Oral Roberts University

Senior Project

Final Report

New Graph Approach to Biovista Primitives

Prepared by

Andrew Westlund

With the Guidance of

Dr. Stephen Wheat

Submitted as partial fulfillment of the requirements for the degree of

Bachelor of Science, Computer Science

Contents

[New Graph Approach to Biovista Primitives 4](#_Toc133041531)

[Introduction 4](#_Toc133041532)

[Graph Properties 4](#_Toc133041533)

[Biovista’s Graph 4](#_Toc133041534)

[System Description and Overview 5](#_Toc133041535)

[Project Objectives 5](#_Toc133041536)

[Technical Architecture 5](#_Toc133041537)

[Attributes of the Graph Data Structure 5](#_Toc133041538)

[ vertex \*vList 6](file:///C:\Users\Andrewski\Documents\ORU%20Stuff\Senior%20Project\Final%20Report.docx#_Toc133041539)

[ UINT64 vertexCount 6](file:///C:\Users\Andrewski\Documents\ORU%20Stuff\Senior%20Project\Final%20Report.docx#_Toc133041540)

[ UINT64 totalDegree; 6](file:///C:\Users\Andrewski\Documents\ORU%20Stuff\Senior%20Project\Final%20Report.docx#_Toc133041541)

[ INT16 allocation; 6](file:///C:\Users\Andrewski\Documents\ORU%20Stuff\Senior%20Project\Final%20Report.docx#_Toc133041542)

[Methods of the Graph Data Structure 7](#_Toc133041543)

[Building the Graph 8](#_Toc133041544)

[Original Building of the Graph 8](#_Toc133041545)

[Improved Building of the Graph 8](#_Toc133041546)

[Binary Reading and Parallel Building of the Graph 9](#_Toc133041547)

[Building G2 10](#_Toc133041548)

[G2 Purpose and Description 10](#_Toc133041549)

[Naïve Building of G2 10](#_Toc133041550)

[Improved Building of G2 11](#_Toc133041551)

[Parallel Building of G2 12](#_Toc133041552)

[Incrementally Updating G and G2 13](#_Toc133041553)

[Incremental Updating Purpose and Description 13](#_Toc133041554)

[Implementation of Incremental Updates 13](#_Toc133041555)

[Performing the Primitives 14](#_Toc133041556)

[Primitives’ Purpose 14](#_Toc133041557)

[Expand 14](#_Toc133041558)

[Connect 15](#_Toc133041559)

[Bridge 16](#_Toc133041560)

[Bibliography 17](#_Toc133041561)

[Transaction Descriptions 18](#_Toc133041562)

[Results 18](#_Toc133041563)

[Memory 18](#_Toc133041564)

[Speed 19](#_Toc133041565)

[Population of G and G2 19](#_Toc133041566)

[Addition of Incremental Edges 20](#_Toc133041567)

[Primitive Performance 20](#_Toc133041568)

[Conclusion 21](#_Toc133041569)

[Potential Improvements to Be Made 21](#_Toc133041570)

[Sources Cited 21](#_Toc133041571)

# New Graph Approach to Biovista Primitives

## Introduction

Biovista’s Project Prodigy is an AI engine that processes a large graph of published papers to develop diagnoses and propose treatment plans for complex illnesses. The purpose of the project was to make a system that allows Biovista to update their data incrementally and rapidly without sacrificing time in their performance of primitives.

Biovista’s update process to incorporate new publications currently takes three weeks, meaning that at any given time, Biovista’s analytics are at least three weeks behind the most up-to-date data. Having the ability to rapidly add incremental data would allow a potential diagnosis or proposed treatment plan to incorporate the latest science.

It is important for medical data analytics to be accurate, and it would be good if this project’s results increase the currency of Biovista’s Project Prodigy so that they can develop more informed diagnoses and propose more relevant treatment plans.

## Graph Properties

A graph, G, can be defined as a union of a set of vertices, V, and a set of edges, E, that connect such vertices together: G = (V, E). These edges may be directed or undirected, and they may have weights. A vertex’s out-degree indicates the number of outgoing edges it has, and the in-degree indicates how many vertices have an edge pointing to it. For an undirected graph, the in-degree and out-degree are equivalent.

A bipartite graph splits up the set of vertices into two sets, V1 and V2. Vertices from V1 can have edges that connects to vertices from V2, but they cannot have edges that connect to other vertices within V1. The same is true for V2 edges.

## Biovista’s Graph

The graph provided by Biovista is a bipartite, directed graph. V1 is the set of published medical papers and V2 is the set of entities, or subjects, that those papers speak about to a certain extent. Thus, papers can point to entities but not to other papers, and entities can point to papers but not other entities. However, in Biovista’s graph, only the papers point to the entities and not the other way around. An example of such a graph is displayed in Figure 1.

