Summer 8-5-2020

QWASI: The Quantum Walk Simulator

Warren V. Wilson
CUNY Hunter College

QWASI: The Quantum Walk Simulator

by

Warren Virgil Wilson

Submitted in partial fulfillment
of the requirements for the degree of
Master of Arts Physics, Hunter College
The City University of New York

2020

08/04/2020	Professor Mark Hillery
Date	Thesis Sponsor
08/04/2020	Professor János Bergou
Date	Second Reader

Table of Contents

Abstract

As quantum computing continues to evolve, the ability to design and analyze novel quantum algorithms becomes a necessary focus for research. In many instances, the virtues of quantum algorithms only become evident when compared to their classical counterparts, so a study of the former often begins with a consideration of the latter. This is very much the case with quantum walk algorithms, as the success of random walks and their many, varied applications have inspired much interest in quantum correlates. Unfortunately, finding purely algebraic solutions for quantum walks is an elusive endeavor. At best, and when solvable, they require simple graphical structure, exploitable symmetries, and case by case analyses. In more complex instances, as with most arbitrary graphs, the pursuit of a general solution is untenable. Enter Qwasi (**Q**uantum **Wa**lk **Si**mulator), a software tool that allows for the composition of arbitrary graphs by a user, includes a mathematical backend that can construct the Hilbert Space from these graphs, and simulates the evolution of the quantum walk as a set of numerical data. This simulation can be observed in real time, compiled into graphical plots, or exported as a CSV file. At present, the software focuses upon discrete-time scattering quantum walks, but expansion to other graph-based algorithms – anything that operates on a collection of vertices and connected edges – would not be a demanding task.

Quantum Computation

The field of quantum computation has enjoyed a surge of attention over recent decades, having grown from an intellectual curiosity into a burgeoning field of cutting-edge research. This proliferation has followed from the discovery of some groundbreaking quantum mechanical algorithms, the importance of which derive from powerful speedups over their classical counterparts. In the most extreme cases, such algorithms boast deterministic solutions to classically nondeterministic problems, which is an exponential improvement in runtime. This means that some problems which are intractable using classical computers are solvable using quantum ones. Because this inherent superiority remains theoretical, the goal of demonstrating it has been termed *quantum supremacy*. The goal of demonstrating less dramatic speedups for classically deterministic algorithms is called *quantum advantage*, and still offers considerable gain.

The specific purpose of this paper is to introduce a software tool which I have developed and coded meant to aid in the research of quantum walks. However, deliberation on its usefulness and abilities is impossible without proper context, so a suitable background in a few different areas will first be established. We will start by discussing algorithmic runtime behavior (which is essentially how long an algorithm takes to complete) and then move on to the topic of *determinism* and its meaning in the field of *computational complexity*. This will equip us with the necessary foundation to discuss quantum algorithms, and in particular, how they are able supersede classical ones. A mathematical formalism of quantum walks will follow which will then lead us into our discussion of the software. But first, I will briefly introduce it here. Named Qwasi (**Q**uantum **Wa**lk **Si**mulator), it can simulate scattering quantum walks in the space of arbitrary graphs, quantifying the evolution of a system that is (in general) unobtainable by algebraic

means. The raw output is a set of numerical data that represents the state of the walk after each step, which consists of the basis coefficients within a Hilbert space generated from a user constructed graph. Furthermore, Qwasi carries with it powerful analytics so that this data can be viewed more intuitively, skirting the need to thumb through pages of numbers to construct any meaningful behavior.

Computational Complexity Theory

To understand the importance of investigating new quantum algorithms, including quantum walks, a basic understanding of both classical and quantum computation is necessary. To start off, let us explore some of the basics regarding the field of *computational complexity*. The efficiency of any algorithm, classical or otherwise, is a quantitative measure of the resources required to run it. Usually, this refers to *time complexity*, aka the number of iterative cycles needed to complete a task, and is given as a function on the numerical size of the input dataset. When there is ambiguity, worst-case runtime is usually the value of interest since it represents an upper bound, but sometimes (and in cases when they are not the same) average runtime is prioritized. Syntactically, computational complexity (also called asymptotic complexity) is given using "Big O" notation – where it is borrowed from its uses in other mathematics – and is of the form

$$\mathcal{O}\big(f(N)\big). \tag{1}$$

Here, f represents the time requirement of an algorithm as a function over the parameter N – the integer size of the input data – for the case of large N. A simple example of this, which is linear in N, is a classical search for a data element in an unsorted list. Because the list is unsorted, the only means by which one can search through it exhaustively (on classical hardware) is to check each element in it. This leads to a worst-case scenario of N processing cycles, so the asymptotic complexity of such a search is $\mathcal{O}(N)$.

Nearly all algorithms used in computation today fall into the classification of *polynomial time* algorithms, including the linear search example above. As the name suggests, polynomial time algorithms are those whose time complexity can be expressed as a finite polynomial on the size of the data given it. As always when dealing with computational complexity, attention is paid to systems as their size becomes very large, collapsing the polynomial nature of the runtime to the behavior of the leading term. Coefficients of these terms are also ignored, since they present as merely a difference in the execution time of a single computational cycle, which will in general vary from machine to machine. To illustrate this, consider again the linear search example above. Let's say for the sake of argument that we choose the time complexity of the algorithm to be based on its average case performance. Since statistically we find the element we're looking for after checking only half of the list, it would be reasonable to assume that the time complexity of the algorithm would be $\mathcal{O}(N/2)$. However, the time it takes one machine to search through $N/2$ items is the same as it takes another with twice the speed to search through N. Since processor speed is an arbitrary factor in this analysis, we drop all coefficients.

As another, slightly more complex example, consider a process that first sorts a list of names, then enumerates them again to print them out on a screen. Let us also suppose that the sorting algorithm used is selection sort, which for each element in the list, takes a separate pass over the remaining unsorted ones. If $f(N)$ represents the number of computational cycles needed for a list of size N, then

$$f(N) = [N + (N-1) + (N-2) + \cdots + 2 + 1] + N \tag{2}$$

or

$$f(N) = \left[\frac{N^2 + N}{2}\right] + N \tag{3}$$

$$= \frac{1}{2}N^2 + \frac{3}{2}N. \tag{4}$$

Therefore, the asymptotic complexity of $f(N)$ is

$$\mathcal{O}(N^2). \tag{5}$$

Though our analysis of computation has thus far been classical, it has been necessary background for two reasons: first, time complexity in the quantum case is measured in the same way, and second, the virtues of quantum computation can really only be observed through the lens of comparison.

As a last point on the topic of computational complexity, and one which is essential to understanding quantum supremacy, there are two distinct classes of algorithms within which the vast majority lie: *polynomial time* algorithms, and *nondeterministic polynomial time* algorithms (or P and NP for short). While we have already discussed polynomial time algorithms, nondeterministic polynomial time algorithms are a bigger beast. Formally, the set of NP problems consists of all problems for which no known polynomial time algorithm can solve, but that should a potential solution to the problem be given (an oracular provision, let's say), its *validity* can be checked in polynomial time. The use of nondeterministic in this context reflects the exponential growth of the algorithm's runtime over the size of its input. For if an algorithm is labeled NP, then by definition, there is no known polynomial time solution. But NP problems can be easily checked, so any algorithm that tries to solve such problems must do so heuristically, generating *guesses* at a solution and then checking it for validity. To illustrate this, consider of a person trying to log into a locked computer without having any clue as to the password, but knowing that it is a four-digit number. The only course of action is to start with 0000 and iterate all possible

combinations, which is 10^4, or ten thousand. Now if we add just *one* more digit, we've increased all possible combinations by a multiple of ten, so to try them all now would require 10^5 or one hundred thousand tries. The exponential nature of this type of brute force method should now seem clear, and though algorithms that solve NP problems usually have some optimizations, they are still at their core some version of a guess and check. Anyway, it suffices to know that P problems are called *easy* or solvable, and NP problems are called *hard*.

A famous NP problem, and one at the heart of quantum supremacy, is integer factorization. For any given integer, determining its prime factors is an algorithmically hard task, and the best techniques we have run in exponential time with respect to the cube root of the number of binary digits. But any solution to the problem can be easily checked, for if a list of prime factors is proposed, one only need multiply them together to verify if their product recovers the original integer. Aside from the mathematician's enduring fascination with prime numbers, the algorithmic difficulty of integer factorization is the backbone of internet encryption, so an algorithm that can factor them in polynomial time would have enormous and far reaching consequences.

But, while factorizing integers using classical algorithms eludes us (for all we know it's impossible), this is not the case in the quantum world. Peter Shor devised a *quantum* algorithm[1] that can factor integers with asymptotic complexity of $\mathcal{O}(N^3)$, a polynomial time cubic over the number of integer digits N. Quantum algorithms that can turn classically hard problems into solvable ones are what *define* the term "quantum supremacy" and have galvanized major interest toward the creation of hardware upon which they are able to run. It is worth mentioning however that this divide along computational efficiency which separates quantum algorithms from classical ones is not predicated on any proof or core principal, but

rather on the state of mathematics today. For example, an unanticipated breakthrough in prime number theory could shift the landscape entirely and generate a polynomial time factorizing algorithm that runs on classical hardware. On the other hand, there are likely quantum algorithms still yet to be discovered that will trivialize our most intractable problems today.

Quantum Walks

As with digital hardware, quantum computers operate on a network of circuitry comprised of logic gates where information in the form of quantum bits, or *qubits*, can enter and be processed. Though the inheritance of the term "bit" here correctly indicates a binary set of states, the qubit is not strictly one or the other in the manner of a classical bit and can instead occupy a superposition of the two. This grants the qubit an informational capacity that is far greater than its classical counterpart since the full range of freedom allowed by its state space is the set of all surface points on the Bloch sphere. While of course the coefficients of a qubit's basis states are not directly accessible to measurement, they are essential to the unique functioning of quantum logic gates. Therefore, the preservation of coherence in the state of the qubit is essential until measurement is desired, which is an enduring technological challenge in the field.

So far, our discussion of quantum computation has been rather general, but the groundwork has been sufficiently laid to move on to the focus of this paper: quantum walks. Quantum walks are essentially quantum mechanical adaptations of random walks, so a condensed review of the latter would be a natural starting point for our discourse. A random walk is a process that tracks the stochastic evolution of an object from within a given space through the successive iteration of discrete steps, each taken at random and with equal probability from all local adjacencies. For continuity later on with the quantum case, we

will say that each step in the walk is the result of applying a *step operator* to the current state of the system, which in the classical case is just the location of the object within the chosen space. The space of the walk can be quite general, so long as it constitutes a discrete set of locations that are each arbitrarily adjacent to a subset of the others. A classic example is a one-dimensional line of integers, with every number admitting occupancy and having two adjacent neighbors (the integers directly above and below it). A more general space for walks, classical and quantum, is a *graph* (Figure 1) – a structure composed of a set of N vertices and a set of M edges, and where the edges connect the vertices together. Edges are uniquely identified by the pair of vertices they connect, and each vertex can have anywhere from 0 to $N-1$ edges. In a random walk, an object positioned at any one of the N vertices transitions to one of its connected neighbors and does so with equal probability. Depending on the application, graphs can take on wildly different configurations with varying degrees of complexity.

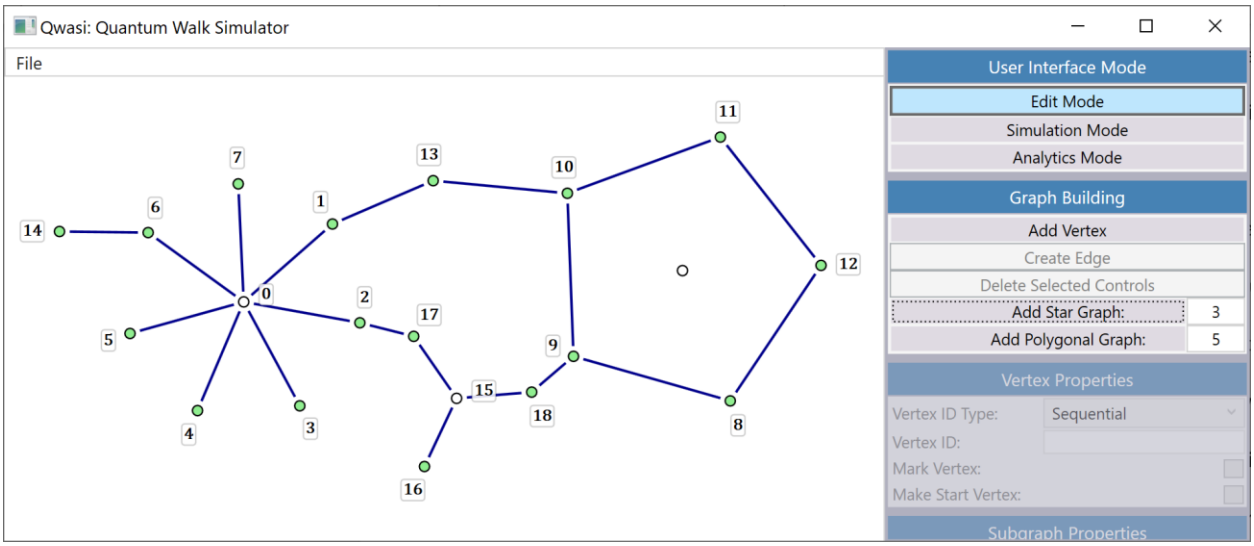


Figure 1: *An example of a graph, built using Qwasi.*

A quantum walk can be adapted from a classical random walk in a few ways, of which there are continuous-time and discrete-time subtypes.[2] [3] [4] In this writing, we will focus on a class of discrete-time quantum walks called scattering quantum walks.[5] Unlike the classical case, these do not admit stochastic

behavior from the random transitions between states, but instead from the probabilistic nature of quantum measurement itself as determined by the Hilbert space coefficients. Another chief difference is that the occupiable states of the scattering quantum walk reside on the edges instead of the vertices, with each ordering of the edge's vertices corresponding to an independent basis state. Specifically, if we let v and w represent connected vertices on the graph, then $|v,w\rangle$ and $|w,v\rangle$ would be the two states corresponding to that edge. We will describe the ordering of vertices in each edge-state as pointing *from* the first vertex *to* the second, so that the state $|v,w\rangle$ could be identified as the state in which v points to w.

