Final Project

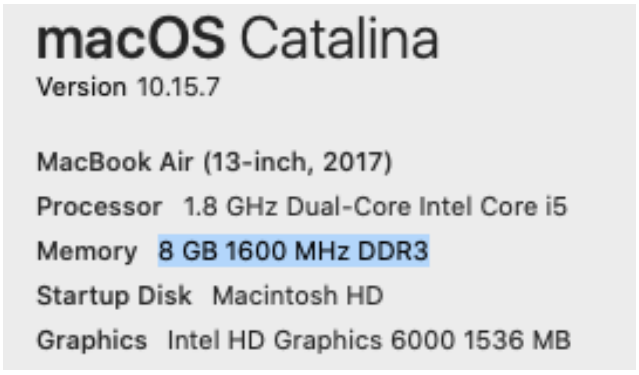
NICOLE SOOD

FALL 2022

Table of Contents:

My Computing Environment:

For this project, I am using a 2017 Intel MacBook which is running off the following specifications:



macOS Catalina
Version 10.15.7

MacBook Air (13-inch, 2017)
Processor 1.8 GHz Dual-Core Intel Core i5
Memory 8 GB 1600 MHz DDR3
Startup Disk Macintosh HD
Graphics Intel HD Graphics 6000 1536 MB

In addition, I completed this project in C++ and used Clion as my IDE. For the execution of my project, I was running in Release mode over Debug mode so that I am not generating debug files every time I executed my program. This sped up my timing slightly.

I am specifically using:

CLion 2020.3.4

Runtime version: 11.0.10+8-b1145.96 x86_64

VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o.

macOS 10.15.7

Memory: 1987M

Cores: 4

I have found that because of the age of my computer, sometimes I can get every extreme value with regards to timing. As a result, I took multiple timings so that they hopefully become as consistent as possible.

Conflict Graph Generation:

For all the following conflict graphs, a temporary initialised graph was created. This created a graph with V number of vertices. Each vertex was assigned an ID, a degree of 0 and colour of 0. This was done so I can update and access that data later.

1. *Generating a complete graph: This type of graph is going to have an edge from every vertex and as a result will have n – 1 neighbour.*

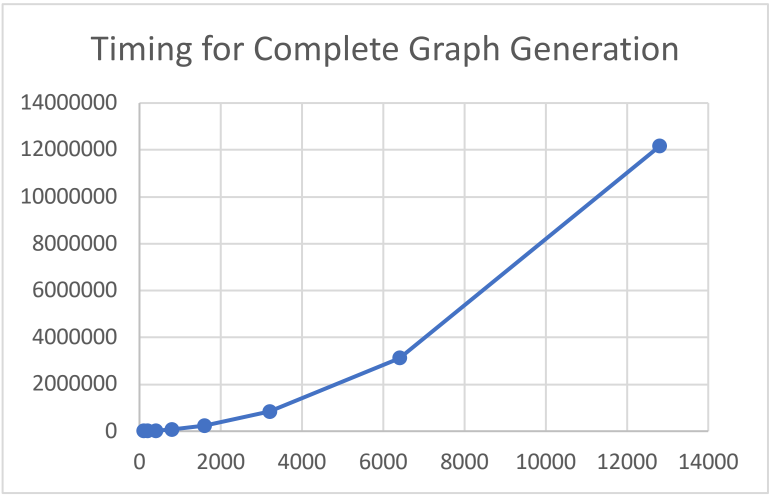
```
void Graph::completeGraph() {
    for(int vert = 0; vert < size; vert++){
        for(int edge = 0; edge < size; edge++){
            if (vert == edge) continue;
            addEdge(vert, edge);
        }
    }
}
```

To test the runtime of this, I generated various sized graphs, doubling the number of vertices each time. Below a table with the runtime values in microseconds that I was receiving.

Timing for Complete Graph Generation

V	T1	T2	T3	Average
100	768	741	789	766
200	3505	2925	2977	3135.66667
400	17624	13956	12550	14710
800	74269	58330	57439	63346
1600	204541	266476	227531	232849.333
3200	883850	777171	847741	836254
6400	3229502	2977229	3149927	3118886
12800	12408908	12015668	12038146	12154240.7

After analysing the timing for this graph, I have determined that a complete graph will generate in $theta(n^2)$ because as the input size of the graph doubled, the time got roughly four times larger.