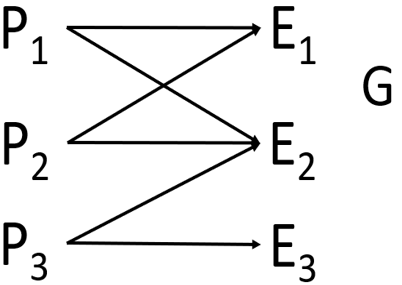


Figure 1:

A graph that consists of three papers and three entities. The paper P1 references both entities E1 and E2, while entity E3 is only referenced to by paper P3.

The graph that Biovista provides is a sequence of approximately 1.2 billion edges. Each edge has four elements: a paper’s identification number (paperID), the publication year for the paper (publicationYear), an identification number for an entity, or subject, that the paper speaks about (entityID), and an identification number to identify what type of entity the entityID is (entityTypeID). The order for each edge is paperID, publicationYear, entityTypeID, entityID; each element of the edge was tab-separated on the same line with the next edge being a line below, and all the edges were originally stored in an ASCII file.

## System Description and Overview

The system utilizes several programs, including a main .cpp file, a .cpp and .h file for the graph data structure, and a .cpp and .h file for a helper AVL Tree structure. The data is read in and turned into a graph, G; this process can be done through the use of one or more threads. G is then converted into G2, and this process can also be done through the use of one or more threads. Data can be incrementally added to G2. Biovista’s primitives can then be performed on G2.

## Project Objectives

The primary goals of this project were to create a program that could build G and G2 in less than a week, for the incremental edges to be implemented into G and G2 in less than 10 milliseconds per paper, and for the primitives’ performance times to be comparable to Biovista’s current methods.

## Technical Architecture

The architecture of the system needs only one node to store G and G2. The program that incrementally updates G and G2 and performs primitives on the graphs is able to run on a single node. The system will be stand-alone on Titan. Future versions of this project could perhaps convert G to G2 on multiple nodes and store G and G2 on multiple nodes. The distribution of G and G2 across multiple nodes is comparable to the University of Illinois at Urbana-Champaign’s large sparse system “distributed across np processes such that each process holds a contiguous block of rows from the matrix, and equivalent rows from the vectors” (Bienz); their article also details alternative ways that it could be stored, such as to “split the rows of [the sparse matrix] on a single process into two groups: an on-process block, containing the columns of the matrix that correspond to vector values stored locally, and an off-process block, containing matrix non-zeros that are associated with vector values that are stored on non-local processes” (Bienz). It is probable that the graph structure is more compatible with the first method, and thus G and G2 could be stored as contiguous blocks on multiple nodes in the future, but since the University of Illinois’ team’s results “show that the communication time dominates the computation as the number of processes is increased, thus decreasing the scalability” (Bienz), the optimal number of processes for the most efficient time- and space-complexity should be found through testing.

### Attributes of the Graph Data Structure

The system heavily depends on the graph engine.

The graph consists of:

vertex count, which stored how many vertices are in the graph (should be the maximum number of entities + the maximum number of papers);

total degree, which stored how many edges exist in the graph;

allocation, which would specify how much to increase the size of a vertex’s neighbor list once it was full;

padding variables to round the data structure’s memory to be a multiple of 64 bits;

and vList, which is an array of vertex objects.

Each entityID and paperID would be assigned its own vertex on the vList; the entity’s or paper’s ID would be its index in vList, such that vList[paperID] is the paperID’s vertex. Accordingly, vList’s length would be vertex count.

Each vertex object consists of:

forward, a Neighbors object for the vertex’s outgoing edges;

and backward, a Neighbors object for the vertex’s incoming back-edges.

Each Neighbors object consists of:

nList, an array of Edge objects (indicates the vertex’s edges or back-edges);

degree, which indicates the number of edges the nList contains;

capacity, which indicates how many edges the nList could currently hold without growing;

myPos, which identifies the hypothetical index of the vertex if it existed on its own nList;

and padding, to round out the object to a multiple of 64 bits.

Before the Neighbors class existed, nearly identical data and methods for a vertex’s edges and back-edges existed, which was redundant and tedious. If a method was to be changed for a vertex’s edges, a similar change would have to be made for the appropriate back-edge method. This redundancy allowed for human error, and thus the creation of the Neighbors class is deemed practical.

Each Edge object consists of:

vid, a vID object that holds the ID of the vertex that the edge/back-edge points to;

weight, the weight of the (back-)edge from the vertex to the vid;

and padding, which rounds out the memory of each Edge to be a multiple of 64 bits.

