The Gremlin Graph Traversal Machine and Language (Invited Talk)

Marko A. Rodriguez

Director of Engineering at DataStax, Inc.
Project Committee Member of Apache TinkerPop
Santa Fe, NM USA
marko@datastax.com

Abstract

Gremlin is a graph traversal machine and language designed, developed, and distributed by the Apache TinkerPop project. Gremlin, as a graph traversal machine, is composed of three interacting components: a graph, a traversal, and a set of traversers. The traversers move about the graph according to the instructions specified in the traversal, where the result of the computation is the ultimate locations of all halted traversers. A Gremlin machine can be executed over any supporting graph computing system such as an OLTP graph database and/or an OLAP graph processor. Gremlin, as a graph traversal language, is a functional language implemented in the user's native programming language and is used to define the traversal of a Gremlin machine. This article provides a mathematical description of Gremlin and details its automaton and functional properties. These properties enable Gremlin to naturally support imperative and declarative querying, host language agnosticism, user-defined domain specific languages, an extensible compiler/optimizer, single- and multi-machine execution models, hybrid depth- and breadth-first evaluation, as well as the existence of a Universal Gremlin Machine and its respective entailments.

Categories and Subject Descriptors G.2 [*Discrete Mathematics*]: Graph Theory

Keywords graph traversal, finite automata, functional languages, virtual machines

1. Introduction

A graph is a structure composed of vertices and edges. Graphs have seen a resurgence in the database community with the growth of graph database technology [1]. The query language of a graph database typically promotes either a *graph traversal* or a *graph pattern match* perspective. In the traversal model, traversers walk a graph according to particular user provided instructions and the result of the traversal is the locations of all halted traversers. In the pattern match model, a subgraph containing variables is created by the user and all graph elements that bind to those variables are returned as the result set. Gremlin supports both the imperative traversal-style and the declarative pattern match-style within the same framework. Furthermore, beyond supporting both popular models of graph querying, Gremlin's machine and language structures naturally facilitate Gremlin being 1.) embedded in a host programming language, 2.) extended by users wishing to leverage the terminology of their problem domain, 3.) optimized via an extensible set of compile-time rewrite rules, 4.) executed within a multi-machine compute cluster, 5.) evaluated in a depth-first, breadth-first, or hybrid ordering, and finally, 6.) represented within the graph itself via the theoretical existence of a Universal Gremlin Machine.

DBPL'15, October 27, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3902-5/15/10...$15.00
http://dx.doi.org/10.1145/2815072.2815073

2. Graph Traversal Machine

Gremlin, as a graph traversal machine, is composed of three components: a graph G (data), a traversal Ψ (instructions), and a set of traversers T (read/write heads). Conceptually, a collection of traversers in T move about G according to the instructions specified in Ψ. The computation is complete when either 1.) there no longer exists any traversers in T or 2.) all existing traversers no longer reference an instruction in Ψ (i.e. they have halted). For the former, the result set is the empty set. For the latter, the result set is the multi-set union of the G locations those halted traversers reference.

2.1 The Graph

Gremlin operates over a multi-relational, attributed, digraph $G = (V, E, \lambda)$, where V is a set of vertices, $E \subseteq (V \times V)$ is a multi-set of directed binary edges, and $\lambda : ((V \cup E) \times \Sigma^*) \rightarrow (U \setminus (V \cup E))$ is a partial function that maps an element/string pair to an object in the universal set U (excluding vertices and edges as allowed property values). Given λ, every vertex and edge can have an arbitrary number of key/value pairs called *properties*. For example, vertices may have name, age, latitude properties and edges may have weight, date, permission properties. The universal set U contains the set of all property values. These values may be restricted to longs ($\mathbb{N}$), doubles ($\mathbb{R}$), strings (Σ^*), etc., or subsets thereof, and thus given the schema of the graph, U can be constrained accordingly.

2.2 The Traversal

A traversal Ψ is a tree of functions called *steps*. Steps are arranged in the following two ways:

1. **Linear motif**: The traversal $f \circ g \circ h$ is a linear chain of three steps where the output traversers of f are the input traversers of g. Likewise, the output of g is the input of h.
2. **Nested motif**: The traversal $f(g \circ h) \circ k$ contains the nested traversal $g \circ h$ which is an argument of the step f. In this way,

f will leverage $g \circ h$ in its mapping of its input traversers to its output traversers which are then provided as input to k.

A step $f \in \Psi$ defined as $f : A^* \rightarrow B^*$ maps a set of traversers located at objects of type A to a set of traversers located at objects of type B. The Kleene star notation A^*, when used in the context of an interpretation function, denotes that multiple traversers may be at the same element in A. However, what is mapped is a unique set of traversers to a unique set of traversers. It is only the multi-set union ($\uplus$) of their locations in G that may contain duplicates.

The Gremlin graph traversal language defines approximately 30 steps which can be understood as the *instruction set* of the Gremlin traversal machine. These steps are useful in practice, with typically only 10 or so of them being applied in the majority of cases. Each of the provided steps can be understood as being a specification of one of the 5 general types enumerated below.[1]

1. **map** : $A^* \rightarrow B^*$, where $|\text{map}(T)| = |T|$. These functions map the traversers T at objects of type A to a set of traversers at objects of type B without altering the traverser set size.
2. **flatMap** : $A^* \rightarrow B^*$, where the output traverser set may be smaller, equal to, or larger than the input traverser set.
3. **filter** : $A^* \rightarrow A^*$, where $\text{filter}(T) \subseteq T$. The traversers in the input set are either retained or removed from the output set.
4. **sideEffect** : $A^* \rightarrow_x A^*$, where $\text{sideEffect}(T) = T$. An identity function operates on the traversers though some data structure x (typically in G) is mutated in some way.
5. **branch** : $A^* \rightarrow^b B^*$, where an internal branch function $b : T \rightarrow \mathcal{P}(\Psi)$ maps a traverser to any number of the nested traversals' start steps.

All the above steps can be represented as a specification of flatMap() – i.e. map one set of traversers to another set of traversers. For instance, map() as a flatMap() simply maps each traverser in T (at A) to a single traverser in B. For filter(), flatMap() either includes the original input set traverser or removes it, where $A = B$. If flatMap() can mutate outside data structures, then sideEffect() is simulated, where the input traverser set T is the output traverser set. Finally, branch() is simulated by ensuring the internal logic of flatMap() include rules for choosing different mappings of the traversers in T given their state.

2.3 The Traverser

A traverser unifies the graph and the traversal via a reference to an object in the graph and a reference to a step in the traversal. Formally, a traverser t is an element in the 6-tuple set

$$T \subseteq \left(U \times \Psi \times (\mathcal{P}(\Sigma^*) \times U)^* \times \mathbb{N}^+ \times U \times \mathbb{N}^+\right).$$

The first element of the tuple is the traverser's location in the graph G (e.g. $v \in V$, where $V \subset U$).[2] The second element is the traverser's step location in the traversal Ψ. The third element is a sequence of sets of strings and objects called a *labeled path*. For example, $(((a), x), ((b, c), y), (\emptyset, z))$ denotes the traverser's path $x \rightsquigarrow y \rightsquigarrow z$ with respective step labels at each location. The forth element is the traverser's *bulk* which denotes the number of equivalent traversers this particular traverser represents.[3] The firth element is the traverser's *sack* which is a local variable of the traverser. The sixth, and final element, is the traverser's *loop counter* which specifies the number of times a traverser has gone through a loop sequence. The following functions project the aforementioned components of a traverser to their respective values.

