Assignment #4 Graph Theory and Parallel Computations

Problem 1. Molecular Structures using Graphs

1.1 Graph Theory

I chose to create a `GraphBase` class that would serve as the basis for a graph structure and implement many of the common graph tasks, then created child classes that implement the specific adjacency list and matrix list edge storage paradigms.

For example, the `GraphBase` defines the following:

- whether graph is directed or undirected
- number of edges
- number of vertices
- edge ids
- edge pairs
- printing
- BFS (using the virtual getNeighbors function)
- distance matrix (using BFS, which calls virtual getNeighbors function)
- attribute handling**

This allowed me to be as DRY as possible, separating only the actually different aspects of adjacency list versus adjacency matrix in their respective child classes.

** Futhermore, I wanted to challenge myself to mimic some of the graph analysis features that I am familiar with in RDKit and IGraph (both of which I use extensively for work). Both these libraries allow users to define node/attribute features of various types at runtime. I realized how suprisingly complex this feature was and wanted to attempt to build it into my graphs for this assignment. To facilitate different typed vertex and edge attributes, I created a base `Attribute` class and a templated `AttributeValue` class that inherits from `Attribute` and actually stores the values for the particular attribute type. Then, in my `GraphBase` I have a map of `v_attr_` which stores the name of the attribute mapped to a vector of pointers to `Attributes`. This allows me to create instances of `AttributeValue` with a particular type that are stored as pointers to an `Attribute`. Then, when retrieving or using the values, I can use `dynamic_cast` to cast the base `Attribute` back to the desired `AttributeValue<T>` type!

Because of these templated attribute features, the `GraphBase` is provided in the `include/graph_base.h` header file, allowing maximum flexibility in terms of user defined Vertex and Edge attribute types! To provide differentiation between directed and undirected graphs, I define to-from as the row and from-to as the column. Therefore, in a undirected class the row and column at a particular index are equal, but this is not necessarily true for a directed graph unless all edges are explicitly created in both directions.