2. *Generating a Cycle Graph: For this graph, each vertex had two edges one pointed to the vertex numbered n+1 in front and another numbered n-1 behind.*

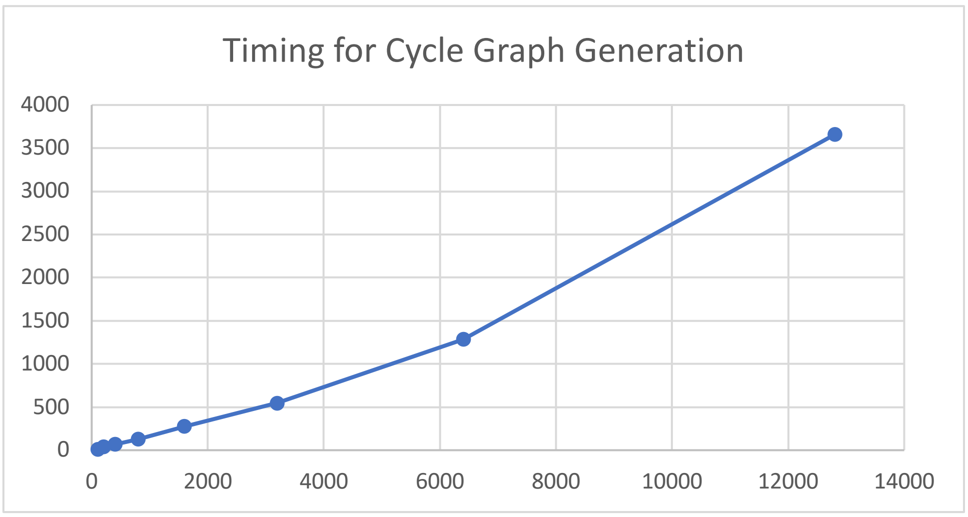
```
void Graph::cycleGraph() {
    for(int x = 0; x < size-1; x++) {
        const int temp = x;
        addEdge(temp, x + 1);
        addEdge(temp + 1, x);
    }
    addEdge(size - 1, 0);
    addEdge(0, size - 1);
}
```

Like the complete graph, I tested this function by using various vertex sizes and doubled the number of vertices in the graph. Below a table with the runtime values in microseconds that I was receiving.

Timing for Cycle Graph Generation

V	T1	T2	T3	Average
100	13	13	13	13
200	37	39	41	39
400	62	85	60	69
800	124	127	132	127.666667
1600	276	280	273	276.333333
3200	560	532	545	545.666667
6400	1251	1271	1331	1284.33333
12800	3745	3789	3436	3656.66667

After analysing the timing for this graph, I have determined that a cycle graph will generate in $O(\text{n})$ because as the input size of the graph doubled, the time got roughly twice as large.



3. *Generating a graph with an even distribution: For this graph, I used a random function to generate an evenly distributed graph. The idea behind this is that every vertex is equally likely to be chosen for a conflict.*

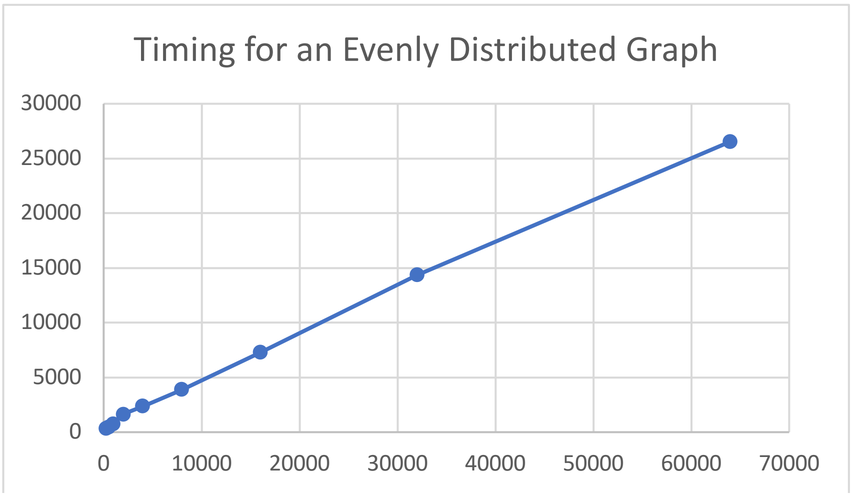
```
void Graph::evenGraph(const int E) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distr(0, size-1);

    int maxEdges = (size * (size -1 )) /2;
    if (E > maxEdges){
        cout <<  "Too many edges";
        exit(0);
    }

    for(int i = 0; i < E; i++){
        int v1 = distr(gen);
        int v2 = distr(gen);
        if (v1 == v2 || vertices[v1].edges[v2] || v1, v2 == size){
            i--;
            continue;
        }
        addEdge(v1, v2);
        addEdge(v2, v1);
    }
}
```

For this part of the analysis, I pre-generated a graph with 10,000 vertices. I test this algorithm would by increasing the number of edges. I increased the number of edges by two for each test. Below a table with the runtime values in microseconds that I was receiving.

