

Assignment 2: Dijkstra's Algorithm

By Joel Pillar-Rogers

Completed: 17 June 2020

Task A: Read in data from a GraphML file (5%)

Task B: Build an adjacency list or matrix representation of the graph (15%)

The first two tasks in Assignment 2 could be implemented simultaneously. There were several decisions to make. I decided to use an adjacency list to store the vertices in my graph as this is what I used in Lab03. Many of the graphs to build are sparse, and adjacency lists are generally faster than adjacency matrices for those.

```
Graph g = new AdjListDirectedGraph();
```

The Graphs in Lab03 used Strings as the Vertex values, so to switch to integers I just substituted Integer for String throughout. As these are both objects, the existing methods still worked. I couldn't use primitive ints because many of the data structures are Lists and Maps, which require objects to work.

```
public class Vertex implements Comparable<Vertex> {
    /**
     * The uniquely identifying label for the Vertex
     */
    private Integer label;
```

Finally, I added weights to each edge using a Map structure, that stored the target Vertex and the weight as an Integer. As in Lab03, each source Vertex is associated with a Map of its edges. I use the *vertices* List to index into the *adjList* List. It is important that the two ArrayLists remain in the order of insertion. Without ordering, I couldn't use the *vertices* list to index into the *adjList* structure.

```
public abstract class Graph {
    /**
     * Lists to maintain the vertices and a list of adjacent vertices with weights for each edge
     */
    protected List<Vertex> vertices = new ArrayList<>();
    protected List<Map<Vertex,Integer>> adjList = new ArrayList<>();
```

This also means that my *addEdge()* method needed to change to accommodate the weight input. The implementation of the Map is a HashMap, so that it is fast to access the Edges for each Vertex, and there is no need for the edges to stay sorted (if there is, I can export to a List and sort at the time).

```
void addEdge(Vertex v, Vertex w, int weight) {
    if(!vertices.contains(v)){
        vertices.add(v);
        adjList.add(new HashMap<>());
    }
    if(!vertices.contains(w)){
        vertices.add(w);
        adjList.add(new HashMap<>());
    }
    adjList.get(vertices.indexOf(v)).put(w,weight);
}
```

Here is the call to this function in GraphBuilder.java.

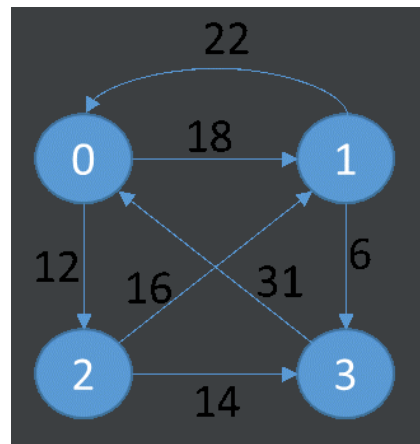
```
// here you could add an edge to your graph from source to target with the weight of nV
g.addEdge(nS,nD,nV);
```

I wrote a little print program to test my implementation to this point.

```
public void print(){
    for(Vertex vt : vertices){
        System.out.println("Node: " + vt.getLabel());
        System.out.print("Edges: ");
        Map<Vertex,Integer> m = adjList.get(vertices.indexOf(vt));
        for(Map.Entry<Vertex,Integer> e : m.entrySet()){
            System.out.print(e.getKey() + "(w:" + e.getValue() + "), ");
        }
        System.out.println();
    }
}
```

This produced the following accurate output for *graphSpecExample.graphml*.

```
graphSpecExample: 11
number of nodes: 4
number of edges: 7
max weight: 31
min weight: 6
Node: 0
Edges: 1(w:18), 2(w:12),
Node: 1
Edges: 0(w:22), 3(w:6),
Node: 2
Edges: 1(w:16), 3(w:14),
Node: 3
Edges: 0(w:31),
```



Task C: Use Dijkstra's to find shortest paths from a specified vertex (25%)

Implementing *dijkstraMap* and *Dijkstra* variables

To implement Dijkstra's Algorithm with an adjacency list, I decided to separate the Dijkstra data from the Vertex and Graph implementations. This would allow me to use the Vertex and Graph classes in other implementations later on. These data are saved in a separate class called *DInfo*, with these initial values. As these are public variables, I can easily update them from elsewhere in the program. I think this outweighs the benefits of encapsulation.

```
public class DInfo {
    /**
     * Information for Dijkstra's Algorithm
     * These values are always relative to a source Vertex
     */
    public boolean known = false;
    public int distance = 100000;
    public Vertex predecessor = null;
}
```

I used a HashMap instead of a List to save the state of each Vertex, with the state value represented by a *DInfo* object. Using a HashMap made look ups fast, and there are lots of look ups needed. There is also no need to order the contents. Also, this allows quick $O(1)$ look up of an Edge HashMap Vertex in this Dijkstra HashMap, without the need to search a list for an Edge target Vertex. This doesn't change the general structure or overall purpose of this Assignment but helps to avoid V^3 time later on.

```
protected Map<Integer, DInfo> dijkstraMap = new HashMap<>();

public void clearState(){
    dijkstraMap.clear();
}

public void setState(Vertex v, DInfo state){
    dijkstraMap.put(v.getLabel(), state);
}

public DInfo getState(Vertex v){
    if (dijkstraMap.containsKey(v.getLabel())){
        return dijkstraMap.get(v.getLabel());
    }
    // if the Vertex v has no state, return null
    return null;
}
```

At this point I had three key data structures:

- *vertices* - an ArrayList that holds my Vertex objects; used to index into the *adjList*
- *adjList* - an ArrayList that holds HashMaps for each Vertex that contain all its edges (target Vertices) and associated weights
- *dijkstraMap* - a HashMap that holds all the Dijkstra variables (DInfo) for each Vertex, relative to a given source Vertex

Implementing Dijkstra's Algorithm

There were three steps to implement Dijkstra's Algorithm. First, I initialised a DInfo with default values for each Vertex in the *dijkstraMap*. This is an $O(V)$ operation.

```
for(Vertex vt : vertices){
    setState(vt, new DInfo());
}
```

Secondly, I looped over each Vertex, updating the distance for each connected edge. The *adjacentTo()* method indexes into the *adjList* array to retrieve the weights of each Edge for this Vertex. Since the *adjList* is a HashMap, this takes $O(1)$ time. The whole operation is therefore $O(V)$ across all loops.