Now, the *step* operator $\hat{S}$ will be defined as a unitary, linear operator on the state space of the graph, and is therefore locally defined as a unitary operator on each of the separate basis states. To construct this behavior, let us consider a particle occupying an edge-state $|v,w\rangle$. We want $\hat{S}$ to advance the walk by *scattering* the particle through the vertex w, and we want the definition of this localized step operator to be valid across all edge-states with w as the target. Since w is the *only* variable upon which this definition is chosen, we can refer to our local step operator as $\hat{S}_w$.

Now let us explore our use of the term "scattering," and what we mean by it in the present context. As is typical when one hears the term, there is the anticipation of components for both transmission and reflection, the sum of which can then be taken to form $\hat{S}_w$. Let us again turn to a particle in the initial state $|v,w\rangle$. For transmission, we want the particle to end up in a superposition of the set of edge-states $\{|w,x\rangle \mid x \text{ is connected to } w, \text{and } x \neq v\}$, with each state having equal amplitude to the rest. Let us call this amplitude t. For reflection, we want the particle to bounce back onto the edge of origin, but in the

opposite, vertex-swapped state $|w, v\rangle$. This amplitude we'll denote as $-r$. Combining these behaviors, we get the localized action of $\hat{S}_w$ to be,

$$\hat{S}_w|v, w\rangle = -r|w, v\rangle + t \sum_{\forall x, x \neq v} |w, x\rangle. \tag{6}$$

Enforcing the unitary requirement for $\hat{S}_w$ determines r and t exactly, up to an arbitrary choice of relative phase. For if $\hat{S}_w$ is unitary, then it a) is normal and b) preserves orthogonality between input and output states. The former condition implies that

$$\left|\hat{S}_w|v, w\rangle\right|^2 = 1 \tag{7}$$

from which it follows that

$$|r|^2 + |t|^2 \sum_{\forall x, x \neq v} \langle w, x|w, x\rangle = 1 \tag{8}$$

$$\Longrightarrow |r|^2 + |t|^2(n-1) = 1 \tag{9}$$

where n is the number of edges connected to w. Preservation of orthogonality between input and output states implies that

$$\left\langle \hat{S}_w(z, w) \middle| \hat{S}_w(v, w) \right\rangle = 0 \tag{10}$$

for all z connected to w, and where $z \neq v$. This gives us

$$-r^* t \sum_{\forall x, x \neq v} \langle w, z|w, x\rangle - t^* r \sum_{\forall y, y \neq z} \langle w, y|w, v\rangle + t^* t \sum_{\forall y, y \neq z} \sum_{\forall x, x \neq v} \langle w, y|w, x\rangle = 0 \tag{11}$$

$$\Longrightarrow -r^* t - t^* r + |t|^2(n-2) = 0. \tag{12}$$

Both conditions together, with the further stipulation that both r and t be real, yield

$$r = \frac{n-2}{n} \quad \text{and} \quad t = \frac{2}{n}. \tag{13}$$

Since $\hat{S}_w$ is defined locally only for the target vertex w, we can say that

$$\hat{S}_w|v,x\rangle = 0 \quad \forall\, x \neq w. \tag{14}$$

Then, our global step operator can now be concisely defined across all vertices as

$$\hat{S} = \sum_w \hat{S}_w\,. \tag{15}$$

To complete this picture, we must choose a start vertex for the graph that will be used to determine our initial state $|\psi\rangle$. We will define this state to be a normalized superposition of all edge-states for which our start vertex is the origin, or

$$|\psi\rangle = \frac{1}{\sqrt{n}} \sum_{\forall x} |v,x\rangle \tag{16}$$

where n represents the number of edges connected to v. The last ingredient which we will throw into the mix is a *marked vertex*, a type of vertex that flips the phase of the particle it scatters. If this feels like an artificial construct to the reader, then it can be thought of as a simple quantum logic gate, but its implementation is tangential for our purposes. Note that it is an optional component and can be placed anywhere and in any number throughout the graph.

Now that we have mathematically formalized our scattering quantum walk, the reader might be turning their attention to pragmatic questions at this point. What can it do? What problems can it solve? Well, because we have chosen to explore this algorithm in the abstract space of graphs, we have endowed it with a very broad versatility. For instance, the Grover search algorithm can be rephrased as a scattering

quantum walk using a simple "star" graph (Figure 2), where a central, starting vertex connects directly to a set of perimeter vertices that encircle it.[6] Using this setup, the perimeter vertices represent a list of memory addresses with a marked vertex among them, and the marked entry itself indicates the presence of the search condition. Incidentally, the Grover algorithm demonstrates a quadratic speedup over the classical linear search, boasting a runtime complexity of $\mathcal{O}(\sqrt{N})$. In a more general sense, this type of quantum walk is simply defined by its behavior within the idealized hardware that gives it its unique and desirable properties. Like the classical random walk, its purpose is to be utilized in situations where that behavior has meaning.

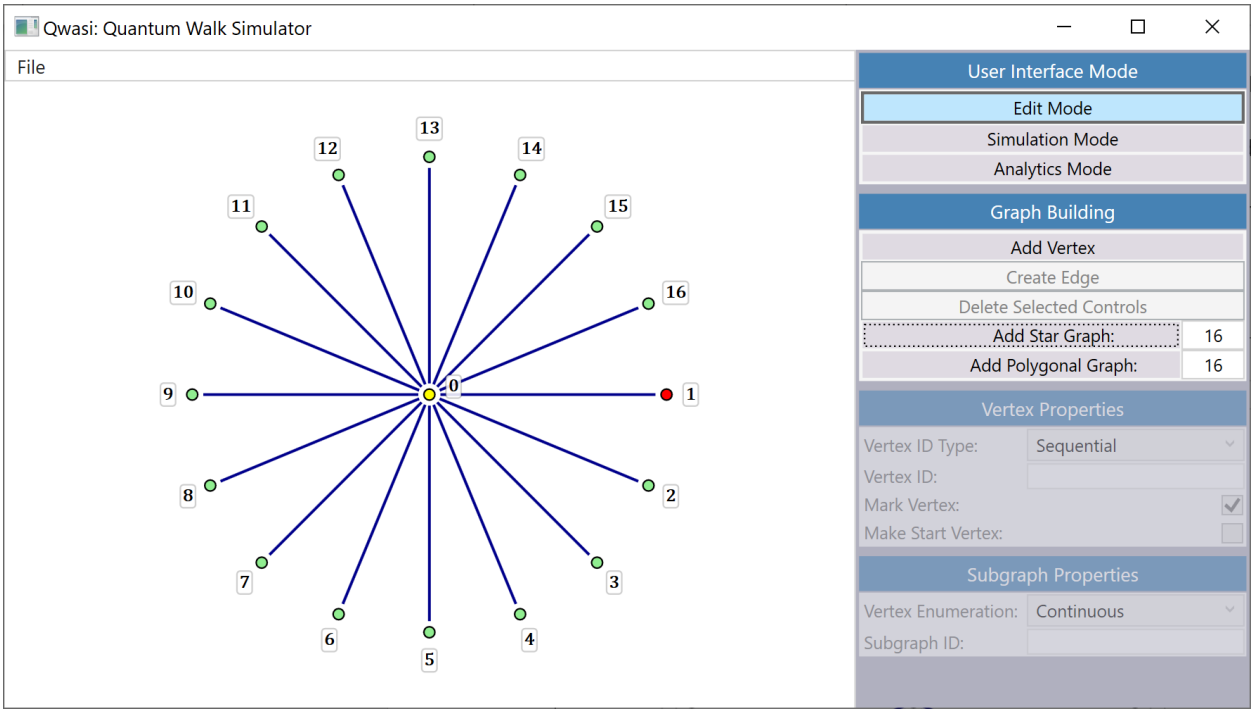


Figure 2: *A star graph with 16 perimeter vertices built in Qwasi's "Edit Mode." The yellow fill of the central vertex declares the starting position of the graph, while the red fill in vertex 1 indicates that it is "marked."*

However, we can probe a little further. The algorithm's most obvious strength is omnipresence within the graph space, where its ubiquity is established after very few steps. It is poor, however, at processing data since we have deliberately excluded any logic gates (with the exception of marked vertices). Personally, I

see power in using scattering quantum walks to building circuits that interact with memory, where vertices depict RAM blocks or CPU registers and edges represent connecting wires. Under favorable conditions, the scattering within these circuits can establish novel ways of accessing data, using statistics and interference to interact with large quantities of information at once. For the empiricists who might need to visualize some form of the hardware, one possible implementation is using photons, where fiber optic cables act as edges and an array of half-silvered mirrors perform the scattering at the nodes.

So, we have thus established the virtues of studying scattering quantum walks on arbitrary graph structures, but we have not touched upon *how* to do it. Regrettably, there is no way to algebraically analyze them in the general sense for the vast majority of graphs. Even when it is possible in simple cases, such as with the Grover algorithm, it requires symmetrizing basis states into a smaller subspace, constructing a matrix form of the step operator, solving for its eigenvectors and eigenstates, and then making asymptotic simplifications for large datasets. Even minor complexities render a graph algebraically intractable, which is what makes Qwasi a vital heuristic tool in this area of research.

This is what Qwasi can do. It allows for the creation of an arbitrary graph by the user and simulates a scattering quantum walk within it. The nuts and bolts of the software will be covered later in detail, and with greater attention paid to the computer science involved. Here, we will focus on the software's power to generate numerical solutions to quantum walks on *any* graph given to it. The most basic uses of Qwasi are likely to follow a similar pattern. First, construct a graph within which exists one or several points of interest. One might need to use marked vertices or extra edges to break symmetries. Second, use the step operator to find an evolution of the walk where the amplitudes are maximized at these points. Third, ensure such amplitudes are significantly greater in magnitude than those in other parts of the graph. And

fourth, using the probabilities derived from squaring these amplitudes, find a suitable number of times the walk can be repeated to acceptably minimize the chances of error. More sophisticated uses of Qwasi might easily stray from this formula. For instance, by performing the walk several times but with different step counts, one might be able to isolate regions of the graph using the combination of the separate interference patterns. Because investigating this behavior requires the composition of data across a range of step numbers, Qwasi is equipped with an analytics mode that allows edge-state coefficients to be plotted across this domain. This differs from simulation mode, which allows for the viewing of the graph as a whole at incremental values of the step operator. During the simulation in this mode, the edges take on color coding to allow for a pictorial view of the state, and the spectrum transitions in ascending fashion from white to red to black. This shading represents the probability of occupancy for that edge, calculated from the sum of the squares of the two edge-states coefficients. From here forward, the uses of Qwasi are only limited by the imagination, for the set of all graphs is unbounded, and my hope is that this tool finds good use in the exciting research of today.

Computer Science and Code Structure

Qwasi is written in C#, a C++ derived, object-oriented language similar to Java that is part of the Microsoft .NET suite of development software. The graphical and user interface portions of the code are built using WPF (Windows Presentation Foundation), a .NET library of graphical controls with event-based interactivity. The free and public "LiveCharts" and "Extended WPF Toolkit" libraries, both obtainable on NuGet, were also used. Though C# is a multiplatform language, the WPF library is not (at the time of this writing), so some version of the "Windows" operating system is required to run the software. The code is freely accessible at the following GitHub repository, https://github.com/warrenvirgilwilson/Qwasi, as are the precompiled binary releases, https://github.com/warrenvirgilwilson/Qwasi/releases. Qwasi is composed of four main parts: the mathematical engine, the user interface, the graph-building components, and the simulation/analytics modes.

Mathematical Engine

This portion of Qwasi is largely comprised of the data structures that build mathematical expressions and hold them in memory; a demanding task as it turned out. The average algebraic binary tree was insufficient here because entities beyond real valued functions, such as the generalized vectors of Hilbert space, required their own expression trees as well. Therefore, to circumvent a system with a ton of redundant code, a ground up inheritance scheme was implemented with future modularity in mind. The objects themselves comprise a collection of interfaces and classes that make heavy use of both *generics* and *default interface methods* (a rather new feature of C# at the time of this writing) to maintain the highest level of abstraction.

For example, the interface *IBasicExpression* inherits from the general *IMathExpression*, and defines a type of entity upon which the operations addition, subtraction, and negation are defined. Since all derived expression types employ these basic operations, such as *IAlgebraicExpression* and *IVectorExpression*, they all inherit from this interface. The generic argument *<TExpression>* seen throughout is used as a self-referential parameter type in the inheritance scheme, so that classes derived from these types can preserve strong-typing with regard to their methods and properties. For instance, *IAlgebraicExpression* inherits from *IBasicExpression<IAlgebraicExpression>* so that the basic operations of addition, subtraction, and negation mentioned above take and return arguments of the type *IAlgebraicExpression*. It should be noted that this type of structural hierarchy can admit additional mathematical constructs with very little code redundancy. For example, though the need for complex numbers in the present discourse did not arise, they could be easily implemented as an algebraic value type should they be required in a future context.

The principal need for this expression tree-structure is to build the recurrence relation for the quantum walk *step operation*. Qwasi will take an arbitrary Hilbert space vector – essentially a dictionary of basis states paired with variable coefficients – and build a general *stepped* vector from it. This latter vector will also be a dictionary of basis states and coefficients, but the coefficients in this case are algebraic functions over the set of variable coefficients of the *unstepped* vector. This recurrence relation, when supplied with the appropriate start vector, will provide the evolution of the quantum walk across any number of step operations. Important to this functioning is that the expression tree which represents the recurrence relation is immutable, and remains unchanged by the evaluation of a step. Thus, the numerical coefficients obtained by one step operation can be passed back into the recurrence relation for the evaluation of the

next. To perform this computation, one instantiates a MathContext object and populates it with the basis-coefficient pair values of the current state, then passes it into the *Evaluate(…)* method of the recurrence vector.

The remainder of the mathematical backend achieves the following. First, it translates the custom graph built by the user into the basis edge-states that populate the Hilbert space, then it builds the *step operator* object used to compile the recurrence relation. Since the step operator is a linear operator, it is constructed as the global sum of all local step operations on each edge-state, the formal behavior of which has been defined in the first section of this paper.

User Interface

Since WPF is an event based graphical library (similar to JavaScript objects), Qwasi's user interface is closely tied to its graphical presentation. However, some effort was made to segregate the two when possible, not only to keep them independent but also to lessen the degree to which WPF is entwined within the code. In this, a trait-based approach took advantage of *default interface methods* in order to keep behavioral logic and graph controls (vertices, edges, and subgraphs) as separate as possible.

