Quantum Walks with Gremlin∗

Marko A. Rodriguez
Director of Engineering, DataStax Inc.
Project Management Committee, Apache TinkerPop

Jennifer H. Watkins
Information Systems and Modeling, Los Alamos National Laboratory
(Dated: November 20, 2015)

A quantum walk places a traverser into a superposition of both graph location and traversal “spin.” The walk is defined by an initial condition, an evolution determined by a unitary coin/shift-operator, and a measurement based on the sampling of the probability distribution generated from the quantum wavefunction. Simple quantum walks are studied analytically, but for large graph structures with complex topologies, numerical solutions are typically required. For the quantum theorist, the Gremlin graph traversal machine and language can be used for the numerical analysis of quantum walks on such structures. Additionally, for the graph theorist, the adoption of quantum walk principles can transform what are currently side-effect laden traversals into pure, stateless functional flows. This is true even when the constraints of quantum mechanics are not fully respected (e.g. reversible and unitary evolution). In sum, Gremlin allows both types of theorist to leverage each other’s constructs for the advancement of their respective disciplines.

I. INTRODUCTION

Gremlin is a graph traversal machine and language developed and distributed by the Apache TinkerPop project of the Apache Software Foundation [15]. The Gremlin language is a human readable/writable graph programming language used to create Gremlin traversals (programs). Gremlin traversals are evaluated by the Gremlin traversal machine. The Gremlin traversal machine is a distributed virtual machine that can execute traversals over graphs contained on a single computer or represented across a multi-machine compute cluster. The traversal machine is agnostic to the underlying graph computing system and is supported by numerous OLTP/transactional graph databases and OLAP/batch graph processors. Gremlin is used to create and analyze directed, binary, attributed, multi-graphs known as property graphs. The expressivity of Gremlin, along with its Turing Complete [17] nature, enables it to simulate discrete quantum walks. Unlike classical walks, a quantum walk places a traverser into a superposition of both locations and “spins” across the graph [11, 9]. Quantum walk theory provides a “coin/shift”-model capable of universal quantum computing and as such, Gremlin may prove useful as a general-purpose quantum programming language when real-world quantum computers come to fruition [6].

Quantum walks on graphs have been studied extensively on lattices (one- and two-dimensional) and arguably less so on arbitrary undirected and directed graphs [11]. In quantum walk theory, the traverser’s location in the graph and its spin are represented in a geometric, complex vector space known as a Hilbert space. Due to quantum superposition, a traverser may be in multiple locations in the graph at the same time as well as have multiple spins at all such locations. When a traverser realizes multiple choices (e.g. multiple incident edges to its current vertex location), the traverser is cloned/split and its respective spin components are projected accordingly. When multiple traversers merge (e.g. multiple paths incoming to a particular vertex), all co-located traverser spins are summed via complex vector addition. Traverser spin merging can effect constructive (non-orthogonal) and destructive (orthogonal) wave interference because complex numbers are a superset of the reals (R ⊂ C) and can “rotate” around the two-dimensional Argand plane [12]. The potential for destructive interference makes the long run behavior of a quantum walk significantly different than its classical walk counterpart. In a classical walk, only constructive interference exists (known in Gremlin as “bulkling”). Moreover, it is the complexity of these wave interactions that make analytical solutions to quantum walks difficult beyond a two-dimensional lattice. Numerical simulation is currently the only feasible means to study quantum walks on complex graph structures of an arbitrary size.

Quantum researchers should note the natural way in which the Gremlin language and machine can be used to represent and execute quantum walks on
graphs at any scale.\textsuperscript{3} Graph theorists should note how quantum traverser spin and interference semantics can be leveraged when designing graph algorithms/queries. Finally, the authors note that by studying discrete quantum walks, various insights were gleaned which may be applied to future versions of the Gremlin machine architecture. Section §2 will introduce quantum walk theory using standard Gremlin constructs. Section §3 will demonstrate one-dimensional quantum walks as well as the famous two-dimensional double-split screen experiment using Gremlin. Finally, section §4 will present a collection of property graph traversal motifs that leverage quantum walk concepts and which may prove fruitful in the advancement of Gremlin.

II. INTRODUCTION TO QUANTUM WALKS

A quantum graph walk can be understood as a propagation of an undulating wave across the set of elements of a graph. Given that waves can have crests and troughs, quantum walks yield constructive and destructive interference patterns analogous to those found in natural systems such as sound and water waves. Wave dynamics are not leveraged in classical walks and thus, quantum walks differ significantly in both their representation and semantics. The quantum walk machinery formalized herein not only enables the simulation of real-world quantum systems, but it also provides a new degree of freedom called “spin” in the definition of a Gremlin traversal. This section will review the Gremlin graph traversal machine and language and discuss how quantum traversals can be expressed. For the sake of discussion and notation simplicity, all examples will only use vertex locations. That is, the examples will never assume the traverser is located at an edge or property in the property graph. However, in practice and as allowed by Gremlin, traversers can exist at any such element in the property graph. Many of the introductory concepts around complex numbers, vector spaces, Dirac notation, and quantum mechanics in general are reviewed in lucid detail by \textsuperscript{19}. Section §3 will introduce quantum walk concepts and which may prove fruitful in the advancement of Gremlin.

A. The Gremlin Traversal Machine and Language

The Gremlin traversal machine is defined by three structures: a property graph $G$, a set of traversers $T$, and a traversal $\Psi$ \textsuperscript{15}. A property graph is defined as $G = (V, E \subseteq (V \times V), \lambda: (V \cup E) \times \Sigma^* \to U \setminus (V \cup E))$, where $V$ is the set of vertices, $E$ is the set of directed binary edges, and $\lambda$ is the property function that maps an element and character string “key” to a non-relational object in the universal set $U$ (minus vertices and edges).\textsuperscript{4} For example, the name property value of vertex $v$ is “marko” and is denoted $\lambda(v, \text{name}) \to \text{marko}$. The traversers $T$ execute the instructions specified in $\Psi$ in order to effect an algorithmic walk over $G$. The result of the computation is the location of all halted traversers in $T$ and any side-effect data structures yielded during the process (e.g. a $\text{groupCount}$(), $\text{sum}$(), $\text{mean}$(), etc. types of reduction). Every traversal in $T$ is composed of $6$ properties which are discussed in detail in \textsuperscript{15}. For the presentation herein, only 4 traversal properties need to be reviewed.

1. $\mu : T \to U$: the graph location of the traverser (e.g. a vertex, edge, property, etc.).
2. $\psi : T \to \Psi$: the traversal location of the traverser (i.e. program counter).
3. $\beta : T \to \mathbb{N}^+$: the “bulk” of the traverser (i.e. its representative count).
4. $\varsigma : T \to U$: the current “sack” value of the traverser (i.e. a local mutable data structure).

Visually, a traverser $t \in T$ is a “bundle” of local variables with a projection to a location in the graph $G$ and a projection to a location in the traversal $\Psi$ and is analogous, in many ways, to the definition of a central processing unit (CPU).

$$G \leftarrow \mu \quad t \in T \quad \frac{\beta, \varsigma}{\psi} \quad \Psi$$

The Gremlin language is used by humans to create traversals. Traversals are composed of primitive functions called steps. The instruction set of the Gremlin traversal machine is called the step library. Every step is either a $\text{map}$-, $\text{flatMap}$-, $\text{filter}$-, $\text{sideEffect}$-, or $\text{branch}$-step. For a review of these functional programming constructs, please see \textsuperscript{7}. The Gremlin language supports three step composition motifs: $f \circ g \circ h \circ k$ (linear), $f(g \circ h) \circ k$ (nested), and $f(g, h) \circ k$ (parallel). The general idea is that traversers are generated at the start/leftmost step of the traversal and propagate from left to right while being modulated by each step along the way. Steps can grow ($\text{flatMap}$) or shrink ($\text{filter}$) the stream. The result(s) of the traversal are found at the end of the last/rightmost step. A collection of self-explanatory traversals are presented below to give a flavor of the Gremlin language. Later, when presenting quantum traversals, each traversal will be described in detail.

---

\textsuperscript{3} The largest publicly known Gremlin processed graph in existence is the Amazon.com order fulfillment network which is approximately 1 trillion edges at the time of this writing.

