CSC2001F Assignment 5 Report: Dijkstra’s Algorithm Measurement.

By Noah Gonsenhauser,
GNSNOA001

Object Oriented Design:

In this assignment I used the provided classes in order to retain functionality with Dijkstra's algorithm, along with creating my own separate class called "GenerateGraphs.java" which contains a function called "GenerateNewGraphs() "which allows for it to be called at a later stage by another program but if "GenerateGraphs.java" is run on it's own it will call "GenerateNewGraphs()" which will generate said new graphs. This is useful as when running the make file it allowed me to add a separate make command which only ran the graph java file which generated all the data about Dijkstra's algorithm for analysis without constantly generating new data every time. For instance, if I type:
"make run": it will first generate new graphs, then generate the data about said graphs.
"make graph": will only generate the data about the graphs without generating new graphs
"make generate": will generate only new graphs without outputting the data
And along with "make": on it's own, along with all of the aforementioned make commands will also generate javadocs under the "docs" folder and class files under the "bin" folder.
"make clean": will delete all javadocs, graphs, data output and class files in the bin folder, while "make alt": will delete all of the same things minus the graphs, again useful for when editing the graph file.

The "GenerateNewGraphs()" function will generate graphXXX.txt files under the "data" folder, where "XXX" would be the number of the given graph. When running the Graph.java program it interacts with Path, Vertex, Edge and GraphException classes in order to run the Dijkstra algorithm over each of the graph files, and then print the information on each graph to the log, and to a text file misspelt to "Djikstraout.txt" in order to be analysed later. The class I created "GenerateGraphs.java" does not directly interact with "Graphs.java" and rather only generates the data that "Graphs.java" will use when run. The Path Edge and Vertex classes are needed by the "Graphs.java" program in order to be stored in the priority queue and then processed by the algorithm. The GraphException class is used to throw errors with the graph such as "Graph has negative edges" as Dijkstra's algorithm cannot process negative edges, or the "NoSuchElementException."

Goal:

My Goal for this experiment is to determine the relationships between the number of vertexes of a graph, number of edges of a graph, number of operations needed to find the shortest path in a graph, and the time complexity of that graph. It should also be expected that the upper bound for performance of Dijkstra's algorithm is $|E|\log_2|V|$ as the algorithm uses a binary heap to implement the priority queue, which involves dividing the elements into two halves repeatedly. I decided that in order to best compare the given variables, I would select a set of vertices (10,20,30,40,50) and for each of them, in order to avoid sparse graphs, as the maximum number of edges for a given graph is $|V|^2$, I decided to get 10 edges that were between 20% and 80% of $|V|^2$ for any given graph. I settled on incrementing this number by 6% each time in order to get 10 equal graphs between them, so in order to determine the number of edges for a given graph, I would take $|V|^2$ and times it by 0.26, then 0.32, 0.38 and so on until I had the corresponding number of edges for all given graphs. I also could modify these values very easily if I so desired and would then be able to get more or less graphs should I need them and furthermore I believed that this method would best show the relationship between the number of edges and the number of operations. I would execute this experiment by running the graph generation algorithm which randomly generated paths and edges that followed the rules of not making the starting and ending vertex equals and not generating paths that already existed. I would then run my graph generation program which would run the algorithm on each graph, counting the number of operations by incrementing the variable "opcount." The operation counter is be incremented by log(pq.size)/log(2) when a comparison is made in the priority queue while it loops over the verticies. it does this by estimating the number of times the priority queue was split in half during the binary heap implementation. it also increments by 1 every time an edge is processed and again by log(pq.size)/log(2) every time a new vertex is added to the priority queue for the same reason as before. This results in a relatively accurate operation counter to compare with the O complexity of each |V| and |E| combination. Once the "Djikstraout.txt" file has been created by the Graphs program, I would import the given data from the txt file to an Excel spreadsheet, wherein I would graph the given data in different ways to analyse the relationships between the different sets of data. A given set of data might look like this once imported into a dataset software.

graph	**PotentialVertexes**	**VertexMapSize**	**Edges**	Operations	O Complexity (|E|log|V|)
0	10	10	26	52	86,37
1	10	10	32	44	106,30

Potential Vertexes is the maximum number of Vertexes that were allowed to be generated for a given graph, VertexMapSize is the actual number of vertexes seen and used by the algorithm, Edges is the number of edges generated for a given set of Vertexes, Operations was the count of operations performed by the algorithm, and O complexity is calculated for each graph based on the number of edges and the VertexMapSize.