To accomplish this, all graph controls implement the interface *IWPFContainer*, which defines within it a property – *WPFPrimaryElement* – that acts as a bridge between what is shown on the screen and how it responds to user interaction. This property is of type *UIElement*, an abstract class in the WPF library from which most visual entities inherit, and allows for the binding of event handlers to achieve the desired UI

behavior. With this feature of *IWPFContainter* exposed, the event logic is accessible to the user interface when necessary, and the remainder of the UI behavior is established in a set of *trait*-like interfaces: *IQGDraggable*, *IQGSelectable*, *IQGMouseHandler*, *IQGMouseDraggable*, *and IQGUserDeletable*. They are *trait*-like in the sense that their logic is already coded in the interface definition and that their action, for the most part, is included simply via inheritance. For example, an object that inherits from both *IQGSelectable* and *IQGUserDeletable* can be selected and deleted by the user, but if the latter interface is removed, it can be selected but not deleted. (User deletion can also be blocked within objects that inherit from *IQGUserDeletable* by setting the Boolean value *UserDeletable* to false). Objects may also override or implement certain members of these interfaces to add customized logic.

The rest of the UI logic resides mostly in the GUI (Graphical User Interface). There is a lot of code in this, but most of it is clear and unambiguous, so there's little incentive to expound here. I will note that much of the graphical styling that went into this section could probably have been done more easily in Microsoft's *XAML*, which is built for this kind of user presentation, but I didn't go this route for the simple reason that I am not familiar with it.

Graph Building

The graph building part of Qwasi is arguably the heart of the software, since it is here that the program's ease of use will translate into its effectiveness as a research tool. A graph building tool that performs poorly at building graphs is of use to no one, regardless of how powerful it is at compiling analytics. For this reason, many features are incorporated into the software here strictly for user convenience.

One such feature is the inclusion of *subgraphs*. Subgraphs are themselves graph controls (like vertices or edges) but contain within them a subset of vertices and edges of their own. As of now, there is one type of subgraph included with two variations. It is called a *perimeter graph*, since it describes a graph whose vertices are a fixed radial distance from the center. The first variation of this is the *star graph* (Figure 3.a), where all perimeter vertices are connected to a central vertex. The second variation is the *polygonal graph* (Figure 3.b), where the center vertex is a dummy vertex (it serves only as a means for selecting and moving the control by the user) and the outer vertices are connected to their two proximate neighbors. Additional types of subgraphs would not be an onerous effort to code, and because subgraphs are also implementations of *graph layers*, subgraphs within subgraphs are possible.

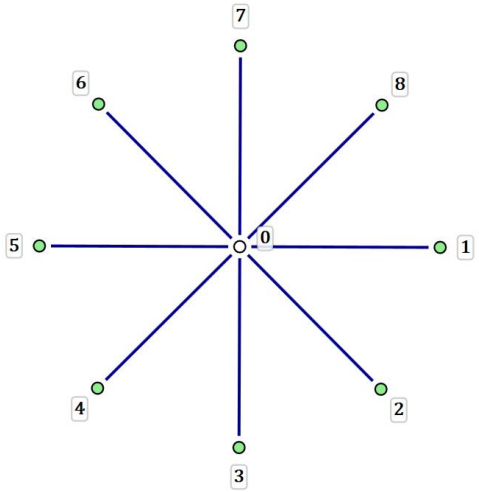


Figure 3.a: *A Qwasi star graph with eight perimeter* vertices.

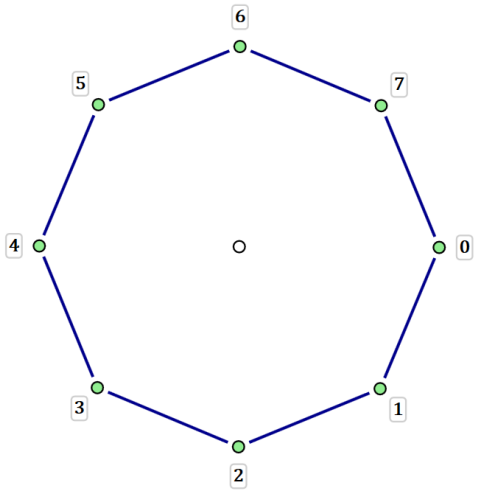


Figure 3.b: *A Qwasi polygonal graph with eight perimeter vertices.*

A *layer* type, which inherits from *IQGLayer*, is intended to represent the most abstract form of a container object for control types, placing them relative to the layer's own local coordinates. There are only two types of layers: *graph layers* and *binding layers*. A graph layer derives from the interface *IQGGraphLayer*, a subtype of *IQGLayer*, and is built to contain and display the controls on the graph. It is also itself a control

in that it inherits from *IQGControl*. In the case of a perimeter graph, the entire layer can be moved by dragging its center. It can also be rotated and resized by dragging one of its outer vertices. The *WPFPrimaryElement* of a graph layer is a *Canvas* control, which is used here because it allows for numerical coordinate placement in the layout of its children. The root layer of the graph is itself a derivation of *IQGGraphLayer*. It should be noted that while building a subgraph could be accomplished "by hand," vertex by vertex and edge by edge, such tedium would undermine the usefulness of the software. Qwasi's power is tied to its ability to make changes on a trial and error basis – *build, simulate, analyze, revise* – so priority was paid to the ease with which one can make manipulations. The second type of layer is the binding layer (*QGBindingLayer*), which allows for binding controls to other controls. In this context, the term control refers to a set larger than just the graph controls, including the index labels seen next to vertices and the edge labels of simulation mode that display the Hilbert space coefficients. *QGBindingLayer* is a property of all graph controls and organizes the local placement of its children relative to that of its parent. The implementation of this in WPF makes use of the *adorner layer* of the control's *WPFPrimaryElement*.

Another core feature of the graph builder is the ability to save graphs to a .qwg (**Q**uantum **W**alk **G**raph) file and reopen them later. To do this, the graph is serialized into and deserialized from XML using a completely readable schema, so any third-party software can make use of this filetype without much difficulty. The serialization library was written from scratch in order to implement a design concept of mine. In this architecture, serializer objects inherit from an abstract class that takes for a generic argument the type of the class that it serializes. XML attributes in the serializer are registered to it by invoking a function of three arguments: the XML name of the attribute and a pair of serialization/deserialization methods. The serialization method takes an object and returns a string which encodes its value, and the deserialization method takes a string, parses it, and then integrates it back into an instance of the

deserialized object. Using lambda functions, registering an XML attribute can often be done in a single line of code. XML child elements can be registered in a similar fashion, with a function that takes a child serializer, an element selector method, and a content integrator method as arguments. The child serializer already contains the XML element name as a property, and can then take an object instance of the registered type and serialize/deserialize it in recursive fashion. The content given to the child serializer can be either a single element or a list and is selected from the parent object with the element selector method provided. To recover the serialized data, the child serializer will provide the parent with an object of the registered type, and the content integrator method will reattach the child object(s) to the parent. Registration of XML elements and XML attributes is best taken care of in the serializer's constructor. This way, serializers can be inheritable in the same fashion as the objects they act on, which provides for clean and readable code. For example, in Qwasi's implementation, *QXVertexSerializer* inherits from *QXControlSerializer* in the same way the *IQGVertex* inherits from *IQGControl*.

Lastly, much of the use of default interface methods in Qwasi includes the atypical use of *state* information, which is considered by some to be taboo in languages such as C# and Java. With this I disagree, and believe C++ to be superior in this one aspect since it allows for state in its unfettered approach to multiple inheritance. However, I acknowledge how uncomfortably close this statement veers to a debate that's raged for decades, and to bring it to the present discourse would be a mistake. I will simply proceed by explaining *how* Qwasi manages state within C# interfaces and not whether it should. All interfaces that want to claim state information (fields and event delegates) can inherit from *IInterfaceStateAgent<TInterface>*, where the generic type *<TInterface>* is again self-referential. For example, *IQGDraggable* inherits from *IInterfaceStateAgent<IQGDraggable>*. The generic parameter in this case serves to differentiate implementations of *IInterfaceStateAgent* between interfaces, so that each of their state information remains distinct when combined within a single derived type. Inheriting

IInterfaceStateAgent exposes a set of protected methods that interact with a data storage backend. For variables, these methods are

IInterfaceStateAgent<TInterface>.GetInstance(variableId),

IInterfaceStateAgent<TInterface>.GetInstance(variableId, initialValue), and

IInterfaceStateAgent<TInterface>.SetInstance(variableId, objectInstance).

For events, they are

IInterfaceStateAgent<TInterface>.AddEventHandler<TEventArgs>(eventId, eventHandler),

IInterfaceStateAgent<TInterface>.RemoveEventHandler<TEventArgs>(eventId, eventHandler), and

IInterfaceStateAgent<TInterface>.RaiseEvent<TEventArgs>(eventId, args).

The variableId and eventId arguments in the functions above are of a struct type that takes a *<TInterface>* generic argument (to tie the field to the relevant interface) and a string parameter which indicates the name of the field. Use of these functions within property/event accessors binds interface values to state, eliminating the need to implement them in the derived classes. Finally, interfaces can also override the *IInterfaceStateAgent<TInterface>.Constructor()* method to add constructor logic to the interface. This method is called once immediately before the first field or event accessor is executed.

The design of *IInterfaceStateAgent* went through two revisions. The first employed a single, static dictionary, keyed by object instance, that contained a second dictionary, keyed by identifier object, to store state information. While it worked, it was scrapped when I realized that the references in such a static dictionary prevented any of its members from being garbage collected. Rather than dealing with object disposal logic, I replaced the static dictionary with an abstract *InterfaceStateCatalog<TInterface>* object to be implemented by an inheriting class. This local entity keeps track of an interface's state within the variable scope of the class so no disposal issues would ensue during garbage collection.

The *QwasiEngine* Class and User Interface Modes

Most of the remaining logic in Qwasi is relegated to the *QwasiEngine* object, a broad, application wide class whose only instantiation is owned by the main application window. It is not a singleton class, however, as its scope is not global. The range of tasks it performs is diverse and numerous, but generally involves mediating the interactivity between the separate modules of code discussed. The following are its most notable responsibilities. It coordinates the mathematical engine described earlier in this section, overseeing the conversion of graphs into Hilbert space, defining the linear step operator, and constructing the walk's recurrence relation. It is also tasked with keeping track of simulation variables, such as the current step number and the coefficients of the corresponding Hilbert space vector. One of its more essential roles is in Qwasi's behavioral logic where it facilitates the transition between user interface modes, taking on such related features as the color coding of edges during a simulation or the compiling of Hilbert space coefficients into plots. While some of these tasks might rightly be better handled elsewhere, and with greater modularity, the present implementation should be thought of as stopgap toward that goal. But some tasks will always rely on its higher-level access, such as binding the simulation/analytics labels to their edges or serializing the graph into XML.

One final point regarding code relates to the implementation of the simulation/analytics labels themselves, though what they do and how they operate will be covered in the *Data Presentation* section below. Like every other item on the graph canvas, they derive from *IQGControl*, but they are not graph controls. They are part of a class of controls called bound controls, so named for their positional binding to the entities they modify, following them on the graph as they move. This is achieved by registering them to their target control's binding layer. Programmatically, bound controls are not a control type of their own, but a conceptual denomination based on how they are used. During a switch into either

simulation mode or analytics mode, the *QwasiEngine* class is tasked with creating and binding these labels, which includes registering the necessary event logic with the edge control to allow for interactivity.

Data Presentation

One endemic challenge to working with large Hilbert spaces is that monitoring basis coefficients is like searching for a needle in a haystack. It's hard to identify the relevant data when it's presented as an endless list of numbers, and any significance about the system's global behavior is buried in its cryptic presentation. For instance, in a scattering quantum walk, a graph of just twenty-five edges creates a Hilbert space of fifty basis states, so a dataset gathered after just two step operations contains one hundred and fifty coefficients (if one includes the starting state). As mentioned earlier, Qwasi offers two distinct modes to overcome this obstacle: *simulation mode* and *analytics mode*. Both offer powerful visual control over graph data and an intuitive focus on what's most relevant. To demonstrate each, let us consider the Qwasi built graph in Figure 4.

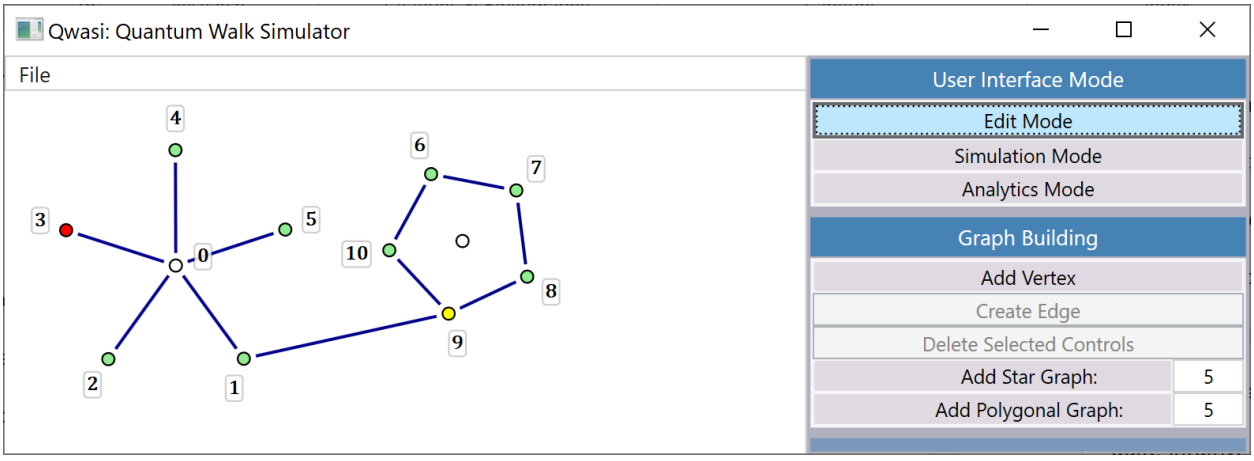


Figure 4: *An example graph constructed from a star subgraph and a polygonal subgraph, attached by the edge* $(1,9)$*. The red vertex is marked, and the yellow vertex is the starting position.*

Simulation Mode

Simulation mode is centered around the real-time monitoring of a quantum walk as the user advances it through the step operator. In this mode, edges of the graph are color coded relative to the probability of their occupancy, the value of which is itself derived from the two edge-state amplitudes. As mentioned earlier, the spectrum transitions from white to red to black corresponding to ascending probability. The exact numerical weight used for this shading however is not directly linear to this value, but instead its square root. This is to bring out the detail in the lower value regions, since high probabilities along a single edge are unlikely for larger graphs. This method of visualization allows for quick dismissal of trivial areas, a process that would otherwise require great scrutiny. However, for those edges that do show interesting behavior, a popup label displaying their two basis state coefficients appears upon hovering the mouse over them. The label can then be pinned so that it remains visible throughout the simulation and the user may pin as many labels as they like. This way, the numerical data that is specifically chosen is available for each iteration of the walk. Let us explore this behavior using the graph from Figure 4. After a simulated walk of 150 steps only looking at edge colors, the darkest shadings were observed along edges $(0,1)$, $(1,9)$, and $(6,7)$. The simulation was then reset, and these edges were pinned. Figure 5 shows the starting state of this walk.