\textsuperscript{4} Properties cannot reference vertices or edges and are typically used to reference primitives such as integers, doubles, strings, etc. Also note that $U$ will later be used to denote a unitary operator. The universal set $U$ is only used in this subsection.
B. Traversal Location and µ-Superposition

A classical random walk (traversal) is composed of a single walker (traverser) moving about the graph according to the instructions (steps) in $\Psi$ and the topology of $G$, where there will never be more than one traverser throughout the course of the computation (i.e. $|T| = 1$). However, in a quantum walk, there are two types of traversers: a single “classical traverser” and set of “quantum traversers.” The classical traverser represents the initial state of the system. This state/traverser has a definite traverser will clone itself across all choices. For instance, in that whenever it meets a decision in a quantum walk, traversers must be able to constructively and destructively interfere with one another. As of the time of this writing, a traverser’s bulk is represented by a 64-bit integer in $\mathbb{N}^+$ and thus, in order to represent “bulks” with phases and amplitudes, the current bulk-construct must be abandoned (i.e. disabled) and replaced by a complex vector in the traverser’s sack called the traverser’s spin.

---

5 The notation $|x\rangle$ is known as “bra(c)ket” notation and was introduced by Paul Dirac as it conveniently denotes row ($x$) and column ($y$) vectors as well as operations such as inner products $\langle x|y \rangle$ and outer products $|y \rangle \langle x|$. When represented programmatically, these vectors and matrices are arrays of complex numbers.

6 A potentially useful visualization is that of a quantum traverser at every vertex in the graph. These traversers can be thought of as a “rubber sheet” across the vertices of the graph. As the quantum computation proceeds parts of the sheet will have more or less amplitude around 0, where there will never be more amplitude across the sheet than what was provided by the initial classical traverser (i.e. the initial perturbation). Like other natural wave systems, quantum systems respect the conservation of energy.

7 A complex number is an element in $\mathbb{C}$. Every complex number is of the form $c = a + bi$, where $c \in \mathbb{C}$, $a, b \in \mathbb{R}$, and $i = \sqrt{-1}$. The $a$ component is known as the real component and the $bi$ component is the imaginary component. Imaginary numbers “rotate” a real number about the two-dimensional Argand plane because $i^0 = 1$, $i^1 = i$, $i^2 = -1$, $i^3 = -\sqrt{-1} = -i$, and $i^4 = 1$. Thus, multiplying a complex number by $i$ rotates it 90°. Multiplying a real number by $i^2$ rotates it 180°, turning a positive real to a negative real and vice versa. There exists a bijection that takes a complex number to a polar form $(\rho, \theta)$, where the magnitude $\rho = \sqrt{a^2 + b^2}$ and the phase $\theta = \tan^{-1} \frac{b}{a}$.
C. Traverser Spin and ψ-Superposition

Quantum walks require both a location and a spin superposition to guarantee reversible, unitary evolution—a primary requirement of quantum processing. Location superposition does not contain sufficient information to ensure that a complex wave dynamic is reversible. In Gremlin, a traverser has two “locations.” A graph location in \( G (\mu) \) and traversal location in \( \Psi (\psi) \). If only the graph location is in superposition, then information about the traversal location is lost. When a traverser has multiple options in \( G \) (a topological branch), it must undergo graph location superposition. When a traverser has multiple options in \( \Psi \) (a program branch), it must undergo spin superposition. For each program branch option, there exists a complex number to represent it. For example, if the graph is a one-dimensional lattice (a line) and the traverser can either go left or right (out(‘left’, ‘right’)), then the traverser’s spin is encoded in its sack \( \psi(t) \in \mathbb{C}^2 \) which is composed of 2 complex numbers and is denoted

\[
|\psi(t)\rangle = c_1 |←\rangle + c_2 |→\rangle,
\]

where

\[
c_1 |←\rangle + c_2 |→\rangle = c_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + c_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}.
\]

The vectors \( |←\rangle \) and \( |→\rangle \) are the spin basis vectors and represent two orthogonal states that the classical traverser can be in. That is, when initialized or measured, the classical traverser will either be spinning left (\( |←\rangle \)) or spinning right (\( |→\rangle \)). However, while undergoing quantum processing, the quantum traversers can simultaneously have both a left spin and a right spin. The amount of spin in either direction is specified by \( c_1 \in \mathbb{C} \) (left) and \( c_2 \in \mathbb{C} \) (right). If there are four options in a traversal, then the traverser’s spin vector would be \( |\psi(t)\rangle \in \mathbb{C}^4 \).

A traverser’s path is completely determined by its spin. If the traverser is on a line graph and its spin is \( [1, 0]^T \), then the traverser will go left on the graph. If its spin is \( [0, 1]^T \), then the traverser will go right on the graph. If its spin is in the superposition \( \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \), then the traverser will split itself and the left clone will go left with a spin of \( \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \) and the right clone will go right with a spin of \( \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \). When two traversers exist at the same location in the graph and are within the same equivalence class \( [t] = \{ t' \in T \mid \mu(t) = \mu(t') \} \)\(^{10}\), the \( [t] \) -traversers will merge to a single traverser whose spin is determined using standard, pair-wise vector addition

\[
\zeta([t]) = \sum_{t' \in [t]} \zeta(t').
\]

It is the merging of traversers at a vertex location that yields the constructive and destructive wave interference patterns—e.g. one traverser’s spin may be positive \( (i^0) \) while another’s may be negative \( (i^2) \).

Given both location and spin superposition, the complete state of the classical traverser undergoing quantum processing is defined as the tensor product of the two superposition states

\[
|\mu(t)\rangle \otimes |\psi(t)\rangle \equiv |\mu(t) \otimes \psi(t)\rangle \equiv |\mu(t), \psi(t)\rangle,
\]

where

\[
|\mu(t), \psi(t)\rangle = c_{1,1} |v_1, ←\rangle + c_{1,2} |v_1, →\rangle + c_{2,1} |v_2, ←\rangle + c_{2,2} |v_2, →\rangle + \cdots + c_{v_1,1} |v_1, ←\rangle + c_{v_1,2} |v_1, →\rangle.
\]

A quantum system on a line graph can be described by a single complex vector in \( \mathbb{C}^{2|V|} \). In quantum mechanics, this complex vector is known as the wavefunction of the quantum system (i.e. the classical traverser in superposition). However, in Gremlin, this representation is distributed across the graph, where each quantum traverser’s location \( \mu \in V \) represents the classical traverser’s location superposition and each quantum traverser’s spin is a length 2 complex vector \( \psi \in \mathbb{C}^2 \).\(^{11}\)

It is important to note that the wavefunction does not determine the location of the classical traverser as, in this representation, the traverser is a wave, not a particle. To transform the wave representation (which is encoded across all quantum traversers) into a particle representation, the system must be measured/observed. The act of measuring first transforms the wavefunction into a probability distribution, where for any vertex \( v \), the probability of the classical traverser being at \( v \) is defined by the square of the modulus of the total spin at that vertex.\(^{12}\) For instance, if the quantum traverser \( t \)

---

8 Quantum walk theory uses the term “spin superposition” to describe the various degrees of freedom in the walker’s movement. However, in the terminology of graph traversals, a better term may be “traversal superposition,” where every branch in the program/traversal is a degree of freedom.

9 The state \( |←\rangle \) represents the column vector \( [1, 0]^T \). The state \( ⟨→| \) represents the conjugate row vector \( [1^* , 0]^T \), where for a real number, the complex conjugate is the number itself and for a complex number \( (a + bi)^* = a - bi \).

10 Traverser equivalence classes in Gremlin can be configured. For the concepts presented in this article, assume that two traversers map to the same equivalence class if their graph location is the same.

11 As of Gremlin 3.1.0, there is a tension between the traverser’s bulk and its sack. A future version of Gremlin may generalize the bulk construct to any object that can be merged, split, and has a magnitude and thus, in such a situation, the traverser’s spin would be encoded in its bulk.

12 The modulus of the vector \( x \) is its magnitude \( |x| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} \).
has $\mu(t) = v$ and $\zeta(t) = [c_1, c_2]^\top$, then the probability of
the traverser being at $v$ is the inner product

$$\langle \zeta(t)|\zeta(t) \rangle = [c_1^*, c_2^*] \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = |c_1|^2 + |c_2|^2,$$

where for all quantum traversers on the line graph at any iteration $n$,

$$\sum_{t \in T} |\zeta(t)|^2 = 1.$$

Thus, the initial classical traverser, when $n = 1$ and $|T| = 1$, has an inner product equal to 1. Quantum processes respect the conservation of wave energy. This is analogous to other natural waves such as a water wave. The initial energy placed into the water will diffuse across the surface but the total energy of the system will never increase nor decrease (barring friction – i.e. decoherence).