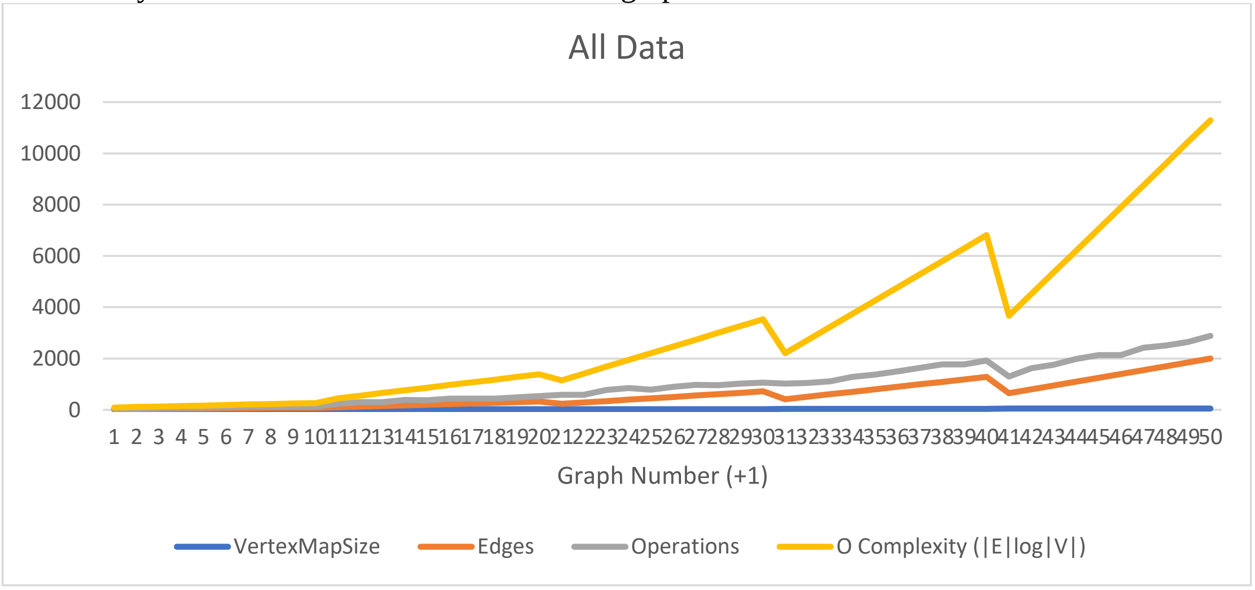
Results:

One of the datasets on which I based my findings can be seen below:

graph	**PotentialVertexes**	**VertexMapSize**	**Edges**	Operations	O Complexity (|E|log|V|)
0	10	10	26	52	86,37
1	10	10	32	44	106,30
2	10	10	38	72	126,23
3	10	10	44	78	146,16
4	10	10	50	86	166,10
5	10	10	56	96	186,03
6	10	10	62	118	205,96
7	10	10	68	130	225,89
8	10	10	74	118	245,82
9	10	10	80	126	265,75
10	20	20	104	232	449,48
11	20	20	128	298	553,21
12	20	20	152	302	656,93
13	20	20	176	383	760,66
14	20	20	200	372	864,39
15	20	20	224	438	968,11
16	20	20	248	442	1071,84
17	20	20	272	439	1175,56
18	20	20	296	483	1279,29
19	20	20	320	541	1383,02
20	30	30	234	586	1148,21
21	30	30	288	580	1413,18
22	30	30	342	768	1678,16
23	30	30	396	854	1943,13
24	30	30	450	787	2208,10
25	30	30	504	904	2473,07
26	30	30	558	972	2738,04
27	30	30	612	966	3003,02
28	30	30	666	1020	3267,99
29	30	30	720	1066	3532,96
30	40	40	416	1021	2213,92
31	40	40	512	1044	2724,83
32	40	40	608	1114	3235,73
33	40	40	704	1282	3746,64
34	40	40	800	1378	4257,54
35	40	40	896	1492	4768,45
36	40	40	992	1638	5279,35
37	40	40	1088	1776	5790,26
38	40	40	1184	1772	6301,16
39	40	40	1280	1924	6812,07