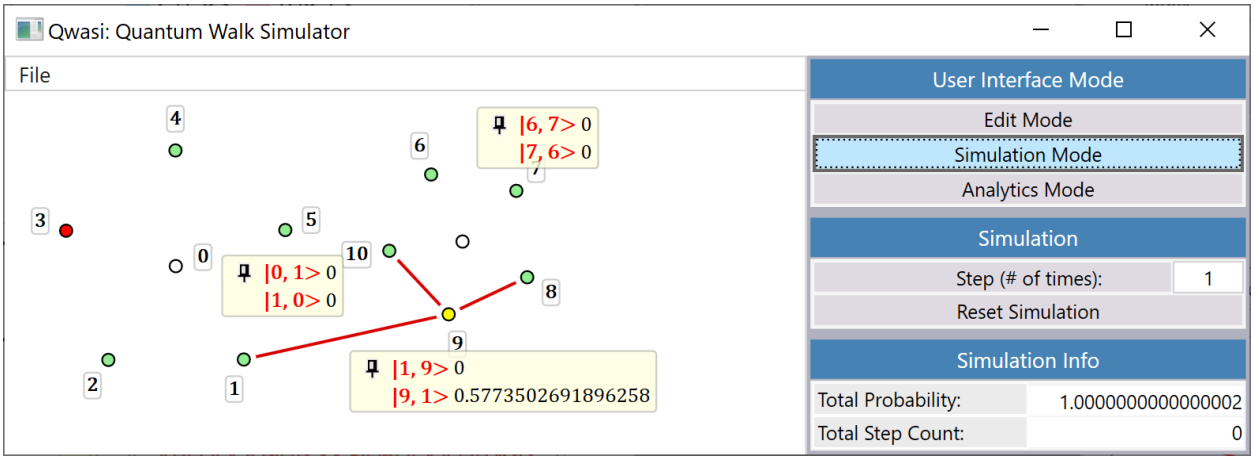


Figure 5: *The starting state in simulation mode for the graph in Figure 4. The edges* $(0,1)$, $(1,9)$, *and* $(6,7)$ *displayed interesting behavior on a first pass, so the simulation was reset and these labels were pinned.*

The walk was then iterated step by step. Figure 6, Figure 7, and Figure 8 all show the state of the walk at progressive step values where the greatest amplitudes were observed.

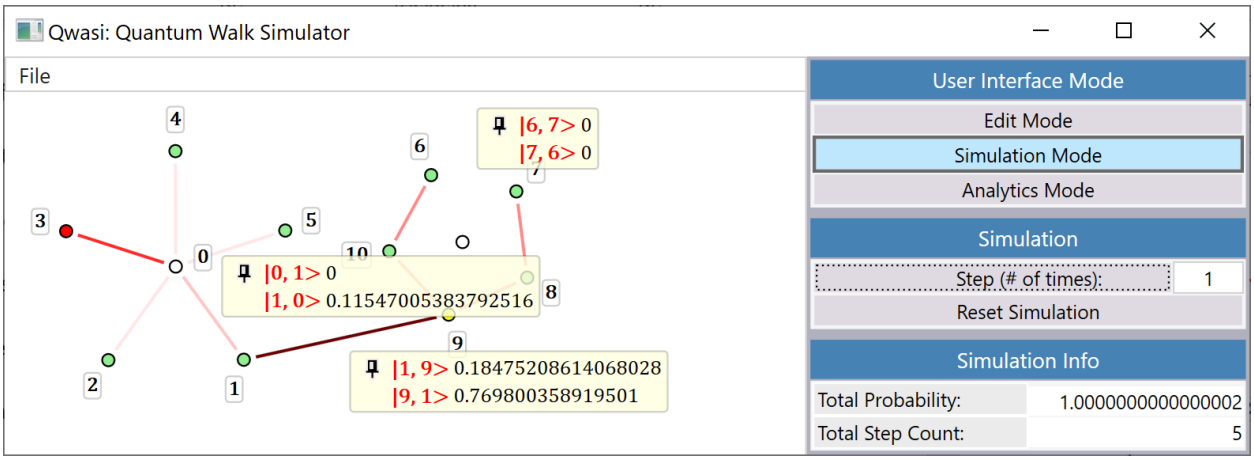


Figure 6: *The state of the walk from Figure 5 after 5 steps.*

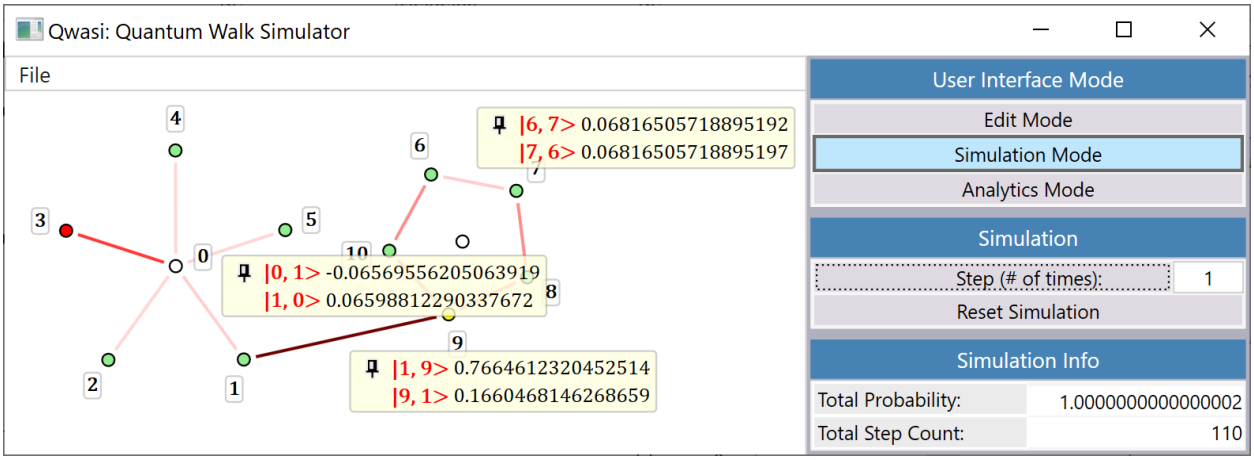


Figure 7: *The state of the walk from Figure 5 after 110 steps, showing the largest edge-state amplitudes across* $(1,9)$ *since step 5.*

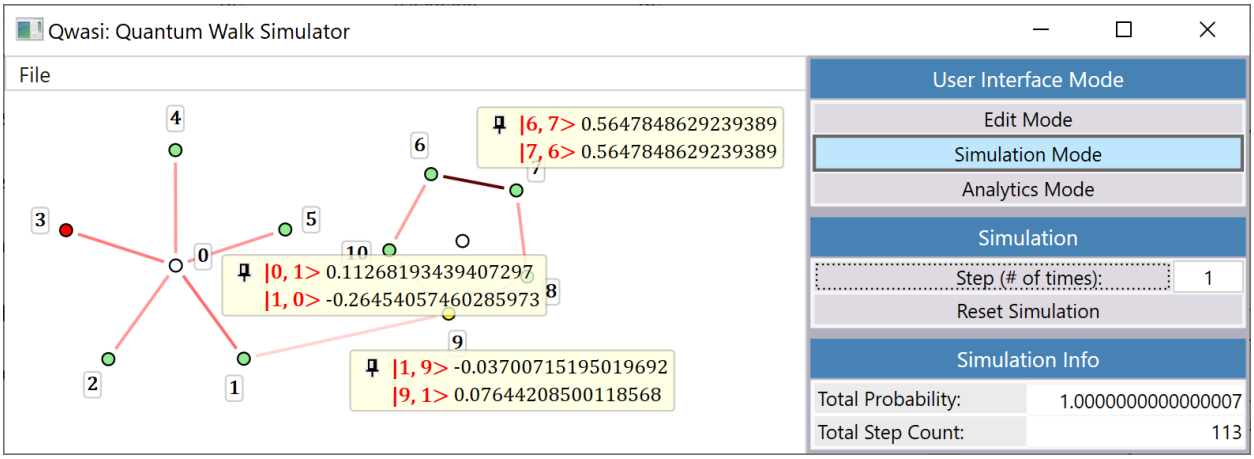


Figure 8: *The state of the walk from Figure 5 after 113 steps, showing the largest edge-state amplitudes across* $(6,7)$ *since step 2 (not shown here).*

In this particular example, it quickly becomes obvious that the two basis states for edge $(6,7)$ are always equal in amplitude. While this may not seem very surprising after noticing the symmetry involved, the ability to track these amplitudes in real time made this peculiarity unmissable.

Analytics Mode

Analytics mode is designed with a slightly different goal in mind, which is to focus on long-term behavior over a large domain of steps. Rather than view the entirety of the graph for each step taken, Qwasi can compile the walk data over a specified range with bounds that the user can choose. Then, similar to simulation mode, each edge will feature a unique label assigned to it with data specific to that edge. Instead of the edge coefficients of simulation mode, however, the label displays a graphical plot of their behavior over the interval of steps specified (Figure 9). This plot is a thumbnail that when clicked will open a larger, customizable plot (Figure 10). This second plot can then be saved to a PNG file for later viewing (Figure 11). Lastly, the compiled analytics data can be exported to a CSV (Comma Seperated-Values) file en masse, so that it can be opened as a dataset in a spreadsheet program like Microsoft Excel.

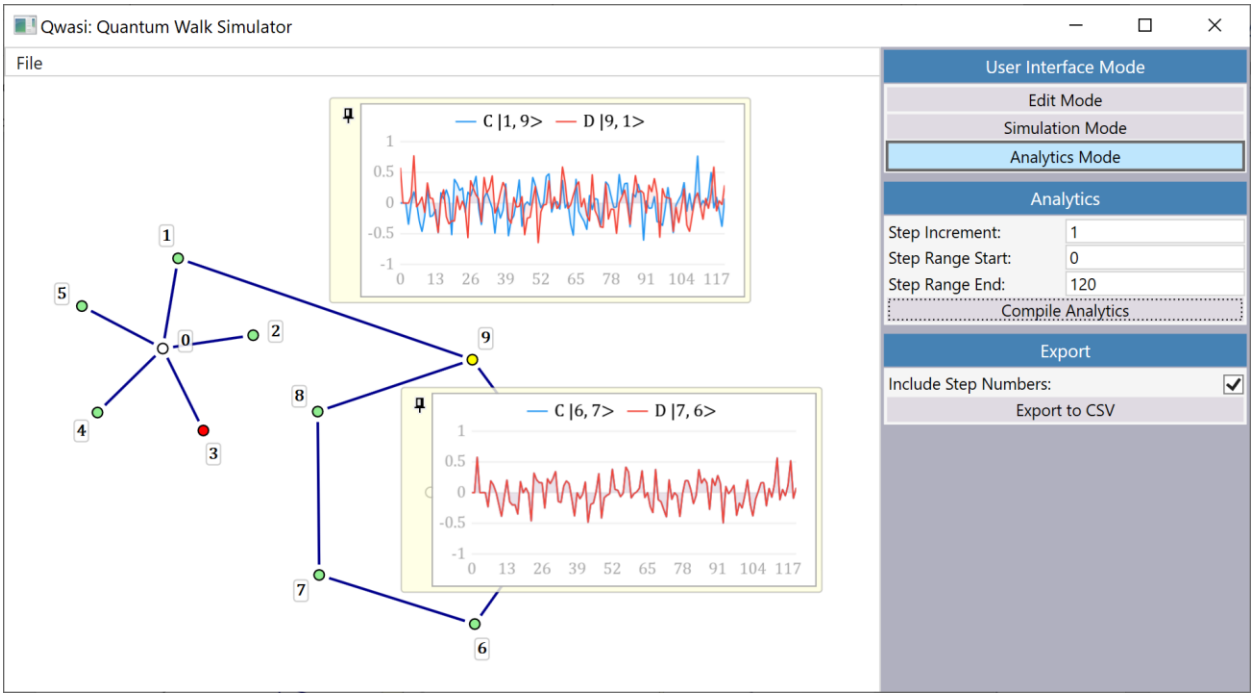


Figure 9: *The graph from Figure 4 in analytics mode. The contents have been moved and rotated in order to accommodate the thumbnail plots for edges* $(1,9)$ *and* $(6,7)$*. The equality between the coefficients of states* $|6,7\rangle$ *and* $|7,6\rangle$ *is obvious here, for the thumbnail of that edge has only one line.*