The second highlighted line is an inner for loop that will update the distance to a Vertex for each Edge in the Graph. Over the whole algorithm, this will only run once for each Edge, so is $O(E)$ time. Processing time for each inner loop will vary, but since *getState()* accesses the *dijkstraMap*, which is a HashMap, this method will run in $O(1)$ time. Everything else is either a getter method or an assignment so will also run in $O(1)$ time, i.e. won't scale with V or E . So, the inner loop still only adds $O(E)$ time for the whole algorithm.

```
for(int i = 0; i < vertices.size(); i++) {
    v.known = true;
    Map<Vertex, Integer> m = adjacentTo(current.index);
    if (!m.isEmpty()) {
        for (Map.Entry<Vertex, Integer> e : m.entrySet()) {
            DInfo w = getState(e.getKey());
            if (!w.known) {
                if (v.distance + e.getValue() < w.distance) {
                    w.distance = v.distance + e.getValue();
                    w.predecessor = current.getLabel();
                }
            }
        }
    }
}
```

Thirdly, I chose the next closest Vertex to the source where known is false. In the first highlighted line, an inner loop runs over each Vertex in *dijkstraMap* to find the closest unknown one. This will always run V times for each outer loop. So, for the whole algorithm will add $O(V^2)$ time.

The second highlighted line uses the key label of the closest Vertex to return the actual Vertex. This will be $O(V)$ time at worst, so will add up to $O(V^2)$ time, but often a lot less.

```
int min = 100001;
int next = -1;
for (Map.Entry<Integer, DInfo> e : dijkstraMap.entrySet()) {
    DInfo n = e.getValue();
    if(!n.known){
        if (n.distance < min) {
            min = n.distance;
            next = e.getKey();
        }
    }
}
if(next != -1){
    current = getVertex(next);
    v = getState(current);
    int d = 0;
}
```



Overall, this algorithm is $O(2V^2 + 2V + E)$, which will reduce to $O(V^2 + E)$ as V and E increase. I could probably avoid some of the extra V and V^2 with a bit more refactoring, but my priority was avoiding V^3 or E^2 operations, which I've achieved here.


Retrieving the shortest path

Retrieving the shortest path from each Vertex to the source was a bit trickier than I'd hoped. I had an outer loop running over each Vertex in *vertices*. Then for each Vertex, I built a path back to source by pushing predecessors onto a Stack. This would be $O(VE)$ at worst, if the Graph was more like a LinkedList, but usually a lot less, much closer to $O(V)$. I then popped each predecessor off to get a String of Vertex labels in the right order, which would also be $O(VE)$ at worst. Finally, I put these strings into a TreeMap so they would be sorted, which would be $O(\log V)$ for insertion. Printing each string would be an $O(V)$ operation at least. Overall, this method is $O(2VE + V + \log V)$, reducing to $O(VE)$ for larger Graphs and may be slower than my Dijkstra's algorithm implementation.

```
public void printShortestPaths(){
    TreeMap<Integer, String> m = new TreeMap<>();
    for(int i = 0; i < vertices.size(); i++){
        Vertex current = vertices.get(i);
        String str = "shortest path to " + current.getLabel() + ": ";
        Stack<Integer> s = new Stack<>();
        DInfo d = getState(current);
        int cost = d.distance;
        while(d.predecessor != -1){
            s.push(d.predecessor);
            d = dijkstraMap.get(d.predecessor);
        }
        while(!s.empty()) {
            str += s.pop() + " ";
        }
        str += current.getLabel() + ": cost = " + cost;
        m.put(current.getLabel(), str);
    }
    for(String s : m.values()){
        System.out.println(s);
    }
}
```

Running this code passed all the online tests.

Input	Expected	Got
 data/graphs/graphTutorialExample.graphml 1	shortest path to 0: 1 0: cost = 4 shortest path to 1: 1: cost = 0 shortest path to 2: 1 0 2: cost = 6 shortest path to 3: 1 0 3: cost = 5 shortest path to 4: 1 4: cost = 2 shortest path to 5: 1 0 2 5: cost = 9 shortest path to 6: 1 4 6: cost = 5	shortest path to 0: 1 0: cost = 4 shortest path to 1: 1: cost = 0 shortest path to 2: 1 0 2: cost = 6 shortest path to 3: 1 0 3: cost = 5 shortest path to 4: 1 4: cost = 2 shortest path to 5: 1 0 2 5: cost = 9 shortest path to 6: 1 4 6: cost = 5 

Passed all tests! 

I tested it on a much bigger Graph (graph1000) and also got the correct results.

shortest path to 995: 0 636 648 213 678 414 921 93 864 81 698 258 975 287 261 782 481 889 983 93	996	996	shortest path to 995: 0 636 648 213 678 414 921 93 864 81 698 258 975 287 261 782 481 889 983 93
shortest path to 996: 0 566 391 592 475 919 598 584 146 783 58 986 543 57 682 648 882 843 487 45	997	997	shortest path to 996: 0 566 391 592 475 919 598 584 146 783 58 986 543 57 682 648 882 843 487 45
shortest path to 997: 0 636 648 213 678 414 921 93 864 81 698 258 975 288 181 292 161 788 289 27	998	998	shortest path to 997: 0 636 648 213 678 414 921 93 864 81 698 258 975 288 181 292 161 788 289 27
shortest path to 998: 0 636 648 213 678 414 921 93 864 81 698 258 975 288 181 292 161 788 289 27	999	999	shortest path to 998: 0 636 648 213 678 414 921 93 864 81 698 258 975 288 181 292 161 788 289 27
shortest path to 999: 0 636 648 213 678 414 921 93 864 81 698 258 975 287 261 782 481 889 983 93	1000	1000	shortest path to 999: 0 636 648 213 678 414 921 93 864 81 698 258 975 287 261 782 481 889 983 93

My commit number was **3602** at this point. I'll implement the Java PriorityQueue before doing profiling.

Task E: Compare the performance of a Map, Vertex-only and Java's PriorityQueue for storing the tentative distances (20%)