40	50	50	650	1301	3668,51
41	50	50	800	1625	4515,08
42	50	50	950	1760	5361,66
43	50	50	1100	1987	6208,24
44	50	50	1250	2134	7054,82
45	50	50	1400	2135	7901,40
46	50	50	1550	2419	8747,98
47	50	50	1700	2512	9594,56
48	50	50	1850	2644	10441,13
49	50	50	2000	2882	11287,71

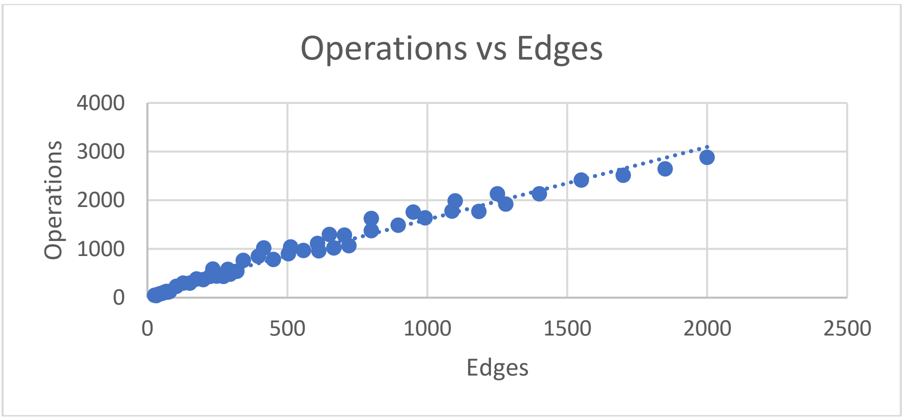
A summary of all of this data can be seen in the graph below:



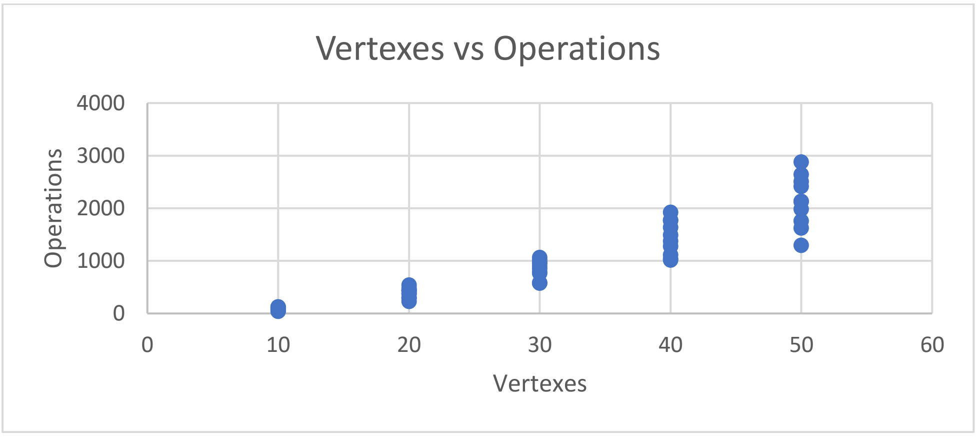
The Graph above does not include the "Potential Vertexes" section as they are equal to the VertexMapSize.

As we can see in the above graph, we can already see some trends emerge, as the O Complexity and Number of operations reduce when the number of edges and vertexes are lower, but are clearly much more correlated to the number of edges, due to it being visible that when the number of edges goes down slightly, there is a massive drop in the O complexity, while an increase in the number of vertexes isn't noticeable until there are more edges.