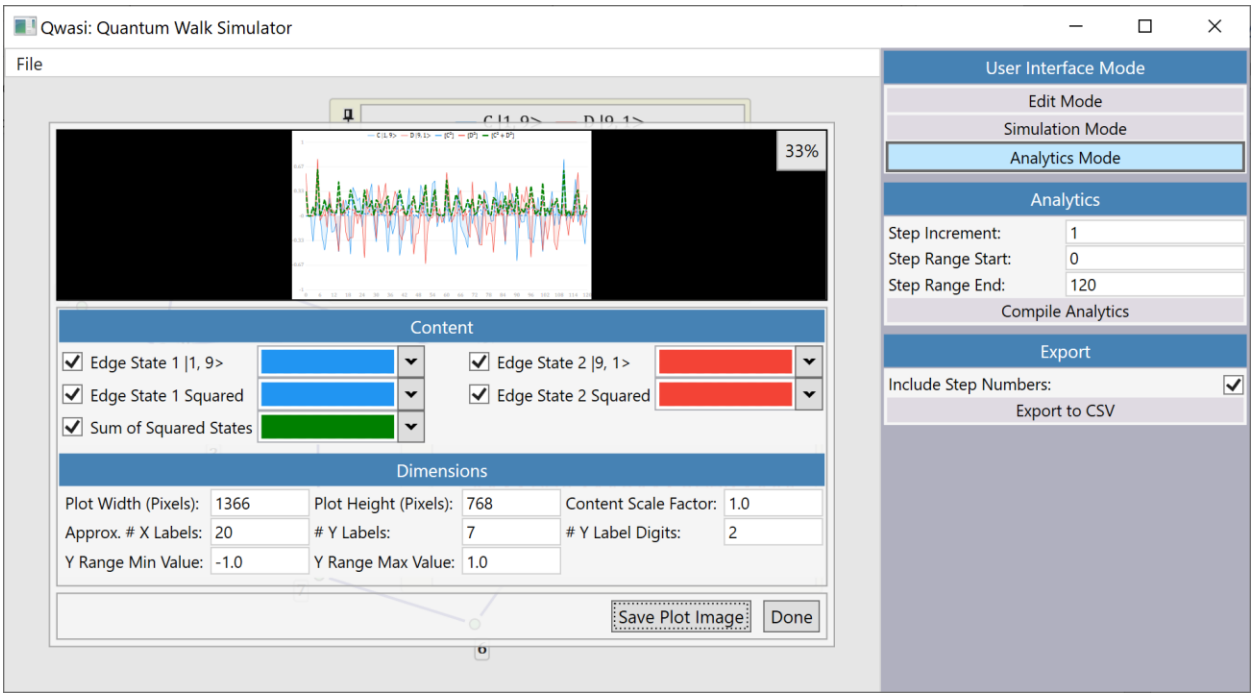


Figure 10: *The plot customization frame for edge* $(1,9)$ *of the graph in Figure 9. The choice of what data to include is given by five toggleable line series, whose colors are user selectable. The dimensions of the rendering and the resolution can also be freely chosen.*

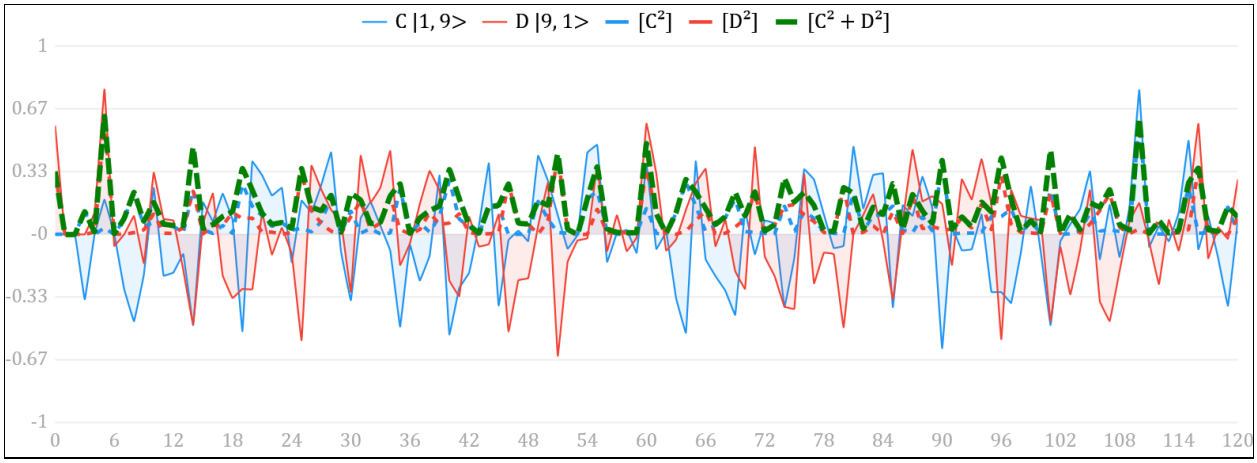


Figure 11: *A rendered PNG plot for edge* $(1,9)$ *in a 120 step walk over the graph in Figure 4. The peak probability values (green dotted line) at steps 5 and 110 – as was noticed in simulation mode – are readily visible here.*

Conclusion

Quantum computation is a thriving field which holds exciting possibilities for the future but finding techniques with which to research it can be challenging. Sometimes algebra does the trick and our models are readily interpretable. Other times it comes desperately short. Since the time that computers have been able to offer assistance, numerical solutions to complicated problems have become the norm in many areas of research. But quantum mechanics often involves large systems, abstract vector spaces, and complex operations that can make data quite cryptic. In fields like fluid dynamics, visualization is often quite intuitive, whereas translating quantum data into some form of visual entity often requires great effort and insight. But I also believe doing so is necessary for fruitful research. That is why I think Qwasi is important, and it is why I spent so much time creating it. In the future, I hope to see others pick up on this and expand it to fit their own applications, but if this an overly optimistic wish, I will be content just to see it used for the purpose that it's been designed.

---

[1] P. W. Shor in Proceedings of the 35th Annual Symposium on the Foundations of Computer Science, Santa Fe, NM, p. 124 (1994).
[2] Y. Aharonov, L. Davidovich, and N. Zagury, Phys. Rev. A 48, 1687 (1993).
[3] D. Aharonov, A. Ambainis, J. Kempe, and U. Vazirani, in Proc. of the 33rd ACM STOC, 50--59 (2001).
[4] E. Farhi and S. Gutman, Phys. Rev. A 58, 915 (1998).
[5] M. Hillery, J. Bergou, and E. Feldman, Phys. Ref. A 68, 032314 (2003).
[6] J. Bergou and M. Hillery, Introduction to the Theory of Quantum Information Processing, (springer, New York, 2013) p. 105.

Appendix: User Instructions

Much effort went into the design of Qwasi to prioritize its ease of use. Therefore, my hope is that the software is sufficiently intuitive that this appendix is just a supplementary resource; to be consulted as opposed to perused. At any rate, I aim to make this section as thorough as possible so that the mechanics of all parts of Qwasi are discussed in detail.

Overview

The overall presentation of Qwasi aims to be sparse by design, intended to give a maximum amount of screen space to the graph canvas. The controls for every action in the program (except saving and opening graphs) are stacked to the right in their own panel, which in the code is referred to as the *command panel*. The three different UI modes are always docked at the top of this panel in a frame labeled "User Interface Modes," with the currently active mode highlighted in blue. All content below this is dynamic and presents relevant information, settings, and commands based on context.

We'll start first with a brief mention of the menu bar. The only menu here currently is the *File* menu, which has the *Open* and *Save* commands. This is a meager set of tasks to warrant an entire menu bar, so I confess that its presence here is mostly to accommodate future functionality. For example, I hope for it to one day contain an *Edit* menu, as currently there isn't one. Indeed, this means (for now) no copying and pasting of graph controls or undoing of previous commands. The lack of an *undo* feature can be circumvented by saving frequently, but there is no denying that some standard functionality is currently lacking.

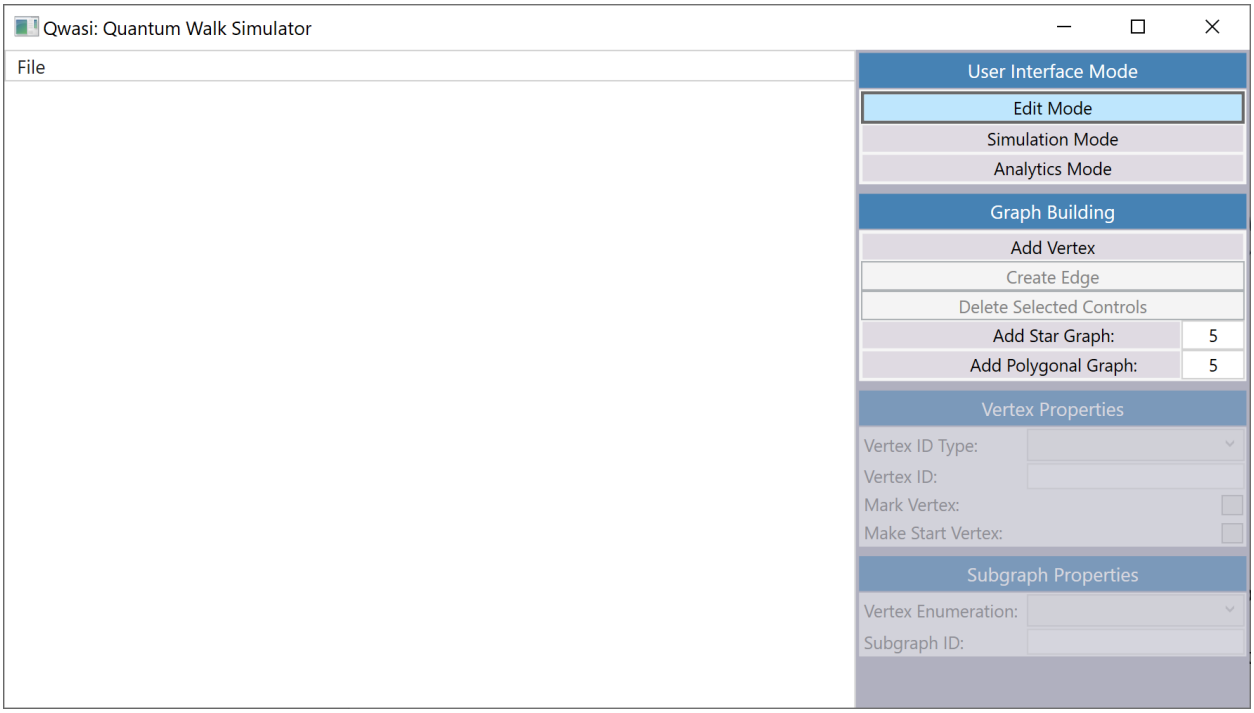


Figure 12: *The initial layout of Qwasi after launching the application.*

Edit Mode

Edit mode is the default starting mode when opening the application and can be seen as such by its blue highlighted status in the "User Interface" pane on the command panel. Right below this is the "Graph Building" pane, which houses a series of buttons for creating and deleting graph controls. The first of these is the "Add Vertex" button, which does just that. The next one down is the "Create Edge" button, which is always *grayed out* unless exactly two vertices are selected (Figure 13). When this condition is met, clicking the button will generate an edge between the them. Skipping the "Delete Selected Controls" button for now, the remaining two options create the *star graphs* and the *polygonal graphs* described in the main body of this paper. Each of these buttons has a textbox to its right whose default value is 5, but which can be changed to hold any integer greater than or equal to 3. This value determines the number of perimeter vertices a subgraph will have when generated by the button to its left (Figure 14).

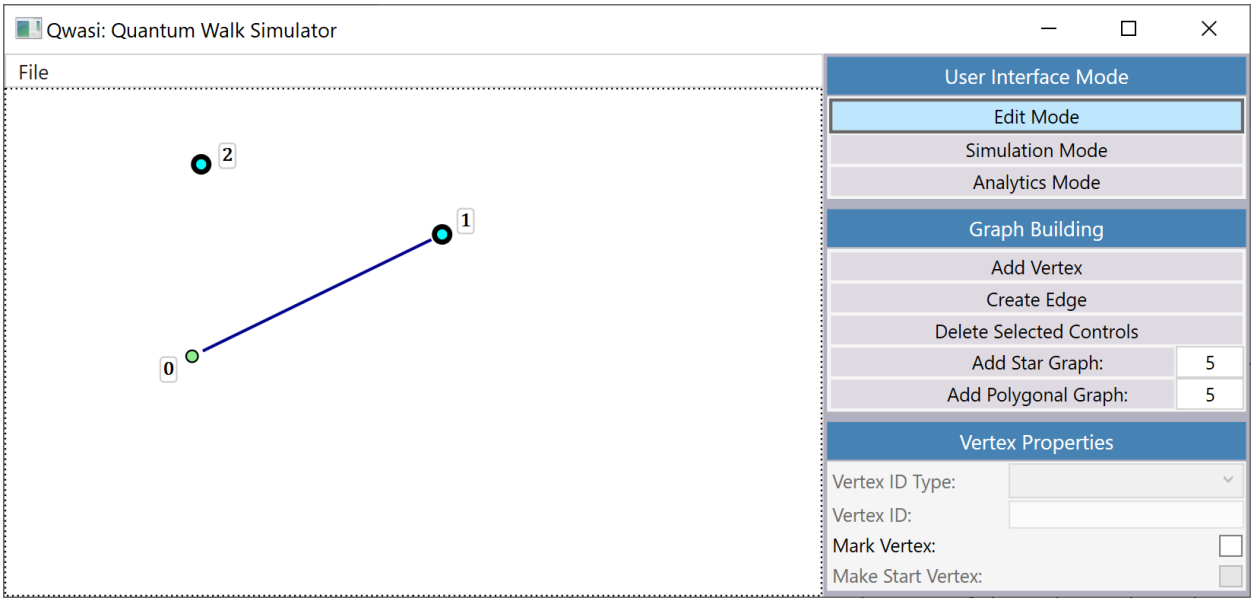


Figure 13: *Selecting exactly two vertices activates the "Create Edge" button in the "Graph Building" pane.*

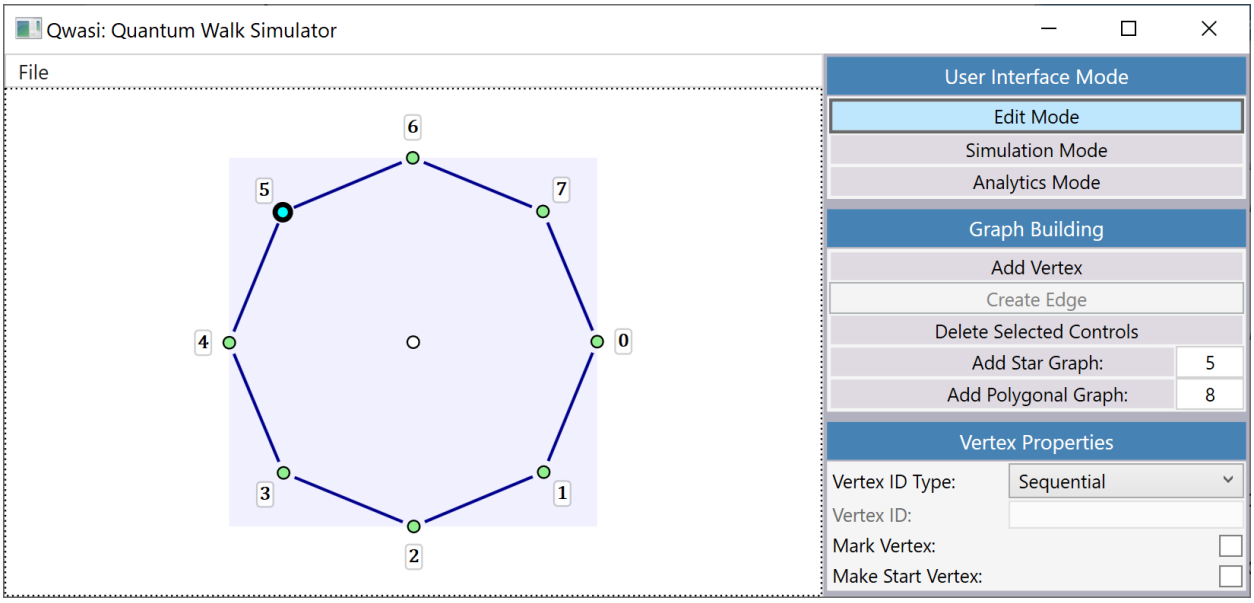


Figure 14: *A polygonal graph with 8 perimeter vertices, generated by typing 8 into the textbox next to the "Add Polygonal Graph" button. Also note the dummy vertex at the subgraph's center, identifiable by its lack of label, as well as the white fill that all central vertices have.*

When a *perimeter graph* is created, the vertex at its center is its proxy, and moving or deleting it will do the same to its parent. In the case of the polygonal graph, its center is merely a dummy vertex; a means

by which to manipulate the subgraph control without contributing any mathematical significance (Figure 14). One can identify central vertices by their white color, rather than the standard green of the others, except in cases where this is overridden such as when a vertex is *marked*. Whereas moving the center vertex of a subgraph will move the subgraph as a whole, moving a *perimeter vertex* will instead transform it. Dragging it in the radial direction will change the size of the subgraph, while moving it at any angular displacement will generate an overall rotation (Figure 15). This feature keeps subgraphs clean and symmetric so that working with them remains tractable, especially in cases where they are densely populated with vertices and edges.

