Analyzing the Family Tree

Daniel W. Rapp
FamilySearch
50 E North Temple St.
Salt Lake City, UT
rappdw@familysearch.org

Michael P. Jones
FamilySearch
50 E North Temple St.
Salt Lake City, UT
jonesmp@familysearch.org

ABSTRACT

FamilySearch holds one of the largest collections of linked family history data in the world. Nearly one billion records of individuals, both deceased and living, have been recorded and placed together into a common tree ("The Family Tree"). The study of this ancestral relationship graph consists of the largest family history network ever analyzed. We have found a number of interesting properties in the network using common graph analysis techniques.

We examine the topology of the graph by calculating the connected components within the graph. The total network consists of one giant component consisting of many millions of records plus millions of very small components. We also describe how this topology has changed over time. The paper further describes how an analysis of the strongly connected components and the graph's diameter can be used to assess the quality of the data. Finally, we describe a heuristic algorithm to determine the "connectedness" of our patrons and find that those who have logged into the system are significantly more connected than those that have not. One third of the potential users are connected to the giant component while 80% of the active users are. We discuss how this analysis could potentially be used to partition the graph to support scaling or distributing the system.

Categories and Subject Descriptors

G.2.2 [**Mathematics of Computing**]: Numerical Analysis—*Graph Theory*

General Terms

Family History, Relationship Graph

1. INTRODUCTION

In 2002, FamilySearch began a project to collect all of its family history data sources, detect duplicate records (i.e., those that identified the same individual across these various data sources), and combine the result into an authoritative view of the family history information that has been collected over the course of many years. The goal was to compile and curate the common pedigree of mankind. The result of this effort is known as the "Family Tree" which represents the largest lineage-linked record of the history of mankind – currently containing approximately one billion records.

To date, the Family Tree has only been available to selected users, although efforts are underway to make it publicly available. This user base is referred to as "patrons" of the system. Not all users have taken advantage of this system; those that have done so are referred to as "active patrons." The Family Tree continues to grow and evolve through contributions by active patrons and internal processes.

The resultant data set is a rich source of information. Because it is a pedigree the inclination might be to model this data as a tree. However, the complexity of the data set suggests that it is more properly modeled as a graph. The complexity in the data arises from two primary reasons. Human relationships are often more complex than a single set of parent-child and spousal relationships, e.g. adoptions, divorces, etc. Also, discrepancies have been introduced by algorithmic and human error while combining records. In the graph model, the vertices represent individuals and the edges represent different relationships (e.g. spousal, parent child, etc.) between individuals.

With this relationship graph in place, it becomes interesting to utilize some of the standard tools in network analysis to examine family history data. In this paper, we present the methodology and system employed to examine the family history network. We present findings that are of interest both from a genealogical perspective and from the perspective of a computer scientist.

2. METHODOLOGY/PROCESS

The family tree is stored in a large RDBMS system. The data model for the family tree is beyond the scope of this paper, but suffice it to say that the relationships are modeled much as you would expect. Namely there is a relationship table that contains the relationship type and the unique identifiers of the records participating in the relationship. Very early on, we determined that performing graph analysis against the database was simply too expensive (in terms of time spent waiting for the DB to respond). We began development of a distributed graph service consisting of a number of commodity class servers that each held a portion of the graph in memory. We then implemented

RootsTech Technology Workshop February 2012, Salt Lake City, UT
.

distributed algorithms to compute various graph properties [7, 6, 3]. This approach proved to be an effective representation of the graph, particularly for drawing pedigrees and calculating relationships between individuals.

However, for many of the analyses we wanted to perform (namely those described in this paper), it became evident that the machinery involved in working with a distributed graph was simply too heavy-weight to allow for effective analysis. In fact, all that was needed for much of this analysis was simply the graph structure. We were introduced to the WebGraph [2] framework in late 2008. WebGraph provides a highly optimized graph representation that allows us to represent our nearly one billion node graph and nearly one and a half billion relationships on a single 8-core machine with 16GB memory. All the analyses described in this paper were performed on that one machine.

Many of the algorithms we use are the standard algorithms for computing various graph properties and most are available as part of WebGraph. We have modified a few of the algorithms to run in the memory constraints imposed by the hardware we are using. We have also developed a few algorithms that are unique to the application of network analysis to the realm of family history. (We have included a modification to Tarjan's algorithm that uses less memory and our heuristic diameter approximation algorithm in the Appendix.)