Once the wavefunction has been converted into a probability distribution, that distribution is sampled and the traverser is then localized to the respective sampled vertex. The wavefunction is said to collapse to some classical, basis state vertex $v \in V$ and as originally, $|T| = 1$. Thus, a classical traverser undergoes quantum processing to yield numerous quantum traversers which are then sampled at some point in the future to yield the new location/spin of the original classical traverser.

A natural (non-simulated) quantum walk is both breadth-first and depth-first at the same time. It is breadth-first because at every step, all legal incident edges are traversed. It is depth-first because only one traverser (classical particle) is actually ever observed at the end of a path.\(^\text{13}\) When simulating a quantum walk, the execution order must be breadth-first because each iteration needs to merge all co-located quantum traversers in order for wave interference to take place. In applied graph computing, this execution model is known as bulk synchronous parallel, where a single traverser takes one step in the graph and does not proceed to do another until all other traversers have completed their current step.\(^\text{15}\) The benefit of this is that breadth-first execution can be easily parallelized/distributed. However, it requires more memory than depth-first as each traverser at every step must be represented. The trade-off between a breadth-first search and a depth-first search is typically a tradeoff between time (breadth-first) and space (depth-first). Interestingly, a natural quantum walk has the benefit of both without the drawback of either granted that the location of the observed particle is the correct answer to the search.

**D. Unitary Operators**

Quantum mechanics is about reversible computing. Reversibility is a constraint that significantly limits the types of operations that can be performed on a quantum system. It requires that every operation have an inverse that returns the system to its previous state. Moreover, that inverse operation is the operation’s adjoint (i.e. complex conjugate).\(^\text{14}\) Specifically, all operations must be unitary. A unitary matrix $U$ is any matrix that satisfies the relation $U^\dagger U = I$, where $U^\dagger$ is the complex conjugate of the matrix $U$. It is important to note that the set of all $n \times n$ unitary matrices forms an algebraic group, where matrix multiplication yields a unitary matrix $(A \cdot B)$, each unitary matrix has an inverse $(A^\dagger)$, and the multiplicative identity is the unit identity matrix $I$. Furthermore, the tensor product of two unitary matrices is a unitary matrix $(A \otimes B)$.

When the quantum system’s state is altered by $U$, its inner product remains 1. That is $(U|x\rangle)\langle U'|x\rangle = \langle x|U^\dagger U|x\rangle = \langle x|I|x\rangle = \langle x|x\rangle = 1$. All unitary operations are isometric in that they preserve this distance. However, if an operation is not unitary, it can still be “quantum” in nature, but it will distort the geometry of the system. This is known as decoherence and decoherence is fundamentally a process of information loss and thus, irreversibility.

A quantum walk has two states in superposition—location and spin. There are two unitary operations that act on these states: a “coin” operator and a “shift” operator. The coin-operator transforms a traverser’s spin vector $\zeta(t)$ into a new vector which is then propagated piecewise by the shift-operator to update $\mu(t)$. In a one-dimensional line graph, every coin-operator is a $2 \times 2$ unitary matrix in $\mathbb{C}^{2 \times 2}$. Two typical $2 \times 2$ coin operators used in quantum walks are the unbalanced Hadamard coin

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

and the balanced coin

$$Y = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & i \\ i & 1 \end{bmatrix}.$$  

\(^{13}\) The de Broglie-Bohm pilot-wave interpretation of quantum mechanics states that there is only ever one particle (traverser) in a quantum process. A consequence of this line of reasoning is that there is no such thing as a particle superposition and a wavefunction collapse. The classical particle is simply guided by its own “wave” (perturbation) in some yet unknown medium.\(^\text{3}\) Thus, the wave is doing a breadth-first walk, but the particle is doing a depth-first walk.

\(^{14}\) The complex conjugate of a complex matrix is defined as

$$[a + bi \ c + di]^\dagger = [a + bi \ e + fi]\] = [a - bi \ e - fi].$$
where $H^\dagger H = I$ and $Y^\dagger Y = I$. The altered spin vector of traverser $t$ is thus $H|\zeta(t)\rangle$. Once the spin has been updated by the coin-operator, the traverser’s $\mu(t)$ location is updated by the shift-operator

$$S|\mu(t),\varsigma(t)\rangle = |\mu(t)-1,\varsigma(t)_0,0\rangle^\dagger + |\mu(t)+1,\varsigma(t)_1,0\rangle^\dagger,$$

where $\mu(t)-1$ ($\mu(t)+1$) is the vertex to the left (right) of $\mu(t)$. In other words, the left-traverser moves left one vertex and has its right spin component set to 0. Similarly, the right-traverser moves to the right one step and has its left spin component set to 0. The complete unitary operation of the entire system (not just a single quantum traverser) is defined as

$$U = S \cdot (I \otimes C),$$

where $I \in \{0,1\}^{|V|\times|V|}$ is the identity matrix. The formulation implies that the coin operator $C$ is “copied” to each vertex in the graph and the shift operator then propagates spin accordingly. However, in Gremlin, this single matrix representation is distributed across the graph where the logic of the coin- and shift-operators are evaluated by each quantum traverser. At each step, each quantum traverser evaluates $C$ and then $S$ to yield a new set of quantum traversers. Finally, because unitary matrices form an algebraic group with matrix multiplication, $U^n$ is unitary and it will iterate this process $n$-times, propagating the wavefunction $n$-steps on the graph. Thus, an $n$-step quantum walk is unitary.

III. QUANTUM WALK EXPERIMENTS

A quantum computation starts with a classical traverser in a basis state representing the initial state of the quantum process. The classical traverser is then operated on by a unitary operator. This operation puts the system into a superposition. When the system is measured, a classical traverser is yielded whose location is determined by the wavefunction of the quantum system. In the natural-world, the wavefunction can never be directly observed, only a resultant basis state is observed after measurement. However, when simulating a quantum system, the wavefunction forms the primary data structure of the computation and thus, it is subject to runtime analysis. This section will present a collection of quantum experiments on simple graph structures to demonstrate the representation of quantum walks in the Gremlin language.

A. Classical Walk on a Line Graph

The simplest walk to execute is one that takes place on a one-dimension lattice graph $G$ (a line graph) with no boundaries. To demonstrate such a walk, a line graph with $|V| = 100$ is constructed, where each vertex has one outgoing $\text{left}$-edge and one outgoing $\text{right}$-edge.

| $n/V$ | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | ... |
|-------|----|----|----|----|----|----|----|----|----|------|
| 1     | ... | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | ...  |
| 2     | ... | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | ...  |
| 3     | ... | 0  | 0  | 1  | 0  | 2  | 0  | 1  | 0  | ...  |
| 4     | ... | 0  | 1  | 0  | 3  | 0  | 3  | 0  | 1  | ...  |
| 5     | ... | 1  | 0  | 4  | 0  | 6  | 0  | 4  | 0  | 1 ... |

TABLE I: The frequency distribution of traversers at each iteration in a classical walk on a line graph.

In order to ensure that no boundaries are touched, the walk will start at $v_50 \in V$ and iterate for 50 iterations. To demonstrate the difference between a classical walk and a quantum walk, this section will first present the traversal and results of a classical walk using Gremlin. A classical random walk, where $|T| = 1$ at every iteration, can be evaluated using the $\text{sample}$-step.

```java
// repeat(out('left', 'right')).sample(1).times(50)
```

A traverser is placed on vertex 50 via $V(50)$ and then for 50-iterations, the traverser will go both left and right on the line graph. However, while two traversers are created at each iteration (one left and one right), one of the two will be filtered by $\text{sample}(1)$. This ensures that only a single traverser exists at each time step and that that traverser is a “random walker.”

```java
g.V(50).repeat(out('left', 'right')).sample(1).times(50)
```

The vertex frequency distribution can be generated by postfixing a $\text{groupCount}()$-step. This step returns a Map<Vertex, Long> denoting how many traversers are located at each vertex location.

```java
g.V(50).repeat(out('left', 'right')).times(50).
groupCount()
```

The frequency distribution of the first 5 steps of the classical walk is provided in Table I. Note that at step 3, the two 1s in step 2 split left and right and merge at vertex 50 to constructively interfere to create a count of 2. Again, in a classical walk, only constructive/additive interference ever occurs.15 These counts are normalized and presented in Table I. Note that Table I also