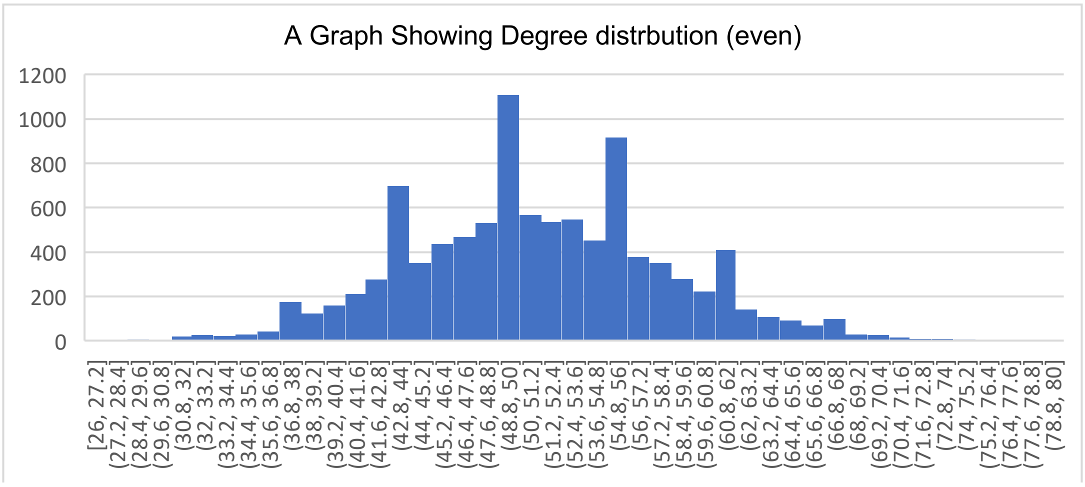
Timing for Evenly Distributed Graph Generation

E	T1	T2	T3	Average
250	316	356	294	322
500	465	444	497	468.666667
1000	789	791	723	767.666667
2000	1855	1460	1598	1637.66667
4000	2139	2497	2505	2380.33333
8000	4007	3509	4148	3888
16000	7295	7300	7201	7265.33333
32000	12923	16661	13504	14362.6667
64000	25369	28456	25809	26544.6667

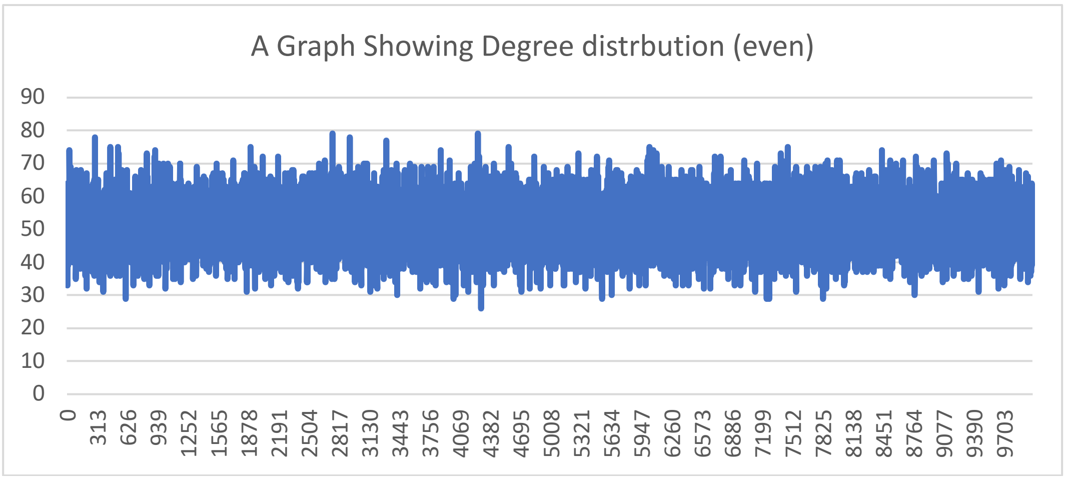
After analysing the timing for this graph, I have determined that an evenly distributed graph will be in $\boldsymbol{O}(\mathbf{n})$ because as the input size of the graph doubled, the time got roughly twice as large. However, the worst case could be infinity if the two randomly selected numbers are always the same.



Diving deeper into how the edges were distributed across the vertices, I looked at the induvial degrees for each vertex. Below shows both a histogram showing the distribution as well as a line graph illustrating the conflicts. Both graphs are what I expected for an even distribution



Histogram showing the dispersion of conflicts across an evenly distributed graph.



A line graph showing the number of conflicts for each vertex.