We construct seven distinct graphs for different types of analysis. All graphs have the same set of vertices but different edges.

Graph Designation	Description of Edges	Size
C	directed, child to parent	1.7 GB
P	transpose of C	1.7 GB
CP	union of C & P	4.0 GB
H	directed, husband to wife	0.7 GB
W	transpose of H	0.7 GB
HW	union of H & W	1.2 GB
U	union of CP & HW	4.0 GB

Table 1: Distinct Graphs

Because our relationships (child to parent, husband to wife) are directed, we find it useful for some analysis to use the transpose of the "natural" direction and to use the union of a graph and its transpose to simulate an undirected graph. Thus the three union graphs may be though of as undirected graphs and the other graphs as directed graphs.

3. RESULTS

3.1 Connected Components (Distinct Trees)

A graph can be partitioned into components such that each vertex within a component can be reached from all other vertices in that component by following **undirected** edges. This is illustrated in Figure 1 and Figure 2.

One of the stated purposes of FamilySearch is to provide a common pedigree to facilitate collaborative work on family history. A key factor on whether this is facilitated can be measured by how many disjoint trees there are in the system. If there is a single common tree then all additions can be effectively shared. If patrons are working in disjoint trees then the collaborative possibilities diminish. The number of

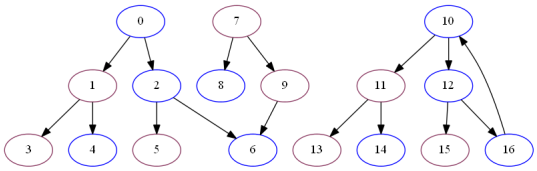

Figure 1: Sample Family History Graph

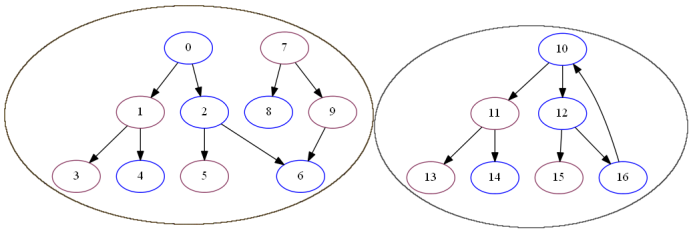

Figure 2: Connected Components of Graph

disjoint trees can be identified by computing the connected components of a the graph.

Tarjan's algorithm is the standard algorithm for computing connected components. There is an implementation of Tarjan's algorithm in WebGraph. Unfortunately, this implementation will exhaust memory on the hardware we have available (16 GB main memory). In order to compute the connected components, we have implemented a slight variation on Tarjan's algorithm that requires less memory at the expense of more execution time. (See Appendix A)

In order to identify the connected components of the family tree, we use the U graph as input. We find that there is a giant component that consists of about 20% of the vertices in the graph. Additionally there is a large number of trees of size 1, 2, or 3. Collectively, these trees hold approximately 63% of the vertices. This is largely an artifact of the extraction projects in which vital records are indexed and added to the Family Tree. These disjoint trees are an indication of work to be completed; specifically, they need to be joined into the common tree. Finally, there is a large number of trees that hold anywhere from 4-20k records. The remaining 17% of the vertices are in trees of these sizes. Figure 3 shows the distribution of the number of trees of given sizes.

