1. There are no known bugs or problem with the code.
2. High Level Descriptions

DiskMultiMap class:

The DiskMultiMap class had its data stored in a BinaryFile which was used to create a hash table that was written onto disk. This hash table consisted of the number of buckets that were passed in when each DiskMultiMap was created. Each of these buckets held a binary offset value which were initialized to -1 to indicate that they did not lead to any other location on the disk. When something is inserted into the DiskMultiMap, a hash function is used and the modulus operator is applied to it of the number of buckets in the DiskMultiMap and then this value is multiplied by the size of a binary offset and then reading this value results in the binary offset value that this bucket essentialy “points” to (which could be -1 indicating nowhere). Through this hash table structure, which is stored entirely on disk, with a given key, all associations with this key, a struct containing both its value and context can essentially be found in constant time. Each data struct, named MultiMapNode contains a key , value, context, and binary offset value. Key, Value, and Context are all stored as c-strings so that they can be written to the disk. The binary offset value is essentially the “next pointer”, so that each bucket in the hash table points to a linked list of nodes (of any size 0 or greater). Therefore given a key, a linked list of all Nodes that contain that key can be accessed in constant time. There are only a few RAM data structures that are stored in the DiskMultiMap class, the offset of the first free space on the disk to write new nodes to, the offset of the list of previously erased nodes which takes priority over expanding the disk when writing to the disk, and the number of buckets of the hash table. These three data structures are written at the very beginning of the disk file and they are read off and copied to the RAM data structures, which are just binary offset types, when the disk is opened, and when the disk is closed, these values are written to those offset locations. This way if the disk is closed and reopened by a different DiskMultiMap class then it will be able to perform all functions as if it were the same MultiMap

**Bool DiskMultiMap::insert(string& key, string& value, string& context)**

If any of the parameter strings are too long

Return false

Generate a hash value for the key string and read the bucket that the key hash value corresponds to

If the bucket is empty

If there are previous freed nodes to reuse

Set the bucket to point to that node and write the parameters there

Update the freed nodes list

If there are not

Set the bucket to the next free location on disk and write parameters

Update the next free location

If the bucket is not empty

Access the node the bucket points to, traverse to the end, if there is a free node to reuse, set the last node in the list to point there, if not, create the node in the furthest free area on the disk

Return true

**Iterator DiskMultiMap::search(const string& key)**

Generates hash value for the key parameter

Read the bucket that the hash value corresponds to

If the bucket is empty

Return an invalid iterator

Create a queue of MultiMapTuples

Loop through all the nodes that contain the parameter key string

Access the key, value, and context of them and store into a MultiMapTuple

Push that MultiMapTuple into the queue

If the queue is empty

Return an invalid iterator

Return a valid iterator with the queue passed in as a parameter

**Int DiskMultiMap::erase(string& key, string& value, string& context)**

Generate hash value for the key string

Read the bucket that the hash value corresponds to

If the bucket is empty

Return 0

Loop through the list of nodes with the corresponding key

If all of the values in the node math the parameters, remove it from the list updating the bucket and previous nodes if necessary

Increment the number of removals

Return the number of removals

Iterator class:

The iterator class stores a boolean representing if it is in a valid state or not. A singe multiMapTuple which stores all the values of the current node it is pointing to, a c-string representing the current key of its node, as well as a queue of multiMapTuples that was passed using one of the overloaded constructors. Because the iterator itself is unable to read and write to the disk, the iterator must be able to access all of the nodes of a specific key, therefore when it is constructed, the search function of the DiskMultiMap class adds the data from all of the nodes in the linked list of a specific key and puts it all into a queue for the iterator to be able to iterate through itself, without needing to access the disk once it is created.

Iterator& Iterator::operator++()

If the iterator is not valid

Return the current iterator

If the queue of associations is empty

Set the iterator to false

Return the current iterator

Set the parameters of the MultiMapTuple to the front of the queue of MultiMapTuples

Pop the queueOfAssociations

Return the current iterator

IntelWeb class:

The intelWeb class only has two member variables, two DiskMultiMaps. One diskMultiMap consists of all the log files read in the form of mapping the source to destination with sources as keys and destinations as values. The other multimap is the opposite order with destinations as keys and sources as values. This is done so that for a given entity, both multimaps can be searched to find all the instances of this entity as both a source and a destination of any associations.

**Bool IntwelWeb::ingest(string& telemetryFile)**

Open the file for input

If the file can not be opened

Return false

Reads each line and checks that the formatting fit what is being searched

If the format does not match, the line is skipped

Insert into the sourceToDestination multiMap in the form (key, value, context)

Insert into the destinationToSource multiMap in the form (value, key, context)

Return true

**Unsigned int crawl(string& indicators, unsigned int prevalenceToBeGood, vector<string>& badEntitesFound, vector<InteractionTuple>& interaciotns)**

Starts by generating various data structures that are used to facilitate storing the information

Creates two sets to store the bad entities as well as interactions

A set is used because it performs two important tasks that are necessary for the bad entities and interactions, they are sorted alphabetically (the InteractionTuple has its comparison operators defined) as well as each element in the set is unique

A queue of strings representing the malicious associations, this is used to perform a breadth-first search through all the connected files that are marked as malicious

A set of strings representing the known “good” entities (those whose prevalence is above the threshold) which is an instance of caching, so that we don’t need to keep checking the prevalence if we’ve already done it once.

Pseudocode:

Push all the elements in the indicator vector into the queue

While the queueOfMalicious associations is not empty

The top value of the queue is taken and stored as the current malicious entity

The top value is popped off

If the value is present in the knownGoodEntities set

Remove it from the Malicious Entities Set

Skip to the next iteration of the loop

Check the prevalence, if it is above the threshold

Remove it from the Malicious Entities Set

Add it to the known good entities set

Skip to the next iteration of the loop

Generate iterators for both multiMaps using the current malicious entity as a key

Loop through both iterators

If the value associated with the key is not in the bad entities set

Add it to the set

Push it onto the queue

Add the interaction to the set of interactions

Increment the iterator

If at least one association was logged, add the malicious entity into the set

Clear both resulting vectors, in case they weren’t empty to being with

Copy the values from the sets into their corresponding vectors

Return the size of the bad entities found vector

**Bool IntelWeb::purge(string& entity)**

Generate iterator through the sources to destination multi map

While the iterator is valid

Create a temporary MultiMapTuple and assign it the dereferencing of the iterator

Erase from the sourceToDestination map passing in (key, value, context)

Erase form the DestinationToSource map passing in (value, key, context)

Increment the iterator

Perform the exact same operation on the destination to sources multi map but flip the ordering for when erasing from the maps

Return if at least one entity was removed

3. Each of the methods satisfy the given big-O requirements. The ingest function takes a long time to operate when the number of entries becomes very large, I do not believe this is a big-O issue since I’m pretty sure that my implementation of ingest() has the right value but rather that writing and reading from the disk takes a long time.