4. *Generating a graph with a tiered distribution: For this graph, the first 50% of the edges will be randomly chosen from first 10% of the vertices. The remaining 50% of the graph will be randomly chosen from the last 90% of the vertices.*

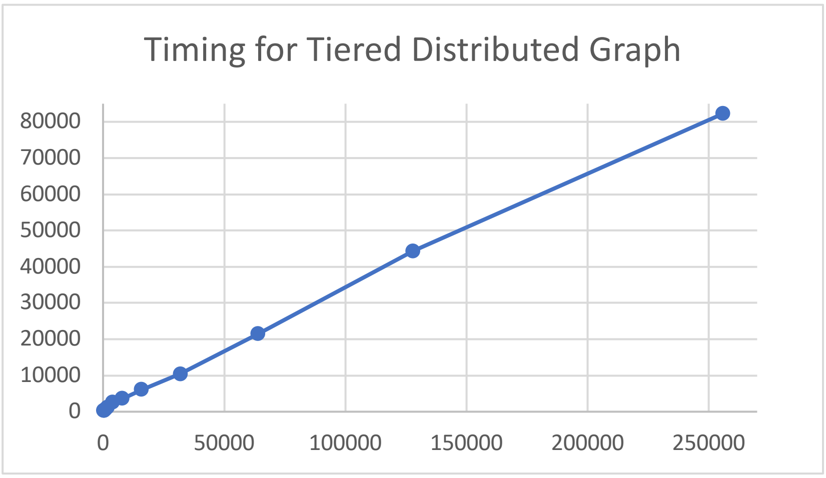
```
void Graph::tieredGraph(const int E) {

    int maxEdges = (size * (size -1 )) /2;

    if (E > maxEdges){
        cout << "Too many edges";
        exit(0);
    }

    //Evenly distributed across the first 10% of vertices with 1/2 of
the choices
    int tempSize = size / 10; //to get roughly 10% of the data

    std::random_device rd; // obtain a random number from hardware
    std::mt19937 gen(rd()); // seed the generator
    std::uniform_int_distribution<> distr(0, tempSize-1);

    for(int i = 0; i < E/2; i++){ //for the first 1/2
        int v1 = distr(gen);
        int v2 = distr(gen);

        if (v1 == v2 || vertices[v1].edges[v2]){
            i--;
            continue;
        }
        addEdge(v1, v2);
        addEdge(v2, v1);
    }

    std::uniform_int_distribution<> distr2(tempSize, size-1);
    for(int j= 0; j < E/2; j++){ //for the 2nd 1/2
        const int vt1 = distr2(gen); // to get a range between the
first 10% and last 100%
        const int vt2 = distr2(gen);

        if (vt1 == vt2 || vertices[vt1].edges[vt2]){
            j--;
            continue;
        }
        addEdge(vt1, vt2);
        addEdge(vt2, vt1);
    }
}
```

For this part of the analysis, I generated a graph with 10,000 vertices. From there, I increased the number of edges I wanted on my conflict graph by two.