Creating a Vertex-only implementation

After some more thought, I realised the Map implementation for holding distances had a lot of unnecessary overhead. Instead of using a separate DInfo object, why not save these variables directly in each Vertex? Then I could also get rid of the Map structure for holding distances altogether. Here is a side by side comparison of these implementation:

```
public void runDijkstra(Vertex s){
    clearState();
    for(int i = 0; i < vertices.size(); i++){
        setState(vertices.get(i), new DInfo());
        vertices.get(i).index = i;
    }

    Vertex current = s;
    DInfo v = getState(current);
    v.distance = 0;

    for(int i = 0; i < vertices.size(); i++) {
        v.known = true;
        Map<Vertex, Integer> m = adjacentTo(current.index);
        if (!m.isEmpty()) {
            for (Map.Entry<Vertex, Integer> e : m.entrySet()) {
                DInfo w = getState(e.getKey());
                if (!w.known) {
                    if (v.distance + e.getValue() < w.distance) {
                        w.distance = v.distance + e.getValue();
                        w.predecessor = current.getLabel();
                    }
                }
            }
        }
        int min = MAX + 1;
        int next = -1;
        for (Map.Entry<Integer, DInfo> e : dijkstraMap.entrySet()) {
            DInfo n = e.getValue();
            if (!n.known) {
                if (n.distance < min) {
                    min = n.distance;
                    next = e.getKey();
                }
            }
        }
        if (next != -1) {
            current = getVertex(next);
            v = getState(current);
        }
    }
}
```

```
public void runDijkstra(int s){
    Vertex v = null;
    for(int i = 0; i < vertices.size(); i++){
        Vertex vt = vertices.get(i);
        vt.distance = MAX;
        vt.known = false;
        vt.predecessor = null;
        vt.index = i;
        if (vt.getLabel() == s) {
            v = vt;
        }
    }
    v.distance = 0;

    for(int i = 0; i < vertices.size(); i++) {
        v.known = true;
        Map<Vertex, Integer> m = adjacentTo(v.index);
        if (!m.isEmpty()) {
            for (Map.Entry<Vertex, Integer> e : m.entrySet()) {
                Vertex w = e.getKey();
                if (!w.known) {
                    if (v.distance + e.getValue() < w.distance) {
                        w.distance = v.distance + e.getValue();
                        w.predecessor = v;
                    }
                }
            }
        }
        int min = MAX + 1;
        for (Vertex n : vertices) {
            if (!n.known) {
                if (n.distance < min) {
                    min = n.distance;
                    v = n;
                }
            }
        }
    }
}
```

This new implementation saves time and space. There is a whole map data structure of Integer and DInfo objects gone, and these both scaled at $O(V)$. In terms of time efficiency:

- Loop 1 will remain $O(V)$.
- Loop 2 will remain $O(V)$ and Operation 3 $O(1)$ will still combine to make up to $O(V)$ overall
- Loop 4 will remain $O(E)$
- Loop 5 will remain $O(V)$, and combined with Loop 2 will remain $O(V^2)$ overall

This implementation is $O(V^2 + 2V + E)$, while the Map implementation was $O(2V^2 + 2V + E)$. Both reduce to $O(V^2 + E)$ for larger graphs, but the Vertex-only implementation will be faster.

A key point for both implementations is that the longest part is Loop 5, finding the next closest Vertex. It is a full V^2 operation, with no shortcuts. The next section will explore how a PriorityQueue might shorten this.

Creating a Java PriorityQueue implementation

Java has a built in PriorityQueue, which I used for the next implementation. The PriorityQueue's only purpose is to return the smallest (or largest) object in a collection by some inherent value. I had some trouble visualising what objects to put in my PriorityQueue: Integers, Vertexes, DInfo? In the end, I settled on creating a new object type that I had complete control over, Distance objects. These simply hold a Vertex and a distance int, representing how far this Vertex is from the source Vertex. They also have a simple *compareTo()* implementation to allow the PriorityQueue to find the smallest.

```
protected PriorityQueue<Distance> distances;

public class Distance implements Comparable<Distance>{
    int distance;
    Vertex vertex;

    public Distance(Vertex v, int d){
        vertex = v;
        distance = d;
    }

    @Override
    public int compareTo(Distance d) {
        return distance - d.distance;
    }
}
```

As with the Vertex-only implementation above, I had to make sure that each Vertex object was only made once and shared by reference to all the data structures. Otherwise updating their values in one structure wouldn't them update in others. This requires tight control of any *new Vertex()* operation.

A side by side implementation vs the Vertex-only implementation is below. In terms of size efficiency, the PriorityQueue adds a new data structure full of Distance objects. There will eventually be a Distance object for every Edge, $O(E)$, because old Distance objects are not removed. This saves time but obviously increases space. The PriorityQueue implementation might therefore be even worse than the Map implementation in terms of space efficiency.

The time efficiency (using the red numbers below) will look like this:

- Loop 1 will remain $O(V)$.
- Operation 2 is a PriorityQueue *add()* operation, but since it is adding new Distance objects with the same distance value, it shouldn't need to do any reordering after insertion. So, I expect this to be an $O(1)$ operation.
- Operation 3 is a new operation, inserting into the PriorityQueue, that will take $O(\log V)$ this time
- Loop 4 will remain $O(V)$
- Operation 5 will remain an $O(1)$ operation, indexing into the *adjList* Edge HashMap
- Loop 6 will remain $O(E)$

- Operation 7, adding smaller distances to the PriorityQueue, will start as an $O(\log V)$ operation, but since larger distances for each Vertex aren't being removed, this will degenerate to an $O(\log E)$ operation. Combined with Loop 6, this becomes an $O(E \log E)$ operation overall.
- Loop 8 is a while loop that discards known Vertices. It will usually run once, but sometime may run a few times at most. So, this is $O(V)$ overall.
- Operation 9, removing the top element of the PriorityQueue, will also degenerate to an $O(\log E)$ operation over time, as in Operation 7. Combined with Loop 4, this becomes an $O(V \log E)$ operation overall.

```

public void runDijkstra(int s){
    Vertex v = null;
    for(int i = 0; i < vertices.size(); i++){
        Vertex vt = vertices.get(i);
        vt.distance=MAX;
        vt.known=false;
        vt.predecessor=null;
        vt.index = i;
        if(vt.getLabel() == s){
            v=vt;
        }
    }
    v.distance = 0;

    for(int i = 0; i < vertices.size(); i++) {
        v.known = true;
        Map<Vertex, Integer> m = adjacentTo(v.index);
        if (!m.isEmpty()) {
            for (Map.Entry<Vertex, Integer> e : m.entrySet()) {
                Vertex w = e.getKey();
                if (!w.known) {
                    if (v.distance + e.getValue() < w.distance) {
                        w.distance = v.distance + e.getValue();
                        w.predecessor = v;
                    }
                }
            }
        }
        int min = MAX + 1;
        for (Vertex n : vertices) {
            if (!n.known) {
                if (n.distance < min) {
                    min = n.distance;
                    v = n;
                }
            }
        }
    }
}

```

```

public void runDijkstra(int s){
    Vertex v = null;
    distances = new PriorityQueue<>();
    for(int i = 0; i < vertices.size(); i++){
        Vertex vt = vertices.get(i);
        vt.distance = MAX;
        vt.known = false;
        vt.predecessor = null;
        vt.index = i;
        distances.add(new Distance(vt, MAX));
        if (vt.getLabel() == s) {
            v = vt;
        }
    }
    v.distance = 0;
    distances.add(new Distance(v, 0));

    for(int i = 0; i < vertices.size(); i++) {
        v.known = true;
        Map<Vertex, Integer> m = adjacentTo(v.index);
        if (!m.isEmpty()) {
            for (Map.Entry<Vertex, Integer> e : m.entrySet()) {
                Vertex w = e.getKey();
                if (!w.known) {
                    if (v.distance + e.getValue() < w.distance) {
                        w.distance = v.distance + e.getValue();
                        w.predecessor = v;
                        distances.add(new Distance(w, w.distance));
                    }
                }
            }
        }
    }
    while(v.known && !distances.isEmpty()) {
        v = distances.poll().vertex;
    }
}

```

The PriorityQueue implementation is therefore $O(V + \log V + V \log E + E \log E)$ overall. This will reduce to $O(V \log E + E \log E)$ at higher numbers. By comparison, the Vertex-only implementation of the algorithm was $O(V^2 + 2V + E)$, which reduces to $O(V^2 + E)$. This is precisely what is suggested by theory, which is nice to see.