Figure 2:

Neighbors:

* Edge \*nList
* UINT32 Degree
* UINT32 capacity
* UINT32 myPos

Data Model (Padding implied):

graph:

## vertex \*vList

## UINT64 vertexCount

## UINT64 totalDegree;

## INT16 allocation;

.

vID:

* UINT64 id

Edge:

* vID vid
* UINT32 weight

vertex:

* Neighbors forward
* Neighbors backward

### Methods of the Graph Data Structure

The graph has several public functions that allow for its creation, updates, and analyses, such as:

* void setAllocation(INT16 a): Will set allocation to a positive number up to 16; affects how much edge/back-edge arrays grow when full
* isEdge(vID u, vID v, optional UINT16 &index): Returns whether there is an edge going from vertex u to vertex v; sets index to v’s position on u’s edge list.
* isBackEdge(vID u, vID v, optional UINT16 &index): Returns whether there is a back-edge going from vertex u to vertex v; sets index to v’s position on u’s back-edge list.
* bool addEdge(vId u, vId v, int weight = 1, bool inc = false): Add an edge from u’s vertex to v’s vertex; if u does not have an edge to v, then an edge with the passed weight from u to v will be added, and a similar back-edge from v to u will be added. If inc is false and u already has an edge to v, the weight from u to v will be overwritten by the new weight; if inc is true, then if an edge already exists, the edge will increase by the passed weight. If the edge already existed, false will be returned; otherwise, true is returned.
* bool addG2Edge(vID u, vID v, UINT32 weight = 1, bool inc = false): Same as addEdge, but instead of setting v’s back-edge to u to reflect u’s forward edge to v, v sets its own forward edge to you. The effects of already existing edges and the inc parameter, as well as the return value, have not changed.
* void createV(vID v, UINT32 capacity, UINT32 backCapacity): Empty edge/back-edge arrays of sizes capacity and back-capacity respectively will be set for v’s vertex; reduces linear copying over and affective with addEdge2.
* void addEdge2(vID u, vID v, UINT32 weight = 1): An edge from u to v will be added quickly from

u to v without seeing if it already exists and without searching where it belongs; appends an edge to u's neighborList and v's backNeighborList; most efficient if used after createV function sets the size of u and v's edge/back-edge lists.

bool delEdge(vID u, vID v, UINT16 weight = 0): If the edge already exists, the edge that points from vertex u to vertex v is removed and true is returned; otherwise, nothing happens and false is returned.

* getVertexCount(): Returns |Vertices|.
* getTotalDegree(): Returns |Edges|.
* printGraph(): Will print the list of all vertices and their adjacent vertices.

These functions also have the same methods for the back-edges:

* void printVneighbors(vID v): The vertex's neighbors’ IDs are printed.
* void printVneighborsWeights(vID v): The vertex's neighbors’ IDs and edge weights are printed.
* UINT32 getVoutDegree(vID v): Returns the vertex outdegree.
* UINT32 getVmyPos(vID v): Returns the vertex's hypothetical position in its own edge array.
* Edge \* getVneighbors (UINT64 v): The list of vertex v’s forward Edges is returned.

# Building the Graph

## Original Building of the Graph

The program originally read in each line of the ASCII file and added the edge to the graph. There is a need to distinguish the papers’ IDs and entities’ IDs, since they would be on the same graph structure and overlap between their sets of ID numbers existed. Thus, a namespace was created for the paperIDs; the largest entityID is currently less than two million, so the entityIDs would remain as they were given while each paperID would be increased by two million. Let paperID indicate the paperID specified in the file plus two million from here forward.

The program creates an array to store the entityTypeID for each entityID. For each edge, if it is the first edge that points to the entityID, then the entityID will become assigned to the entityTypeID that accompanies it on the edge.

The edge from the paperID to the entityID would be added using the graphs addEdge function, and thus a back-edge from the entityID to the paperID would also be added. The standard addEdge function has a runtime of O(n), where n is the number of Edge objects on a vertex’s nList. It is linear because the function involves a linear shifting of the Edges in the nList until the new edge’s position is found. It also uses the isEdge function, which is a binary search for whether the Edge already exists; this has a runtime of O(log2(n)). Furthermore, this all happens again when the back-edge from the entityID to the paperID is formed.

Graphical user interface, text, application

Description automatically generated

Figure 3:

Pseudocode for naïvely adding edges to build G.

This form of building a graph that consisted of 1.2 billion edges, with some entities being referenced to by over 16 million papers, was deemed insufficient in terms of speed. Furthermore, due to all the reallocations made for each edge array, there was much unused space created.