---

15 It is interesting to note that a traverser’s bulk is represented by a 64-bit integer. Programatically, if the bulk is larger than what a 64-bit integer can represent, then a number overflow occurs and the bulk becomes negative and thus, destructive interference occurs upon merging a non-overflow with an overflow bulk. A speculation is that the natural world is in fact premised on bulking/bulking, but due to numeric precision issues (fidelity), the wave top crests and becomes negative. In fact, similar re-
Gremlin uses frequentist “bulking” when merging traversers in the tum walk traversal, three functions are defined. First, the Hadamard coin, 
\[
\text{hadamard} = \{ a, b \to [ (1/\text{Math.sqrt}(2)) \ast (a[0] + a[1]), \\
(1/\text{Math.sqrt}(2)) \ast (a[0] - a[1]) ] \} .
\]
Lastly, the shift function forces a traverser’s spin vector into a basis state by either setting the left- or right-spin component to 0, where \( b = [1,0] \) will project the left-spin component and \( b = [0,1] \) will project the right-spin component.
\[
\text{shift} = \{ a, b \to [ a[0] \ast b[0], a[1] \ast b[1] ] \} .
\]
A quantum walk is started with a classical traverser located at vertex 50 with a spin in the basis state \([1,0]^{\top}\). In Dirac notation, this is the state \( |_{50}, + \rangle \).^{18}
\[
g.\text{withSack}([1,0], \text{merge}).V(50).
\]
\[
\text{repeat}(
\text{g sack(hadamard)} .
\text{union(}
\text{sack(shift).by(constant([1,0])).out('left'),}
\text{sack(shift).by(constant([0,1])).out('right'))}).
\text{times}(50)
\]
The withSack([1,0], merge) parameterization defines the initial state of the traverser’s sack along with the merge function to use when two traversers meet at the same vertex location. The traverser is then placed at vertex 50 via V(50). Finally, 50 iterations are executed, where for each iteration, all traverser sacks are evolved by the unitary hadamard function and then two traversers are created each with the original traverser’s left (right) sack component zereed out by the shift function. These traverser children are then propagated left or right on the line graph accordingly.

In order to yield the probability that the classical traverser will be at some vertex in \( V \), the wavefunction (quantum traversers) must be turned into a probability distribution. This is accomplished by grouping and normalizing the sack of each traverser. If
\[
\text{norm} = \{ \text{sack} \to \text{Math.pow(sack.get()[0],2) + Math.pow(sack.get()[1],2)} \}
\]
then the probability distribution is computed using
\[
g.\text{withSack}([1,0], \text{merge}).V(50).
\]
\[
\text{repeat(}
\text{g sack(hadamard)} .
\text{union(}
\text{sack(shift).by(constant([1,0])).out('left'),}
\text{sack(shift).by(constant([0,1])).out('right'))}).
\text{times}(50).\text{group()}.by().by(sack().map(norm))
\]

\[\begin{array}{cccccccc}
 n/V & 46 & 47 & 48 & 49 & 50 & 51 & 52 & 53 & 54 & \ldots \\
 1 & \ldots & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \ldots \\
 2 & \ldots & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & \ldots \\
 3 & \ldots & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \ldots \\
 4 & \ldots & \frac{1}{8} & 0 & \frac{3}{8} & 0 & \frac{3}{8} & 0 & \frac{3}{8} & 0 & \ldots \\
 5 & \ldots & \frac{1}{16} & 0 & \frac{6}{16} & 0 & \frac{6}{16} & 0 & \frac{6}{16} & 0 & \ldots \\
 \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 50 & \ldots & 0.096 & 0 & 0.108 & 0 & 0.112 & 0 & 0.108 & 0 & 0.096 & \ldots \\
\end{array}\]

TABLE II: The probability distribution of a traverser’s vertex location at each iteration in a classical random walk.

shows the probability distribution at iteration 50. At iteration 50, the number of traversers at vertex 50 is 126,410,606,437,752 (≈ 126 trillion traversers) and the total number of represented traversers across the whole graph is 1,125,899,906,842,624 (≈ 1 quadrillion traversers).^{16}

B. Unbalanced Quantum Walk on a Line Graph

A classical walk can be compactly described in Gremlin. On the other hand, a quantum walk in Gremlin is more complex given that it requires that each step modulate the traverser’s spin and projects that spin component-wise to the adjacent left and right vertices accordingly. This complexity exists because spin superposition, unlike location superposition, is not natively supported in Gremlin. Before presenting the Gremlin quantum walk traversal, three functions are defined. First, the merge function is responsible for constructive and destructive wave interference. It merges the sacks (spins) of any two traversers at the same vertex location via pairwise vector addition.
\[
\text{merge} = \{ a, b \to [a[0] + b[0], a[1] + b[1]] \} .
\]
Second, the Hadamard coin,
\[
H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} ,
\]
is computed with the hadamard function, where the b-argument is null.\(^{17}\)}

\[\begin{array}{cccccccc}
 n/V & 46 & 47 & 48 & 49 & 50 & 51 & 52 & 53 & 54 & \ldots \\
 1 & \ldots & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \ldots \\
 2 & \ldots & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & \ldots \\
 3 & \ldots & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \frac{1}{4} & 0 & \ldots \\
 4 & \ldots & \frac{1}{8} & 0 & \frac{3}{8} & 0 & \frac{3}{8} & 0 & \frac{3}{8} & 0 & \ldots \\
 5 & \ldots & \frac{1}{16} & 0 & \frac{6}{16} & 0 & \frac{6}{16} & 0 & \frac{6}{16} & 0 & \ldots \\
 \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 50 & \ldots & 0.096 & 0 & 0.108 & 0 & 0.112 & 0 & 0.108 & 0 & 0.096 & \ldots \\
\end{array}\]

sults using overflow on bounded precision bulks (16-bit, 32-bit, etc.) can effect results similar to the quantum experiments to follow.\(^{16}\)

Gremlin uses frequentist “bulking” when merging traversers in the same equivalence class \([t]\). Thus, while 1 quadrillion traversers are represented, they are not individually enumerated.

\(^{17}\) All sack-steps take a binary function. However, if the second argument is not required because the function itself contains the full logic for mutating the first argument, then it can be safely ignored.

\(^{18}\) In Gremlin 3.1.0, a withBulk(false) parameterization is required after g to tell the Gremlin machine to use a traverser’s sack (not its bulk) as the determinant of the traverser’s magnitude/count. This parameterization is left out of the presented Gremlin quantum examples. As stated previously, a future version of Gremlin may generalize the bulk construct to be anything that can be “split” and “merged” and thus, at that point, the traverser’s spin will be represented in the bulk of the traverser, not its sack.
The group-step takes two by-modulators. The first is the group “key” where by() is the identity and thus, the vertex location. The second is the modulus squared of the spin at that vertex. The probability distribution of the traverser’s location for the first five and last iteration is presented in Table III. Note that at step 4, unlike a classical walk, destructive interference at other vertices increases the probability of locating the traverser at vertex 49. The traverser moving right is 180° out of phase with the traverser moving left.\textsuperscript{19}

The probability distribution at iteration 50 for both classical and quantum Hadamard walks is diagrammed in Figure 1. Given that the Hadamard is a biased coin, the traverser’s location is biased to the left. Moreover, the traverser is less likely to be located at the center of the line which is contrary to what is expected from the classical walk. For a classical random walk, after \( n \)-steps, the traverser will most likely be at a vertex \( \sqrt{n} \)-step away from \( v_50 \). In a quantum walk, the traverser will most likely be at a vertex \( n \)-steps away from \( v_50 \). This feature makes quantum walks interesting as an algorithmic technique as, with the correct coin and initial state, quantum walks can be self-avoiding in search of novel, less reverberant/noisy areas of the graph \[2\].