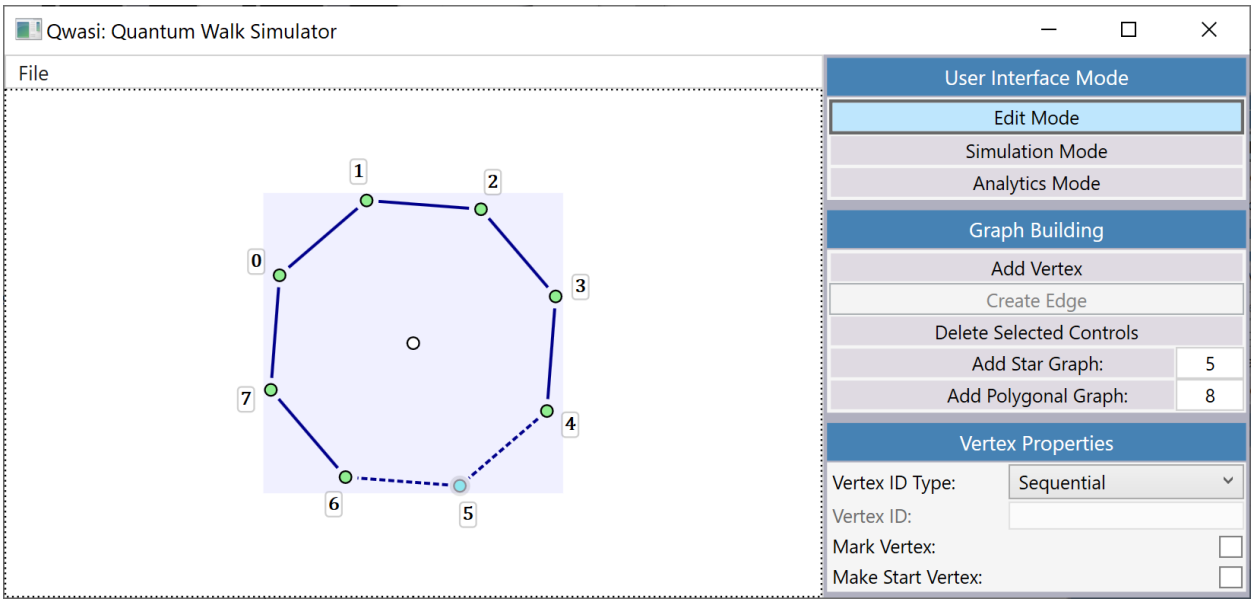


Figure 15: *The polygonal graph from Figure 14, rotated and resized by dragging vertex "5." Also on display here is the background purple highlight for the subgraph, visible because one of its children is selected.*

When any member of a subgraph is selected – edge or vertex – the subgraph itself is also selected as shown graphically by a purple square highlight around its contents (Figure 15). When this is the case (and when only one subgraph is highlighted) the "Subgraph Properties" pane in the command panel will become active (Figure 16), giving the user control over the subgraph's properties. In this section, the "Add

Perimeter Vertices" button allows the user to add vertices to the subgraph in the quantity specified to its right (Figure 17). Above this, the "Vertex Enumeration" menu offers a powerful tool by which to organize the graph, but it requires a bit of background first on vertex IDs.

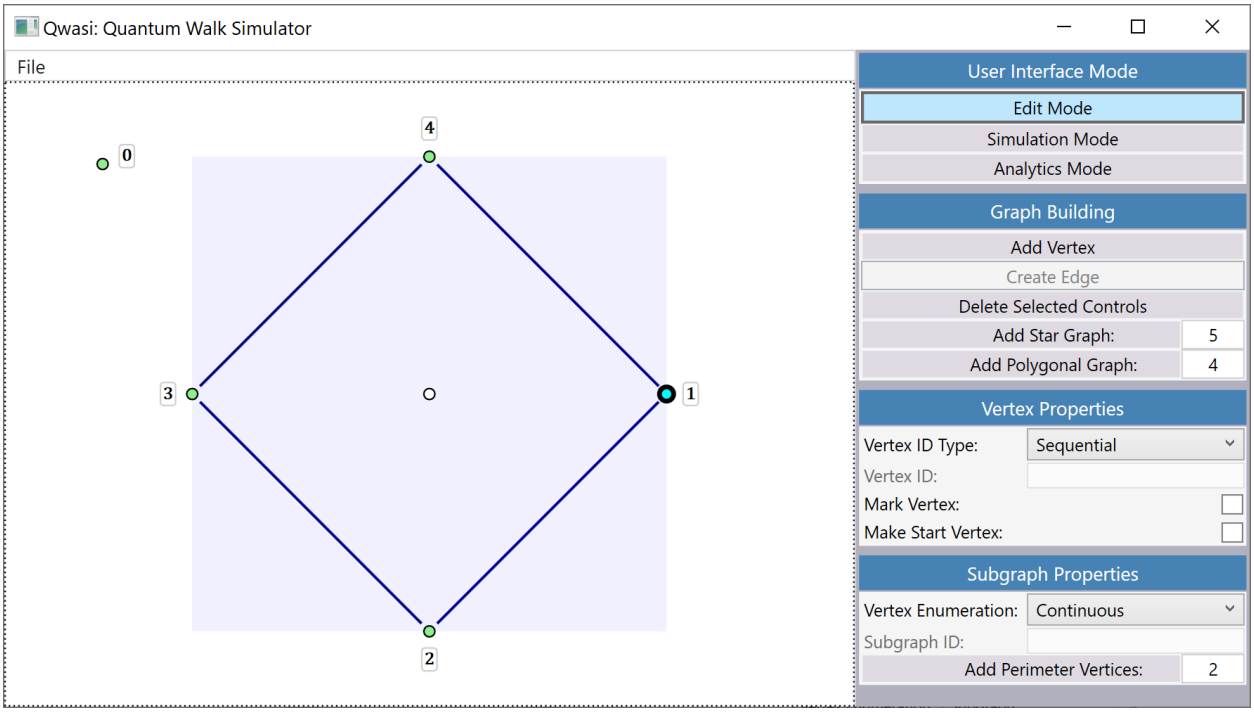


Figure 16: *A diamond polygonal graph of 4 vertices. Because the child vertex "1" is selected, the "Subgraph Properties" pane in the command panel is visible.*

All vertices have an ID, which by default is assigned using a global enumeration of integers starting at 0. If a subgraph's "Vertex Enumeration" property is changed from "Continuous" to "Subgraph" however, it will be taken off the global enumeration queue and instead generate its own local indices. This requires providing an input for the "Subgraph ID" field directly below the "Vertex Enumeration" menu, which will then generate vertex IDs of the form "[subgraph ID]:[vertex ID]". For example, if a subgraph has an ID of "SG," then its child vertices will be locally enumerated starting with the ID "SG:0" (Figure 17). This feature is purely for organization and has no mathematical impact on the graph whatsoever. Its inclusion is intended to make very large or very complex graphs more manageable.

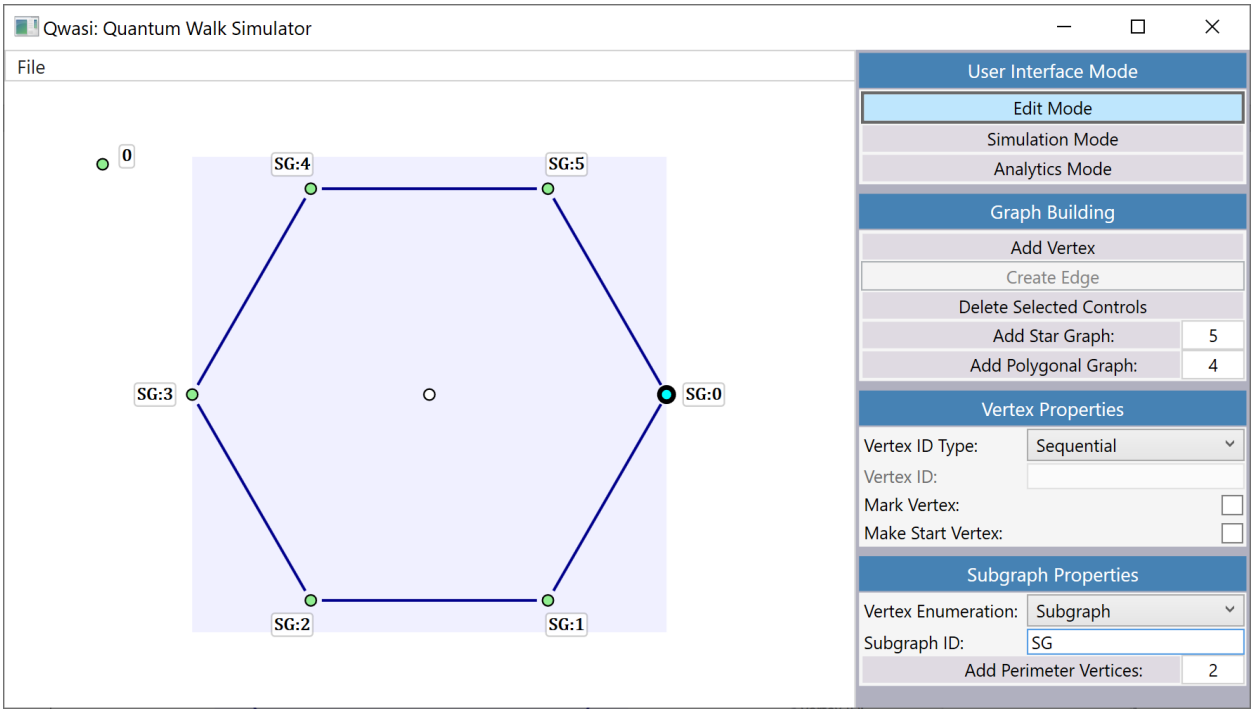


Figure 17: *The graph from Figure 16 with 2 vertices added using the "Add Perimeter Vertices" button. Additionally, the "Vertex Enumeration" property has been set to "Subgraph" and the subgraph has been given an ID of "SG." Note that whereas in Figure 16 the global enumeration of vertices gives the subgraph a starting index of 1, the local enumeration of vertices in this case starts it at 0.*

When a user selects one and *only* one vertex, the "Vertex Properties" pane in the command panel will become active. The first property in this control is the "Vertex ID Type" menu, which continues the above discourse on how vertex IDs are assigned. "Sequential" is the default value and enumerates vertices as discussed. "Custom Global" removes indexing for that vertex altogether, permitting an entirely arbitrary input such as "exit" or "start" or anything that adds clarity to the layout. "Custom Local" works the same way as "Custom Global," but will prefix the subgraph ID to the input. Again, using the example above of a subgraph with ID "SG," if one of its child vertices has a "Vertex ID Type" set to "Custom Global" and a "Vertex ID" input of "global-vertex," then it will have a full vertex ID of "global-vertex," whereas if it has a "Vertex ID Type" set to "Custom Local" and a "Vertex ID" input of "local-vertex," then it will have a full vertex ID of "SG:local-vertex." This might sound confusing, but seeing it represented visually in Figure 18

should help. The two remaining checkboxes in this category can set the status of the vertex, either making it the *start vertex* or *marking* it.

