CTA200H Final Project

Anatoly Zavyalov

May 14, 2021

1 Installation & Setup

The following steps outline how to install JanusGraph[1] along with the Apache Cassandra storage backend[2] and the Elasticsearch indexing backend[3], as well as setting up Gremlin-Python[4], which is used for querying the JanusGraph backend from a Python interface. The operating system used is **Ubuntu 20.04.2 LTS**.

1.1 Installing Java

JanusGraph is built on top of Apache TinkerPop[5], which, in turn, is built on top of Java and hence requires Java SE 8. The implementation of Java that we will install is OpenJDK 1.8. First, we refresh the list of available packages:

```
$ sudo apt update
```

Next, we install OpenJDK 1.8:

```
$ sudo apt install openjdk-8-jre
```

To verify that the correct version has been installed, we run `java -version`. We should see something similar to `openjdk version "1.8.0_292"`.

1.2 Setting the `$JAVA_HOME` environment variable

Next, we must set the `$JAVA_HOME` environment variable. First, we head to `/usr/lib/jvm/` and locate the intallation of the JDK. It should look similar to `/usr/lib/jvm/java-11-openjdk-amd64`. Next, we set the `$JAVA_HOME` environment variable to point to the installation of the JDK:

```
$ export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
```

We doublecheck that this is successful with `echo $JAVA_HOME`.

[1] https://janusgraph.org/
[2] https://cassandra.apache.org/
[3] https://www.elastic.co/elasticsearch/
[4] https://pypi.org/project/gremlinpython/
[5] https://tinkerpop.apache.org/

1.3 Setting up JanusGraph

From the JanusGraph Releases[6], we download the .zip of the "full" installation of JanusGraph (the file name should resemble `janusgraph-full-X.X.X.zip`, where `X.X.X` is the version number), and extract the contents. This "full" installation includes JanusGraph, as well as pre-configured Apache Cassandra and Elasticsearch.

From here, we start the JanusGraph server by running

```
$ bin/janusgraph.sh start
```

1.4 Connecting to JanusGraph

1.4.1 Gremlin Console

We can open the Gremlin console by running

```
$ bin/gremlin.sh
```

Next, we may create a remote connection to the JanusGraph server:

```
gremlin> :remote connect tinkerpop.server conf/remote.yaml
```

In order to use variables when remotely accessing the Gremlin server using Gremlin console, we can also connect to the server with a session:

```
gremlin> :remote connect tinkerpop.server conf/remote.yaml
   session
```

From here, we can send commands to the JanusGraph server by preceding them with `:>`. We can avoid this by running

```
gremlin> :remote console
```

which will enable sending all queries directly to the JanusGraph server and avoid the need of `:>`.

From here, we may instantiate a new graph by running

```
gremlin> graph = JanusGraphFactory.open("conf/janusgraph-
   cassandra-es.properties")
gremlin> g = graph.traversal()
```

to create a graph instance.

1.4.2 Gremlin-Python module

We can also access the JanusGraph server from a Python interface. First, we install the `gremlinpython` Python module by running

```
$ pip install gremlinpython
```

Note that a minimum `gremlinpython` version of `3.4.7` must be used for full functionality. Now, we may create a Python file to connect to and query the graph:

[6] https://github.com/JanusGraph/janusgraph/releases

```
# Import full gremlinpython functionality
from gremlin_python import statics
from gremlin_python.structure.graph import Graph
from gremlin_python.process.graph_traversal import __
from gremlin_python.driver.driver_remote_connection import
    DriverRemoteConnection

# Instantiate a Gremlin Graph clientside
graph = Graph()

# Connect to the server, instantiate traversal of graph. Note
    that the server is opened on port 8182 by default.
g = graph.traversal().withRemote(DriverRemoteConnection('ws://
    localhost:8182/gremlin','g'))

# Get the vertices of the graph as a list, and print them.
print(g.V().toList())
```

If a graph has already been instantiated in the JanusGraph server (see **2.4.1**), this code will print a list of the vertices of the graph (which, by default, is the empty list).

2 Timestamp System

One of the more desired functionality required for the HIRAX project involves determining the graph's structure at some point in time. Specifically, we wish to know, at a specific date and time, whether two vertices were connected and what the properties of the elements represented by those vertices were. Unfortunately, neither the core Apache TinkerPop nor JanusGraph support such complex timestamp queries. In the following subsections, we describe methods to implement such functionality.