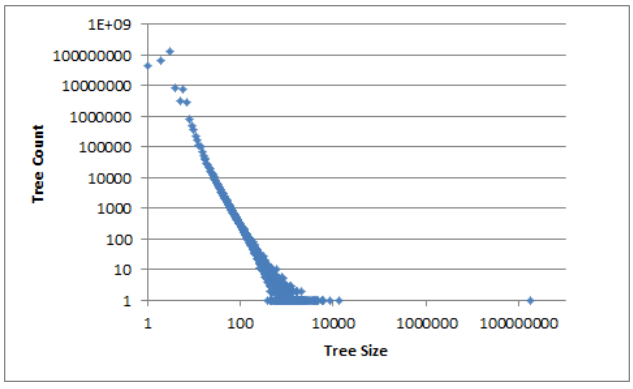

Figure 3: Tree Size Distribution

Figure 4 shows how the percentages of records contained

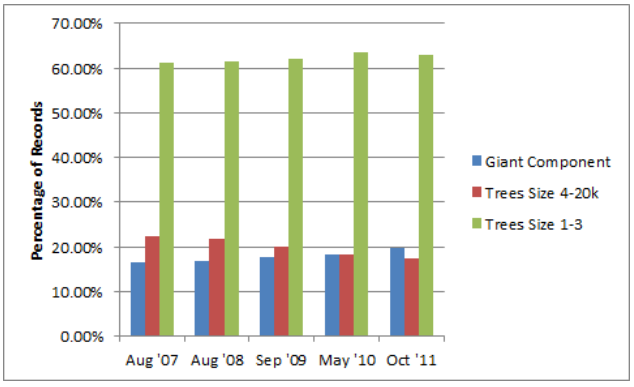

Figure 4: Changes in Trees Over Time

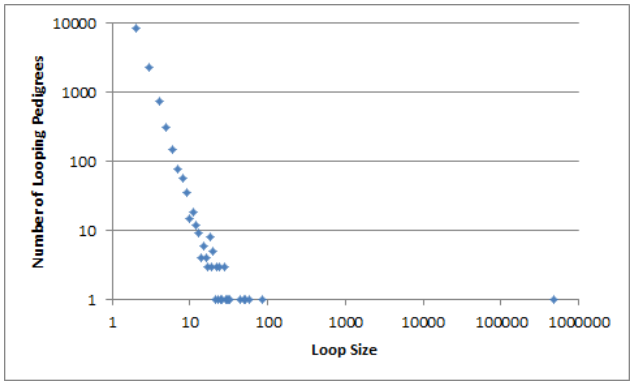

Figure 6: Distribution of number of looping pedigrees against size of loop

within various tree sizes have changed over time. It is significant to note that the giant component has increased from holding 16% of the records to holding 20% of the records. This is a desired result in the system in that it is indicative of better collaboration.

3.2 Strongly Connected Components (Looping Pedigrees)

Another property of graphs is the identification of strongly connected components. These are the components of the graph in which each and every vertex can reach all other vertices within that component by following **directed** edges, as depicted in Figure 5.

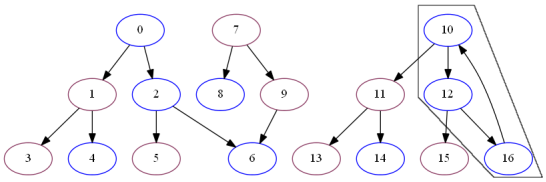

Figure 5: Strongly Connected Components of Graph

If we run Tarjan's algorithm against the C graph instead of the U graph, we identify pedigrees that are looping or tangled. Obviously it is impossible for me to be my own grandpa in a biological sense, but it is possible in the broader context of the graph (when step-parent relationships are considered as in the old song "I'm my Own Grandpa" [10]). It is also possible due to human or algorithm error as trees and/or records are merged and invalid or impossible relationships are introduced.

Running the algorithm identifies a moderate number of looping pedigrees with the number of records participating in looping pedigrees following a power law like distribution with an extremely long tail as shown in Figure 6.

The number of records involved in looping pedigrees is minimal (0.06%), but has been increasing. (An increase from 0.05% in June of 2010 to 0.06% in October of 2011.) For the most part, we would not expect loops in the relationship graph and thus the analysis gives insight into the quality of the data.

3.3 Diameter Analysis

The diameter of the graph is the number of arcs between the two vertices in the graph that are furthest apart. Computing the diameter of a large graph is an expensive (execution time) operation and therefore not practical. Typically heuristics must be used to approximate the diameter of large graphs to make the calculation computationally tractable. While we examined the heuristics developed by other researchers[5], we found that the nature of our data leads to a novel heuristic.

Our patrons are represented as records in the common pedigree. Because these records represent currently living individuals, they are likely to be on or very near an edge of the graph. Therefore, it seems reasonable that at least one of those records would likely be on the graph's diameter. Starting with an input set of patron records, concurrently crawl back from that set in the C graph. The longest path encountered in this method is an approximation of the diameter. (A pseudo code implementation of this algorithm is found in Appendix B.) This is only an estimate of diameter as we can not ensure that the diameter runs from one of the vertices in the input set. Additionally, in order to speed computation, we short circuit retrying partial paths that have already been encountered without regard to whether they may indeed be part of a longer chain from the one which they were initially computed.