## Improved Building of the Graph

To increase the runtime efficiency of the building of G, AVL trees were introduced. An AVL tree is a binary search tree that balances itself so that every insertion, deletion, and lookup from the tree is O(log2(n)). Every entity was assigned its own AVL tree through an array of AVL trees where the entityID was the index of the entity’s AVL tree. The program would iterate through the edges of the graph, and every edge’s paperID was inserted into the edge’s entityID’s AVL tree. Another array, paperCountArray, kept track of how many entities each paperID referenced. Once the graph is finished being read, each entity’s AVL tree contained every paperID that referenced it.

The program would then use the createV function to allocate an empty edge array of the right size (the number of entities the paper references, found in the paperCountArray) for every paperID and an empty back edge array of the right size (the number of papers that referenced the entity, found in the entity’s AVL trees node count) for every entityID. Then, each AVL tree in the AVL array would recurse in an in-order traversal and add an edge from the tree’s node, which stored a paperID, to the entity that the tree belonged to. The edge was added through the addEdge2 function, which would append the edge to the end of the paperID’s edge array and to the end of the entityID’s back-edge array.

The original building of the graph used the linear addEdge function, which took O(n) time, where n is the number of edges in the vertex’s edge array and performed it 1.2 billion times. Considering that there were approximately 28.4 million(M) papers and 1.1 million(M) entities, each entity would on average be referenced by 1100 papers and each paper would reference approximately 40 entities. However, one paper references 3185 entities and one entity is referenced by over 18M papers. Nonetheless, according to the average values, the original building of G would lead to O( (1.2B \* (1140) ). However, for the improved building of the graph, each entity’s AVL tree would insert a paperID at O(log2(n)), where n is the number of papers in the AVL tree. The insertion into the trees would be O( |Entities| \* log2( |Papers per entity| ) ), or O( ( 1.1M \* log2(1100) ) ). This results in a massive increase of time efficiency.

The transfer time to add the edges from the trees to the graph relies on the number of nodes in the AVL array. For G, there should be 1.2B nodes, each node representing an edge. However, the addEdge2 function occurs in constant time. Thus, the total time taken to build G occurs in O( ( 1.1M \* log2(1100) ) + 1.2B) time.

## Binary Reading and Parallel Building of the Graph

Reading in an ASCII file and converting the characters to numbers one number at a time takes time; the number of ASCII conversions that occur are (1.2B edges \* 4 values per edge = 4.8B); this method is not the most effective way of scanning data. Thus, the program was altered to scan in a binary graph. To do this, another program was created to convert the ASCII file into a binary file, which takes as much time as it would to read in the original ASCII file on the main program, but it would only have to be performed once and would reduce the time of the building of G for every execution of the program. The full graph as a binary file can be read into the program in approximately 22 seconds, which is far faster than the reading of the ASCII file.

When the project’s main program scans the binary file, it loads all of its contents to an edgeArray. Each edge would by four contiguous values on the array, and edgeArray[i \* 4] would thus return you the ith edge’s paperID, with the next being the publicationYear, then the entityTypeID, then the entityID. Similarly to the improved building of the graph, this version would go through each edge of the array and insert the paperID to the entityID’s AVL tree. Then it would also transfer the edges to the graph in the same way the improved building of the graph did.

However, with this new version of having an array with the edges, the paperIDs can be inserted into the entities’ AVL trees in parallel. To do this, the program assigns each thread with a range of entities that it can modify the trees of. Then, each thread iterates through the edgeArray, and if the edge that the thread is currently looking at has an entityID that is within the thread’s range, it inserts the edge’s paperID into the entityID’s AVL tree. However, the transfer of the edges from the AVL trees to the graph is currently not parallelized due to some race conditions and the logistics of appending to the paperID’s back-edge array in order. Thus, the time of the tree insertions can be divided by the number of entities there are, becoming O( |Entities| \* lg( |Edges|/|Entities| ) / |threads| ), while the transfer time remains the same. Thus, G can now be built in O( ( ( 1.1M \* log2(1100) ) / |threads| ) + 1.2B) time.

# Building G2

## G2 Purpose and Description

As previously mentioned, Biovista’s Project Prodigy is an AI engine that processes the graph to develop diagnoses and propose treatment plans for complex illnesses. The methods used to analyze the graph for useful information look at the relationships between entities. The relationships between entities that are observed in this project are called co-occurrences, “coocs”; a cooc is the number of common papers between two entities. Recall in Figure 1 that paper P1 references entities E1 and E2, and that P2 also references E1 and E2. Because of the entities sharing two papers in common, E1 and E2 have a cooc of two.

To have quick access to the coocs of any two given entities, this project squares the graph, G, to construct G2. Although the graph provided by Biovista was bipartite, G2 is not, for an entity will have an edge that points to any other entity that it has a cooc with, and the weight of the edge is the magnitude of the cooc. The result of squaring the graph provided in Figure 1 produces the graph provided in Figure 4.