The previous traversal generated a probability distribution from the wavefunction amplitudes. However, the final step is to ultimately collapse the wavefunction to a basis state. This is accomplished by sampling the probability distribution. Thus, the full quantum traversal, from classical start state to classical end state is below.

\[
g.withSack([1,0],merge).V(50).
\]

\[
\text{repeat(}
\quad \text{sack(hadamard).}
\quad \text{union(}
\quad \quad \text{sack(shift).by(constant([1,0])).out('left'),}
\quad \quad \text{sack(shift).by(constant([0,1])).out('right'))}.
\]

\[
\text{times(50).}
\]

\[
g.withSack([1,0],merge).V(50).
\]

\[
\text{repeat(}
\quad \text{sack(hadamard).}
\quad \text{union(}
\quad \quad \text{sack(shift).by(constant([1,0])).out('left'),}
\quad \quad \text{sack(shift).by(constant([0,1])).out('right'))}.
\]

\[
\text{times(50).}
\]

\[
\text{probDist(50).}
\]

\[
\text{The last line of the traversal samples the group-step probability distribution. It unfolds the key/values pairs of the Map<Vertex,Double> and samples 1 key/value pair where the probability distribution is over the set of all key/value pair values. Given that the basis state is a vertex and not a probability, the key of the sampled key/value pair is projected with select(keys). In review, the first line of the traversal creates the classical initial state (naturally observable), the repeat(...).times(50) lines are the evolution of the quantum state (naturally unobservable), and the last two lines represent the wavefunction collapse (unobservable) which ultimately yields a single classical traverser at a vertex (observable).}

Finally, it is important to note that the traversal, prior to wavefunction collapse, is unitary. This can be demonstrated by iterating for 50 steps and then iterating the inverse 50 steps. After doing so, vertex 50 will have a single traverser with spin \([1,0]^{\top}\) and probability \(|1|^2 + |0|^2 = 1.0\).\textsuperscript{20} That is, the classical traverser can be reconstructed by running the quantum process in reverse.

\[
\text{g.withSack([1,0],merge).V(50).}
\]

\[
\text{repeat(}
\quad \text{sack(hadamard).}
\quad \text{union(}
\quad \quad \text{sack(shift).by(constant([1,0])).out('left'),}
\quad \quad \text{sack(shift).by(constant([0,1])).out('right'))}.
\]

\[
\text{times(50).}
\]

\[
\text{probDist(50).}
\]

19 The complex number 1 + 0i (1) is 180° out of phase with 0 + 1^2 (-1). The polar form \((ρ, θ)\) of these two complex numbers is \((1, 0)\) and \((1, 180°)\), respectively.

20 In practice, a complex number is represented by two 64-bit floating point numbers. Numerous operations on such numbers typically incur floating point errors and thus, a perfect 1.0 reconstruction of the classical traverser’s spin is unlikely. Usually, results are of the form 0.99999999.\}

\[
\begin{array}{cccccccccccccccc}
\hline
n/V & 46 & 47 & 48 & 49 & 50 & 51 & 52 & 53 & 54 & \ldots \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\
2 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & \ldots \\
3 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & \ldots \\
4 & 0 & \frac{1}{8} & 0 & \frac{1}{8} & 0 & \frac{1}{8} & 0 & \frac{1}{8} & 0 & \ldots \\
5 & 0 & \frac{1}{16} & 0 & \frac{1}{16} & 0 & \frac{1}{16} & 0 & \frac{1}{16} & 0 & \ldots \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
50 & 0.015 & 0 & 0.014 & 0 & 0.013 & 0 & 0.012 & 0 & 0.011 & \ldots \\
\hline
\end{array}
\]

TABLE III: The probability distribution of the traverser’s vertex location at each iteration using an initial spin of \([1,0]^{\top}\) and the \(H\) Hadamard coin.
repeat(
    union(
        sack(shift).by(constant([1,0])).in('left'),
        sack(shift).by(constant([0,1])).in('right')),
    sack(hadamard)).
times(50)

C. Balanced Quantum Walk on a Line Graph

The Hadamard coin \(H\) is useful as an introductory example because it is a coin in \(\mathbb{R}^{2 \times 2}\). However, \(\mathbb{R} \subset \mathbb{C}\) and thus, for arbitrary complex coins and spins, the previous quantum walk traversal must be generalized to support complex numbers. The Apache Commons Math3 library’s \org.apache.commons.math3.complex.Complex\ can be imported into Gremlin using the \:\texttt{import} command.\footnote{The Apache Commons Math3 \texttt{Complex} number is encoded as a two dimensional \texttt{double} array where the first component of the array is the real component (\(a\)) and the second is the real number in the imaginary component (\(b\)), where \(a + bi\). Thus \(i^2 = (1,0)\), \(i^3 = (0,1)\), and \(i^4 = (1,0)\), etc.}

From there, if
\[
\text{zero} = \text{Complex.valueOf}(0,0)
\]
\[
\text{one} = \text{Complex.valueOf}(1,0)
\]
\[
i = \text{Complex.valueOf}(0,1)
\]

then the \(2 \times 2\) balanced unitary coin
\[
Y = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & i \\ i & 1 \end{bmatrix}
\]
is denoted
\[
\text{balanced} = \{ a,b \rightarrow \\
[(i/\text{Math.sqrt(2)}) \ast a[0].add(i.mult(a[1])), \\
(i/\text{Math.sqrt(2)}) \ast i.mult(a[0]).add(a[1]) \}.
\]

Furthermore, the \texttt{merge} function must be redefined to support pairwise complex vector addition.\footnote{The binary operators + and * can only be used with standard Java primitives like \texttt{long}, \texttt{double}, etc. To add and multiply a \texttt{Complex} number, the respective \texttt{Complex} methods must be used.}

\[
\text{merge} = \{ a,b \rightarrow [a[0].add(b[0]),a[1].add(b[1])] \}
\]

The balanced coin \(Y\) uses the imaginary number \(i\) to rotate the traverser’s spin by \(90^\circ\) on each iteration. The following Gremlin traversal leverages complex numbers and the balanced \(Y\)-coin.

\[
g.withSack([\text{one},\text{zero}],\text{merge}).V(50).
\]
\[
\text{repeat(}
\quad \text{sack(\text{balanced}).}
\quad \text{union(}
\quad\quad \text{sack(shift).by(constant([\text{one},\text{zero}])).out('left'),}
\quad\quad \text{sack(shift).by(constant([\text{zero},\text{one}])).out('right'))}.times(50)
\\text{)}\]

\[
\]

\begin{table}
\begin{tabular}{lrrrrrrrrrrr}
\hline
n/V & 46 & 47 & 48 & 49 & 50 & 51 & 52 & 53 & 54 & \ldots \\
\hline
1 & \ldots & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \ldots \\
2 & \ldots & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & \ldots \\
3 & \ldots & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & \ldots \\
4 & \ldots & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & \frac{1}{2} & \ldots \\
5 & \ldots & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & \frac{1}{2} & \ldots \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
50 & 0.013 & 0 & 0.013 & 0 & 0.013 & 0 & 0.013 & 0 & 0.013 & \ldots
\hline
\end{tabular}
\end{table}

\textbf{TABLE IV}: The probability distribution of the traverser’s vertex location at each iteration using an initial spin of \([1,0]^\top\) and the balanced \(Y\)-coin.

The probability distribution generated by the \(Y\)-traversal is diagramed in Figure 2 and the evolution of the probability distribution is provided in Table IV. The \(Y\)-coin is balanced and thus, both the left and right components of the distribution centered at vertex 50 are identical. However, like \(H\), the probability of locating the traverser at vertex 50 is minimal compared to the vertices on the line’s ends.

\begin{figure}
\centering
\includegraphics[width=0.4\textwidth]{figure2.png}
\caption{The probability of locating a classical walker (gray dashed line) and an unbiased \(Y\)-coin quantum walker (black solid line) at a particular vertex on a 100 vertex line graph after 50 steps.}
\end{figure}

D. Quantum Walk on a Line Graph with Boundaries

The previous simulations were run with an initial traverser placed at the middle of the line and using \(|V|/2\) number of iterations (e.g. 50 steps for a 100 vertex line graph). This ensured that the traversers did not try and move beyond the boundary vertices \(v_1\) and \(v_{|V|}\). However, it is rarely the case in real-world situations that the quantum process will be conveniently confined to an unbounded space. In the case where a traverser with spin \([c_1,c_2]^\top\) reaches \(v_1\), the unitary operator for the next
step generates two traversers where the right-traverser is placed at \( v_2 \) with spin \( [0, c_2] \top \) (typical) and the left-traverser remains at vertex \( v_1 \) with spin \( [0, c_1] \top \) (atypical). In words, the left traverser remains at \( v_1 \) with its left-spin component now being its right-spin component (i.e. reflection). If

\[
\text{reflect} = \{ a, b \mapsto [a[1], a[0]] \}
\]

then a unitary reverberant quantum walk using \( H \) for 100 iterations is defined in Gremlin as

\[
g \text{.withSack}([1, 0], \text{sackSum}).\text{V}(50).\text{repeat}(\text{sack(hadamard)}.\text{union}(\text{choose(out('left').count().is(gt(0)), sack(shift).by(constant([1, 0])).out('left'), sack(shift).by(constant([1, 0])).sack(\text{reflect})), \text{choose(out('right').count().is(gt(0)), sack(shift).by(constant([0, 1])).out('right'), sack(shift).by(constant([0, 1])).sack(\text{reflect}))})).\text{times}(100).
\]

When there are only two options in a choose-step, the meaning of choose is “if-then-else.” If there is a left (right) vertex, then the traverser is cloned and shifts as previous. Else, if no such vertex exists (i.e. a boundary is reached), then the traverser’s spin is shifted and then reflected. There are four branching options in the boundary-aware traversal above – two unions each with two chooses \( (2 \times 2 = 4) \). However, because the choose-step branches are selective and based on the state of the quantum traverser, only one branch will ever be taken for each quantum traverser. Thus, the sack still only needs to be in \( \mathbb{C}^2 \). The probability distribution generated from the traversal above is plotted in Figure 3.