This heuristic yields a number of furthest ancestor records. This set of records can in turn be used as input to the algorithm, however using the P graph rather than the C graph. Performing this extra step reveals several things. First, our heuristic was a good estimator. Second, there are longer paths to be found because neither end of the diameter is actually in our initial input set.

Using our algorithm, we estimate the diameter of the giant component of the graph as 286. The diameter of the entire graph comes from one of the disconnected trees and is 400+. It represents someone's private conjecture of Hawaiian royalty with ancestry of which most predates historical records and some predates biblical records.

In examining some of the longer paths found in the graph, we find that this analysis is a good detector of apocryphal or even fictional data that has been added to FamilySearch.

3.4 "Connectedness" of Patrons

One of our early applications of graph analysis was the exploration of algorithms to efficiently calculate how two individuals are related. While that work is beyond the scope

of this paper, we did notice an interesting property. Many cases showed surprisingly close relationships. In almost all cases common ancestors were found within a 12 generation horizon of the input parameters. While some of this was due to data quality issues, [1] many of the discovered relationships were indeed valid. In many ways this is similar to some of the other "degrees of separation" experiments [1, 8] that have been conducted.

We conducted an analysis to see how rapidly our patron set converges into a common "extended family". The algorithm employed is:

1. Begin with a set of vertices determined by some heuristic. These are the partition sets at generation n_0. This partitioning will not cover the entire graph. At this point, each vertex represents a partition that contains only itself.

2. For each partition, take all vertices in that partition and follow all out edges, adding encountered vertices to that partition. If another partition is encountered, merge the partitions.

3. The partitions resulting from this become the input set for the next iteration (partition sets at generation $n_0 + 1$) and represent a partial partitioning of the graph.

4. Repeat for n iterations (generations).

There are a number of useful properties in this analysis including: the number of partitions of the graph at each step; the number of total elements covered by the partial partitioning at each step; the percentage of the initial seeds that reside in a given partition. Both the rate of reduction of partitions and the extent of reduction of partitions indicate the degree of "connectedness" of the input seed set. The total number of elements covered indicate the connectedness to the heart of the graph at large. The percentage of initial seeds that reside in the same partition is an indication of how well this heuristic works at uniformly partitioning the graph. Ideally, the input seeds would be evenly distributed among partitions after n steps.

To compare how rapidly our patron set converges into a common "extended family", we begin with two different initial partition sets. The first is a set of vertices randomly selected from the graph. The second is the set of patrons of the system. We ran this algorithm over both the entire graph and over just the giant component. We also ran the analysis over the C graph and the U graph. We present results from running over the giant component of the C graph for brevity and as the results were similar. Also, the results from the U graph represent results similar to social networks as the addition of spousal links tend to join disparate "extended families" resulting in both a greater and quicker convergence as well as a significantly higher coverage of the graph.

Figure 7 shows a significant difference in how quickly patrons merge into "extended families" when compared to a randomly selected sample of initial seeds. Another way of looking at the data is to determine what percentage of the seed set resides in the largest "extended family" at each iteration as indicated in Figure 9.

[1]As pointed out in the details of the strongly connected component section, i.e. anyone that connects into one of the strongly connected components of the graph will be related to all others that also connect into that component

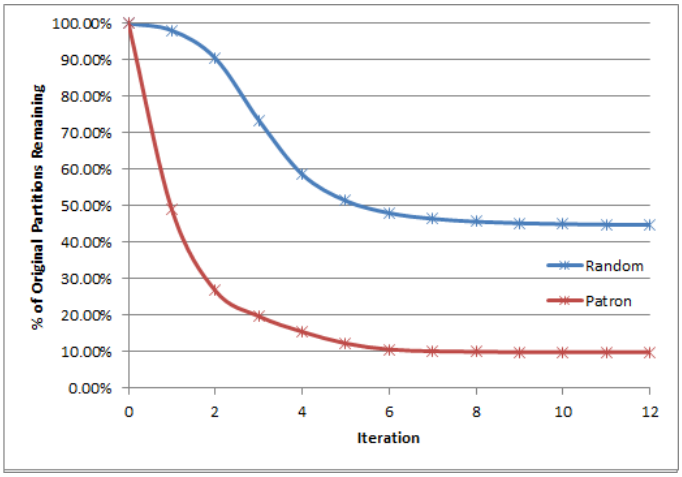

Figure 7: Number of Partitions (Giant Component)