To summarise my implementations to this point, these are my theoretical space and time efficiencies, which I'll test empirically in a moment. The Vertex-only graph should be the most space efficient, while the Java PriorityQueue will quickly become the least space efficient as the E to V ratio increases. For time, Map is the slowest, while Vertex-only and Java PQ will compete, based on the ratio of E to V. At lower ratios, the Java PriorityQueue should perform better. At higher ratios, the Vertex-only implementation should perform better.

Efficiency/GraphType	Map	Vertex-only	Java PriorityQueue
Space	$O(2V + E)$	$O(V + E)$	$O(V + 2E)$
Time	$O(2V^2 + 2V + E)$	$O(V^2 + 2V + E)$	$O(V + \log V + V \log E + E \log E)$

Profiling

To profile my implementations, I used the profiler I built (based on Trent's example) for Assignment 1. This uses the *System.nanoTime()* method to collect timestamps and the Runtime class for memory testing and garbage collection.

For both time and memory tests, I ran a for loop over each file, and an inner for loop over each implementation. I added a switch statement to GraphBuilder.java to account for this.

```
// main loop to test implementations
// loops over the filenames in files
for (String file : files) {
    // loops over the 4 different graph types
    for (int i = 0; i < 4; i++) {

        // set the file for this loop
        filename = file;

        // set the Graph type for this loop
        graphType = i;

        // Test this implementation
        timeTest( loops: 1); // initial test to warm things up
        timeTest( loops: 10);
    }
}
```

```
Graph g = null;
switch(type){
    case 0:
        g = new DistanceInMapGraph();
        break;
    case 1:
        g = new DistanceInVertexGraph();
        break;
    case 2:
        g = new DistanceInJavaPQGraph();
        break;
    case 3:
        g = new DistanceInMyPQGraph();
        break;
}
```