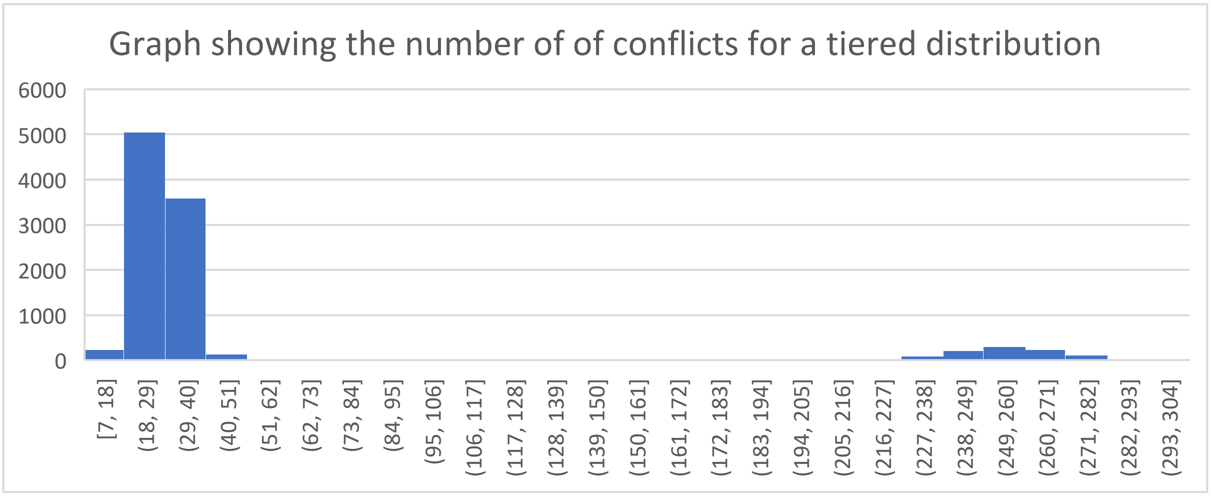
Timing for Tiered Distributed Graph

n	T1	T2	T3	Average
250	238	209	237	228
500	375	382	348	368.333333
1000	652	673	774	699.666667
2000	1118	1088	1212	1139.33333
4000	3134	2440	2246	2606.66667
8000	3562	3414	3750	3575.33333
16000	5851	5927	6481	6086.33333
32000	10409	10942	9877	10409.3333
64000	22502	22322	19648	21490.6667
128000	40088	42742	50288	44372.6667
256000	81246	77427	88158	82277

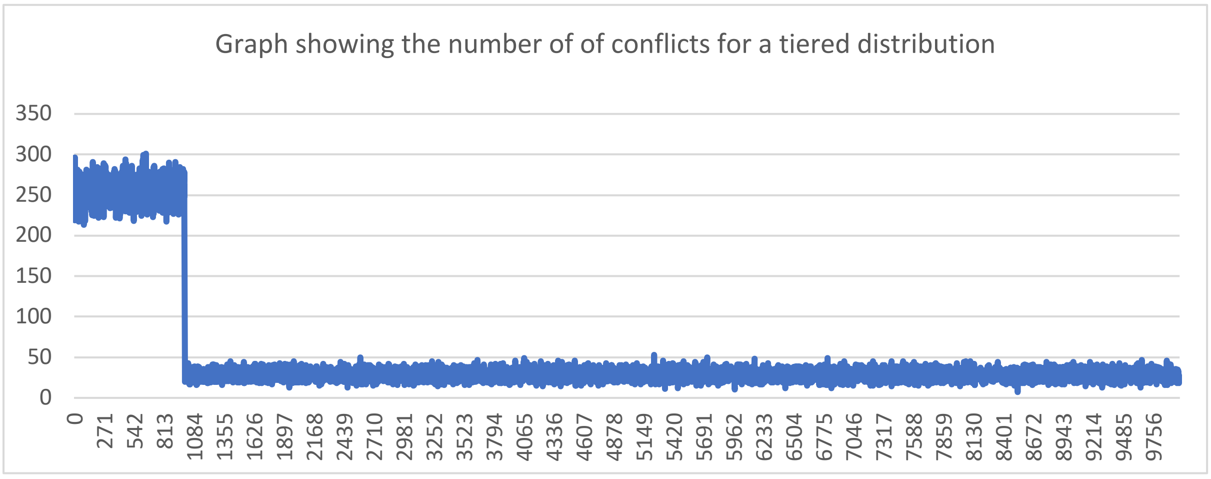
After analysing the timing for this graph, I have determined that a skewed distributed graph will also be in $\boldsymbol{O}(\mathbf{n})$ because as the input size of the graph doubled, the time got roughly twice as large. Like the even graph, the worst case could be infinity if the two randomly selected numbers are always the same.



Diving deeper into how the edges were distributed across the vertices, I looked at the induvial degrees for each vertex. Below shows both a histogram showing the distribution as well as a line graph illustrating the conflicts. Both graphs are what I expected for their type of tiered distribution. For the histogram, I expected the first 10% of the vertices to be the most densely populated which is what is illustrated below.



Histogram showing the dispersion of conflicts across a tired distributed graph



A line graph showing the number of conflicts for each vertex.

5. *Generating a graph with a distribution of my own choosing: For this, I decided to do a distribution like the one above, however 1/3rd of the graph will be generated using only the first 1/4th of available vertices. The last 2/3rd will be generated with the remaining vertices. The idea behind this is to mimic a student schedule where theoretically 25% of students would access roughly 1/3rd of the same classes (for example if they are Lyle students) and the remaining classes would represent classes all students would need to takes (for example: an English class)*