A picture containing text, clock, gauge

Description automatically generated

Figure 4:

P1 and P2 both pointed to E1 and E2, and so they have a cooc of 2, as reflected by the undirected edge from E1 to E2. P3 pointed to both E2 and E3, and thus they share a cooc of 1.

## Naïve Building of G2

Although it would have been possible to store G and G2 separately, this project put G2 onto G’s graph for the sake of convenience. Thus, every entity’s vertex on the graph would both show which papers reference it and which entities it coocs; the papers’ vertices remain unchanged. To build G2, the program iterates through each paper and gets the list of entities it references via the getVneighbors function. Starting from the first entity that the paper references, a pseudo-undirected edge is added between that entity and every subsequent entity on the list, then this process occurs again for the second entity and every subsequent entity, up until there has been an edge added for every pair of entities in the paper’s edge array.

The pseudo-undirected edge is formed by calling the addEdge function twice, with the two vertices adding both an edge and a back-edge to each other. This results in each entity having the entities it coocs on both its edge and back-edge array. Furthermore, note how in the standard use of the addEdge function, the inc variable is false, resulting in the edge being set to one. However, for G2, an entity should record how many times it coocs another entity. Thus, the inc variable is set to true and the weight variable is set to 1, and consequentially, every time two entities add an edge to each other, the cooc size is incremented.

Text, letter

Description automatically generated

Figure 5:

Pseudocode for the naïve algorithm to build G2.

Each paper has on average 40 entities it references, which means each paper has on average 40C2, or 780, pairs of entities. Each pair of edges would add the edge twice, and the naïve approach adds each edge in linear time, so it would have a runtime of (|Papers| \* |Pairs of entities per paper’s neighbor list| \* 2), or O(28.4M \* 780 \* 2) which is O(44.3B).

## Improved Building of G2

To increase the time efficiency, the program uses the same array of AVLs as previously mentioned. After each edge of G is moved from the AVL to the graph, the node of the edge is deleted. Thus an array of clear AVL trees is available upon the transfer. The AVL’s node class was also modified to store an additional data variable, weight, which would store how many times that node’s value has been inserted into the tree. This means that if a value is inserted into the tree for the first time, its node will start with a weight of one, and every time the value is inserted after, the weight will increment.

The program still starts by iterating through all the papers and grabbing the list of entities it references; however, for each pair of entities on the list, instead of adding a pseudo-undirected edge between them, they are inserted onto each other’s AVL tree. Recall how the naïve building of G2 would result in each entity having both an edge and a back-edge to each entity it had a cooc with. It was determined that this redundancy used up unnecessary space, and it would interfere with the speedy transfer of G2’s edges from the trees to the graph.

Thus, the concept of the graph was modified so that each entity’s set of neighbors is a subset of the set of entityIDs, while its set of back-neighbors is a subset of the set of paperIDs. Each paper’s set of neighbors is a subset of the set of entityIDs, and its set of back-neighbors would be empty. Recall G from Figure 1 and G2 from Figure 3. Figure 6 displays how paper P1’s and entity E2’s neighbor and back-neighbor lists would appear.

A picture containing text

Description automatically generated

Figure 6:

Observe the weights of each neighbor’s edge; each paper’s outgoing edge has a weight of 1, as does each entity’s incoming edge. E2’s edge to E1 would have a weight of 2 (their cooc) and E2’s edge to E3 would have a weight of 1 (their cooc).

Each entity’s AVL stores which entities it has a cooc with, how many entities it had cooc with, and the weight of each cooc, and thus the transfer of G2 from the trees to the graph may commence. To do this, the createV function is once again used for each entity to set its neighbor list to be the size of how many edges are to be put in it (the entity’s AVL node count). Then, traversing through each entity’s tree in-order, the addEdge2 function is used again to append an edge from that entity to each entity it has a cooc with, using the node’s weigh (the cooc weight) as a parameter for the weight of the edge to be added on the graph. This version of addEdge2 would not add a back-edge to either entity. Since both the iteration through the array of AVLs and the traversal of those AVLs occur in order, G2 is built onto G successfully.

Though each paper would still have to add an edge twice for each pair of entities, the insertion of the edge would reduce the runtime of the addition of an individual G2 edge from O(n) to O(log2(n)). The runtime of inserting all the edges becomes (|Papers| \* log2( |Pairs of entities per paper’s neighbor list| ) \* 2), or O( 28.4M \* log2(780) \* 2).

Again, the transfer of each edge from the tree to the graph is O(1) and it happens once per edge. G2 adds approximately 3.2B edges to the graph, so the transfer would now take O(3.2B). Therefore, the improved runtime to build G2 is O(28.4M \* log2(780) \* 2) + O(3.2B) = O(3.7B).