First, I doubled checked that the output for each implementation was identical. Here are some snippets from the *random_v100_e1000_w50* test:

Map

```
shortest path to 96: 0 33 19 96: cost = 27
shortest path to 97: 0 50 97: cost = 22
shortest path to 98: 0 50 42 6 98: cost = 34
shortest path to 99: 0 50 62 99: cost = 11
Loops: 1
```

Vertex-only

```
shortest path to 96: 0 33 19 96: cost = 27
shortest path to 97: 0 50 97: cost = 22
shortest path to 98: 0 50 42 6 98: cost = 34
shortest path to 99: 0 50 62 99: cost = 11
Loops: 1
```

Java PriorityQueue

```
shortest path to 96: 0 33 19 96: cost = 27
shortest path to 97: 0 50 97: cost = 22
shortest path to 98: 0 50 42 6 98: cost = 34
shortest path to 99: 0 50 62 99: cost = 11
Loops: 1
```

Although they look identical, they are screen captures for different Graphs, so this was a good confidence builder that everything was okay. They also matched perfectly with the expected output.

Next, I ran the *timeTest()* below and recorded the details for each Graph in an Excel sheet.

```
public static void timeTest(int loops) throws FileNotFoundException, IOException, JDOMException {

    System.out.println("timeTest");
    System.out.println(filename);
    System.out.println("Graph: " + graphType);

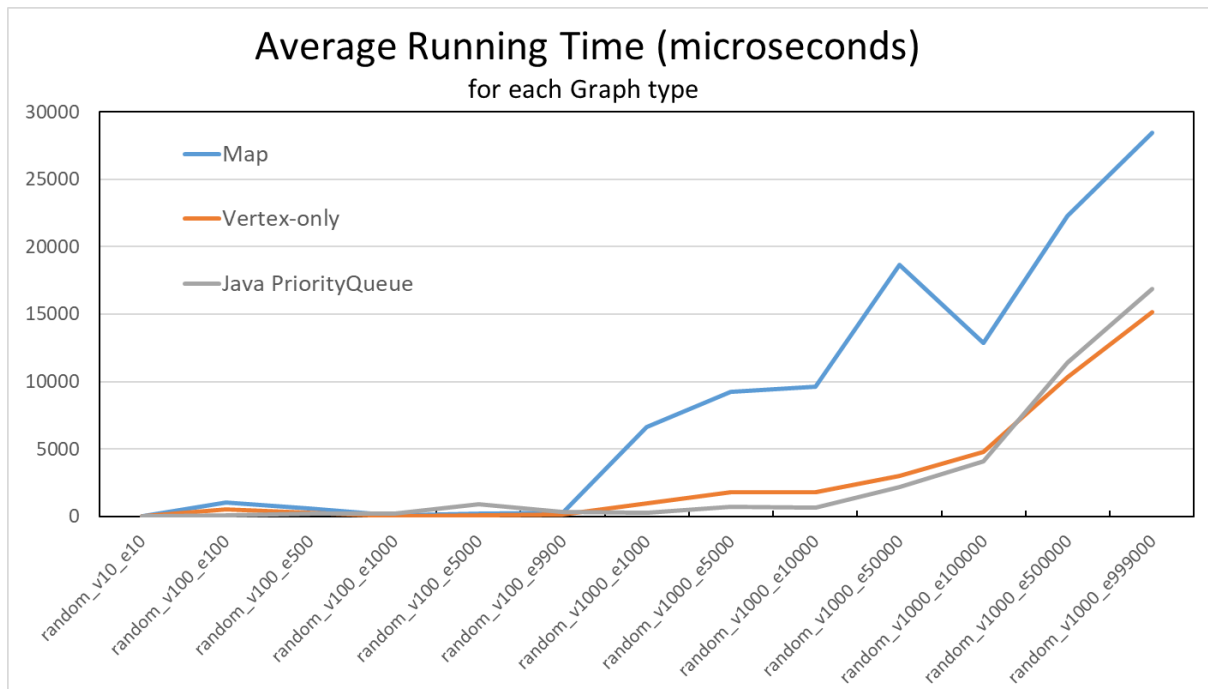
    // instantiate the profiler object
    GraphProfiler profiler = new GraphProfiler();

    // run loops
    profiler.avgReset(loops);
    for(int i = 0; i < loops; i++){
        // build graph
        Graph g = GraphBuilder.buildFromGraphML(filename, graphType);

        profiler.avgTic();
        // run Dijkstra's algorithm
        g.runDijkstra(sourceVertex);
        profiler.avgToc();

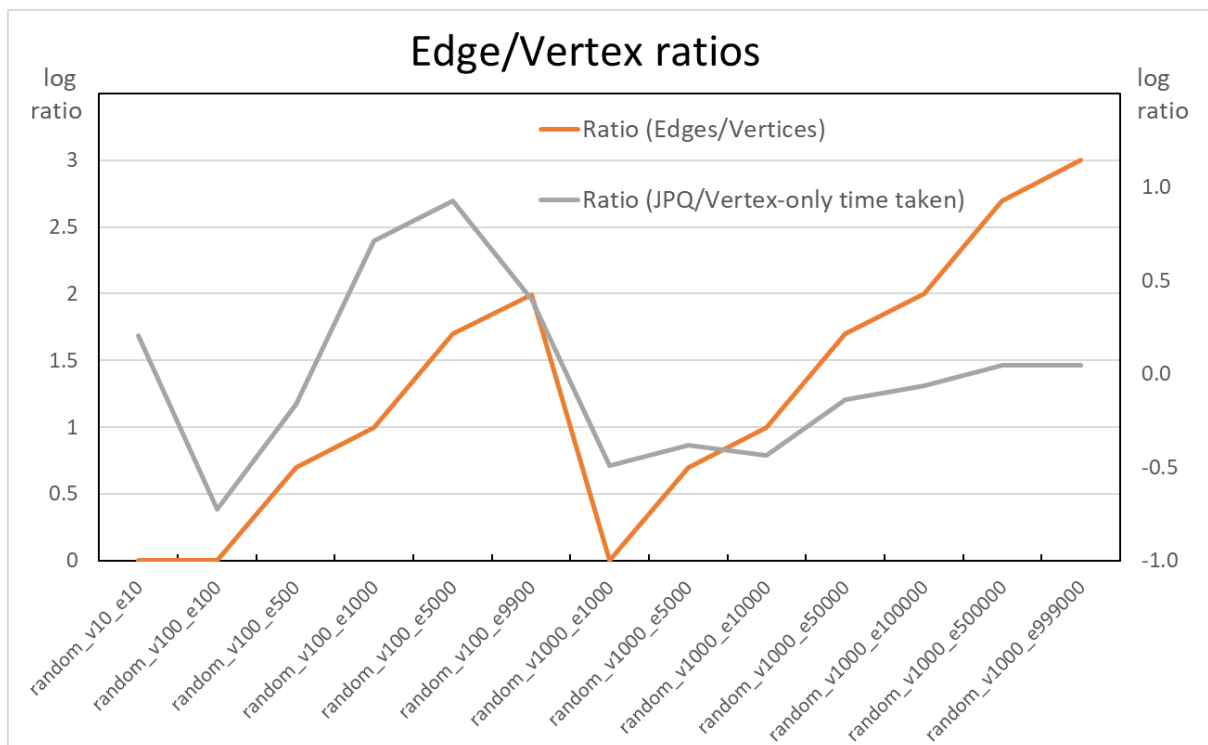
        g = null;
        profiler.takeOutGarbage();
    }
    System.out.println("Loops: " + loops);
    System.out.println("Average time (ns): " + profiler.avgCalc());
    System.out.println("Average time (ms): " + GraphProfiler.nsToMs(profiler.avgCalc()));
}
```