1. $\mu : T \rightarrow U$ maps a traverser to an object in U (i.e. its location in the graph).
2. $\psi : T \rightarrow \Psi$ maps a traverser to a step in Ψ (i.e. its location in the traversal – program counter).
3. $\Delta : T \rightarrow (\mathcal{P}(\Sigma^*) \times U)^*$ maps a traverser to its labeled path (i.e. its history in the graph).
4. $\beta : T \rightarrow \mathbb{N}^+$ maps a traverser to its bulk.
5. $\varsigma : T \rightarrow U$ maps a traverser to its sack value.
6. $\iota : T \rightarrow \mathbb{N}^+$ maps a traverser to its loop counter.

Visually, a traverser $t \in T$ is a "bundle" of local variables (metadata) with a projection to a location in the graph G and a projection to a location in the traversal Ψ.

$$G \longleftarrow_{\mu} \frac{t \in T}{\{\Delta, \beta, \varsigma, \iota\}} {}_{\psi}\longrightarrow \Psi$$

3. Graph Traversal Language

Gremlin, as a graph traversal language, is a *functional language*. The purpose of the language is to enable a human user to easily define Ψ and thus, program a Gremlin machine. The simplicity of Gremlin's grammar enables it to be embedded in the native programming language of the user.[4] In this way, for a developer, there is no discontinuity between their software code and their graph analysis code.

In order for a language to host Gremlin, the language needs to support *function composition* and *function types* (i.e. functions as first-class entities or enable it via "function objects."). With method chaining (a type of function composition), a natural fluent syntax is possible. For instance, the traversal $a \circ b \circ c$ is denoted `a().b().c()` in the dot notation-syntax of modern object-oriented programming languages. With function arguments, traversal nesting is possible. For instance, $a(b \circ c) \circ d$ is denoted `a(b().c()).d()`.

3.1 A Simple Traversal

The most basic graph traversal is one that moves traversers through the steps of Ψ in a sequential order ($\Psi_1 \rightsquigarrow \Psi_2 \rightsquigarrow \ldots \rightsquigarrow \Psi_{|\Psi|}$) and where no step maintains internal, nested traversals. The example traversal below is a simple linear traversal that determines, in plain language, the age of the oldest person that Marko knows (assuming, for the sake of simplicity of discussion, that each vertex in the example graph has a unique name).

```
g.V().has("name","marko").
  out("knows").values("age").max()
```

The first term, V_g (`g.V()`), is the definition of a traverser set bijective to V, where $\biguplus_i \mu((V_g)_i) = V$. The above traversal can

[1] If the underlying host language supports lambda functions (and `LambdaVerificationStrategy` is disabled), then it is possible for users to leverage the common lambda parameterization idiom of functional programming. For instance, users can do `filter{t.loops() < 5}`. However, this is strongly discouraged as the provided lambda can not be subjected to Gremlin's compiler optimizations. Instead, the pure traversal form `loops().is(lt(5))` should be used, where is() is a type of filter() step.

[2] The "graph location" of a traverser is in U as opposed to only $V \cup E$ because a traverser can move beyond vertices and edges by referencing arbitrary objects associated with the graph such as property keys, property values, and side-effect data structures.

[3] Traverser bulk is useful as an fundamental optimization, though it is not theoretically required.

[4] TinkerPop distributes a Gremlin machine implemented in Java8 and a Gremlin language binding in both Java8 and Groovy. With the JVM being host to numerous programming languages, the wider TinkerPop community has provided Gremlin language bindings in Scala, Clojure, Ruby, JavaScript, and more. Conceptually, TinkerPop's Gremlin machine is a virtual machine implemented in Java that can be programmed via the numerous Java-based programming languages in existence.

be written in curried functional notation as

$$\max(\text{values}_{\text{age}}(\text{out}_{\text{knows}}(\text{has}_{\text{name=marko}}(V_g)))).$$

The starting traverser set V_g is first processed by $\text{has}_{\text{name=marko}}$. A traverser set is returned that only contains a single traverser at the vertex named "marko." The step $\text{out}_{\text{knows}}$ then maps the *marko*-vertex traverser (parent) to a traverser set (children) located at those vertices that are outgoing *knows*-adjacent to the *marko*-vertex. The children have a new graph location, traversal step location, and a path that is the concatenation of their parent's path and their current location. An example child traverser, at this point, will be the 6-tuple

$$(y, \text{values}_{\text{age}}, ((\emptyset, x), (\emptyset, y)), 1, \emptyset, 0),$$

where $\lambda(x, \text{name}) = \text{marko}$ and $\lambda((x, y), \text{label}) = \text{knows}$. Next, $\text{values}_{\text{age}}$ maps to a traverser set where each child traverser is located at the integer value of their current vertex's *age*-property. Finally, max() transforms the traverser's at $\mathbb{N}^*$ to a single traverser at a number representing the maximum number in the previous set (i.e. the oldest *age*-value).

$$\begin{array}{rll}
V_g & : \mathbf{0} \to V^* & \text{flatMap} \\
\text{has}_{\text{name=marko}} & : V^* \to V^* & \text{filter} \\
\text{out}_{\text{knows}} & : V^* \to V^* & \text{flatMap} \\
\text{values}_{\text{age}} & : V^* \to \mathbb{N}^* & \text{map} \\
\max & : [\mathbb{N}^*] \to \mathbb{N} & \text{map}
\end{array}$$

It is important to note that the domain of the $step_n$ is equal to the range of $step_{n-1}$. Furthermore, the domain of max() is $[N^*]$ and the range is N. The step max() is "blocking" in that the entire traverser set is required as input before the single traverser t is outputted, where $\mu(t) \in \mathbb{N}$. The notation $[A^*]$ denotes a barrier. Steps that reduce a traverser set to a single traverser by way of some binary operation are called *reducing barrier steps*.

In Gremlin, a traverser set can grow and shrink over the course of the computation. Traverser sets typically shrink due to filter() steps removing traversers, map()/flatMap() partial functions mapping to undefined locations in G (e.g. Marko may not know anyone), or reducing barrier steps going from many-to-one.[5] Traverser sets grow due to one-to-many flatMap() steps. Gremlin traversers are *furcating automata* in that if multiple options are met, then all options are taken. For instance, if a traverser is at a single vertex and the current step $\psi(t)$ maps to many adjacent vertices (e.g. $\text{out}_{\text{knows}}$ and Marko knows many people), then the traverser "splits" (clones itself) and each child is placed at each adjacent vertex. The only modification to the child clones are new locations in G and Ψ as well as a new labeled path Δ which is a one-step extension of the parent traverser's path.

The language used in the discussion thus far states that a "set of traversers" is being mapped between each step of the traversal. However, traversers are isolated entities maintaining their own metadata/state, where the step functions themselves have no state. This type of traverser isolation enables a traversal's evaluation order to change at different points in Ψ as sometimes it is useful to use depth-first (one traverser at each step) and sometimes breadth-first (sets of traversers at each step). A Gremlin machine implementation can make use of a dataflow/stream construct [9] and simulate breadth-first evaluation at particular points in the traverser stream via the insertion of an *identity barrier step* with interpretation function barrier : $[U^*] \to U^*$.

[5] As will be explain later, traverser sets also shrink when multiple traversers arrive at the same location in G and, at which point, these traversers merge into a single traverser with a respective "bulk" equal to the sum of the bulks of all merged traversers. However, while the set shrinks, the same logical number of traversers still exists.