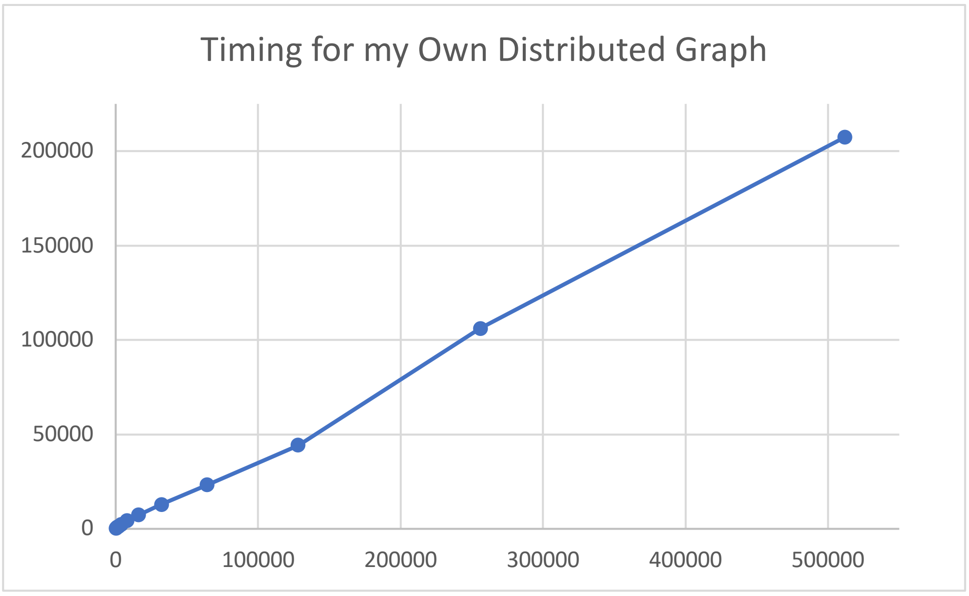
```
void Graph::myOwn(const int E) {

    //1/3 of edges to be in the first quarter of vertices
    //2/3 of edges to be in the three quarters of vertices
    std::random_device rd; // obtain a random number from hardware
    std::mt19937 gen(rd()); // seed the generator
    std::uniform_int_distribution<> distr(0, (size/4)-1);

    int maxEdges = (size * (size -1 )) /2;

    if (E > maxEdges){
        cout << "Too many edges";
        exit(0);
    }

    for(int i = 0; i < E/3; i++){
        int v1 = distr(gen);
        int v2 = distr(gen);

        if (v1 == v2 || vertices[v1].edges[v2]){
            i--;
            continue;
        }
        addEdge(v1, v2);
        addEdge(v2, v1);
    }

    std::uniform_int_distribution<> distr2(size/4, size-1);
    for(int i = 0; i < (2*E/3); i++){
        int v1 = distr2(gen);
        int v2 = distr2(gen);

        if (v1 == v2 || vertices[v1].edges[v2]){
            i--;
            continue;
        }
        addEdge(v1, v2);
        addEdge(v2, v1);
    }
}
```

For this part of the analysis, I once again generated a graph with 10,000 vertices. From there, I increased the number of edges I wanted on my conflict graph by two.