This produced the following graph.



As predicted, the running time for the Map was the worst, while the Vertex-only and Java PriorityQueue were similar. There was an interesting thing happening with those latter two lines however, which is illustrated in another graph.

As the ratio of Edges to Vertices increased, the time taken by the Java PriorityQueue increased more than the Vertex-only graph. This was theoretically predicted, because the Vertex-only graph runs $O(V^2 + E)$ time, whereas the Java PriorityQueue runs in $O(V \log E + E \log E)$. So, this was a nice confirmation.



I now ran similar tests for memory. My memory test method is below. I separated out the memory used for the Graph from the memory used by Dijkstra's algorithm. This might yield valuable insights in the results.

```

public static void memTest() throws FileNotFoundException, IOException, JDOMException {

    System.out.println("memTest");
    System.out.println(filename);
    System.out.println("Graph: " + graphType);

    // instantiate the profiler object
    GraphProfiler profiler = new GraphProfiler();

    // build graph
    profiler.ticMem();
    Graph g = GraphBuilder.buildFromGraphML(filename, graphType);
    long graphMemory = profiler.tocMem();

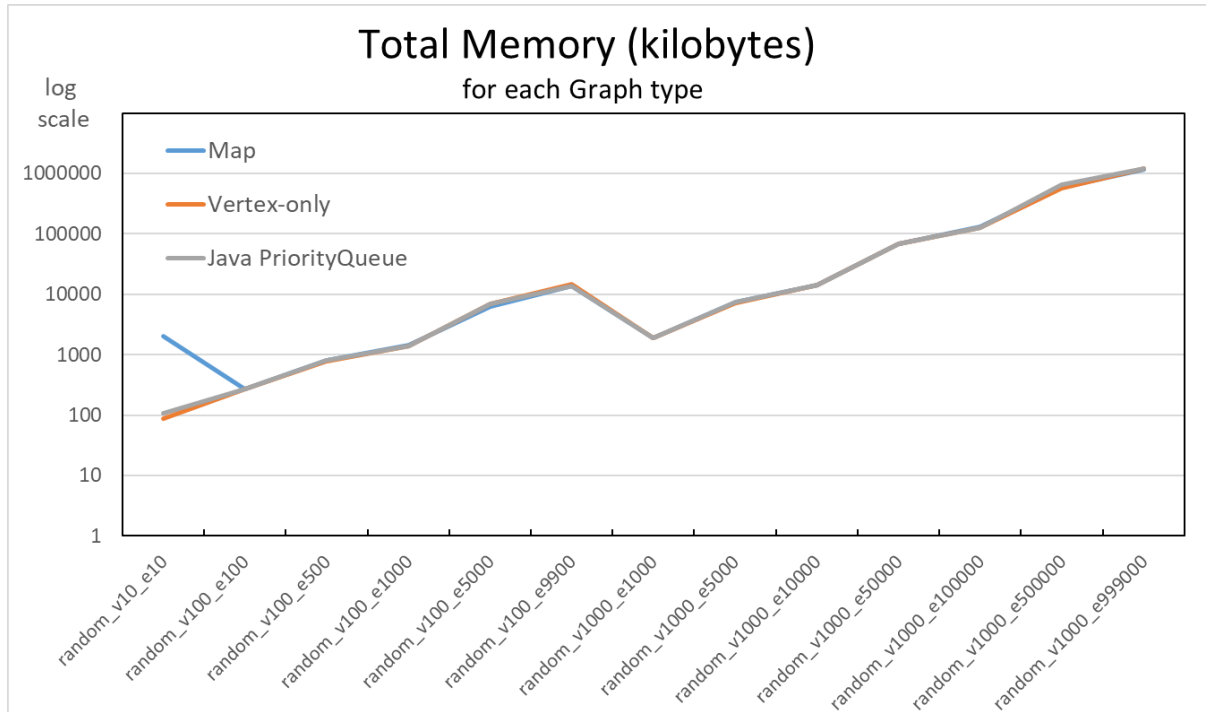
    // run Dijkstra's algorithm
    profiler.ticMem();
    g.runDijkstra(sourceVertex);
    long dijkstraMemory = profiler.tocMem();

    g = null;
    profiler.takeOutGarbage();

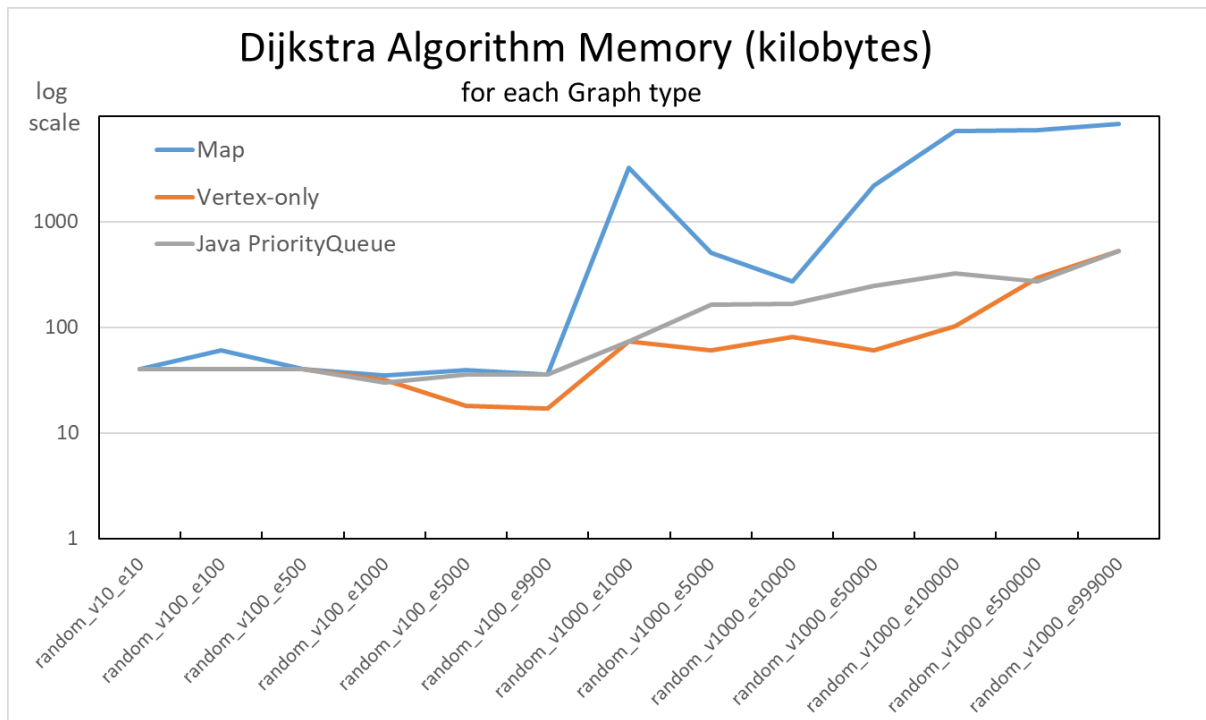
    System.out.println("Graph memory (kB): " + graphMemory/1000);
    System.out.println("Dijkstra memory (kB): " + dijkstraMemory/1000);
    System.out.println("Total memory (kB): " + (graphMemory + dijkstraMemory)/1000);
    System.out.println();
}

```

I recorded the details for each Graph memory test in an Excel sheet. Total memory was virtually identical for each Graph type. They all build the same graph, and graph memory is 99% of total memory, so this should have been expected. On reflection, my space theoretical estimates above weren't correct because even though each Graph has a different number of data structures, I generally only have one copy of each Vertex and Edge, with references to their memory locations stored in each data structure.



The more interesting differences were in the memory used for the Dijkstra implementation. In the chart below, the Vertex-only Graph uses the least amount of memory, as expected. It doesn't have a PriorityQueue filled with Distance objects, or a Map filled with DInfo objects. More interestingly, the Map uses the most memory by far (this is a log scale). I believe this is because in that implementation, I didn't strictly monitor uses of *new Vertex()*, because the structure was tolerant of different Vertices with the same *label* variable. So, it is probably creating a lot more Vertices overall.



Finally, the hit to memory wasn't really noticeable on my simple Windows Explorer profiler until the number of Edges got to 500,000 and 999,000. This program didn't blow up my memory like the SuffixTries in Assignment 1. From my data, the maximum memory used by any Graph was around 1.2 gigabytes.

Memory

16.0 GB DDR3

Memory usage

15.9 GB



My code to this point was committed as revision **3645**.

Task F: Implement your own Priority Queue and compare (15%)

Implementing my own PriorityQueue

I now turned to implementing my own PriorityQueue. The goal was to create one at least as efficient as Java, or potentially more efficient, given I could tailor it to suit my specific needs.

However, it should be noted that one potential gain mentioned by Trent in the instructions is probably not possible. The instructions say making your own PriorityQueue: “should allow optimisations... e.g. **insert new value** instead of **full update**”. However, the insert new value methods for Java's priority queue, *add()* or *offer()* only do a single sift up operation, in $O(\log n)$ time (see the Java 13 API source

code). This is the method I used for adding distances to my PriorityQueue, so it will be hard to improve on. There should be other savings to make, however. I wanted to do a really lightweight implementation of the PriorityQueue, with only the minimum overhead needed to save and retrieve the smallest values in $O(\log n)$ time.