3.2 A Branching Traversal

A branch in Gremlin is a split in Ψ. Formally,

$$\text{branch}_{\text{b}}(t) = b(t)(t),$$

where $b : T \to \Psi$. The branching function b determines, given the state of t, which internal traversal the traverser should follow. Depending on the particular branch step, the traverser t may be sent down a single branch (e.g. choose()), a subset of the branches (e.g. repeat().emit()), or all branches (e.g. union()).

The choose() step is a branch step which provides the common "if/else if/.../else" programming construct.

```
g.V().choose(label()).
  option("person", out("created").count()).
  option("software", in("created").count()).
  option(none, label())
```

In the above traversal, if the traverser incoming to choose() is at a *person*-vertex, then send the traverser down the branch that computes the number of projects that that person has created. If the traverser is at a *software*-vertex, then send the traverser down the branch that computes the number of collaborators on that software project. Finally, if the traverser is located at neither a *person*- nor *software*-vertex, then send the traverser down the branch that yields the label of the vertex, where the *none*-option refers to a branch that should be taken if no other options are valid (i.e. "else"). If the *none*-option did not exist, then choose() would act as a filter removing the option-less traverser from T. Note that option() is not a step, but a *step modulator*. Step modulators are "syntactic sugars" that manipulate the previous step in order to reduce the complexity of the modulated step's arguments (and respective function overloadings). In this way, choose() takes traversals as arguments and thus, maintains internal nested traversals, where the first traversal (label()) plus the option keys (e.g. "person") form the branch function. The above choose() step is represented in curried form as

$$\text{choose}_{\text{label}}(t) = \begin{cases} \text{count}(\text{out}_{\text{created}}(t)) & : \mu(\text{label}(t)) = \text{person} \\ \text{count}(\text{in}_{\text{created}}(t)) & : \mu(\text{label}(t)) = \text{software} \\ \text{label}(t) & : \text{otherwise.} \end{cases}$$

Note that the domain and range of $\text{choose}_{\text{label}}()$ is

$$\text{choose}_{\text{label}} : V^* \to \left(\mathbb{N}^+ \cup \Sigma^*\right)^*.$$

As such, any step following choose() must be able to accept either numbers or strings.

It is worth noting that Gremlin supports a more compact syntax for boolean-based "if/else." If there are only two options, "person" and *none*, then the above traversal would be defined as below.

```
g.V().choose(label().is("person"),
  out("created").count(),
  label())
```

3.3 A Recursive Traversal

The examples presented thus far have the traversers moving from "left to right" through the sequence of steps in Ψ. In order to support recursion (i.e. looping), it is necessary to set the traverser's ψ-program counter back to some previously seen step. An example of such a step is the recursive function

$$\text{repeat}_p(t) = \begin{cases} \text{repeat}_p(t_{\iota+1}) & : p(t) = \textbf{true} \\ t_0 & : \text{otherwise,} \end{cases}$$

where $p : T \to \textbf{Bool}$ is some traverser predicate, $\iota(t_{\iota+1}) = \iota(t)+1$ (i.e. increment the loop counter), and $\iota(t_0) = 0$ (i.e. reset the loop counter).

The following traversal returns the names of the vertices 5 outgoing steps from the vertex named "marko."

```
g.V().has("name","marko").
  repeat(out()).times(5).
    values("name")
```

With times() being a step modulator, the repeat() step is functionally defined as

$$\text{repeat}_{\iota<5}(t) = \begin{cases} \text{repeat}_{\iota<5}(\text{out}(t_{\iota+1})) & : \iota(t) < 5 \\ t_0 & : \text{otherwise.} \end{cases}$$

Suppose it is necessary to get the names of all the vertices encountered along the 5 step walk emanating from the vertex named "marko" (and not just those names 5 steps away).

```
g.V().has("name","marko").
  repeat(out()).emit().times(5).
    values("name")
```

If the traverser loops, it is also emitted along with its recursive mapping. Note that emit(), like times(), is a step modulator of repeat(), where

$$\text{repeat}_{\iota<5,\text{emit}}(t) = \begin{cases} \left(\text{repeat}_{\iota<5,\text{emit}}(\text{out}(t_{\iota+1})), t_0\right) & : \iota(t) < 5 \\ t_0 & : \text{otherwise.} \end{cases}$$

3.4 A Path Traversal

The third component of the traverser t's 6-tuple is its labeled path $\Delta(t) \in (\mathcal{P}(\Sigma^*) \times U)^*$. Whenever a traverser is mapped to a new location in G, this location as well as the set of labels for the respective step in Ψ are appended to the child traverser's path. For example, assume the following traversal.

```
g.V().as("a").out().as("b","c").path()
```

In the traversal above, the traverser t will start at a particular vertex in $x \in V$. That location is labeled "a" via the step modulator as(), where $\Delta(t) = (((a), x))$. Next, the traverser t will split itself amongst all the outgoing adjacent vertices of x, where one particular child traverser's path would be $\Delta(t') = (((a), x), ((b, c), y))$ assuming $(x, y) \in E$. Thus,

$$\text{path}(t) = t_{\Delta(t)},$$

where $\mu\left(t_{\Delta(t)}\right) = \Delta(t)$. A single halted traverser t'' from the traversal above would have

$$\mu(t'') = (((a), x), ((b, c), y))$$

and

$$\Delta(t'') = (((a), x), ((b, c), y), (\emptyset, ((a), x), ((b, c), y)))).$$

That is, the labeled path of t'', up to that point in the traversal, is an element in its path.

A traverser's path history is useful in the following enumerated situations.

1. It is necessary to determine the (shortest)-path from vertex x to vertex y.
2. It is necessary to go back to some previous location of the traverser.
3. It is necessary to determine if a particular location has already been visited.

In terms of items 1 and 3,

```
g.V(x).repeat(out().simplePath()).
  until(is(y)).path().limit(1)
```

will return the shortest, simple (non-looping) path from vertex x to vertex y, where until() is a step modulator for repeat() and the filter

$$\text{simplePath}(t) = \begin{cases} t & : \left|\bigcup_{i<|\Delta(t)|} \mu\left(\Delta(t)_i\right)\right| = |\Delta(t)| \\ \emptyset & : \text{otherwise.} \end{cases}$$

3.5 A Projecting Traversal

In the previous subsection, it was stated that sometimes it is necessary to go back to some previous location in the traverser's path history. The following traversal does just that.

```
g.V().as("a").out("knows").as("b").
  select("a","b").
    by(in("knows").count()).
    by(out("knows").count())
```

When the traverser t reaches select(), there will be two vertices labeled "a" and "b" in its path. The select() step generates two new traversers $t_{\Delta_a(t)}$ and $t_{\Delta_b(t)}$, where $\mu\left(t_{\Delta_a(t)}\right) = \Delta_a(t)$ and $\mu\left(t_{\Delta_b(t)}\right) = \Delta_b(t)$. Traverser $t_{\Delta_a(t)}$ will ultimately determine the number of incoming *knows*-adjacent vertices to the "a"-vertex and traverser $t_{\Delta_b(t)}$ will determine the number of outgoing *knows*-adjacent vertices to the "b"-vertex. The by() step modulator specifies which traversal the "a" and "b" traversers should traverse. The curried function signature is

$$\text{select}_{\text{a,b}} : V^* \to \mathcal{P}(\Sigma^* \times \mathbb{N}^+)^*,$$

where an element in $\mathcal{P}(\Sigma^* \times \mathbb{N}^+)$ is a `Map<String,Long>` data structure in programming. The definition of the $\text{select}_{\text{a,b}}$ function is

$$\text{select}_{\text{a,b}}(t) = \left(\begin{array}{c} \left(a, \text{count}\left(\text{in}_{\text{knows}}\left(t_{\Delta_a(t)}\right)\right)\right), \\ \left(b, \text{count}\left(\text{out}_{\text{knows}}\left(t_{\Delta_b(t)}\right)\right)\right) \end{array} \right).$$