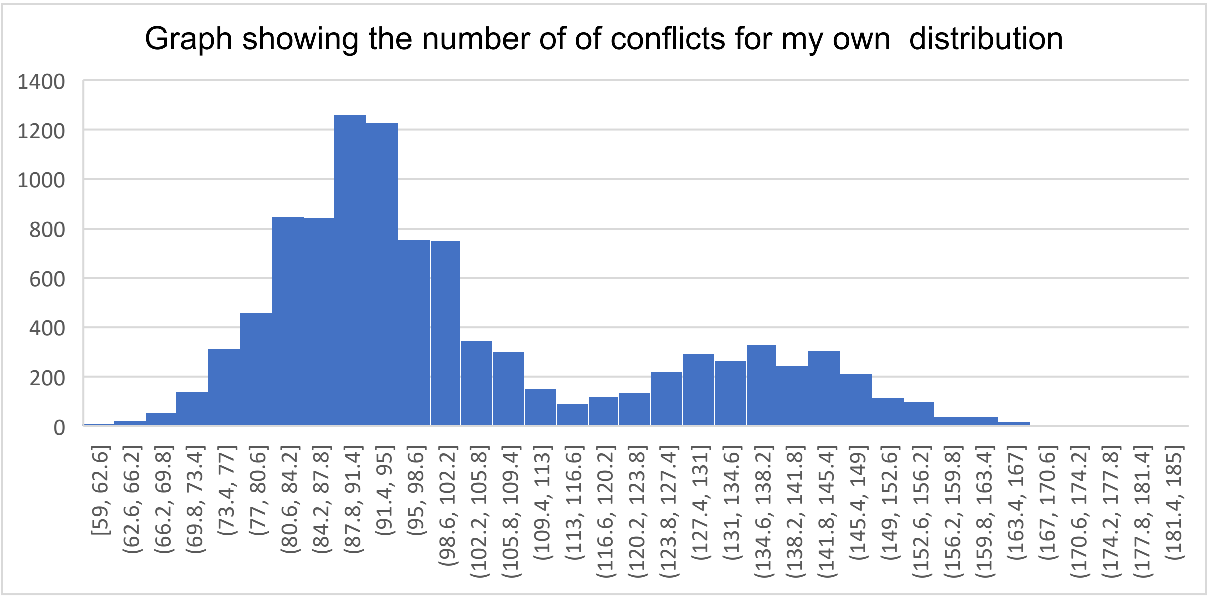
Timing for my Own Distributed Graph

n	T1	T2	T3	Average
250	283	229	301	271
500	477	468	460	468.333333
1000	746	918	782	815.333333
2000	1390	1372	1198	1320
4000	2489	2192	2207	2296
8000	4806	4144	4418	4456
16000	7278	7090	7650	7339.33333
32000	12709	13037	13076	12940.6667
64000	24746	21644	23037	23142.3333
128000	44355	47381	41106	44280.6667
256000	109733	108642	100086	106153.667
512000	193528	213897	214837	207420.667

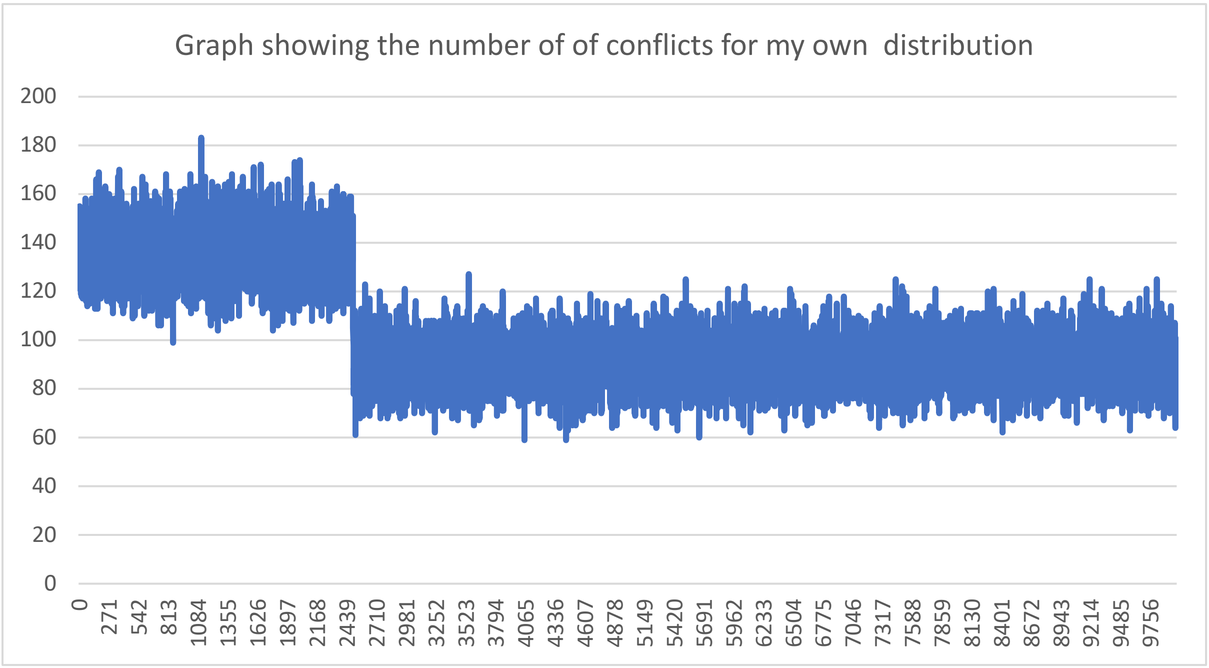
After analysing the timing for this graph, I have determined that this distributed graph will also be in $\boldsymbol{O}(\mathbf{n})$ because as the input size of edges graph doubled, the time got roughly twice as large. Like the even and the skewed graph, the worst case could be infinity if the randomly selected numbers are always the same.



Diving deeper into how the edges were distributed across the vertices, I looked at the induvial degrees for each vertex. Below shows both a histogram showing the distribution as well as a line graph illustrating the conflicts.



Histogram showing the dispersion of conflicts across my own designed distribution



Line Graph showing the dispersion of conflicts across my own designed distribution

Vertex Ordering:

In this section I will be discussing my vertex ordering methods. As a student on the accelerated pathways program, I had to complete six different vertex orderings which includes:

- Random Vertex Ordering (required),
- Smallest Last Vertex Ordering (required),
- Smallest Original Degree (required),

For the following two orderings: I did the reverse ordering of a smallest last vertex and an smallest degree last ordering because I wanted to see if it would have an impact on the colouring aspect of this project.