I started with my key variables, an array and a size int to keep track of the “end” of my array. I made my array a Distance[], my first complexity saving relative to the PriorityQueue which has to be able to handle any object type.

```
Distance[] array;
int size = 0;
```

I also wanted to avoid array resizing operations. I knew the number of Distance objects in my array would approach the number of Edges, plus one for each Vertex initialised to 100,000. This number (Edges + Vertices) can be read directly from the GraphML files. I changed some code to allow this number to be passed to my Graph on construction, and then passed to MyPriorityQueue on its construction.

```
int total = graph.getChildren().size();
Graph g = null;
switch(type){
    case 1:
        g = new DistanceInMapGraph();
        break;
    case 2:
        g = new DistanceInVertexGraph();
        break;
    case 3:
        g = new DistanceInJavaPQGraph();
        break;
    case 4:
        g = new DistanceInMyPQGraph(total);
        break;
}
```

```
/**
 * Constructor for MyPriorityGraph
 * Need to pass in edges + vertices
 */
public DistanceInMyPQGraph(int edges){
    this.edges = edges;
}
```

```
public void runDijkstra(int s){
    // reset all the Dijkstra variables in each Vertex
    // add a new Distance object for each Vertex to the
    // this will be an  $O(V)$  operation
    Vertex v = null;
    distances = new MyPriorityQueue(edges);
```

```
/**
 * Constructor for this data structure
 * Will set up the array as the max needed to hold
 *
 * @param edges is the sum of edges and vertices f
 */
public MyPriorityQueue(int edges){
    array = new Distance[edges];
}
```

The two main operations I needed to be able to perform were *add()*/*insert()* and *poll()*/*remove()*. The *add()* method would require an underlying *siftUp()* method to maintain partial ordering. I referenced both the lectures notes and the Java 13 PriorityQueue source code to build the following methods.

```
public void add(Distance d){
    size++;
    siftUp(size-1, d);
}

/**
 * Add the given Distance object to the array list
 * Needs to repeatedly compare with the parent while it is smaller
 * Maintains partial ordering of the PriorityQueue
 * @param i the index into the array, representing parents/children in binary tree
 * @param d a Distance object to add
 */
private void siftUp(int i, Distance d){
    while(i > 0){
        int parent = (i - 1)/2; // use logical model to know where parent is
        Distance temp = array[parent];
        if(d.compareTo(temp) < 0){ // if d is not smaller than the parent...
            break; // ...break...
        }
        array[i] = temp; // ...otherwise switch the parent into the child...
        i = parent; // ...and keep moving up
    }
    array[i] = d; // if d is not smaller than parent, save d at current index
}
```

The *siftUp()* method doesn't really perform swaps. Rather, it continuously assigns parents to children as long as they are bigger than the newly added object, starting from the last position in the array. Finally, it puts the new object in its correct spot.

The *poll()* and *siftDown()* methods work in a similar manner, although in reverse. After removing the smallest item in position 0, it compares all the children of each parent to see if they are smaller than a reference Distance object, which was the last element in the array. The only additional complication is working out which of the two children is bigger before assigning them to the parent position.

```
public Distance poll(){
    size--;
    Distance d = array[0];
    siftDown();
    return d;
}

/**
 * Sift down by taking the last Distance object in the array,
 * and comparing it with each element from the top down
 */
private void siftDown(){
    int half = size / 2;           // the last row is at least 2x all the previous nodes
    int i = 0;
    Distance d = array[size];      // grab the last element, which is now 1 past "end" of array
    while(i < half){               // if i == half, i has no children; i < half must have children
        int child = (2*i) + 1;
        int rChild = child + 1;
        Distance test = array[child];
        if(rChild < size &&
           array[child].compareTo(array[rChild]) > 0){ // if right child exists...
            // ...and it's bigger than left child...
            test = array[rChild]; // ...test against right child...
            child = rChild;       // ...and update path for next child (if needed)
        }
        if(d.compareTo(test) <= 0){ // if d is smaller or same size as largest child...
            // ...break...
            break;
        }
        array[i] = test;           //...otherwise, move largest child into parent position
        i = child;                 //...and keep moving down
    }
    array[i] = d;                 // save d at it's right position
}
```

The implementation of MyPriorityQueue in a Graph structure was virtually identical to the Java PriorityQueue graph. The only difference was the instantiation of the PriorityQueue structure. Every other usage was exactly the same, including all method calls.

```
distances = new PriorityQueue<>();
```

```
distances = new MyPriorityQueue(edges);
```

```
distances.add(new Distance(vt, MAX));
```

```
distances.add(new Distance(vt, MAX));
```

```
while(v.known && !distances.isEmpty()) {
    v = distances.poll().vertex;
}
```

```
while(v.known && !distances.isEmpty()) {
    v = distances.poll().vertex;
}
```

I tested the results of these implementations and found them working as expected. These are the results for the two implementations for *random_v100_e100_w50.graphml*. This is the correct output for both.

```
shortest path to 96: NO PATH
shortest path to 97: NO PATH
shortest path to 98: 0 7 57 52 31 99 98: cost = 159
shortest path to 99: 0 7 57 52 31 99: cost = 123
Loops: 1
```

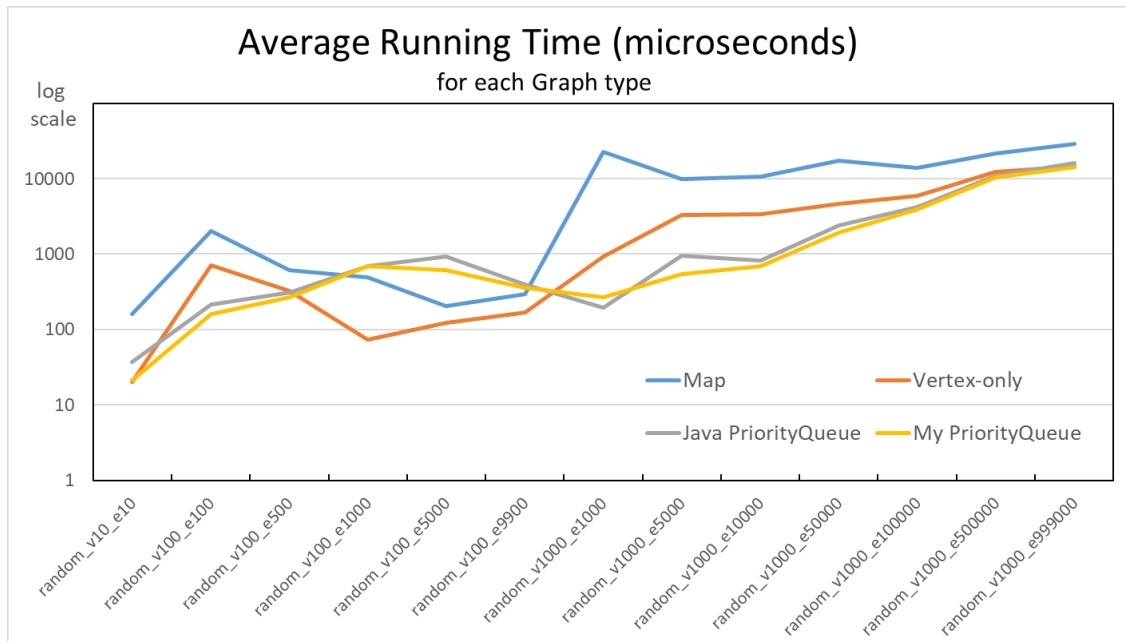
```
shortest path to 96: NO PATH
shortest path to 97: NO PATH
shortest path to 98: 0 7 57 52 31 99 98: cost = 159
shortest path to 99: 0 7 57 52 31 99: cost = 123
Loops: 1
```

I moved on to profiling.