The where() step is similar to select() save that it filters a traverser based on its labeled path. The traversal below does the same selection as above, but only if the traverser's "a" and "b" vertices are not maternal siblings. Thus,

$$\neg\text{where}_{\text{a,b}}(t) = \begin{cases} t & : t_{\Delta_b(t)} \notin \text{in}_{\text{mother}}\left(\text{out}_{\text{mother}}\left(t_{\Delta_a(t)}\right)\right) \\ \emptyset & : \text{otherwise} \end{cases}$$

in the traversal

```
g.V().as("a").out("knows").as("b").
  where(not(
    as("a").out("mother").in("mother").as("b"))).
  select("a","b").
    by(in("knows").count()).
    by(out("knows").count())
```

The above syntax of `as("a")...as("b")` is syntactic sugar for `select("a")...where(eq("b"))`.

3.6 A Centrality Traversal

A graph is a complex structure that is difficult to reason about in its n-dimensional form. In order to extract meaningful information from graphs, numerous statistics have been developed in the domains of graph theory and network science [2]. Every graph statistic maps an n-dimensional graph to some lower dimensional space. Typically, the reduction is either to a 0- or 1-dimensional space. For instance, $\text{size}_V : \mathbb{G} \to \mathbb{N}^+$ is a 0-dimensional statistic that maps a graph to the number of vertices it contains. *Graph centrality* is a 1-dimensional graph statistic generally defined as $\mathbb{G} \to \mathbb{R}^{|V|}$, where the function maps a graph G to a vector of centrality scores for each vertex in V. Centrality measures identify "important," "representative," "connective," "influential," etc. vertices in the graph. In *eigenvector centrality*, centrality is formally defined as the probability that a vertex will be host to some random walker at some random point in time. This description can be represented by the linear algebraic equation $A\mathbf{v} = \lambda\mathbf{v}$, where

$$A_{i,j} = \begin{cases} 1 & : (i,j) \in E \\ 0 & : \text{otherwise} \end{cases}$$

is the adjacency matrix of the graph G and $\mathbf{v}$ is the eigenvector whose components change, with each iteration, according to the scalar λ (the eigenvalue).[6] If the graph is strongly connected and aperiodic [6], then the larger a vertex's value in $\mathbf{v}$, the more central it is in the graph. The equation $A\mathbf{v} = \lambda\mathbf{v}$ is expressed and solved in Gremlin via the traversal

```
g.V().repeat(groupCount("m").out()).
  times(30).cap("m")
```

The groupCount() side-effect step is defined as

$$\text{groupCount}_{\text{m}}(t) =_{\text{m}} t : m[\mu(t)] = m[\mu(t)] + \beta(t).$$

Every time a traverser arrives at groupCount("m"), its vertex location is indexed into the map $m \in \mathcal{P}(V \times \mathbb{N}^+)$, where $m \sim \mathbf{v}$ and m is a `Map<Vertex,Long>`. The keys of m are the vertices in V and the values are the number of times each vertex has been encountered thus far. Thirty iterations is provided as a value that is typically large enough to ensure convergence in natural graphs. It is possible for repeat()'s times() to be replaced with an until() that calculates whether the map's values have reached a steady state distribution. This requires comparing the unit vector $\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$ at iteration n with $\hat{\mathbf{v}}$ at iteration $n+1$. However, for the sake of simplicity, this computation is left to the reader to deduce.

The step cap("m") is a *supplying barrier step* in that is takes takes all the incoming traversers and emits a single traverser that is not a function of the incoming set, but a function of a side-effect data structure. In this case, a single traverser t is emitted where $\mu(t) = m$.

$$\text{cap}_{\text{m}} : [V^*] \to \mathcal{P}(V \times \mathbb{N}^+).$$

3.7 A Mutating Traversal

All of the examples presented thus far have only read from the graph. None have written to it. Gremlin provides a collection of graph mutation steps that can be used to add and remove vertices, edges, and properties. A few of these are outlined below.

addOutE	$: V^* \to E^*$	sideEffect/map
addInE	$: V^* \to E^*$	sideEffect/map
addV	$: U^* \to V^*$	sideEffect/map
property	$: (V \cup E)^* \to (V \cup E)^*$	sideEffect
drop	$: [(V \cup E)^*] \to \mathbf{0}$	sideEffect/map

The two traversals below mutate the graph. The first one adds an inverse *createdBy*-edge for every *created*-edge. The second removes the original *created*-edges.

```
g.V().as("a").out("created").
  addOutE("createdBy","a")

g.V().outE("created").drop()
```

[6] The method of multiplying the vector $\mathbf{v}$ (initially being set to $\mathbf{1}^{|V|}$) against the adjacency matrix A until λ reaches a fixed point is called the *power iteration method*. The resultant $\mathbf{v}$ (when λ converges) is guaranteed to be the primary eigenvector with λ being the largest eigenvalue. λ defines the growth rate of $\mathbf{v}$ over any subsequent iterations. In Apache TinkerPop's Gremlin implementation, the growth rate λ plays an important role in understanding when the "bulk" β of a traverser will overflow its 64-bit long representation.

3.8 A Declarative Traversal

Gremlin supports graph pattern matching analogous, in many respects, to SPARQL [11]. The primary benefit of Gremlin's pattern matching is that it encompasses only a single step within the Gremlin language and thus, it is possible to move from declarative pattern matching, to imperative traversals, and back all within the same traversal definition. Moreover, pattern matching is expressed as a traversal and thus, uses the same Gremlin traversal machine constructs presented thus far.

The match() step's argument is a set of *traversal patterns* that may be not()'d or nested via and() and or(). When a traverser enters match(), it will propagate through each pattern. A traverser that continues on to the step after match() is guaranteed to have had its labeled path values bind (via equality) to all the prefix and postfix variables of the match() traversal patterns.

```
g.V().match(
    as("a").out("created").as("b"),
    as("b").in("created").count().is(gt(3)),
    as("b").in("created").as("c"),
    as("a").out("father").as("c")).
  dedup("a").
  select("a").by("name")
```

The traversal above will return the name of all vertices who created a piece of software in collaboration with at least 4 people with the caveat that one of those collaborators is their father.

$$\text{match}(t) =$$

$$\begin{cases} \text{match}(\text{bind}_{\text{b}}(\text{out}_{\text{created}}(t_{\Delta_a(t) \wedge \Delta_{\text{m1}}}))) & : \Delta_a \neq \emptyset = \Delta_{m1} \\ \text{match}(\text{is}_{>3}(\text{count}(\text{in}_{\text{created}}(t_{\Delta_b(t) \wedge \Delta_{\text{m2}}})))) & : \Delta_b \neq \emptyset = \Delta_{m2} \\ \text{match}(\text{bind}_{\text{c}}(\text{in}_{\text{created}}(t_{\Delta_b(t) \wedge \Delta_{\text{m3}}}))) & : \Delta_b \neq \emptyset = \Delta_{m3} \\ \text{match}(\text{bind}_{\text{c}}(\text{out}_{\text{father}}(t_{\Delta_a(t) \wedge \Delta_{\text{m4}}}))) & : \Delta_a \neq \emptyset = \Delta_{m4} \\ t & : \text{otherwise,} \end{cases}$$

where

$$\text{bind}_{\text{x}}(t) = \begin{cases} t_{\Delta_x(t) = \mu(t)} & : \Delta_x(t) = \emptyset \\ t & : \Delta_x(t) = \mu(t) \\ \emptyset & : \text{otherwise.} \end{cases}$$