![Probability Distribution](image)

FIG. 3: The probability of locating a Hadamard-coin quantum walker with boundary support at a particular vertex on a 100 vertex line graph after 100 iterations.

The reflection operation remains unitary. The original §2 specification of the unitary operator of a quantum walk was \( U = S \cdot (C \otimes I) \), where the coin \( C \) is “copied” to each vertex via the tensor product of the identity matrix \( I \). It is not necessary that the same coin \( C \) be used at each vertex in \( V \). In fact, as long as the coin used is unitary, there can be a heterogenous set of coins used in a quantum walk \([8]\). This is analogous to how quantum circuits work. The graph is a collection of unitary quantum gates (vertices) connected to one another via quantum wires (edges). Each gate can perform a different unitary operation on the qubit (walker). There are two points to be made. First, the reflect operation is simply a different unitary operation being used at the boundary vertices. Second, quantum graph walks are sufficiently expressive for universal quantum computing \([4]\) and thus, Gremlin can serve as a general purpose quantum programming language.

E. A Quantum Walk on a Two-Dimensional Lattice with Two Slits

In 1802, Thomas Young published the results of a light experiment known today as the “double-slit experiment” \([20]\). In this experiment, a light source emits light at a screen. The screen has two slits in it. The light that makes it through the slits ultimately ends up being registered by a light sensitive film. The results of this experiment demonstrated that light behaves as a wave because the film showed interference patterns typical of wave mechanics. However, in 1932, Sir Geoffrey Ingram Taylor repeated the experiment, where instead of the light source emitting a constant stream of light, it emitted a single quanta of light known as a photon (a “feeble” amount of light) \([16]\). Unexpectedly, the same interference pattern emerged in the long run – a single photon can interfere with itself! From the work of both Young and Taylor, light is now understood as being both a particle (at emission from the light source and absorption by the film) and a wave (while in quantum superposition between the start and end states). In other words, light is a particle when in a basis state and is a wave when in a superposition of the basis states.

The double-slit experiment can be repeated using Gremlin. It requires a “space” (a two-dimensional lattice), a slit screen (vertices with and without edges), and a light sensitive film (the back row vertices on the lattice) \([13, 14]\). Figure 4 visualizes the two-dimensional lattice, where the bottom black vertex is the initial location of the classical traverser (the light source), the four vertices with edges at the 10\(^{th}\) and 11\(^{th}\) rows are the two slits, and the 20 vertices at the top of the lattice represent the film.

Given that a two dimensional lattice has 4 directions (left, right, up, and down) and barrier conditions, the coin must be a \( 4 \times 4 \) unitary operator that supports reflection. The balanced Grover coin is a \( 4 \times 4 \) unitary operator that will rotate the traverser \( 180^\circ \) \((i^2 = -1)\) in its current trajectory/spin and only \( \frac{1}{2}\)-scale it in the
FIG. 4: The two-dimensional lattice used to simulate the double-slit screen experiment. The black bottom center vertex is provided a single classical traverser in the basis state \([0, 0, 1, 0]^\top\) (i.e. spin up). Only four vertices in the 10th and 11th rows of the lattice have edges and they represent the two 2-vertex width slits in the screen. Finally, the dark gray vertices at the top of the lattice represent the light sensitive film which will ultimately measure the probability distribution.

A two-dimensional lattice walk with the Grover coin \(R\) is expressed below where vertex 10 is the bottom center vertex (dark black vertex in Figure 4).

\[
g.\text{withSack}([0, 0, 1, 0], \text{merge}).V(10).\text{repeat}(
  \text{sack}(\text{grover}).
  \text{union}(
    \text{choose(out('left').count().is(gt(0)),}
    \text{sack(shift).by(constant([1, 0, 0, 0])).out('left'),}
    \text{sack(shift).by(constant([1, 0, 0, 0])).out('right'),}
    \text{sack(shift).by(constant([1, 0, 0, 0])).out('up'),}
    \text{sack(shift).by(constant([1, 0, 0, 0])).out('down'),}
    \text{sack(reflect).by(constant('ud'))}),
    \text{choose(out('left').count().is(gt(0)),}
    \text{sack(shift).by(constant([1, 0, 0, 0])).out('left'),}
    \text{sack(shift).by(constant([1, 0, 0, 0])).out('right'),}
    \text{sack(shift).by(constant([1, 0, 0, 0])).out('up'),}
    \text{sack(shift).by(constant([1, 0, 0, 0])).out('down'),}
    \text{sack(reflect).by(constant('ud'))}),
  ),
  \text{times}(26)
)\]

Figure 5 shows the probability distribution on the top 20 lattice vertices (the light sensitive film) after 26 iterations. As expected, the waves emanating from the two slits interfere with each other yielding a constructive peak at the center vertex with diminishing intensity towards the ends.

There are 2 types of boundary conditions in the double slit screen experiment. The sides of the lattice and the mid-screen barrier. However, because of the \text{reflect} function, the barriers reverberate the wavefunction back while still preserving the unitary nature of the quantum process. Figure 6 shows the probability distribution at iteration 26, where lighter gray vertices (squares) have a higher probability than the darker gray vertices. The white vertices on the top left and right have no probability and denote the wave front boundary.
Finally, it is important to emphasize that this experiment’s results vary depending on the coin used, the size of the lattice, the depth and width of the slits, as well as the number of iterations allowed until measurement \[14\].

IV. QUANTUM SEMANTICS IN CLASSICAL TRAVERSALS

All quantum walk algorithms share the same common description.

1. Put a single (classical) traverser into a position basis and spin basis.
2. Alter the traverser’s spin state using a unitary operation which conserves the total spin of the system.
3. Given the new spin state, create (quantum) traversers at the respective adjacent vertices with shifted spin.
4. If two (quantum) traversers meet at the same vertex, sum their spin states using pairwise vector addition and make one (quantum) traverser.
5. Repeat steps 2 through 4 for each (quantum) traverser for some number of iterations.
6. Generate a probability distribution based on the sum of the absolute squares of all (quantum) traversers across the graph.
7. Sample that probability distribution and yield a single (classical) traverser in a position and spin basis.

The distinction between a “classical traverser” and a “quantum traverser” is simply that the classical traverser is the initial and end state of the system when \(|T| = 1\) and it is in a basis state (i.e. at a particular vertex with a spin completely in one direction).\[23\] This is a superficial distinction and as such, there is nothing fundamentally different about the two types of traversers in Gremlin. Furthermore, the requirement for wavefunction collapse is simply a requirement of the natural world and not something inherent to simulated quantum walks. By removing probabilistic sampling (steps 6 and 7), a quantum traversal is equivalent to a classical Gremlin traversal save for the notion of “spin.” The concept of spin can be leveraged in Gremlin (irrespective of “quantum processing.”). When spin is used to encode branch frequency (and not branch amplitude), then the quantum concepts presented thus far are useful in “classical” Gremlin traversals.

A. Intersection and Symmetric Difference with Traverser Spin

The branches of a traversal determine the number of components in a traverser’s spin. On a line graph, there are two options that are union-d together – out('left') and out('right'). Thus, a two dimensional spin array is required. The first number in the array is the amount of spin-left and the second, the amount of spin-right. For the lattice example, a four dimensional array was required. Each dimension represented the amount of left-, right-, up-, and down-spin in the traverser. In general, the number of spin dimensions required is equal to the number of options or degrees of freedom in the traversal.

