best result. We were able to work this out from the formula:

$$(sin(2 * nIterations + 1) * arcsin(\sqrt{nMarkedElements} \div nElements))^2$$

Where: nIterations is the number of iterations of the oracle and amplitude amplification
nMarkedElememts is the number of marked elements we are looking for
nElements is the total number of elements

As can be seen, the probability can be described by a wave as nIterations increases. For instance, with a graph (V,E), where V is the vertices and E is the edges, and V = {a, b, c, d}, E = {ab, ac, bc}, and there exists a triangle abc, if we were to use 1 iteration of the oracle and amplitude amplification, and 10 repeats of the search, we will find the triangle almost 100% of the time. Whereas if we were to do 2 iterations, we would only find the triangle around 20% of the time.
However, repeating the full Grover's search does result in a more accurate result, at the cost of time.

5. Conclusions

Problems encountered during design/implementation

Although the entire team was very keen to do the project, and volunteered to do it, we had a few challenges along the way. The first problem that we encountered was our abundance of ideas. Initially, the team considered exploring cryptocurrency in quantum, but we quickly realised that we did not have adequate information about the workings of cryptocurrency, and had to scrap the idea. The next idea involved a quantum circuit simulator that would visualise the circuit and output the results. The client rejected this idea, as we would not be focusing on Q#. We would be looking into building a UI that would

take the inputs more. Finally, we found a solution that suited both our client and us. We decided to implement an algorithm.

The algorithm implementation came with its own challenges. Q# is a new tool, only released in November 2017, so the resources out there were limited. There are sample solutions for a few other algorithms that we referenced. The team had to dedicate the first few weeks to learning to use the development tool and get used to coding in a completely new language.

There were times when the maths was a little more complicated than we expected and we hit a few roadblocks along the way due to this. The team would have to stop development at these stages and begin to research again. Eventually, we were able to overcome these hurdles and implement a quantum solution to the triangle finding problems.

Self Evaluation of how well the project objectives were reached

The main objective was to test whether the quantum development kit accurately represents a quantum computer. To do this, we had to grasp the concepts of quantum better, learn to use Q# and then implement an algorithm. By week 12, the team had implemented the algorithm, so the objectives were reached. The team proposed a one page document on the findings over the course of using the development kit, and whether the quantum algorithm is better than traditional algorithms. This has also been completed and has been presented to the client as part of the handover document.

Self evaluation of working as a group

The group worked well together from the get-go. Everyone was willing to help each other with issues they encountered. The third years and the mentors guided the second years along and helped them with anything that stumped the second years. During the meeting, we managed to stick to the agenda, and assign tasks adequately. We were also able to find time outside of college to meet up and continue working on the project. Overall, the group worked together pretty well.

Description of algorithms and suggestions for enhancements

The triangle-finding problem is the problem of determining whether a given graph contains a triangle (a clique of size 3). Classical algorithms depend on the speed of matrix multiplication which has a time complexity of O(v^log2(3)) or O(v^3). There exist theoretical matrix multiplication algorithms such as Strassen's Algorithm with has one fewer assignment in it's recursive calls bringin the time complexity down to O(v^log2(7)) or ~O(v^2.81..). The best known matrix multiplication algorithms such as the Coppersmith-Winograd algorithm which as an approximate time complexity of O(v^2.4), where v is the number of vertices, however such algorithms can not be implemented practically. There exists a quantum algorithm theorised to run with Õ(v^1.25).