The match() step is a recursively defined branch step, where each pattern/branch is taken once and only once. This is guaranteed by the hidden path label "m#" which is appended to the traverser's labeled path upon entering a branch pattern. Moreover, the prefix label x of a pattern must exist in the traverser's path prior to the traverser taking that particular branch (i.e. $\Delta_x(t)$ can not equal $\emptyset$). Upon completing a pattern, if the traverser already has the postfix label in $\Delta(t)$ then that historic location must equal its current location $\mu(t)$ otherwise the traverser is deemed a non-match and is filtered out. However, if the postfix label does not exist in $\Delta(t)$, then it is added to the path of the traverser and thus, that variable is bound for all subsequent patterns. A traverser is able to exit the match() step when every pattern has been taken. Within the labeled path of a surviving traverser lies the match-variables and their respective bindings – $(\Delta_a(t), \Delta_b(t), \Delta_c(t))$.

The order in which match-patterns are executed is up to the match() step implementation. The only caveat is that the two pattern selection criteria are respected – 1.) the pattern has not been taken before and 2.) the prefix variable of the pattern already exists in the labeled path of the traverser. TinkerPop's Gremlin implementation provides two *match algorithms* called `GreedyMatchAlgorithm` and `CountMatchAlgorithm` [3]. The former simply finds the first pattern in the user provided list that meets the respective constraints and executes that pattern. The latter maintains, for each pattern, a dynamic *multiplicity* variable that is equal to the number of traversers outputted by the pattern divided by the number of traversers inputted to the pattern. It then continually re-sorts the patterns favoring those that have the lowest multiplicity (i.e. it favors patterns that yield the largest set reductions).

3.9 A Domain Specific Traversal

The Gremlin traversal machine supports approximately 30 steps in its instruction set. These steps are deemed the most useful for most any traversal algorithm. The Gremlin traversal language is (nearly) in one-to-one correspondence with these steps. This alignment is due to the fact that the Gremlin language is a "graph specific language" that forces the user to process their data from the graph-perspective of vertices (out(), inV()), edges (outE()), and properties (values()).

Typically, in practice, a graph structure represents some problem-domain that is best modeled as a graph. In these domains, the concept of vertices and edges may be understood as people and social relationships, for example. It is trivial for a user to define a *domain specific language* that, when compiled, generates a traversal in terms of the ∼30 Gremlin steps. For instance, given the example graph used throughout this section, a hypothetical "social traversal language" may allow the following domain specific query to be expressed.

```
g.people().named("marko").
  who().know(well).people().
  who().created("software").
  are().named()
```

Each one of these steps would be the composite or 1 or more Gremlin steps.

```
g.people()           ↦  g.V().hasLabel("person")
named("marko")       ↦  has("name","marko")
who()                ↦  identity()
know(well)           ↦  outE("knows").
                         has("weight",gt(0.75)).
                         inV()
created("software")  ↦  out("created").
                         hasLabel("software")
are()                ↦  identity()
named()              ↦  values("name")
```

While the user expresses queries in the language of their domain, Gremlin ultimately evaluates those queries in terms of the underlying graph structure used to represent that domain.

4. Traversal Strategies

A traverser executes the step functions defined in Ψ. Sometimes a particular step sequence in Ψ can be expressed in another way that is perhaps more efficient to execute. Gremlin *traversal strategies* define translations which rewrite sections of a traversal (with typically, though not necessarily, the same semantics as the original traversal) [16]. There are 5 traversal strategy categories that form a total order, where a category's strategies are evaluated prior to moving to the next category and prior to traversal execution. Furthermore, within a category, strategies form a partial order where some strategies may require the execution of another strategy before (or after) its execution.

1. **Decoration**: Rewrite a traversal given that certain steps serve only as syntactic placeholders.
2. **Optimization**: Rewrite a traversal given that a particular step sequence can be expressed in a more efficient form.
3. **Vendor Optimization**: Rewrite a traversal give that a particular step sequence can be expressed in a more efficient form given the underlying graph system.
4. **Finalization**: Make any final adjustments to the traversal given the "final" compiled form.
5. **Verification**: Analyze the traversal and ensure it is valid given some set of constraints.

The itemization below presents a subset of Gremlin's provided traversal strategies and demonstrates a few rewrite rule examples for each.[7] The rewrite rules are represented such that when the left-hand pattern is matched, it is rewritten as the right-hand pattern.

- **ConjunctionStrategy** (Decoration): Gremlin supports prefix and infix notation for logical connectors. When infix notation is used, it is converted into the respective prefix-based representation.

```
a.and().b          ↦  and(a,b)
a.or().b           ↦  or(a,b)
a.or().b.and().c   ↦  or(a,and(b,c))
a.and().b.or().c   ↦  or(and(a,b),c)
```

- **IncidentToAdjacentStrategy** (Optimization): If a traversal touches the incident edges of a vertex on its way to its adjacent vertices, and executes no step that requires the analysis of those incident edges, then simply jump to the adjacent vertices without manifesting the respective incident edges.

```
a.outE().inV().b         ↦  a.out().b
a.bothE().otherV().b     ↦  a.both().b
```

- **AdjacentToIncidentStrategy** (Optimization): If a traversal only checks for the existence of adjacent vertices, it is typically cheaper to only manifest the incident edges.[8]

```
a.in().count().b        ↦  a.inE().count().b
a.where(out()).b        ↦  a.where(outE()).b
a.and(in(),out()).b     ↦  a.and(inE(),outE()).b
```

- **IdentityRemovalStrategy** (Optimization): If a traversal maintains an identity() step, remove the identity() step and fold any as() modulators into the previous step.

```
a.identity().b  ↦  a.b
```

- **FilterRankingStrategy** (Optimization): All filter() steps either remove or retain traversers in T. Thus, if there is a sequence of filter() steps, reorder them such that cheaper filters are executed first in the hopes of yielding large set reductions prior to executing more costly filters. Note that order() is considered a "filter" even though it only sorts T without ever removing traversers from T.

```
a.and(c,d).has().b     ↦  a.has().and(c,d).b
a.order().dedup().b    ↦  a.dedup().order().b
```

- **RangeByIsCountStrategy** (Optimization): If a traversal only needs to determine if a particular number of elements exist,

[7] It is easy for vendors and users to register new traversal strategies with TinkerPop's Gremlin compiler. This is typically done by vendors as their graph system may maintain different optimizations accessible only through their custom interfaces. In such situations, vendors will write custom steps and traversal strategies that replace particular Gremlin step sequences with their vendor-specific steps.