We can see when comparing edges directly to the O complexity along with adding a line of best fit, we see that:

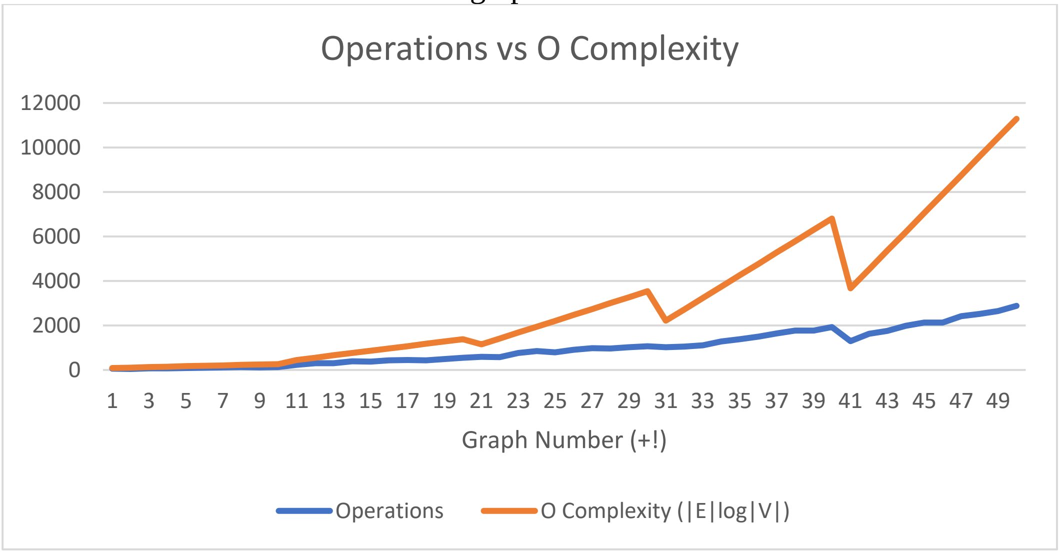


In this graph we can clearly see the correlation between the number of operations completed and the number of edges. We can also see a corrolation between Operations and Vertexes below:



Where a trend also emerges that the number of vertexes clearly affects the number of operations, just not the same degree as the number of edges.

We can also see based on the below graph that:



The O complexity is a maximum for the number of operations, as seen in the data initially presented and in the graphed version here, the number of operations never exceeds that of the O complexity, with the number of operations increasing, but not at the speed of the O complexity.

Discussion:

In essence what I have discovered is that:
The O complexity is the limit for number of operations that will be performed, and at smaller numbers of edges and vertexes it’s very possible to reach that O complexity, but the likelihood of this decreases at |V| and |E| get larger.

I found that as |E| and |V| get larger, the O complexity increases however the |E| is far more impactful on the algorithm. This makes sense as Dijkstra’s algorithm is searching over the edges of a given graph and not the vertexes. The vertexes are just where the algorithm is going to, and thus it makes sense that the number of edges would impact the speed more by increasing the number of operations required. I can also see this to be true as it has the time complexity of O(|E|log|V|) which when plugging numbers in, it can be seen that changes in E dominate the equation far more than changes in V.