2.1 Determining Connections Between Vertices

In a practical use case, users would be able to connect and disconnect vertices in the graph at any time. To implement functionality described above, we will record the time that the connection was changed, as well as whether the vertices have been connected or disconnected. Every time that two vertices are connected and disconnected, we create a new *connection* between the two vertices. Each connection will have a *start time* and an *end time*. When a connection between two vertices is created, its start time is set to the time that it was created, and the end time is set to infinity. When a user disconnects the two vertices, the latest connection's end time is set to the time the user disconnected the vertices. When a user decides to connect the two vertices again, a new connection is created with a new start time and a set time set to infinity, and all previous connections remain for when the graph is to be queried at a past time.

For example, for two vertices A and B, consider the following user queries:

- **Time:** 0. **Query:** Connect vertices A and B.
- **Time:** 1. **Query:** Disconnect vertices A and B.
- **Time:** 2. **Query:** Connect vertices A and B.
- **Time:** 3. **Query:** Disconnect vertices A and B.
- **Time:** 4. **Query:** Connect vertices A and B.

One possible implementation would be, for each connection, to form a new *edge* between the vertices. Each edge will contain the connection information, namely the start and end times of the connection. For example, after the queries above, the following illustration represents the edges between vertices A and B:

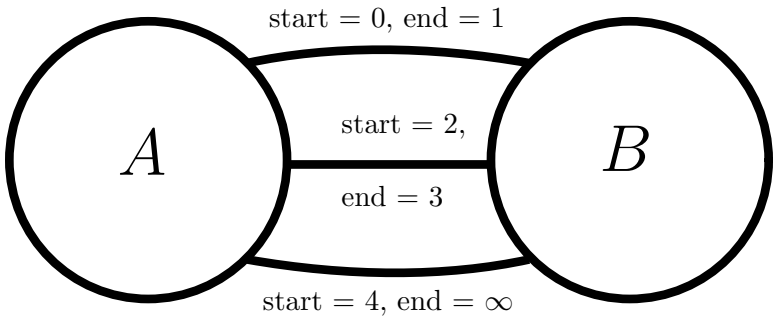

Figure 1: Connections between two vertices along with timestamps for start and end times

When determining whether two vertices were connected at time t, it would be very easy to iterate through all edges and determine whether a connection existed such that t fell in between the start and end times for the connection. If so, then the vertices were connected at this time. Otherwise, no such connection existed.

Such an implementation would trivial in Gremlin: simply filter the edges between two vertices to determine whether an edge exists such that t falls in between its start and end times. The main downfall of this technique is that every edge must be iterated through in order to determine whether a connection existed at some time, resulting in a worst-case time complexity of $\mathcal{O}(n)$, where n is the number of edges, implying that, as more and more connections are created and broken, querying the graph with a timestamp would get slower over time.

Another implementation could be to simply have one edge between two vertices with a single list property. The (sorted) list will contain two-tuples of form $(time, status)$, where *time* denotes the time when the connection was changed, and *status* is a Boolean representing whether a connection was formed or broken. Hence, the above queries would be represented in the list as such:

$$[(0, \text{True}), (1, \text{False}), (2, \text{True}), (3, \text{False}), (4, \text{True})]$$

Given a time t, one may perform a simple binary search on the first elements of the tuples to determine the most recent change that occured in the connection. If the second

element of the specified tuple is True, then the connection existed at time t, and did not exist otherwise. The binary search algorithm has a worst case time complexity of $\mathcal{O}(\log n)$, where n here denotes the number of changes made to the connection between the two vertices. This, asymptomatically, is much faster than the previous implementation. However, it is not trivial to construct such a complex query in a single Gremlin command. Hence, the work may be outsourced to a Python script or a Gremlin script to perform the binary search and return the value. We also find that it would be more efficient to construct this script in Gremlin, as querying the JanusGraph server from Python takes a substantial amount of time when compared to a native Gremlin query (explored in the next section).

2.2 Storing Past Properties of Vertices

Similarly to edges, we wish to know the history of properties of vertices in the graph. Specifically, every time a change is made to a vertex, we wish to store this new change along with all previous properties of the vertex, then, given a timestamp to query on, we retrieve the properties of the vertex at that date and time.

A possible approach is to store the properties in vertices adjancent to the vertex of the element we are looking at. Every time the element is changed, a new vertex is created with its new properties along with a timestamp, and connected to the vertex by an edge. For example, consider the following queries on vertex A:

- **Time:** 0. **Query:** Set the "active" property of A to **True**.
- **Time:** 1. **Query:** Set the "active" property of A to **False**.
- **Time:** 2. **Query:** Set the "active" property of A to **True**.