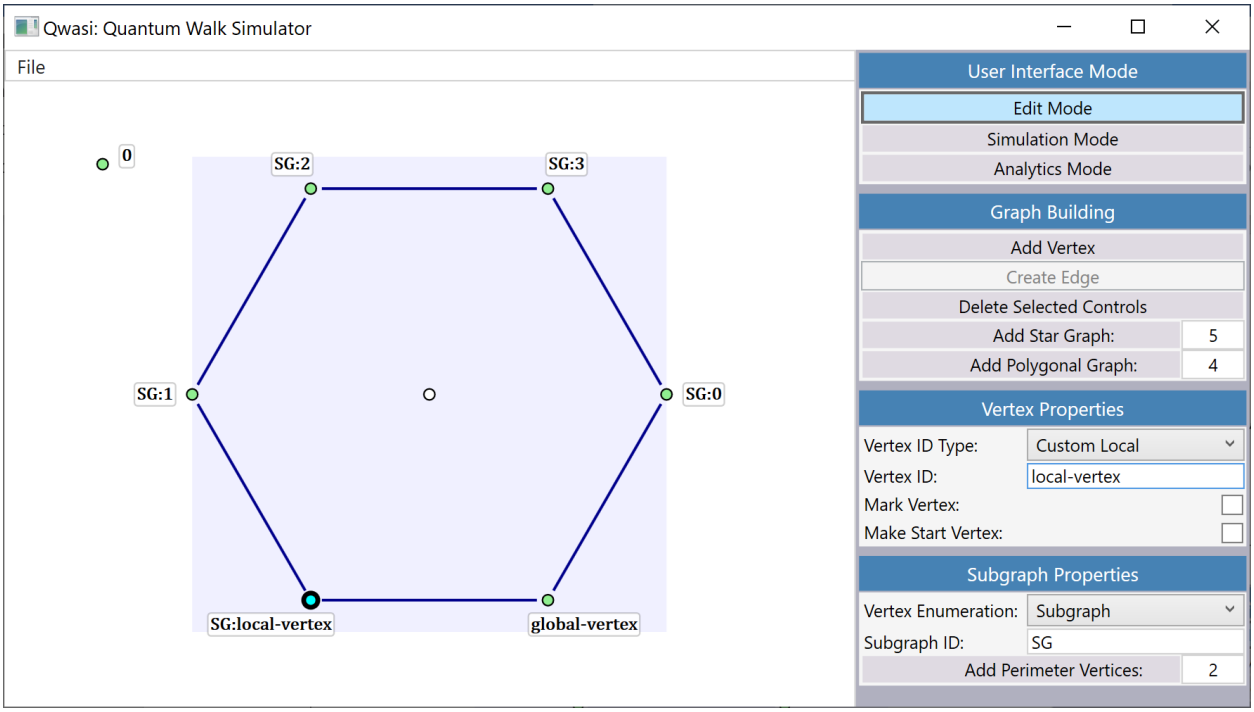


Figure 18: *The graph from Figure 17 with vertex "SG:1" given a "Custom Global" ID of "global-vertex" and vertex "SG:2" given a "Custom Local" ID of "local-vertex." Notice too how the previously indexed "SG:3" vertex is now labeled "SG:1", since the original "SG:1" and "SG:2" from Figure 17 have been taken off sequential enumeration.*

We have talked a lot about the things one can do when there are selected controls on the graph, but we have not talked about *how* to select them. You just click on them. To select more than one control, there are two methods. The first is to hold down the shift key and click on them one at a time. This is effective for a few controls but a nightmare for many. The second method is to click on the graph's empty space, hold the mouse button down, and drag the cursor to form a blue "selection" rectangle (Figure 19). All controls that lie inside this rectangle when the mouse button is lifted will be selected. This feature should be familiar to anyone who knows their way around contemporary operating systems, where files and folders are selected in the same way.

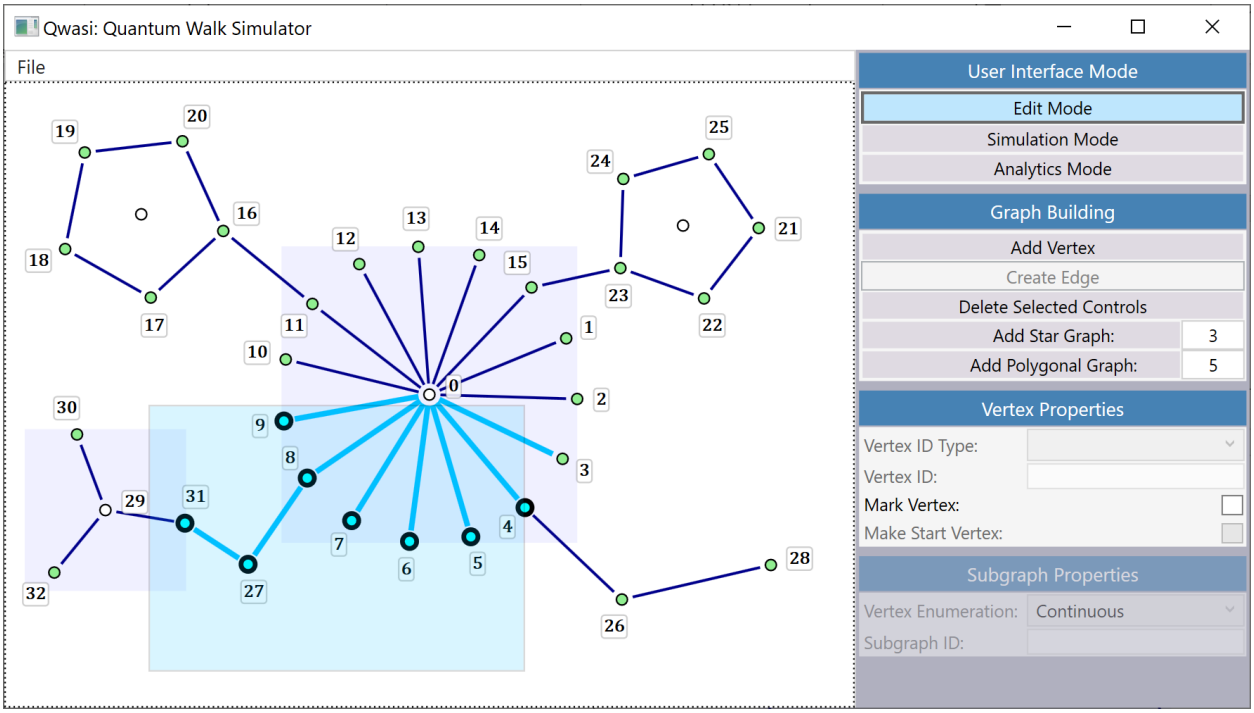


Figure 19: *An example of an arbitrary graph whose controls are highlighted via the "selection rectangle."*

The last aspect of edit mode that warrants discussion is control deletion, which is pretty straightforward. One can either use the "Delete Selected Controls" button in the "Graph Building" pane, or just press the backspace or delete keys on the keyboard. Obviously, for this to work one or more controls must be selected. Deleting a vertex will also delete its edges but deleting edges do not affect vertices (except in subgraphs). The outer vertices in a perimeter graph can be deleted and will shift the others to retain symmetry. The only truly unique behavior is the central vertex of a subgraph, which will delete its parent when it itself is deleted.

Simulation Mode

First, it is important to know that in both simulation mode and analytics mode, the user may jump back into edit mode at any time to make modifications to the graph as needed. Before the user can enter simulation mode, they must first choose a *start vertex* somewhere on the graph. After this criterion is met, the mode becomes accessible. Once in simulation mode, the most obvious change to the graph presentation is that the edge controls have been recolored, taking on various shades of red to represent the probability of occupancy for that edge. This has been covered extensively in the first part of this writing, but as the present discourse is designed to be instructional, we will review it here.

The probability that a particle will be detected on any given edge is the sum of the squares of its two edge-state coefficients. This value determines where on the spectrum the color should lie, with 0 being white, 1 being black, and total red (an RGB value of $[1,0,0]$) sitting at 0.25 (Figure 20). The initial state of the graph will determine the starting colors here, where edges attached to the start vertex present as some shade of red, and all edges elsewhere show up as white. This might seem surprising at first, for the white edges will appear to have vanished! Rest assured this is intentional. Transparency correlates naturally with the probability of measurement, for if an edge cannot permit occupancy then it makes sense for it to be indiscernible. On the other end of the spectrum is an edge which is entirely black, a most imposing presence on the graph, and indicates with total certainty the outcome of a measurement.

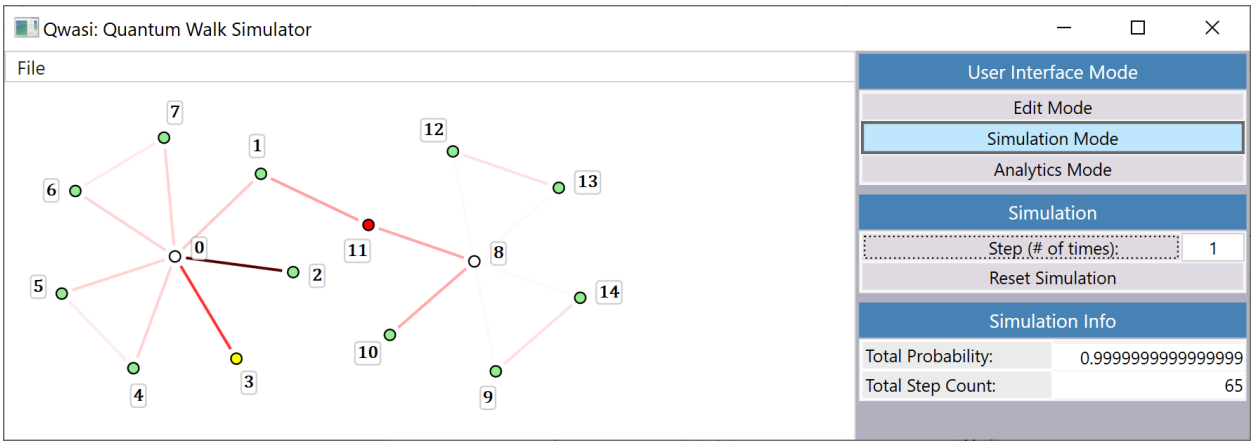


Figure 20: *An example of a walk with different edge shadings. The very pale edges, such as* $(8,13)$*, indicate very low probabilities, whereas the very dark brown of edge* $(0,2)$ *indicates higher probabilities.*

To begin a simulation, one clicks the step button in the "Simulation" pane of the command panel which is only visible in this mode. The field to the right of this button permits the user to choose how many steps to perform for each click. Though this value will typically stay at 1, changing it can be helpful in many circumstances, such as if the user would like to "jump" to a particular step count to progress the walk from there. Right below this is the "Reset Simulation" button, which returns the walk to its starting state. At any stage during the simulation, hovering the cursor over an edge control will display a popup showing the coefficients of its two basis states. Clicking the pin icon on this will allow it to persist on the graph even after the cursor is moved away from the edge (Figure 21).

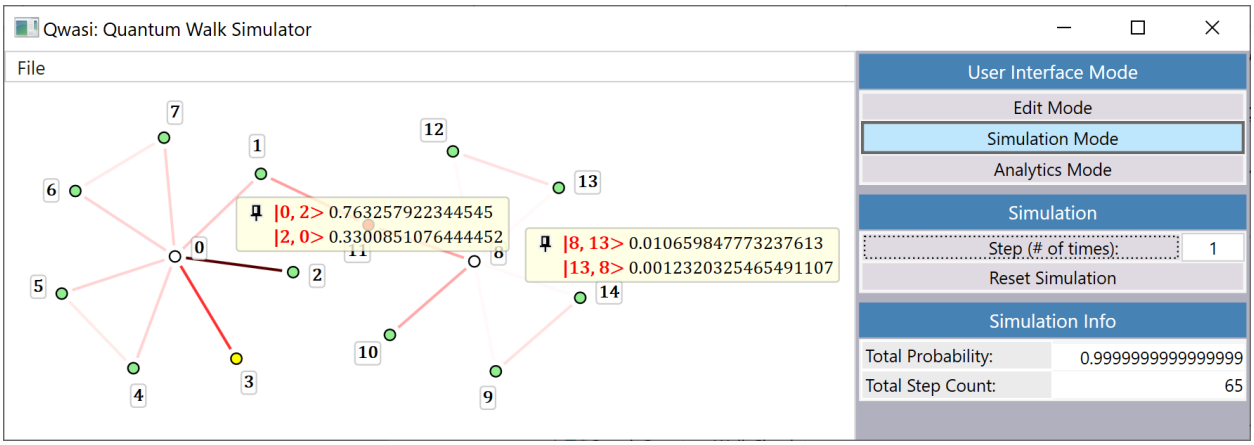


Figure 21: *The walk from Figure 20 with edges* $(8,13)$ *and* $(0,2)$ *pinned. Also notice the deviation of the total probability from 1.0, which gives a sense of the overall error, and the "Total Step Count" showing the walk at step 65.*

The last thing to discuss here is the "Simulation Info" pane in the command panel which, like the "Simulation" pane, only appears in simulation mode. The "Total Probability" label here gives an aggregate of the squares of all edge-state coefficients, which should of course be 1.0 (Figure 21). The inclusion of this field is intended to provide a loose estimate for the degree of error caused by the truncation of 64-bit floating-point numbers. In my experience, this value remains very small even over tens of thousands of steps. The field below this labeled "Total Step Count" displays the step number at which the current state of the simulation resides.

Analytics Mode

Analytics mode is designed to allow for the compilation of large sets of data into a single, readable form. Instead of dealing with each step individually as we do in simulation mode, we can define a range of steps over which our data of interest resides. These settings are contained within the "Analytics" pane in the command panel, which is only visible in analytics mode. The "Step Increment" textbox in this pane

determines the number of steps between each recorded state of the system, and the “Step Range Start” and “Step Range End” fields determine the bounds of the domain (Figure 22). Before anything useful can happen in analytics mode, these settings must be set and the “Compile Analytics” button below them must be clicked.

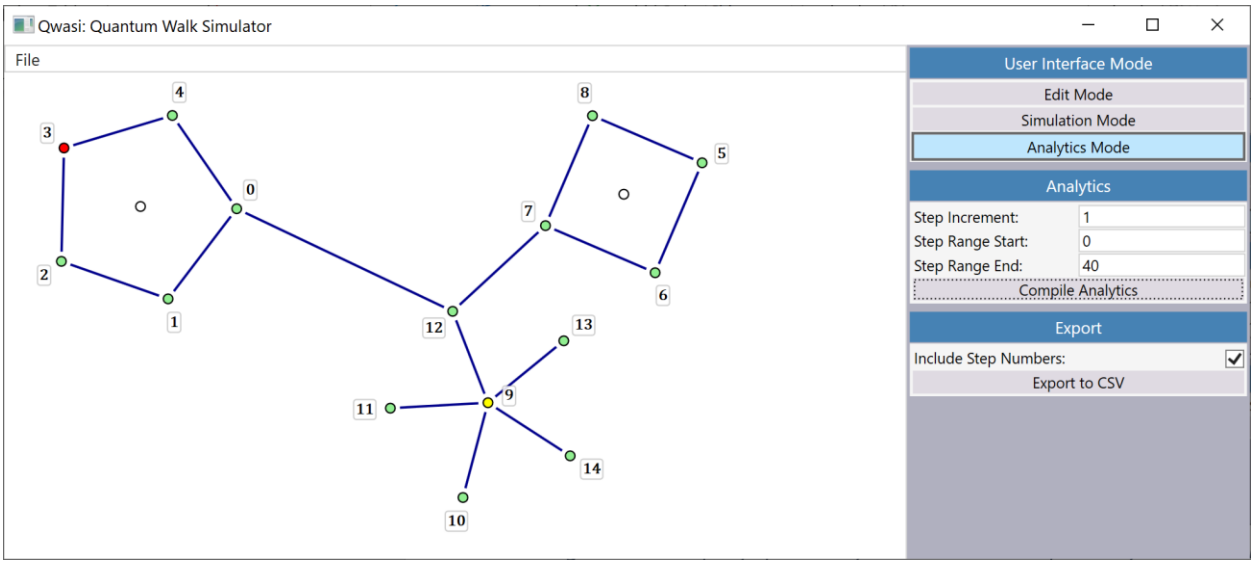


Figure 22: *A sample graph in analytics mode. The “Compile Analytics” button must be pressed before any data can be viewed.*

Once the analytics data are compiled, a popup label – similar to the one in simulation mode – will appear upon hovering the mouse over an edge. Again, this label can be pinned so that it stays visible when recompiling the data (Figure 23). The only content within this popup is a thumbnail plot over which the two edge-state amplitudes are graphed. This is designed only to give a very basic overview of the edge’s behavior.