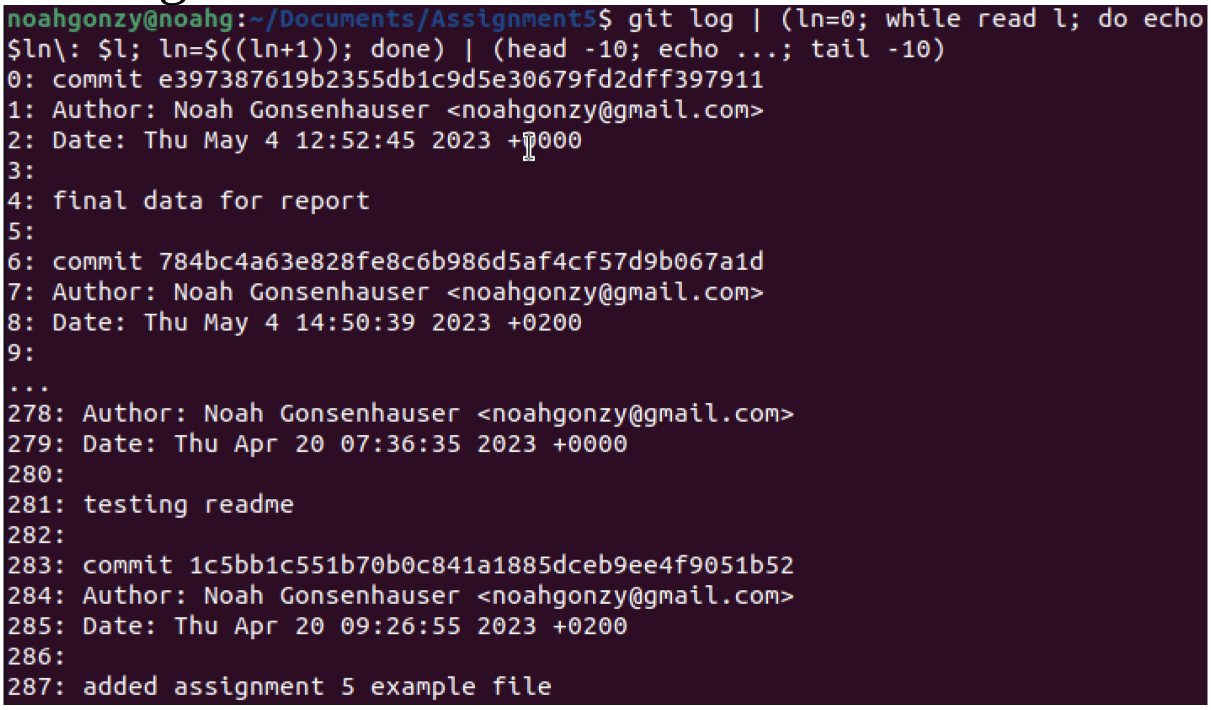
I also discovered a very effective way to count the number of operations taken by the algorithm.

Thus, in conclusion, I proved the theoretical bounds of the algorithm to be true where there won’t be more operations completed by the algorithm than |E|log|V| and I graphically represented the speed of the algorithm at different numbers of edges and vertexes, concluding that the information presented about the algorithm is true, i.e.: edges affect the speed of the algorithm more than the number of vertexes.

Creativity:

For this assignment, I believe I was creative by creating new make file commands for testing my project. I also made use of GitHub as I was switching between 4 different coding environments, my desktop PC, the VM on that machine, my laptop, and the VM on that machine. Presenting an information file to the user as an “infoFile” allows a user to see all the information about a given set of generated graphs in the data folder, and the way in which I programmed my generation program allows for quick integration in other ways, and to pick and choose vertexes to generate edges for.

Git Log:



```
noahgonzy@noahg:~/Documents/Assignment5$ git log | (ln=0; while read l; do echo
$ln\: $l; ln=$((ln+1)); done) | (head -10; echo ...; tail -10)
0: commit e397387619b2355db1c9d5e30679fd2dff397911
1: Author: Noah Gonsenhauser <noahgonzy@gmail.com>
2: Date: Thu May 4 12:52:45 2023 +0000
3:
4: final data for report
5:
6: commit 784bc4a63e828fe8c6b986d5af4cf57d9b067a1d
7: Author: Noah Gonsenhauser <noahgonzy@gmail.com>
8: Date: Thu May 4 14:50:39 2023 +0200
9:
...
278: Author: Noah Gonsenhauser <noahgonzy@gmail.com>
279: Date: Thu Apr 20 07:36:35 2023 +0000
280:
281: testing readme
282:
283: commit 1c5bb1c551b70b0c841a1885dceb9ee4f9051b52
284: Author: Noah Gonsenhauser <noahgonzy@gmail.com>
285: Date: Thu Apr 20 09:26:55 2023 +0200
286:
287: added assignment 5 example file
```