The following diagram illustrates the adjacent property vertices:

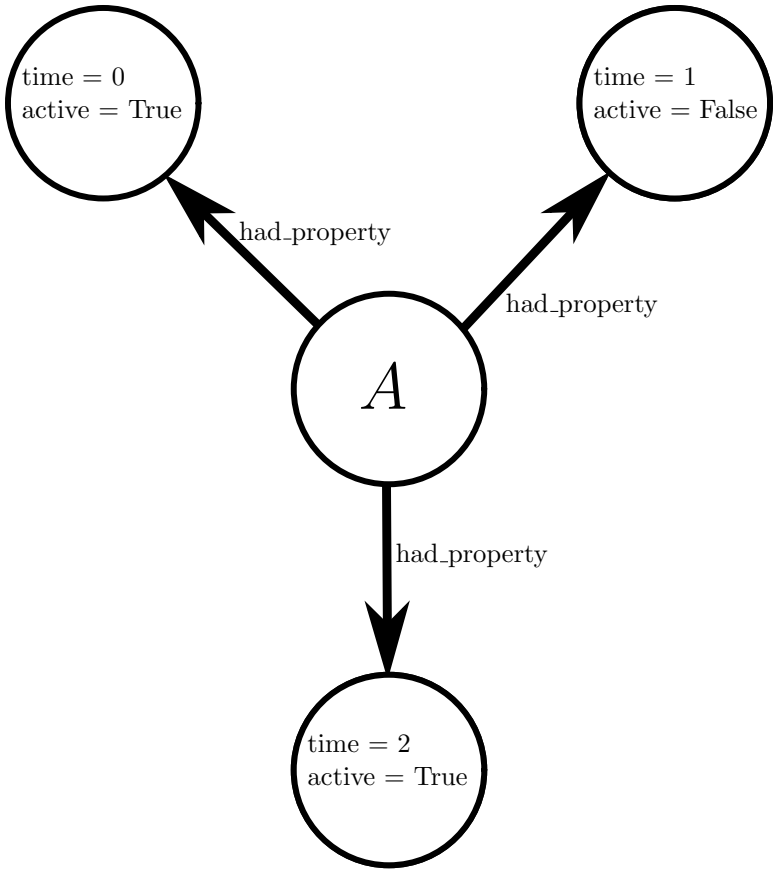

Figure 2: Storing properties as adjacent vertices

Such an implementation would be highly structured, and allow quick returning of all past vertices using built-in Gremlin traversals.

To determine the most recent property given a timestamp, a process similar to the second method in the previous subsection could be performed, involving a binary search to return the most recent property and query the corresponding property vertex to access the properties of the element at this timestamp. Since such complex operations cannot be performed in one Gremlin command, a Python script (implemented in **property_vertices.ipynb**) or a Gremlin script (which would be faster) could be used to make the query.

3 Benchmarking

Not surprisingly, we find that sending queries to the JanusGraph server from the Python interface takes much longer than performing the same queries in native Gremlin. Suppose that we wish to query the number of vertices in our graph 100 times and find the average time it takes to do so. If we were to do so in Python, we would get:

```
>>> Use the timeit module for querying the graph
>>> from timeit import Timer
>>> t = Timer("g.V().count().next()",'from __main__ import g')
>>> t.timeit(number=100) / 100
0.01622656100000313
```

This means that, every query of the graph from Python takes ∼16ms (based on the `timeit` documentation[7], `timeit.timeit` returns the time in seconds). When the same query is run from the Gremlin console, we get:

```
gremlin> clock(100) {g.V().count()}
==>0.006461
```

Note that based on the implementation[8] of the `clock` method in Gremlin, this is in milliseconds! Thus, we find that querying the graph from Gremlin is approximately *2500 times faster* than from Python (of course, this is assuming that there is no additional overhead from using Python's `timeit` module). While the Python queries are not slow (for practical purposes), this shows that queries written in Gremlin are clearly much faster.

4 Scaling

In practice, we are more concerned about paths of vertices than individual ones. Specifically, we often wish to know what components a signal travelled through as it was received by an antenna and travelled through the graph to the rack of GPUs and FPGA boards where it is then processed. Apache TinkerPop provides pre-built Gremlin "recipes" to perform common complex traversal queries[9]. The main recipe that we are interested in for the HIRAX project is the "shortest path" recipe, which returns the shortest path of vertices that connect two specified vertices. The vertices in this path can then be queried with a specific time to determine the properties of the components and their connections in the path at the time an experiment was being conducted. To minimize the number of calls to the database from the Python interface (as we saw is slow), as much of the work as possible must be done by Gremlin before being returned to the Python interface. Hence, prewritten Gremlin queries will be used to handle these common complex traversals that will be used in the HIRAX project that are not offered out-of-the-box by JanusGraph and Gremlin.

[7] https://docs.python.org/3/library/timeit.html

[8] https://github.com/apache/tinkerpop/blob/7c1a48538837a56b4a1e5b73bb9dfb77e0e93575/gremlin-core/src/main/java/org/apache/tinkerpop/gremlin/util/TimeUtil.java

[9] https://tinkerpop.apache.org/docs/current/recipes/