[8] This is especially true in distributed Gremlin where a vertex in a vertex partition maintains direct references to only its properties, incident edges and their properties. Without `AdjacentToIncidentStrategy`, the left-handed patterns would waste network bandwidth as traversers would be sent to adjacent vertices only to be counted/etc.

then instead of counting the full set, limit the count to 1 plus the required number.

```
a.count().is(0)  ↦  a.limit(1).count().is(0)
```

- **XGraphStepStrategy** (Vendor Optimization): Graph database vendors typically maintain indices over the properties of the vertices (and sometimes edges) of the graph. In order avoid $\mathcal{O}(|V|)$ linear costs, fold all has() steps into a vendor specific V() step in order to facilitate $\mathcal{O}(log(|V|))$ indexed-based lookups.

```
V.has().has().b  ↦  V[has,has].b
```

- **MatchPredicateStrategy** (Optimization): If match() is followed by a where() step, fold that where() step into match() as a new branch pattern. In this way, the where()-based pattern is subject to the match() step's `XMatchAlgorithm` runtime optimizer. Furthermore, if match() maintains a has()-based pattern whose prefix variable is the input variable, then pull that pattern out of match() such that it may be used by the vendor for respective index lookups (see **XGraphStepStrategy**).

```
a.match(c,d).where(e).b  ↦  a.match(c,d,e)
a.match(has(),c,d).b     ↦  a.has().match(c,d).b
```

- **ProfileStrategy** (Finalization): If the user wants to get metrics about a traversal's performance, then a terminal profile() step serves as a place holder for later inserting profile() steps between each step in the traversal. Profiling metrics are maintained in a side-effect data structure.

```
a.b.profile()  ↦  a.profile().b.profile()
```

- **ComputerVerificationStrategy** (Verification): Single-machine Gremlin (OLTP) is more flexible in the types of traversals it can execute. When a traversal is executed within a multi-machine compute cluster (OLAP), certain traversal sequences are not allowed. Over time, as advances are made to the distributed Gremlin machine, respective verifications will be removed accordingly.

```
a.order.b                ↦  error
a.local(out().out()).b   ↦  error
```

5. Distributed Graph Traversals

Gremlin's *graph computer* traversal machine was designed to support the distributed execution of a Gremlin traversal via the bulk synchronous parallel (BSP) model of distributed computing [19]. In graph-based BSP, each vertex is a logical processor that receives messages (typically via adjacent neighbors), updates its state given its current state and its received messages, and send messages (typically to adjacent neighbors) [10]. This process continues until there are no more messages being sent. The result of the computation is distributed across the state of all the vertices (e.g. a *score*-property on each vertex). In Gremlin, the vertices receive traversers as messages, execute the traverser's traversal step identified by ψ, and for each traverser generated, sends a message to the respective vertex referenced by the traverser's μ.[9] For those traversers that have halted, the vertex saves the halted traversers in a "hidden" vertex property value. This process continues until there are no more traversers being sent around the cluster. The aggregate of the locations of all halted traversers across all vertices in the graph is the result of the computation.

[9] If the traverser references an edge, then the message is sent to the vertex maintaining that edge. If the traverser references some other object in U, then the traverser remains at the current vertex location.

When a graph is partitioned across a cluster, traversers migrate between the machines as dictated by their $\mu(t)$.[10] For instance, $V_1 \uplus V_2 \uplus V_3 = V$ denote three vertex partitions of V, where each partition is composed of a unique set of vertices of V and their respective incident edges ($V_1 \cap V_2 \cap V_3 = \emptyset$). When traverser t is located at $\mu(t) \in V_1$, the traverser will exist at machine 1. Suppose the step $\psi(t)$ is applied and the following traverser's are generated (t', t'', t''') where, $\mu(t') \in V_2$, $\mu(t'') \in V_3$, and $\mu(t''') \in V_1$. Traversers t' and t'' will be serialized and sent over the network to machines 2 and 3, respectively, for further execution. However, traverser t''' will remain at machine 1 to execute its referenced step $\psi(t''')$. In this model of distributed graph traversal, it is advantageous to create a partition of V that reduces costly inter-machine communication.

The Gremlin graph computer machine executes in a breadth-first manner as each traverser at every vertex is operated on in parallel. As graphs become large, the number of traversers can easily grow exponentially, especially in repeated one-to-many mappings. For example, in a 20x20 lattice with vertices having only one "right" and one "down" edge, the traversal

```
g.V(topLeft).repeat(out()).times(40)
```

will ultimately yield ~138 billion traversers at the lattice's "bottom right" vertex. The general equation for the number of traversers for any nxn lattice is $\binom{2n}{n} = \frac{(2n)!}{(n!)^2}$. While the number of traversers can grow exponentially, the number of traverser locations is always bounded by the size of U, where for this particular lattice traversal, the upper bound is $|V| = 400$. Gremlin *bulking* takes advantage of this traverser-to-location relation by projecting the (potentially exponentially growing) traverser set to a traverser set constrained to an upper limit $|U|$. As a result of this *lossless compression*, in the 20x20 lattice example, the resultant traverser set contains only one traverser t whose bulk is $\beta(t) = 137,846,528,820$. Traversers in T are "bulked" according to their respective equivalence class

$$[t] = \{t' \in T \mid \begin{array}{rcll} \mu(t) & = & \mu(t') & \wedge \\ \psi(t) & = & \psi(t') & \wedge \\ \Delta(t) & = & \Delta(t') & \wedge \\ \varsigma(t) & = & \varsigma(t') & \wedge \\ \iota(t) & = & \iota(t') & \}. \end{array}$$

In other words, all components of the traverser's 6-tuple must be equal amongst all traversers in $[t]$ except for their respective bulk.[11] The traversers in $[t]$ are reduced to a single traverser t'', where $\beta(t'') = \Sigma_i \beta\left([t]_i\right)$ and $|[t'']| = 1$. Instead of enumerating each traverser, each traverser is counted (with only one traverser existing in T for each equivalence class). Bulking ensures that breadth-first Gremlin on large graphs (OLAP) does not require an exponential amount of memory. Furthermore, this optimization is leveraged in depth-first Gremlin (OLTP) via the barrier() step (and `LazyBarrierStrategy`), where

$$\text{barrier} : [U^*] \rightarrow U^*$$

[10] It is not required that the Gremlin graph computer machine operate over multiple physical machines. The only requirement is that the underlying graph system provide a logical partition of the vertices in V and that each vertex in each partition has direct (non-remote) references to its properties, incident edges, and incident edge properties.

[11] Many traversals do not require the labeled path (or a full labeled path) of a traverser and as such, in practice, when the labeled path is not required, then $\Delta(t) = \emptyset$. This helps to ensure a smaller number of equivalence classes in T and thus, a greater likelihood of bulking. Note that when labeled paths are considered, typically, the number of equivalence classes grows proportionate to the number of unique Ψ paths in G which tends to grow exponentially in broad, non-filtering traversals.

and

$$\text{barrier}(T) = \bigcup_{\forall [t] \in T} t_{\beta(t) = \sum_i \beta([t]_i)}.$$

6. Minimal Gremlin Traversal Machines

Gremlin, as a graph traversal machine, is an automaton. Automata are studied in computer science to understand the minimal structural and behavioral requirements necessary for some "abstract machine" to perform a particular type of computation if it were to be physically built or modeled in another machine powerful enough to simulate its behavior [8]. In automata theory, computations are represented as problems in language transduction. Every language is composed of an alphabet of characters in Σ. Strings in the language are in Σ^*. If an input string is "1 + 2" and an automaton produces "3," then the automaton has the requisite computing power to perform addition (assuming it generalizes to any "$x + y$").

As it stands, the *Turing machine* is the minimal abstract machine required for general-purpose computing. A Turning machine can do any known "mechanical" (algorithmic) computation. Restricted forms of a Turning machine are able to solve simpler problems. For instance, a finite-state automaton can be programmed to process regular languages and answer regular expressions. For example, the finite-state automaton programmed as a^*b maps b, ab, aab, $aaab$, etc. to **true**.