When the spin of a traverser is understood as its superposition in \(\Psi\), then the topology of the graph and the traversal can be decoupled. In all the quantum experiments presented thus far, the traversal topology mirrored the graph topology. If the vertices had an outgoing left-edge, then the traversal had a respective out('left')-step. In real-world property graphs, the graph structure is complex with different traversals identifying different features of the graph. For instance, the graph diagrammed in Figure 7 can contain read-, wrote-, and liked-edges throughout, but if a particular traversal is trying to identify the most central person in the implicit co-authorship graph, then the traversal will only contain out('wrote') and in('wrote') branches. Thus, the traverser is constrained to a particular subgraph of \(G\) and that constraint is expressed in the branches of its traversal.

Next, unitary evolution is necessary in the natural world as the total spin of the system must be conserved. In the world of graph computing, this constraint is not required and in fact, if unitary evolution is abandoned,

---

\[23\] A nice mental distinction to make is that when \(|T| = 1\) there exists a “particle” (classical) and when \(|T| > 1\), there exists a “wave” (quantum).
then traverser spin “bookkeeping” denotes how many traversers were generated by each branch of the traversal. With this information it is possible to implement intersection and symmetric difference using traverser spin. To do this, three methods are defined. The methods **merge** and **shift** accomplish pair-wise vector addition and multiplication of an arbitrary dimension, respectively. The method **split** is analogous to the coins defined for quantum walks, but instead of conserving spin (unitary), it sums the vector and places that sum in every dimension. The understanding here is that if the traverser is in $[1,0,0]^T$ spin prior to a three-branch traversal, then when it goes into spin superposition and thus, down all three branches, the traverser’s spin/bulk is $[1,1,1]^T$.

$$merge = \{a, b \to x = []\};$$

$$0..a.size()-1).each\{ i \to x << a[i] + b[i] \}; x \}$$

$$shift = \{a, b \to x = []\};$$

$$0..a.size()-1).each\{ i \to x << a[i] * b[i] \}; x \}$$

$$split = \{a, b \to x = []\};$$

$$a.each\{ x << a.sum() \}; x \}$$

In the following traversal, a traverser is placed at vertex 0 of the graph diagrammed in Figure 7. That traverser’s spin is then **split** to be $[1,1,1]^T$. Semantically, this means that the traverser has gone into a $\psi$-superposition as it will take both branches of the **union**-step. Each branch will **shift** the spin of the traverser to zero out the component of the opposing branch (i.e. a projection). The **barrier**-step aggregates all the traversers and merges those traversers at the same location (as defined by the equivalence class $[t]$) into a single traverser via pair-wise vector/array addition. Finally, the **map**-step serves as a view into the state of each traverser where the **t.get()** is the vertex location of the traverser and the **t.sack()** is its respective sack (i.e. spin).

```plaintext
g.withSack([1,0],merge).V(0).
   sack(split).
   union(  
      sack(shift).by(constant([1,0])).out('read'),
      sack(shift).by(constant([0,1])).out('wrote')).
   barrier().map{t -> [t.get(), t.sack()]}  
```

The output of this traversal over the toy graph in Figure 7 is provided below.

```plaintext
==>[v[1], [1, 1]]
==>[v[2], [1, 0]]
==>[v[3], [2, 0]]
```

The spin state of the merged traverser provides enough information to determine intersection (i.e. and) and symmetric difference (i.e exclusive or), where

$$intersect(t) = \begin{cases}  
true & : \forall i \in \zeta(t) \ i > 0 \\
false & : \text{otherwise}  
\end{cases}$$

and

$$symDiff(t) = \begin{cases}  
true & : \exists i \in \zeta(t) \ i = \sum_j \zeta(t)_j \\
false & : \text{otherwise}  
\end{cases}$$

These predicates are expressed using a **filter**-step. For intersection,

```plaintext
g.withSack([1,0],merge).V(0).
   sack(shift).by(constant([1,0])).out('read'),
   sack(shift).by(constant([0,1])).out('wrote')).
   barrier().filter(notsack().unfold().is(eq(0))).
   map{t -> [t.get(), t.sack()]}  
```

The **filter** only allows those traversers to pass that don’t have a sack component equal to 0. Given that only vertex 1 can be reached by both **read**- and **wrote**-edges, then the intersection of these two branches filters out vertex 2 and vertex 3.

```plaintext
==>[v[1], [1, 1]]
```

For symmetric difference, the **filter** is different as it must ensure that one and only one sack component is greater than 0.

```plaintext
g.withSack([1,0],merge).V(0).
   sack(shift).by(constant([1,0])).out('read'),
   sack(shift).by(constant([0,1])).out('wrote')).
   barrier().filter(notsack().unfold().is(eq(0))).
   map{t -> [t.get(), t.sack()]}  
```

The result of the symmetric difference is below.

```plaintext
==>[v[2], [1, 0]]
==>[v[3], [2, 0]]
```

Note that the number of traversers generated by each branch is contained in the respective spin component. For instance, there are two **read**-edges from vertex 0 to vertex 3 and zero **wrote**-edges from vertex 0. Thus, the traverser located at vertex 3 has a spin state of $[2,0]^T$. Given that the number of traversers generated by each branch is split amongst the spin component, a “measurement” of the spin state of the traverser at its respective vertex location will collapse the spin to a frequency-based basis state.

```plaintext
norm = \{a, b \to x = []\};
x[0] = a.sum(); (0..a.size()-2).each(x << 0); x }
```

```plaintext
g.withSack([1,0],merge).V(0).
   sack(shift).by(constant([1,0])).out('read'),
   sack(shift).by(constant([0,1])).out('wrote')).
   barrier().filter(notsack().unfold().is(eq(0))).
   map{t -> [t.get(), t.sack()]}  
```

In this context, the Gremlin **filter**-step acts in a similar fashion as a **polarized screen** to light. A polarized screen only allows light through that is at a particular spin.
The result of the above intersection-traversal is below. The number of traversers after branching all fold to the first component/branch as the traversal is now back to a linear (non-branching) form.

\[ v[1], [2, 0] \]

The above example had two branches (“read” and “wrote”) and thus, a length 2 spin vector. In the traversal below, there are three branches and thus, a length 3 spin vector is used.

\[ g.withSack([1,0,0],merge).V(0).\]
\[ \text{ sack(split).} \]
\[ \text{ union(} \]
\[ \text{ sack(shift).by(constant([1,0,0])).out('read'),} \]
\[ \text{ sack(shift).by(constant([0,1,0])).out('wrote'),} \]
\[ \text{ sack(shift).by(constant([0,0,1])).out('liked')).} \]
\[ \text{ barrier().map{t -> [t.get(),t.sack()]} \]

The result of the traversal above is below. Using the appropriate filter, intersection or symmetric difference can be effected.

\[ v[1], [1, 1, 1] \]
\[ v[2], [1, 0, 1] \]
\[ v[3], [2, 0, 1] \]

The significance of this model is that there is no side-effect data structure that aggregates all the traversers in each branch and then does an intersection/difference of those aggregations. The traversal is fully functional as the branch statistics are encoded on the traverser’s spin and thus, the “aggregation” is distributed across the traversers and therefore, across the traversal flow.

B. Distributing Global Data Structures Across Local Traverser Spin

The ability to abandon side-effect “bookkeeping” data structures is important because it enables steps to be purely functional (stateless). When steps are stateless, then they can more easily be executed in both a threaded and machine distributed manner. This is perhaps made more salient in the following example where an explicit side-effect can be abandoned in favor of a traverser spin representation.