## Parallel Building of G2

The insertion of G2’s edges to the AVL trees is also parallelizable. Similarly to how the parallel building of G distributed the insertion of the entities’ trees, the building of G2 can also insert the edges of pairs of entities across multiple threads. Recall in Figure 5 how each pair of entities adds each other as an edge. Instead of adding the edge, the thread checks whether the paper’s ith entity is within its entity range; if it is, it inserts the paper’s jth entity onto the ith AVL tree. It then checks if the paper’s jth entity is within its entity range; if it is, it inserts the paper’s ith entity onto the jth AVL tree. Thus, the insertion of G2’s edges into the AVL tree is parallelized.

However, due to the graph’s shared total degree variable, the transfer of G2 from the AVL trees to the graph is not parallelized. Future versions of this project could attempt to parallelize the transfer.

Thus, the runtime efficiency of the insertion of G2’s edges becomes ( ( |Papers| \* log2( |Pairs of entities per paper’s neighbor list| ) \* 2) / |threads| ), and the nodes used to run the program on Titan had 40 threads, so it had a runtime of O( (28.4M \* log2(780) \* 2) / 40), or O(13M).

# Incrementally Updating G and G2

## Incremental Updating Purpose and Description

The current primary issue with Biovista’s Project Prodigy is that it cannot incrementally update; every time a new paper is published, to include the paper in their program takes three weeks, which negatively impacts the currency of their data and diagnoses. Thus, one central purpose of this project was to create a graph where the data could be incrementally updated.

Suppose that G2 has been built and is currently running, but Biovista has received a list of new edges that they want the graph to include. Let a graph that has been successfully updated be called G2’, or G squared prime. For each edge on the update file, the edge’s paperID’s neighbor list should reflect that it references the edge’s entityID, and likewise the entityID’s back-neighbor list should indicate that it is referenced by the edge’s paperID. Furthermore, since paperID now references the entityID, every entityID that the paperID refers to will now have a new cooc with the edge’s entityID, and each of the entities’ neighbor list will need to reflect such.

## Implementation of Incremental Updates

For each edge provided in the incremental update file, which is assumed to have the same format as the graph’s read-in file that was used to build G, an edge from the edge’s paperID to the edge’s entityID is added through the normal addEdge function with its default parameters (a back-edge from the entityID to the paperID is added, weight is 1, inc is false). If the addEdge function returns true, that means that the edge from the paperID to the entityID is new and that G has been updated.

Then G2 needs to be updated; the program gets the paperID’s list of entityIDs it references through the getNeighbors function (with the new entityID included due to the addEdge function). It then goes through every entityID, currEID, on the paperID’s list of entityIDs that is not the incremental edge’s entityID and adds an edge between the incremental edge’s entityID and currEID using the addG2Edge function. This will add (or increment) the forward edges to the two entityIDs that point to each other without modifying the back-neighbor list. It adds the edges in a runtime of O(n), where n is the number of entities that the entity has a cooc with.

Graphical user interface, text, application, email

Description automatically generated

Figure 7:

The pseudocode to add an incremental edge to an existing G2.

Just as the program could read in edges as an ASCII or binary file to build G, the program can also read in the incremental edges in either format. No attempt was made to parallelize the update of G2 to G2’, for the relatively small number of edges to be added at the time and the difficulty of parallelizing the process deem it unnecessary. Thus, the updating of G2 to G2’ takes O(|Number of incremental edges| \* (|average number of entities a paper references| + |average number of papers an entity is referenced by| + (|average number of entities a paper references| \* |the average number of entities an entity has a cooc with|) ) ). Since G2 added 3.2B edges and there are 1.1M entities, the average number of entities an entity has a cooc with 2900, which means the update to G2’ takes O(|Number of incremental edges| \* (40 + 1100 + (40 \* 2900) ) ), or O(117000 \* |Number of incremental edge|). Of course, as more updates are made, the longer each update will take, for the averages will increase.

# Performing the Primitives

## Primitives’ Purpose

As previously mentioned, Biovista uses the relationships between the entities for their analyses. Biovista uses primitives to analyze aspects of the entities’ relationships, and this project focuses on the cooc relationships that the primitives find. There are four primitives this project aimed to perform: expand, connect, bridge, and bibliography.

## Expand

The expand primitive seeks to find the most connected entities to a given entity. Given an entityID and an entityTypeID, the expand primitive should return the ten entityIDs of the input entityTypeID that have the strongest cooc with the input entityID. It will produce a list of Edge objects, or expandArray, where the first Edge’s vid’s id is the entityID that had the strongest cooc, and the edge’s weight will be the magnitude of that cooc, and so on for up to ten Edges.