The types of languages that particular minimal Gremlin machines can process are provided in the table below, where each minimal Gremlin machine is a reduction of the elements in the traverser's 6-tuple structure and/or a reduction in the set of possible steps that can be used to program Ψ. For Gremlin to simulate a Turing machine (as well as any less powerful automata), its instruction set must at minimum support values(), property(), sack(), choose(), repeat(), in(), and out().

Automaton	Minimal Gremlin Machine	Language
FiniteState	$(U_\mu \times \Psi \setminus \text{property} \cup \text{in} \times U_\varsigma)$	Regular
Pushdown	$(U_\mu \times \Psi \setminus \text{in} \times U_\varsigma)$	ContextFree
Turing	$(U_\mu \times \Psi \times U_\varsigma)$	Recursive

6.1 Turing Completeness

This subsection defines a surjective function whose image of the full Gremlin traversal machine is isomorphic to a single-headed Turing machine. The aforementioned components of Gremlin are mapped to the components of a 5-tuple Turing machine

$$M = (Q, F, \Gamma, \Sigma, \delta),$$

where Q is the set of machine states, $F \subseteq Q$ is the set of legal halt states, Γ is the readable/writable alphabet, $\Sigma \subseteq \Gamma$ is the initial input on an infinite one-dimensional tape of cells, and $\delta : ((Q \setminus F) \times \Gamma) \rightarrow (Q \times \Gamma \times \{L, R\})$ is the transition function which updates the machine's state, determines which symbol to write to the current tape cell, and whether it should then go "left" or "right" on the tape.

$$\begin{array}{c} \ldots[\ ][\ ][\gamma][\ ][\ ][\ ]\ldots \\ \uparrow \\ M[\delta] \end{array}$$

A machine is deemed Turing Complete if it can simulate the aforementioned single-headed Turing machine. If so, that machine can be programmed to execute any known algorithm [18].

Theorem. *The Gremlin graph traversal machine is Turing Complete.*

Proof. For the Gremlin traversal machine, the tape is the (infinite) graph G, where each vertex $v \in V$ has one incoming neighbor, one outgoing neighbor, a *symbol*-property value in Γ and initially, the symbols Σ are the *symbol*-property values of a consecutive chain of vertices. In other words, G is a line graph with each vertex representing a cell in the Turing tape with respective input symbols. The Turing machine state is the traverser's sack $\varsigma(t) \in Q$. Assume the existence of only the following Gremlin steps:

$\text{values}_{\text{symbol}}$	$: V \rightarrow \Gamma$	map	read tape
$\text{property}_{\text{symbol} \in \Gamma}$	$: V \rightarrow V$	sideEffect	write tape
sack	$: V \rightarrow Q$	map	read state
$\text{sack}_{q \in Q}$	$: V \rightarrow V$	sideEffect	write state
choose	$: V \rightarrow V$	branch	if/else
repeat	$: V \rightarrow V$	branch	loop
in	$: V \rightarrow V$	map	move left
out	$: V \rightarrow V$	map	move right.

The δ function of the Turing machine is the composition of the steps above to create Ψ. Note that the steps above map to traverser sets of size 1 as no step is a true flatMap() nor filter(). Given that there is only one "left" and one "right" adjacent neighbor to a vertex in the line graph, the above steps will never increase (nor decrease) the size of the traverser set. In this way, any single-headed Turing machine can be simulated. □

As an example, a 3-state "busy beaver" Turing machine is defined, where $\Gamma = \{0, 1\}$, $Q = \{\text{A}, \text{B}, \text{C}, \text{HALT}\}$, and δ is the Gremlin traversal below.

```
g.V(1).
 sack("A").
 repeat(choose(values("symbol")).
  option("0", choose(sack()).
   option("A",
    property("symbol","1").out().sack("B")).
   option("B",
    property("symbol","1").in().sack("A")).
   option("C",
    property("symbol","1").in().sack("B"))).
  option("1", choose(sack()).
   option("A",
    property("symbol","1").in().sack("C")).
   option("B",
    property("symbol","1").out().sack("B")).
   option("C",
    property("symbol","1").out().sack("HALT")))).
 until(sack().is("HALT"))
```

6.2 A Universal Gremlin Machine

A Universal Gremlin Machine (UGM) is a Gremlin machine that can simulate another Gremlin machine within its G, Ψ and T constructs [14, 15]. The encoding to follow will represent both Ψ and T in G [12, 17]. Any step in Ψ can be represented as a vertex $v \in V$, where $\lambda(v, \text{label}) = \text{step}$ and $\lambda(v, \text{op})$ is the operation of that step (e.g. out). If $u \in V$ is another step in Ψ that follows v then there exists an edge $(v, u) \in E$, where $\lambda((v, u), \text{label}) = \text{nextStep}$.[12] A traverser in T can be represented by a vertex $t \in V$ where, $\lambda(t, \text{label}) = \text{traverser}$ and each element of the traverser's 6-tuple is a subgraph in G.

1. $\mu(t)$ is the edge $(t, x) \in E$, where $\lambda((t, x), \text{label}) = \text{mu}$.
2. $\psi(t)$ is the edge $(t, y) \in E$, where $\lambda((t, y), \text{label}) = \text{psi}$ and $\lambda(y, \text{label}) = \text{step}$.
3. $\Delta(t)$ is a collection of edges to vertices that join step labels (as vertex properties) with graph locations in G.

[12] Further complications exist for nested traversals. However, for the sake of brevity, the general representation of such traversals are left to the reader to contemplate.

4. $\beta(t)$ is the traverser vertex's bulk property $\lambda(t, \text{bulk}) = \beta(t)$.
5. $\varsigma(t)$ is the traverser vertex's sack property $\lambda(t, \text{sack}) = \varsigma(t)$.
6. $\iota(t)$ is the traverser vertex's loop counter property $\lambda(t, \text{loops}) = \iota(t)$.

The aforementioned mapping represents both Ψ and T as subgraphs of G such that, in total, G contains the complete representation of the structure (graph) and the process (traversal and traversers) of the computation. However, representation is not execution. In order for this graph structure to evolve (and thus, compute), there must exist a Universal Gremlin Machine traverser and respective traversal that moves between $G \setminus (\Psi \cup T)$ and the traversal $\Psi \subset G$ updating the respective edges and properties of the traversers $T \subset G$. The G-encoded traversal and traversers form a *virtual machine* in the Universal Gremlin Machine. A snippet of the Ψ_{UGM} is presented below where all supported steps would need to be represented in an `option(...)`.

```
g.V().hasLabel("traverser").as("t").
 repeat(
  choose(out("psi").values("op")).
   option("out",
    outE("mu").as("drop").inV().out().
    addInE("mu","t"))
   option("in",
    outE("mu").as("drop").inV().in().
    addInE("mu","t"))
   option(...)
   option(...)
  sideEffect(select("drop").drop()).
  select("t").
  outE("psi").as("drop").inV().
  out("nextStep").addInE("psi","t").
  sideEffect(select("drop").drop()).
  select("t")).
 until(out("psi").count().is(0))
```

The Ψ_{UGM} traversal loops over its repeat() traversal until the G-encoded traverser halts by no longer referencing a step vertex. The result of the computation is the multi-set union of the symbols on the "tape"-subgraph $G \setminus (\Psi \cup T)$. Formally,

$$\text{result} = \biguplus_{i}^{|V|} \begin{cases} \lambda\left(V_i, \text{symbol}\right) & : \lambda\left(V_i, \text{label}\right) \notin \{\text{traverser}, \text{step}\} \\ \emptyset & : \text{otherwise.} \end{cases}$$

In order to provide a Universal Gremlin Machine that can operate on G-encoded Gremlin machines that maintain the same level of expressivity as the Gremlin traversal machine discussed in this article, it would be necessary to extend the above Ψ_{UGM} traversal to account for the growing and shrinking of G-encoded traverser sets as well as all the steps of Gremlin's instruction set.