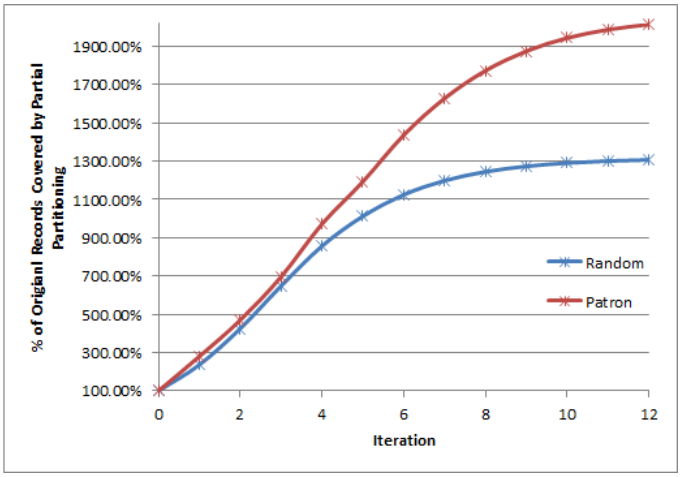

Figure 8: Number of Records in Partitions (Giant Component)

Having nearly 75% of our patrons within the same "extended family" within 4 generations shows a remarkable level of connectedness.

3.4.1 *Application: Graph Partitioning*

A potential interesting area of application for this analysis is whether or not this would facilitate a partitioning of the graph. Graph partitioning is interesting because many web-scale applications use data partitioning for providing better scale characteristics.

We can examine an "extended family" scheme, for its suitability as a potential partitioning strategy that would support multi-generational display. Four generations is a common limit for multi-generational display. Ideally, such a partitioning would be uniform. While seeding the algorithm with random seeds yields promising results, the high degree of "connectedness" of our patrons works against this. If we were to use strictly the "extended family" approach to partition the graph for a 4 generation display, we would find that just over half of our active patrons would be in the same partition. Relying solely on this heuristic does not

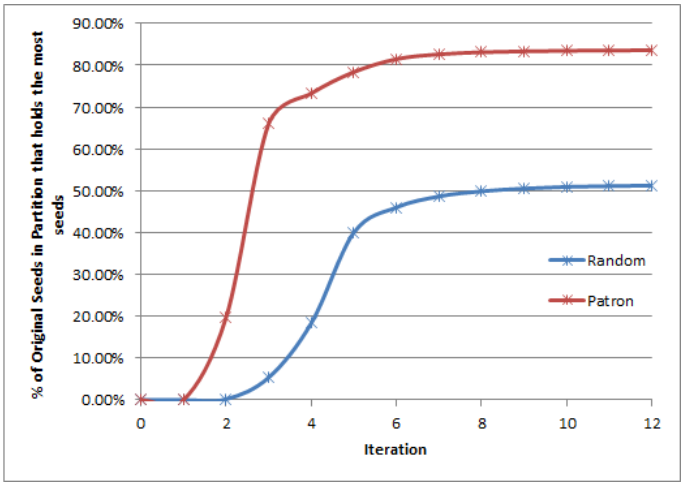

Figure 9: Percentage of Seeds in Largest Extended Family

yield a uniform partitioning with 65% of the initial patron seeds ending up in the same partition after 3 iterations. (See Figure 9) Thus we could at best hope for only a 2x scaling capability.

4. CONCLUSIONS

The idea of applying graph analysis to the study of human relationships is not new, but has generally focused on social relationships rather than familial relationships[8, 1, 9]. While connected component analysis has similar meaning and importance in both contexts, strongly connected components and diameter analysis provides unique insights into the familial relationship graph. Not surprisingly the connected component analysis of the Facebook social network demonstrated similar distribution of component sizes as did the analysis of the Family Tree as they both have small world graph properties, e.g. high degree of connectedness, small diameter compared to number of vertices, etc.[4]

In social networks, strongly connected components are expected or even desired. In a lineage network, strongly connected components are neither desired nor represent a possible truth state in general. We have shown that easily computed graph properties may be used to identify quality issues in genealogical data. Additionally, we have shown how graph analysis can verify that our system is achieving one of its stated purposes, namely curating a common family tree.