Let the input entityID be called inEID. The program iterates through inEID’s list of cooc entities, and if the current entityID from the list, currEID, is of the input entityTypeID type, then it iterates through the list of the top ten entities that have the strongest cooc with inEID until it finds an entityID that has a weaker cooc with inEID. currEID would then take its spot, and all the Edge objects would shift down. If the list is full, then the last entityID is removed from the list.

Text, letter

Description automatically generated

Figure 8:

The pseudocode for the expand primitive.

The runtime for the expand primitive is O(|the number of entities an entity has a cooc with| \* (|the number of entities with the given entityTypeID| / |the number of entities in the graph|) \* 10).

## Connect

The connect primitive seeks to find which entities connect with a certain entity. Given an entityID, an entityTypeID, and a list of entityIDs, the connect primitive determines for which pairs there is a connection between the given entityID and the items of the list. The connect primitive will produce a list of Edge objects, connectArray, that will be of a size ≤ the size of the given list of entityIDs.

For each entityID, currEID, on the input list, the program will check if currEID has the same entityTypeID as the input entityTypeID. If it does, it will do a binary search to find if currEID has a cooc with the input entityID. If it is, then an Edge with the vid id of currEID and the weight of the cooc’s magnitude will be appended onto connectArray.

Graphical user interface, text, letter

Description automatically generated

Figure 9: The pseudocode for the connect primitive.

The runtime for the connect primitive is O( |the number of entities on the input list| \* (|the number of entities with the given entityTypeID| / |the number of entities in the graph|) \* log2(|the number of entities the input entity has a cooc with|) ).

## Bridge

The connect primitive seeks to find the most connected entities between two given entities. Given two entityIDs and an entityTypeID, find the ten entityIDs of entityTypeID that have the greatest combined cooc with the input entityIDs. It will produce a list of Edge objects, coocArray, that store the entityIDs of the ten entityIDs most strongly coocing with both input entityIDs and their cooc sum.

For both input entityIDs, eID1 and eID2, get their neighbor lists. For each entity on either’s list, check if the current entityID, currEID, of both lists are the same. If it is not, then for the list with the smaller currEID, get the next entityID on the list. If the currEIDs do match, then check if it is of the given entityTypeID. If it is, then the program iterates through the list of the top ten entities that have the strongest combined cooc with the two entityIDs until it finds an entityID that has a weaker cooc sum. currEID would then take its spot, and all the Edge objects would shift down. If the list is full, then the last entityID is removed from the list.

Text, letter

Description automatically generated

Figure 10:

The pseudocode for the bridge primitive.

The bridge’s runtime is O( ( |number of entities that have a cooc with both given entities| \* (|the number of entities with the given entityTypeID| / |the number of entities in the graph|) \* 10) + (2 \* |average number of entities an entity has a cooc with| ) ).

## Bibliography

The bibliography primitive seeks which papers supporting a cooc between two entities. Two entityIDs would be given, as well as the number of papers desired to be returned. A list of paperIDs of the given size, biblioArray, will be produced.

For both input entityIDs, eID1 and eID2, get their back-neighbor lists. For each Edge on either’s list, check if the current paperID, currPID, of both lists are the same. If it is not, then for the list with the smaller currPID, get the next paperID on the list. If they are the same currPID, then append it to the end of biblioArray. If the list is full, break.

Graphical user interface, text, application, email

Description automatically generated

Figure 10:

The pseudocode for the bibliography primitive.

The runtime for the bibliography primitive is O( (2 \* |the average number of papers an entity is referenced by| ) + |the number of papers that both entities are referenced by| ).

### Transaction Descriptions

The driver program, graphMain.cpp, is executed with a mandatory flag -f and an argument for the name of the file storing G’s edges, followed by an optional flag -b that indicates the user wants the file to be read is in binary, not ASCII. Currently, a file with the incremental edges is passed by adding a -i flag followed by the incremental update’s file name, and an optional -n to indicate that the file is in binary. However, future versions may keep the program running after G+G2 is built and ask the user what they want to do next. Furthermore, a flag for what primitive(s) is to be performed, -e -c -r -p for expand, connect, bridge, and bibliography may be thrown, followed by the number of tests they desire to perform on the primitive. If a primitive flag is thrown, the user will be prompted for the appropriate arguments.

# Results

## Memory

For this project, time efficiency was a greater concern than memory efficiency. However, it is still relevant. Figure 11 shows the memory consumed by the program at different stages of its life.