6.3 Parallel Universal and G-Encoded Machines

It has been assumed, up to this point, that the traversers in T all reference steps of the same Ψ. However, nothing prevents multiple traverser sets to exist, where each set operates under a different traversal. In fact, regardless of G-encoded machines, this is necessary for allowing parallel, concurrent traversals/queries of G. With respect to G-encoded machines, the Universal Gremlin Machine need not concern itself with which traverser of which traversal it is executing. In fact, the Universal Gremlin Machine simply needs to find any traverser that has yet to halt and execute its next step. The Universal Gremlin Machine acts as a *thread* evolving the state of different traversals/programs. However, in order to get a well defined result set for each traversal, a Ψ-unique identifier would need to be appended to each traverser so that the result of some traversal Ψ^{123} can be unambiguously gathered via

$$\text{result}_{\Psi^{123}} = \biguplus_{i}^{|V|} \begin{cases} \mu(V_i) & : \lambda\left(V_i, \text{label}\right) = \text{traverser} \;\wedge \\ & \quad \lambda\left(V_i, \text{psiId}\right) = 123 \;\wedge \\ & \quad \lambda\left(V_i, \text{psi}\right) = \emptyset \\ \emptyset & : \text{otherwise.} \end{cases}$$

Finally, nothing prevents multiple Universal Gremlin Machines operating in parallel against G locating active traversers and executing a step until no more traversers exist or all traversers have halted. This is, in fact, analogous to a multi-threaded system.

6.4 Traversing a Gremlin Traversal Machine

When the graph G, the traversal Ψ, and the traverser set T are all encoded in G, then all the components of a Gremlin traversal machine exist in the same address space – namely G. A consequence of this co-location is that a traverser can, in principle, traverser its own structure. Similarly, a traverser can traverse its traversal. When a machine has direct reference to its representation, a machine can not only analyze itself via *reflection*, but it can also rewrite itself. The ramifications of this consequence, with respects to applied graph computing, are left to future ruminations.

6.5 A Primordial Graph Traversal Machine

This section describes, at a high-level, a vision of graph computing that is, in many ways, analogous to the token rewrite model of the lambda calculus [4]. A lossless, injective function takes a multi-relational, attributed digraph (**MADG**) to a multi-relational, unattributed digraph (**MDG**), where edges are reified structures and all properties are "property key"-labeled edges incident to "property value" vertices [7]. Next, there exists an injective function that maps a multi-relational digraph to an unlabeled digraph (**DG**), where labels are encoded as "binary vertex chains" [13]. Finally, another injective function has been defined that maps a digraph to an undirected graph (**UG**), where edge directions are represented as topological features of the undirected form [13].

$$\textbf{MADG} \mapsto \textbf{MDG} \mapsto \textbf{DG} \mapsto \textbf{UG}$$

Given the existence of this mapping, the complete state of computing (i.e. G, Ψ, and T) can be represented by a single undirected graph whose structure is solely the composition of "dots and lines" in some n-dimensional space. In this primitive, verbose graph, there are no labels, strings, numbers, etc., simply dots connected to each other by lines. Computing occurs when subgraphs of a particular shape (e.g. a traverser at a location in the graph) morph to form new subgraphs of a particular shape (e.g. new traversers with new graph locations). Computing, in this manner, can be conceptualized as a chemical reaction where "molecular structures" (undirected subgraphs) interact with adjacent structures to yield new structures that may elicit yet more reactions [5]. In this primordial world, the computation is complete when vertices and edges are no longer being created nor destroyed. When the undirected graph reaches an equilibrium with its "laws of physics," the problem is solved – for it has reached a stable state.

7. Conclusion

Gremlin is a graph traversal machine and language. The Gremlin machine specification is simple to describe and ultimately implement. The complexity of the computations that Gremlin enables is not necessarily due to its constructs, but due to the data sets being processed. Graphs are multi-dimensional structures able to model a heterogenous set of "things" related to each other in a heterogenous set of ways – all within a single, connected data structure. When a Gremlin traversal is evaluated against a graph, billions upon billions of traversers can be generated on even small graphs due to the exponential growth of the number of paths that exist with each step the traversers take. With so many forks in the road, traversers continually split themselves in order to explore each option that meets the constraints of the traversal they obey. When these traversers ultimately halt, they provide an answer to the question specified by their traversal, which was programmed by a user via the Gremlin traversal language.

Acknowledgments

The Apache TinkerPop project (`http://tinkerpop.com`) was started in November 2009 and is currently in its third generation of development with TinkerPop3 having been released in July of 2015. Many individuals have contributed to the project and wider ecosystem over the years and their contributions, both theoretical and applied, have been invaluable to the generation of the ideas presented in this article.

References

[1] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computer Surveys*, 40:1–39, February 2008. ISSN 0360-0300. .

[2] U. Brandes and T. Erlebach, editors. *Network Analysis: Methodolgical Foundations*. Springer, Berling, DE, 2005.

[3] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. A budget-based algorithm for efficient subgraph matching on huge networks. In *Workshops Proceedings of the 27th International Conference on Data Engineering*, pages 94–99, 2011. . URL `http://dx.doi.org/10.1109/ICDEW.2011.5767618`.

[4] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[5] P. Dittrich and P. di Fenizio. Chemical organisation theory. *Bulletin of Mathematical Biology*, 69(4):1199–1231, 2007. ISSN 0092-8240. . URL `http://dx.doi.org/10.1007/s11538-006-9130-8`.

[6] O. Häggström. *Finite Markov Chains and Algorithmic Applications*. Cambridge University Press, 2002.

[7] O. Hartig. Reconciliation of RDF* and property graphs. Technical report, University of Waterloo, 2014. URL `http://arxiv.org/abs/1409.3288`.

[8] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[9] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995. ISSN 0018-9219. .

[10] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys*, 2015.

[11] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. Technical report, World Wide Web Consortium, October 2004. URL `http://www.w3.org/TR/rdf-sparql-query/`.

[12] M. A. Rodriguez. Grammar-based random walkers in semantic networks. *Knowledge-Based Systems*, 21(7):727–739, 2008. . URL `http://arxiv.org/abs/0803.4355`.

[13] M. A. Rodriguez. Mapping semantic networks to undirected networks. *International Journal of Applied Mathematics and Computer Science*, 5(1):39–42, February 2008. URL `http://arxiv.org/abs/0804.0277`.

[14] M. A. Rodriguez. *Emergent Web Intelligence: Advanced Semantic Technologies*, chapter General-Purpose Computing on a Semantic Network Substrate, pages 57–104. Advanced Information and Knowledge Processing. Springer-Verlag, June 2010. ISBN 78-1-84996-076-2. URL `http://arxiv.org/abs/0704.3395`.

[15] M. A. Rodriguez. The RDF virtual machine. *Knowledge-Based Systems*, 24(6):890–903, August 2011. URL `http://arxiv.org/abs/0802.3492`.

[16] M. A. Rodriguez and J. Shinavier. Exposing multi-relational networks to single-relational network analysis algorithms. *Journal of Informetrics*, 4(1):29–41, 2009. . URL `http://arxiv.org/abs/0806.2274`.

[17] J. Shinavier. Functional programs as Linked Data. In *3rd Workshop on Scripting for the Semantic Web*, Innsbruck, Austria, 2007.

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

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