A common pattern in graph traversing is to determine if a vertex has already been touched by a previous step in the traversal. For instance: “Who are my friends’ friends that are not my friends?” To answer this question, it is important to know all of the person’s friends, then, for each of those friends, determine their friends while excluding those friends-of-a-friend that are not the original person’s friend. This is currently accomplished in Gremlin by using the aggregate-step which generates a side-effect data structure. In the example, this data structure is referenced by the variable \( x \).

\[ g.V().out('knows').aggregate('x').\]
\[ \text{ out('knows').where(not(within('x'))) \]

The aggregate(‘x’)-step is a barrier in that it blocks until all traversers prior to it have passed through it. These traverser’s locations (i.e. friends) are stored in the set \( x \). Once all the traverser locations have been aggregated, the barrier is “drained” one traverser at a time by the out(‘knows’) -step which computes the friends-of-a-friend.\(^{25}\) The where-step filters out all those traversers that are at the same vertex location as any of those stored in \( x \). This is not a purely functional operation. The problem is that \( x \) is a “global blackboard” that is accessed by all traversers. This limits what can be done in a distributed environment as the \( x \) data structure is external to the traversal flow. However, the previous spin-based technique can be used to implicitly store \( x \) in the traversal flow. The above non-functional traversal can be rewritten in a purely functional way. The final filter ensures that no merged traverser went down the first “identity” branch (i.e. \( x \)).

\[ g.withSack([1,0],merge).V(0).out('knows').\]
\[ \text{ sack(split).} \]
\[ \text{ union(} \]
\[ \text{ sack(shift).by(constant([1,0])),} \]
\[ \text{ sack(shift).by(constant([0,1])),out('knows')).} \]
\[ \text{ barrier().} \]
\[ \text{ filter(sack().unfold().range(0,1).is(eq(0)))} \]

The interesting aspect of this traversal is that there are in fact two branches to union. However, the first branch stays on the friend vertex and simply shifts the traverser’s spin while the other branch shifts the spin and then moves to the friends-of-a-friend via the out(‘knows’) -step. While not necessary, the first branch could have ended with an identity-step to make this “stall” more apparent.

C. Applying Quantum Concepts to Classical Gremlin

The Gremlin compiler makes use of traversal strategies to introspect a traversal prior to its execution. The purpose of these strategies is to rewrite particular sequences of steps into a more optimal representation. For instance, out().count().is(eq(5)) is rewritten to out().limit(6).count().is(eq(5)) by the RangeByIsCountStrategy. It is possible to use such strategies to dynamically introduce spin into the traverser’s sack definition. This would enable the user to simply write

\[ g.V().intersect(out('read'),out('wrote')) \]

and have the compiled version use the aforementioned spin representation. The user would not have to be concerned with sack splitting, shifting, and filtering. In fact, \(^{25}\)

\(^{25}\) A barrier-step, like aggregate and barrier, ensures a breadth-first execution up to that point in the traversal. Gremlin’s architecture supports traversals that go from breadth- to depth-first over the course of the computation.
the concept of spin is alien to the user and only serves as a functional optimization. Next, suppose the following abstract traversal, where X and Y are standard functional step sequences.

\[
g.X.intersect(out('read'), out('wrote')).Y
\]

In this situation, Gremlin would use classical constructs in the X-sequence, quantum constructs in the intersect-sequence, and then “collapse” the system back to a single spin for the Y-sequence. In many ways, this is how quantum computers and classical computers are expected to interact. An algorithm will have classical and quantum components. The classical aspects compute up to the quantum part at which point, the classical data is sent to the “quantum chip” for the quantum part of the algorithm to execute. When the quantum component is complete, the wavefunction is collapsed and the resultant basis state is fed back into the classical part of the algorithm. Gremlin can follow this model, save that a compiler strategy (not a user) would be responsible for breaking up the traversal into classical and quantum components for their execution using “bulks” (classical) or “spins” (quantum), respectively.

V. CONCLUSION

Gremlin is a graph traversal machine and language that can evaluate traversals on graphs represented on a single computer or across an arbitrarily large compute cluster. It can be used to execute any known algorithm and thus, is Turing Complete. Furthermore, Gremlin maintains a step library (instruction set) encompassing numerous graph traversal primitives. The expressivity of its step library enables it to conveniently represent and execute quantum walks. In order to do this, Gremlin traverser “sacks” are endowed with a complex vector. These sacks undergo unitary evolution and in the process, yield constructive and destructive interference as the unitary operation “rotates” them and merges them at vertex locations in the graph. Collapsing the wavefunction to a probability distribution and ultimate classical basis state is also conveniently expressed in Gremlin. The concept of quantum “spin” has been demonstrated to be useful outside of pure quantum simulation, where unitary amplitude evolution is abandoned for a frequentist evolution. The use of spin allows typical non-functional Gremlin constructs to be represented in a purely functional way. Future work in this area will explore other aspects of Gremlin that can take advantage of quantum concepts as well as make it easier for quantum computing practitioners to leverage Gremlin for quantum simulation and potentially, as a general purpose quantum programming language.

Acknowledgments

The authors would like to thank the Apache Software Foundation for their commitment to open source software and specifically for their support of the Apache TinkerPop project. TinkerPop has been developed and maintained by various individuals over the years and each individual’s unique efforts have helped shape and grow Gremlin. Finally, the authors are grateful to Lynn Bender for the creation of the GraphDay conference and his support of the graph computing space.

[1] Y. Aharonov, L. Davidovich, and N. Zagury. Quantum random walks. *Physical Review A*, 48:1687–1690, Aug 1993. doi: 10.1103/PhysRevA.48.1687. URL [link.aps.org/doi/10.1103/PhysRevA.48.1687](http://link.aps.org/doi/10.1103/PhysRevA.48.1687)

[2] A. Ambainis. Quantum walks and their algorithmic applications. *International Journal of Quantum Information*, 01(04):507–518, 2003. doi: 10.1142/S0219749903000383.

[3] D. Bohm. A suggested interpretation of the quantum theory in terms of “hidden” variables. ii. *Physical Review*, 85:180–193, January 1952. doi: 10.1103/PhysRev.85.180.

[4] A. M. Childs. Universal computation by quantum walk. *Physical Review Letters*, 102:180501, May 2009. doi: 10.1103/PhysRevLett.102.180501.

[5] P. A. M. Dirac. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, 35:416–418, 7 1939. ISSN 1469-8064. doi: 10.1017/S0305004100021162.

[6] S. J. Gay. Quantum programming languages: survey and bibliography. *Mathematical Structures in Computer Science*, 16:581–600, 8 2006. ISSN 1469-8072. doi: 10.1017/S0960129506005378. URL [http://journals.cambridge.org/article_S0960129506005378](http://journals.cambridge.org/article_S0960129506005378)

[7] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989. ISSN 0360-0300. doi: 10.1145/72551.72554.

[8] I. G. Karafyllidis and P. I. Hagouel. Multiple quantum walkers on the line using hybrid coins: A possible tool for quantum search. *Electronics and Energetics*, 27(1):103–112, March 2014.

[9] J. Kempe. Quantum random walks: An introductory overview. *Contemporary Physics*, 44(4):307–327, 2003. doi: 10.1080/00107151031000110776. URL [http://dx.doi.org/10.1080/00107151031000110776](http://dx.doi.org/10.1080/00107151031000110776)

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[11] A. Montanaro. Quantum walks on directed graphs. *Quantum Information and Computation*, 7:93–102, 2007.

[12] B. Nachtergaele. *Quantum Spin Systems*. Elsevier, 2004.

[13] A. C. Oliveira, R. Portugal, and R. Donangelo. Decoherence in two-dimensional quantum walks. *Physical Review A*, 74:012312, Jul 2006. doi: 10.1103/PhysRevA.74.012312. URL [link.aps.org/doi/10.1103/PhysRevA.74.012312](http://link.aps.org/doi/10.1103/PhysRevA.74.012312)

[14] A. C. Oliveira, R. Portugal, and R. Donangelo. Sim-
ulation of the single- and double-slit experiments with quantum walkers. Technical report, 2007. URL http://arxiv.org/abs/0706.3181

[15] M. A. Rodriguez. The Gremlin graph traversal machine and language. In Proceedings of the 15th Symposium on Database Programming Languages, DBPL 2015, pages 1–10. ACM, October 2015. ISBN 978-1-4503-3902-5. doi: 10.1145/2815072.2815073. URL http://arxiv.org/abs/1508.03843

[16] G. Taylor. Interference fringes with feeble light. Proceedings of the Cambridge Philosophical Society, 15:114–115, 1909.

[17] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society, 42(2):230–265, 1937.

[18] L. G. Valiant. A bridging model for parallel computation. Communications of the ACM, 33(8):103–111, August 1990.

[19] N. S. Yanofsky and M. A. Mannucci. Quantum Computing for Computer Scientists. Cambridge University Press, New York, NY, USA, 1 edition, 2008. ISBN 0521879965, 9780521879965.

[20] T. Young. The Bakerian lecture: On the theory of light and colours. Philosophical Transactions of the Royal Society of London, 92:12–48, 1802. doi: 10.1098/rstl.1802.0004.