We applied a heuristic approach to partition the graph to look at system scalability design goals. We found that this heuristic would only allow us to bifurcate the working set, thus yielding roughly a 2x improvement in scale. This result alone does not show much promise. Further investigation may be warranted to determine whether this approach, together with other heuristics may prove fruitful for our application. Nevertheless, this analysis did provide insights into how closely connected our patrons are within the Family Tree.

5. REFERENCES

[1] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. Four degrees of separation. `http://arxiv.org/abs/1111.4570`, November 2011.

[2] P. Boldi, M. Santini, and S. Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2):33–38, 2008.

[3] J. Bondy. *Graph theory*. Springer, New York, 2008.

[4] M. Buchanan. *Nexus : small worlds and the groundbreaking science of networks*. W.W. Norton, New York, 2003.

[5] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Fast diameter estimation and mining in massive graphs with hadoop. `http://reports-archive.adm.cs.cmu.edu/anon/ml2008/CMU-ML-08-117.pdf`, 2008.

[6] D. Knuth. *The Art of Computer Programming, Volume 4a: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, Reading, 2011.

[7] N. Lynch. *Distributed algorithms*. Morgan Kaufmann Publishers, San Francisco, Calif, 1996.

[8] S. Milgram. The Small World Problem. *Psychology Today*, 2:60–67, 1967.

[9] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. `http://arxiv.org/abs/1111.4503`, November 2011.

[10] Wikipedia. I'm my own grandpa. `http://en.wikipedia.org/wiki/I'm_My_Own_Grandpa`.

APPENDIX

A. ALGORITHM FOR DETERMINING CONNECTED COMPONENT

```
BitSet verticesThatHaveBeenProcessed = new BitSet(graph.numNodes());
Deque<Integer> verticesToProcessForCurrentComponent = new ArrayDeque<Integer>();
for (int i = 0; i < graph.numNodes(); i++) {
  if (verticesThatHaveBeenProcessed.get(i)) {
    continue;
  }
  verticesToProcessForCurrentComponent.add(i);
  verticesThatHaveBeenProcessed.set(i);
  int componentSize = 1;
  // ... identifying a new component
  while (verticesToProcessForCurrentComponent.size() > 0) {
    final int vertex = verticesToProcessForCurrentComponent.removeFirst();
    // ... record the fact that vertex belongs to current component (if desired)
    final LazyIntIterator successorIterator = graph.successors(vertex);
    int successor;
    while ((successor = successorIterator.nextInt()) != -1) {
      if (!verticesThatHaveBeenProcessed.get(successor)) {
        verticesThatHaveBeenProcessed.set(successor);
        verticesToProcessForCurrentComponent.add(successor);
        componentSize++;
      }
    }
  }
  // ... identified component of size: componentSize, record this fact (if desired)
}
```

B. ALGORITHM FOR DIAMETER ANALYSIS

```
class TraversalContext {
  final int vertexId;
  final BigInteger pathLength;
  TraversalContext(int vertexId, BigInteger pathLength) {
    this.vertexId = vertexId;
    this.pathLength = pathLength;
  }
  public int getVertexId() {
    return vertexId;
  }
  public BigInteger getPathLength() {
    return pathLength;
  }
}

BigInteger diameterApproximation = BigInteger.valueOf(Long.MAX_VALUE);
BitSet visitedNodes = new BitSet(graph.numNodes());
Deque<TraversalContext> workQueue = new ArrayDeque<TraversalContext>();
// ... initialize the workQueue with the vertices that are in the input set
while (workQueue.size() > 0) {
  TraversalContext context = workQueue.removeFirst();
  if (visitedNodes.get(context.getVertexId())) {
    continue;
  }
  visitedNodes.set(context.getVertexId());
  diameterApproximation = context.getPathLength().min(diameterApproximation);
  LazyIntIterator successorIterator = graph.successors(context.getVertexId());
  int successor;
  while ((successor = successorIterator.nextInt()) != -1) {
    final BigInteger newPathLength = context.getPathLength().add(BigInteger.ONE);
    workQueue.add(new TraversalContext(successor, newPathLength));
  }
}
// ... at this point, diameterApproximation contains our best guess at the diameter
```