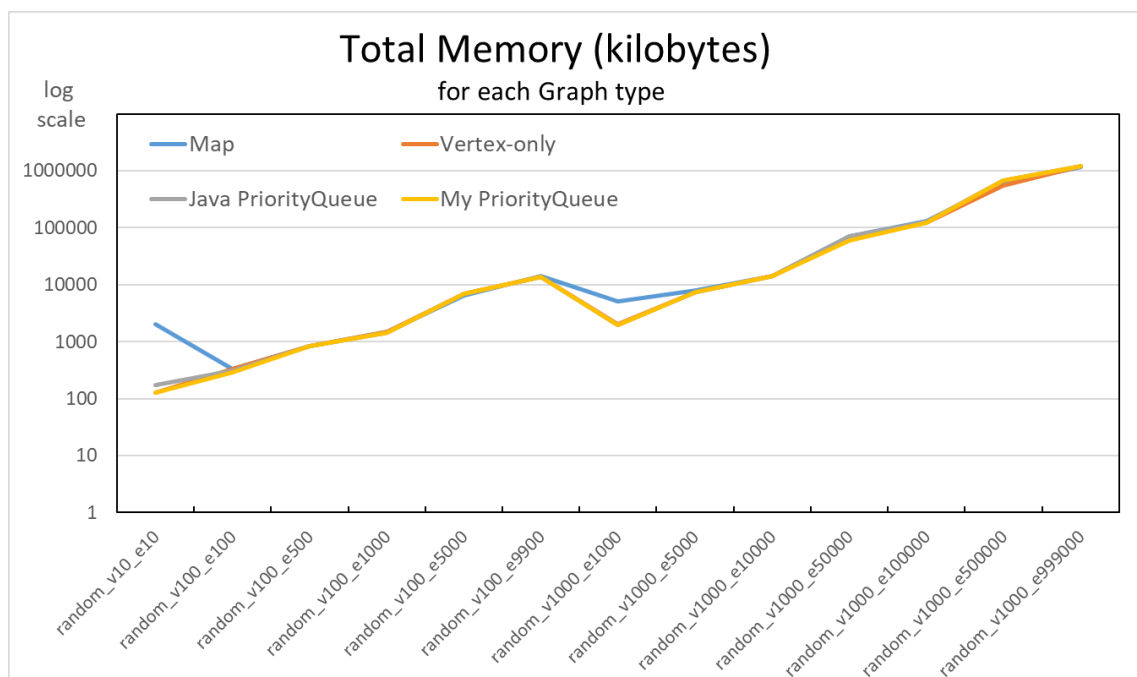
Profiling MyPriorityQueue vs other implementations

I ran the same tests for MyPriorityQueue as I ran for the other implementations. I expected my implementation to be at least as fast as the Java implementation. Without array sizing operations, generics options, comparator/comparable checking, some additional reference passing, and other minor bits of overhead, my implementation should be a tiny bit faster. There are no major theoretical differences in the $O(n)$ time of the two implementations, as far as I can see.

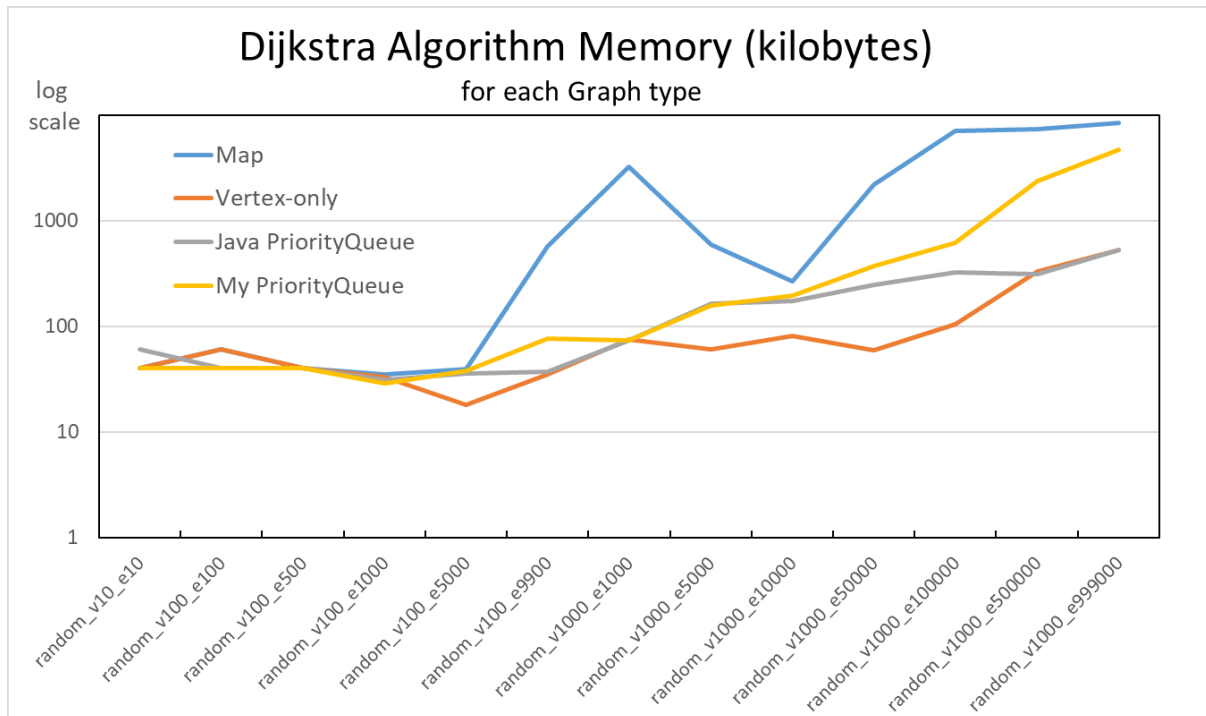
The graph below shows the time taken for each implementation across many tests. The times for the PriorityQueuees are so close as to be negligible, and illustrates they are operating under the same $O(n)$ time. I take some satisfaction that my implementation wins by a small margin more often than not. You can again see clearly that the Map and Vertex-only implementations perform comparatively better than the PriorityQueue implementations at higher Edge-to-Vertex ratios.



I moved on to profiling memory for MyPriorityQueue. The first graph shows again that there is almost no difference in the total memory footprint across the four implementations. This makes a lot of sense, since there is usually only one object for each Vertex, referenced by each data structure. Additionally, all four implementations extend the Graph abstract class, where the main data structures are.



The more interesting graph shows the memory used by the Dijkstra algorithm. My decision to pre-set the size of the underlying array in my PriorityQueue was aimed at avoiding resizes and saving some time. Here, it is obvious that this is leading to a much larger memory footprint than the array resizing in the Java PriorityQueue.



This concludes my analysis and code development on the Dijkstra Algorithm.

My final code commit was **3671**.