Since I am managing complex structures with pointers to objects on the heap (and intentionally not using smart pointers since the graph class should own all this memory), I used Valgrind to check for memory leaks. After finding that I need to ensure the attribute pointers are deleted when overriding their values, my final memcheck output shows no leaks!

```
(base) [cjpeh test]$ valgrind --leak-check=yes
../bin/test_graph_al_directed
==9079== Memcheck, a memory error detector
==9079== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9079== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==9079== Command: ../bin/test_graph_al_directed
==9079==
ALL TESTS PASSED
==9079==
==9079== HEAP SUMMARY:
==9079==     in use at exit: 0 bytes in 0 blocks
==9079==   total heap usage: 398 allocs, 398 frees, 86,028 bytes allocated
==9079==
==9079== All heap blocks were freed -- no leaks are possible
==9079==
==9079== For lists of detected and suppressed errors, rerun with: -s
==9079== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

1.1.1

Inheriting from the `GraphBase`, I define `Graph_AM` in `include/graph_am.h` as my implementation of the adjacency matrix representation of a graph.

Here, I use a vector of vectors to maintain a square matrix where `0` represents no edge and `1` represents an edge between the two vertices. Then, I override the virutal function `getNeighbors` to iterate over the respective row of the adjacency matrix for the given vertex argument and build a neighbors vector with all the non-zero element indicies.

1.1.2

Inheriting from the `GraphBase`, I define `Graph_AL` in `include/graph_al.h` as my implementation of the adjacency list representation of a graph.

Here, I again use a vector of vectors, but now store just the neighbors for each vertex. To overridde the `getNeighbors` function, I simply return the vector of neighbors already stored in the adjacency list for the requested vertex id.

1.1.3

I created an extensive testing protocol that tests all the aspects of the graph storage methods, adding vertices, adding edges, retrieving edge ids from a pair of vertices, getting neighbors, proper handling of directed/undirected, etc.

The test can be found in `test/test_graph_al_directed.cpp`, `test/test_graph_al_undirected.cpp`, `test/test_graph_am_directed.cpp`, `test/test_graph_am_undirected.cpp`.

All tests can me compiled to executable files placed in the `bin` directory by invoking the makefiles `all` target. There are assert statements throughout the test programs that will test for proper conditions. If all tests pass,

`ALL TESTS PASSED` will be printed and the program will return `0`.

After these initial tests and after implementing the BFS, I returned here to also add tests for parsing the ethane and octane example graphs. These can be found at `test/test_ethane.cpp` and `test/test_octane.cpp`. These check the proper parsing of the molecular structure with the known bonds and distance matrix.

1.1.4

The two classes `Graph_AM` and `Graph_AL` both accomplish the same graph operations, but they have contrasting strengths and weaknesses in terms of memory and temporal efficiency.

There are 4 different aspects that I will discuss for each paradigm:

1. memory required to store edges
2. complexity of neighbor lookup
3. complexity of edge query
4. storing edge weights

In general, the analysis is more complex than our standard big O notation analysis since there are two important factors, the number of vertices and number of edges. Both of these input characteristics will be discussed.

Adjacency List

Foremost, the memory requirements for storing adjacency list is heavily dependent on the number of vertices in the graph. For graphs with many vertices but few edges, this method is very efficient because it only stores the neighbors for each vertex, rather than the relationship between all vertices. As the number of edges increases to be a complete graph, we approach the same (or worse) memory requirements as the adjacency matrix. I say potentially worse because the adjacency matrix may be more optimized than I have implemented here. The time complexity to retrieve the neighbors for a particular vertex is O(1) for the adjacency list. This is because the list of neighbors is stored directly at the index of that vertex. Therefore, it is a direct lookup. This makes adjacency list a very good choice if the majority of operations will be finding neighbors. In contrast, to check if an edge exists between two vertices is an O(n) operation. This is because we must first lookup the neighbor list, O(1), then iterate over all those neighbors to see if the desired other vertex is in the neighbor list, an O(n) type operation. This means the adjacency list is not ideal for types of problems where we want to ask if two vertices are neighbors frequently. Lastly, to store edge weights in an adjacency list requires an additional datastructure. This could come in many forms, such as storing a special `Edge` class in the adjacency lists, using a `std::pair`, or forcing the weight to be a normal edge attribute stored in a completley separate list keyed by the edges id. But, all of these solutions are not as memory or time efficient as storing them as the value in an adjacency matrix directly.

Adjacency Matric

Foremost, the memory requirements for storing an adjacency matrix is always O(n^2) based on the number of vertices. Though there are more possibilities for optimization here than for an adjacency list. For example, since we know the matrix is square, we could store just the upper triangle, or at least improve data locality by using a block of memory to store the entire matrix. This would improve both the spacial efficiency as well as potentially improve timings as well if the caching is significant. The time complexity to retrieve the neighbors

for a particular vertex is O(n) where n is the number of vertices. This is because we need to check the given row of the adjacency matrix for non-zero elements to find the neighbors. Therefore, we must always check over every other vertex. Therefore, applications that must retrieve the neighbors for a vertex often are not well suited for adjacency matrix. The time to query an edge is the opposite; The lookup on whether there is an edge between two vertices is an O(1) operation since it is simply a lookup in the matrix. Lastly, storing the data as an adjacency matrix provides a very straightforward way to incorporate edge weights directly with the edges. Here, the weight can simply be the value stored in the matrix. Unlike adjacency lists, this means there is no need for an external datastructure to store weights, providing certaintly better memory efficiency, and potentially better timing as well.

In summary:

```
The adjacency list's best case scenario is when we want quick neighbor
lookups on relatively sparse graphs.
The adjacency matrix's best case scenario is for fast edge queries on dense
graphs.
```

The developer should analyze their use case to see which type of graph operation is most common and choose the paradigm that best suits that use case.

Generally, most chemical structure graphs are not dense. Thinking in the organic space, typically vertices will have at most `4` edges, and typically the types of algorithms used on molecules (fingerprinting, substructure search, walking the graph) all rely on getting a vertices neighbors. Therefore, these applications likely favor the adjacency list approach.

1.2 Breadth First Search (BFS)

1.2.1

I implemented `BFS` as a method of the `GraphBase` class (found in `/include/graph_base.h`), which interacts with the different edge storage paradigms using their respective `getNeighbors` functions. This allowed me to provide a common interface and again maintain DRY code.

To color the different vertices, set the `pi` value showing their parent vertex in the search path, and the current distance, I set `_color` (enum of `Colors`) `_distance` (int),and `_pi` (int) vertex attributes with the dynamic typed attribute setting cababilities described above. The templated attributes allowed me to very easily create, set, and retrieve the necessary designations for the BFS.

I wrote a few test cases to ensure the BFS was correctly working for directed and undirected graphs with both adjacency matrix and adjacency list representations. First, I create a graph with a few connected vertices and one non-connected vertex. I use `BFS` to check that the non-connected vertex can only reach itself. Then I check whether the connected vertices can reach all of each other, but not the non-connected vertex. Finally, I add a new edge from the non-connected vertex to the rest of the vertices and check that now all vertices are found by `BFS`.

1.2.2

My choice to implement the `BFS` at the baseclass level makes it so that I did not have to choose which edge paradigm to implement it for! By providing a consistent API of `getNeighbors` for the adjacency list and adjacency matrix, I was able to provide better functionality in the baseclass. But, if I did need to choose, having the adjacency list provide direct access to neighbors without having to iterate over the row in the adjacency matrix and filter to `1`s would be my choice. This is because getting the neighbors in BFS is the most important manipulation of the underlying graph data.

For example, given vertex `V`, the BFS algorithm needs to retrieve all neighbors for `V`, then all neighbors for all of those neighbors. The adjacency list stores these neighbors in memory with O(1) lookup access. In contrast, we would need to iterate over all elements in the row of the adjacency matrix to find non-zero elements, which are the neighbors. Therefore, this is an O(n) operation for the adjacency matrix. For a small graph like the examples, the timing differences may not be noticable as `n` is small, but as `n` grows and the number of edges in the graph also grows, there will be many many more calls to `getNeighbors` to exhaustively search all vertices.

While adjacency list provides fast access to neighbors and is particularly memory efficient for sparse graphs since it only needs to store the edges that exist, disadvantages of the adjacency list approach is that it can be memory inefficient for large, dense graphs and poor for testing edge existance. This is especially true compared to an optimized undirected graph which could store the upper-triangle of the adjacency matrix to save space. Further, which the lookup of neighbors is O(1), to test for the existance of a particular edge is an O(n) operation. This is because you would need to iterate over the list of neighbors to determine if a particular node was contained in the adjaecncy list. There are some optimizations that could be applied, such as using a set instead of a normal list, but as implemented the test for if there is an edge between vertices 1 and 3 is an O(n) operation.

Ultimately, the complexity of `BFS` comes from retrieving neighbors, not determining the presence of an edge. As a result, adjancency list should be the ideal storage paradigm.

1.3 Distance Matrix

1.3.1

I again implemented the `getDistanceMatrix` method in the `GraphBase` class (found in `/include/graph_base.h`). This method calls `BFS` on each vertex in the graph, which populates the `_distance` attribute of each vertex, which is then extracted into the square distance matrix.

This provides me a great way to compare side to side the hypothesis put forth in answer 1.2.2 that the adjacency list should in theory be faster than the adjacency matrix.

1.3.2

See 1.4

1.4 Algorithmic Analysis

To compute the distance matrix, we must first allocate space for our distances, then execute the `BFS` on each vertex:

```
all_distances = NxN matrix                          // c1, executed 1 time,
allocate space for NxN distance matrix

for vertex_id in length(graph):                     // c2, executed N + 1
times, loop over every vertex in graph

    distances = graph.BFS(vertex_id)                // c3, exeecuted N times,
do the BFS

    all_distances[vertex] = distances               // c4, executed N times,
move/insert distances to vertex_id into matrix
```

Therefore, first we must determine the complexity of BFS to understand the complexity of computing the distance matrix.

The BFS algorithm is as follows:

```
set all vertex colors to white                                  // c1, executed
N times, set each initial 'unseen' color
set all vertex distances to nil                                 // c2, executed
N times, set each initial distances
set all vertex pi to -1                                         // c3, executed
N times, set each initial parent to non-vertex

set vertex V color to grey                                      // c1, executed
1 time, set our query vertex to grey
set vertex V distance to 0                                      // c2, exeucted
1 time, set self-distance

create queue Q                                                  // c4, executed
1 time, set up empty queue
Q.push(V)                                                       // c5, executed
1 time, add element to queue

while Q is not empty:                                           // c6, executed
at most N+1 times, iterate until queue is empty
    vertex = Q.pop()                                            // c7, executed
at most N times, get top of queue
    for each neighbor of vertex:                                // c8, executed
at most N - 1 times, get neighbors with adj list is O(1)
        color = get color of neighbor                           // c9, executed
at most N * (N - 1) times, get neighbors color
        if color == white:                                      // c10, executed
at most N * (N - 1) times, if unseen
            set neighor's color to grey                         // c1, executed
at most N - 2 times, set as seen (already seen our query V, so N-2)
            set neighbor's distance = vertex distance + 1       // c2, executed
at most N - 2 times, lookup distance and set it
            set neighbor's pi to vertex                         // c3, executed
at most N - 2 times, set parent
```

```
            Q.push(neighbor)                                    // c5, executed
at most N - 2 times, add to queue
    set vertex color to black                                   // c1 executed
at most N times, mark that we should not come back

all_accessible = new vector                                     // c11, executed
1 time, get array to store accessible vertices
for color in graph vertex colors:                               // c9 , executed
N + 1 times, iterate over each vertex's color
    if color == black:                                          // c12, executed
N times, if color is back, then we reached that node
        all_accessible.push_back(vertex)                        // c13, executed
at most N times, add accessible node to our list
```

Here I have analyzed solely the as a function of `N` vertices, therefore the maximum number of edges is `N-1`. Therefore, we can sum the constants as a function of `n` to yield: `complexity = c1(3n) + c2(2n) + c3(2n) + c4(1) + c5(n) + c6(n+1) + c7(n) + c8(n) + c9(n^2) + c10(n^2) + c11(1) + c12(n) + c12(n) + c13(n)`

The dominant term is the `n^2` term that comes from the worst case scenario of a complete graph wherein every time we query the neighbors it returns a list of all other vertices. This means that we are looking up the color and testing whether that color is white `n^2` times.

A more accurate description though is based not only on the number of vertices, but also on the number of edges since the case of a complete graph is not very common.

We can reframe this to be more applicable to normal graph use by declaring `V=number of vertices` and `E=number of edges`. Then, the above demonstrates when `E = V^2`. However, for a less densely connected graph we actually see that we still must visit each vertex, O(V). But, the number of neighbors to iterate over is proportional to the number of edges in the graph.

In a directed graph, we will always only travel along an edge one time to get to the neighbor on the other side, and enqueue that neighbor. In other words, the inner for loop runs once for each edge, making the total number of iterations of the inner loop proportional to the number of edges. In an undirected graph since we can move either direction along an edge, we will travel each edge twice. But, still the number of iterations of the inner loop is still proportional to the number of edges `E`.

Combining the queueing of `V` vertices with the visiting of each of the `E` edges yields a complexity of O(E + V).

We can re-capitualte the original analysis by substituting `E = V-1` into O(E + V), giving O(V^2 + V) = O(V^2) as we saw in the worst case of a complete graph.

Now, returning to the original question of the distance matrix computation, we see that the BFS is executed N times. Now that we know the BFS is O(n^2) for a complete graph, we can see the calculation of distance matrix will be N * N^2 or O(n^3).

Again, we can reframe to more accurately reflect the graphs we are dealing with and rephrase this as iterating over `V` vertices, or V * (V + E), leading to a complexity of O(V^2 + VE).

While the time complexity analysis is non-intuative with the solution of O(V + E) and took a while to come to, the spacial analysis is much more straight forward. The following memory allocations are requested during the BFS:

```
- array size V to hold colors
- array size V to hold distances
- array size V to hold pi
- queue with max size of V-1
- neighbor list with max size V
- all_accessible list with max size V
```

Here, all spacial considerations are a function of the number of vertices in the graph.

To support this theoretical analysis, I ran a set of tests to compare how increasing the number of vertices and edges affected the distance matrix calculation times.

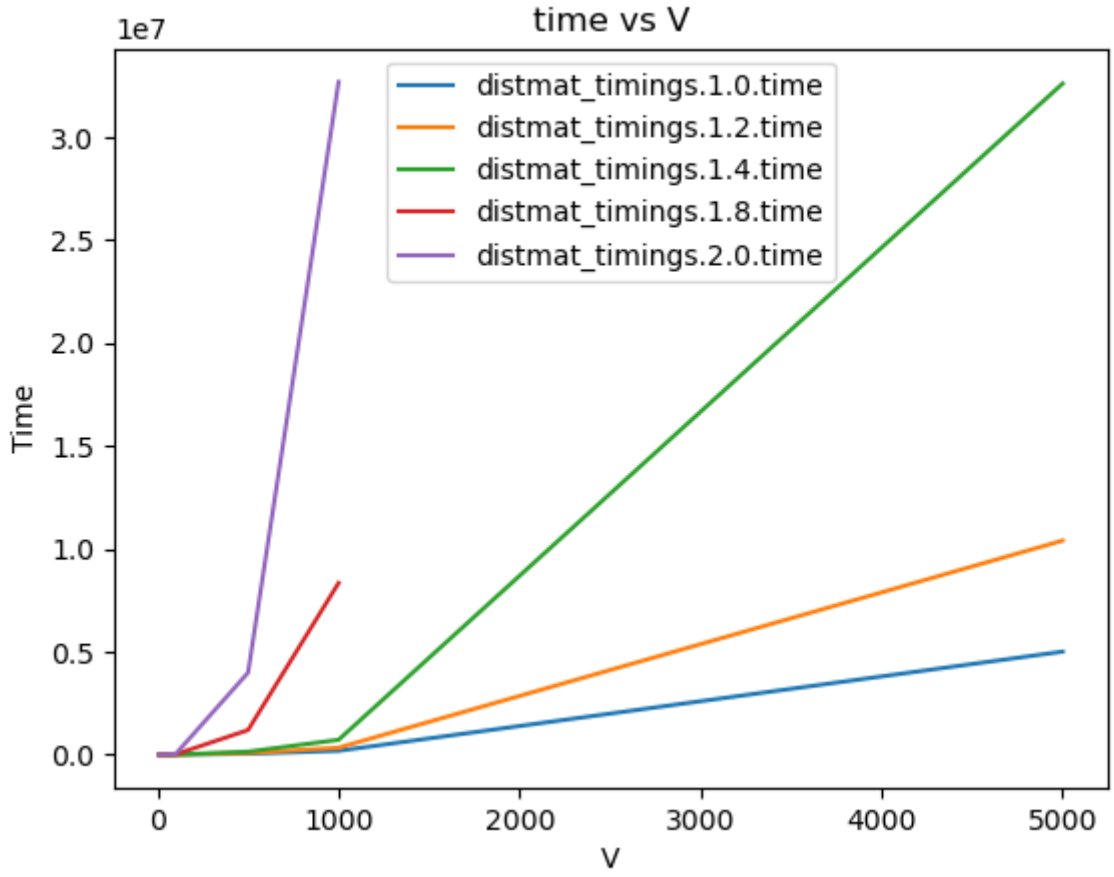
I initially tried using the ethane and octane examples, but they were too small to yield meaningful differences. Therefore, I left the use of those provided graphs simply as tests that the graph structure is correct and created the following test for analysing the time complexity:

```
- increase V from 10 to 5,000
- increase E as a function of V by E = v^e with e from 1 to 2
```

To construct each graph, I first made a vector of all possible edge pairs, then shuffled that vector, and took the first `E` pairings to make the edges. This should give more reasonable results than putting all the edges on the beginning indicies of the graph, for example.

The `test_distmat` target is provided in the `tests/Makefile` to run the series of calculations. The distance matrix calculation is preformed 3 times per graph to ensure accurate measurement.

Below, I plot the time in microseconds as a function of `V`, where the different curve are colored by the `e` value defined above. Therefore, we can see both how the number of vertices and number of edges affect the complexitiy:



Indeed, we see the expected time increase with respect to `V` is moderate, but the affect of `E` is dramatic! Clearly, the BFS term is highly dependent on `E` and again as we saw in the analysis above, it is O(V + E), but when approaching the complete graph that becomes O(V + V^2), dramatically increasing the time complexity. At `V=5000` with a sparse matrix the computation is faster than `V=500` with the complete graph! This helps cement what we learned before, which is the number of edges is proportional to the number of iterations in our inner loop and has a dramatic affect on the BFS algorithm's performance.