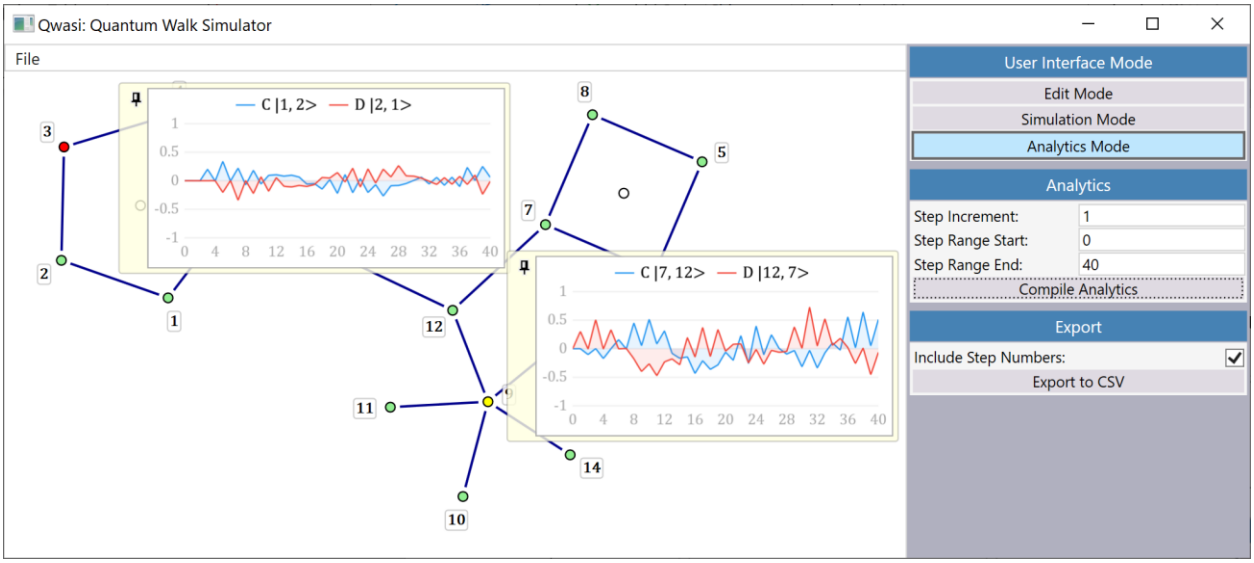


Figure 23: *The graph in Figure 22 with edge labels pinned. These labels contain a thumbnail plot that offers a quick glance at an edge's behavior over the range of steps specified.*

If one clicks on this thumbnail, however, they will be taken to a more comprehensive panel of settings, with a far more detailed graph in a preview frame at the top (Figure 24). Here, housed within the section labeled "Content," one can choose which data to graph (amplitudes, amplitudes squared, or total probability), and the color of the corresponding series. Below this, in a panel labeled "Dimensions," one can specify the resolution of the graph, the number of labels for each axis, and the maximum and minimum values for the Y-axis. The "Content Scale Factor" here refers to the size of the graphics within the plot, such as text and the thickness of the plot lines. After customizing these options, the user can save the final plot as a PNG file for later use or analysis.

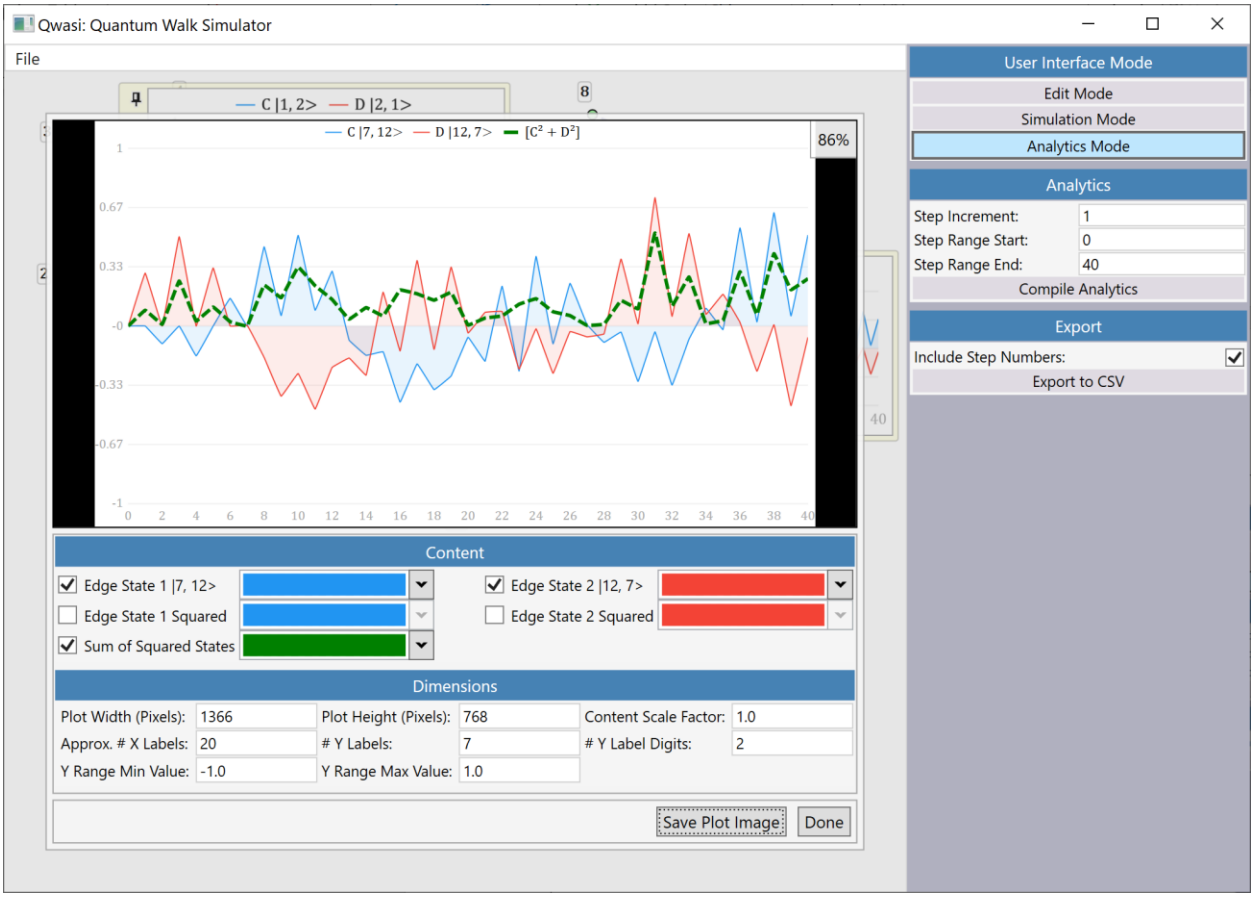
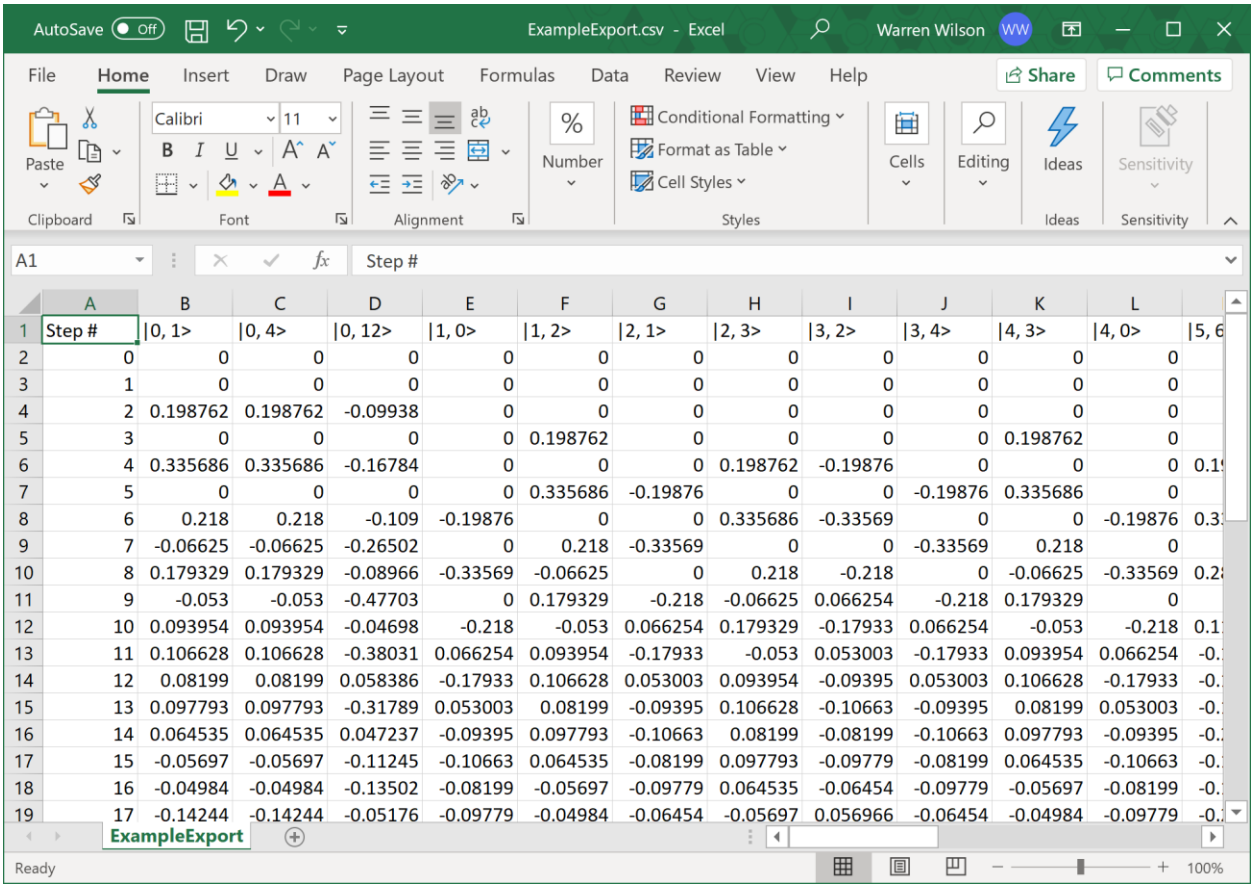


Figure 24: *A customizable plot of edge* $(7,12)$ *from the graph in Figure 22. This inner panel appears when clicking the thumbnail of an edge control's popup label.*

Finally, there is an "Export" pane in the command panel of analytics mode. This will take the analytics data, which again must first be compiled, and save it as a CSV (Comma Separated Values) file (Figure 25). This is a common format for spreadsheet applications like Microsoft Excel.



Step #	|0, 1>	|0, 4>	|0, 12>	|1, 0>	|1, 2>	|2, 1>	|2, 3>	|3, 2>	|3, 4>	|4, 3>	|4, 0>	|5, 6
0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	
2	0.198762	0.198762	-0.09938	0	0	0	0	0	0	0	0	
3	0	0	0	0	0.198762	0	0	0	0	0.198762	0	
4	0.335686	0.335686	-0.16784	0	0	0	0.198762	-0.19876	0	0	0	0.1
5	0	0	0	0	0.335686	-0.19876	0	0	-0.19876	0.335686	0	
6	0.218	0.218	-0.109	-0.19876	0	0	0.335686	-0.33569	0	0	-0.19876	0.3
7	-0.06625	-0.06625	-0.26502	0	0.218	-0.33569	0	0	-0.33569	0.218	0	
8	0.179329	0.179329	-0.08966	-0.33569	-0.06625	0	0.218	-0.218	0	-0.06625	-0.33569	0.2
9	-0.053	-0.053	-0.47703	0	0.179329	-0.218	-0.06625	0.066254	-0.218	0.179329	0	
10	0.093954	0.093954	-0.04698	-0.218	-0.053	0.066254	0.179329	-0.17933	0.066254	-0.053	-0.218	0.1
11	0.106628	0.106628	-0.38031	0.066254	0.093954	-0.17933	-0.053	0.053003	-0.17933	0.093954	0.066254	-0.
12	0.08199	0.08199	0.058386	-0.17933	0.106628	0.053003	0.093954	-0.09395	0.053003	0.106628	-0.17933	-0.
13	0.097793	0.097793	-0.31789	0.053003	0.08199	-0.09395	0.106628	-0.10663	-0.09395	0.08199	0.053003	-0.
14	0.064535	0.064535	0.047237	-0.09395	0.097793	-0.10663	0.08199	-0.08199	-0.10663	0.097793	-0.09395	-0.
15	-0.05697	-0.05697	-0.11245	-0.10663	0.064535	-0.08199	0.097793	-0.09779	-0.08199	0.064535	-0.10663	-0.
16	-0.04984	-0.04984	-0.13502	-0.08199	-0.05697	-0.09779	0.064535	-0.06454	-0.09779	-0.05697	-0.08199	-0.
17	-0.14244	-0.14244	-0.05176	-0.09779	-0.04984	-0.06454	-0.05697	0.056966	-0.06454	-0.04984	-0.09779	-0.

Figure 25: *A CSV file created using the compiled data from the graph in Figure 22, opened in Microsoft Excel.*

This concludes the appendix, which has hopefully been thorough enough to guide an interested party in their research. As I've said before, I encourage anyone who desires to modify Qwasi to do so according to their needs, and hopefully share these revisions with the community. The GitHub link mentioned earlier in this paper is live, so contributions can be made there.