Figure 11

Whenever G2 is finished being built and the program is just about to die, the process is consuming over 230 gigabytes of memory. According to a test, there seems to be something in the program that takes increasingly more memory even when the program is not supposed to be doing anything. Calculations on the predicted memory consumption of all the graph’s variables indicate that the process should consume approximately 85 gigabytes of memory.

The graph has a vList, which is an array of 32M vertices, and each vertex has two Neighbors objects. Each Neighbors object consumes 24 bytes. Therefore, the graph’s vList takes up 1.4 gigabytes. Each Edge object occupies 16 bytes of memory. In G2, there are 1.2B Edge objects in all the graph’s backward neighbors’ nLists. Furthermore, G2 has 4.4B forward Edges in all the graph’s forward neighbors’ nLists. The size of memory occupied by the Edge objects is 5.6B \* 16 = 84 gigabytes. When the vList’s memory consumption and the Edges’ memory consumption are summed together, only 85 gigabytes are justified, which is just more than one-third of the memory being taken. All other factors seem to be relatively insignificant, with no variables taking anything near a gigabyte. Thus, if someone were to continue this project, a more thorough analysis of the memory consumption should occur.

## Speed

Speed was the main focus of this project. Its purpose was to build G and G2 quickly, incrementally update edges rapidly, and perform primitives at a comparable speed to Biovista’s current system.

### Population of G and G2

One objective of this product was to build G and G2 in less than a week. A binary read of Biovista’s graph takes 22 seconds, and this allows for the parallel insertion of G’s edges into AVL trees, which takes approximately 190 seconds (3.2 minutes). To transfer the edges from the tree to the graph takes approximately 470 seconds (7.8 minutes), and thus G is built in just 682 seconds (11.4 minutes).

To build G2, the program inserted the G2 edges to an AVL tree in parallel as well, which takes 3250 seconds (54.2 minutes), and to transfer these edges to the graph engine takes 550 seconds (9.2 minutes). Thus, the total population of G2, after G is completed, is 3800 seconds (63.3 minutes). G’s population time + G2’s population time = 4482 seconds (74.7 minutes). That is 7.4 thousandths of a week, so this goal has been met by the project.

Figure 12

### Addition of Incremental Edges

This project has also successfully implemented a method to incrementally update G and G2. The goal was for an incremental update to be added with a speed of less than 10 milliseconds per edge. The program added a new paper that referenced 15 entities in 73.1 milliseconds, just under five milliseconds per incremental edge. This is twice as fast as the project aimed for. However, when the new paper points to entities with a larger outdegree on G2, more time is taken. A future student could aim to add the incremental edges quickly regardless of which entities are referenced.

### Primitive Performance

Individual runs of the primitives display an inconsistent performance of time, so to find the times of the primitives this project uses, each primitive is performed 100 times and the average time is recorded.

The correctness of the execution of the expand primitives are inarguable; there was a perfect match between Biovista’s expand output and this project’s output. The project’s average expand results are several orders of magnitude faster than Biovista’s. In fact, it is so much faster that it indicates that the method of performing the 100 tests may not be working properly or that the advantage of the cache used is monumental.

The correctness of the execution of the bridge primitives are somewhat less certain. Dr. Wheat claimed that a different approach to the bridge primitive also had output that differed from the benchmark’s output. However, if the timing report is correct and the bridges are truly being found, all of the bridge’s tests perform faster on the senior project graph than on Biovista’s system. However, this project reported to run a supposedly more complex test quicker than it ran an easier one, so if someone were to continue this project, they could go into a deeper investigation of this phenomenon.

# Conclusion

The graph framework can accommodate the speedy building of Biovista’s graph and square the graph relatively quickly. Each incremental edge is added quickly. The primitives perform at a rate comparable to Biovista’s current system. Thus, this project can be used by Biovista to quickly build, update, and analyze their graphs.

## Potential Improvements to Be Made

* Merge this result with Chilu Mutale’s MariaDB Biovista Storage Engine. This allows:
  + A persistent backing store (able to start up instantly from having been shut down)
  + Client/Server usage model consistent with what they already use.
* Implement methods to perform more of Biovista’s primitives.
* Parallelize the transfer of edges from the AVL array to the graph.
* Make the building of G2 linearly proportional to the building of G.
* Find out what is taking up the excess memory.
* Add the incremental edges quickly regardless of which entities are referenced.
* Find out why the timings of the performances of the primitives are inconsistent.
* Develop a dumpGraph function that will dump G2 to a file that is quickly recoverable.

## Sources Cited

Bienz, Amanda, et al. “Node Aware Sparse Matrix–Vector Multiplication.” *ScienceDirect*,Journal of Parallel and Distributed Computing, vol. 130, August 2019, pp. 166-78, www.sciencedirect.com/science/article/pii/S0743731519302321. Accessed on 3 Dec 2022.