- Largest Last Vertex Ordering,
- Largest Original Degree,

My last ordering implementation was:

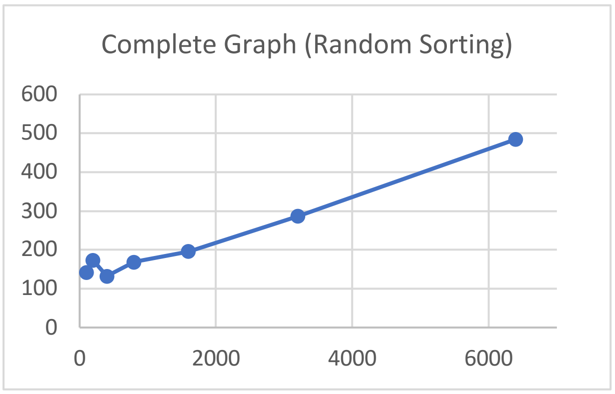
- Depth First Search

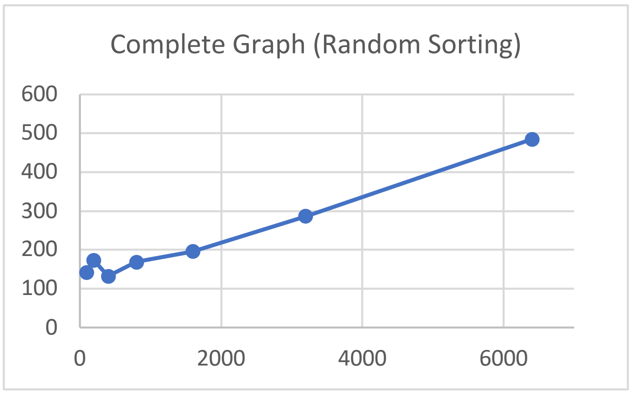
1. *Random Vertex Ordering: For this ordering method, we had to order the vertices randomly. As a result, I simply put the whole graph into my ordering vector and shuffled it so that it would be in a random order.*

```
void Graph::randomOrdering() {

    std::random_device rd; // obtain a random number from hardware
    std::mt19937 gen(rd()); // seed the generator
    std::uniform_int_distribution<> distr(0, size);

    for (vertex& v : vertices){
        ordering.push_back(&v);
    }

    shuffle (ordering.begin(), ordering.end(),
std::default_random_engine(distr(gen)));

}
```

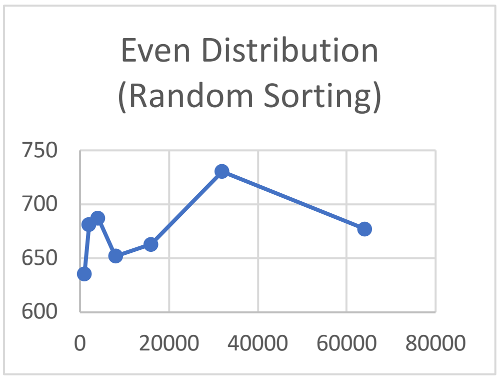
For this ordering, I expected it to be in $\boldsymbol{O}(\mathbf{n})$ because of the nature of the for loop. Once all the nodes were added to the ordering vector, I shuffled them randomly so that they would be put into the colouring algorithm in a random order.

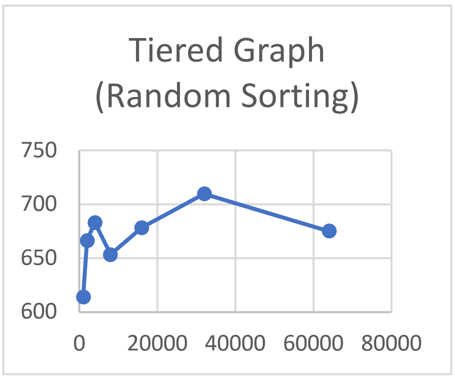
My runtime tables supported my expectations for this algorithm, as the number of vertices doubled, the runtime of the ordering doubled. This is what I expected because the ordering only considered the vertices and did not depend on any of the edges.

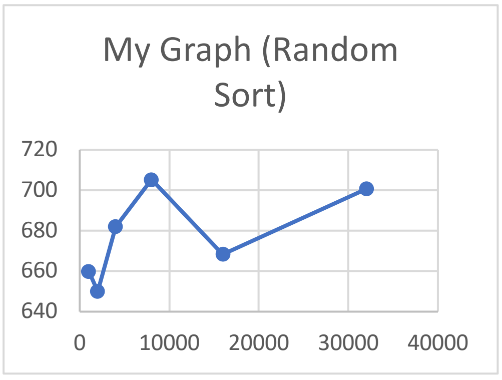




The number of edges did not impact the runtime, and when testing the distributed graphs, the number of vertices stayed the same at 10,000. As a result, I expected these graphs to sort in a constant time. This is supported in the following graph where you see as the number of edges doubled, the runtime mostly constant with some variations which I believe was an effect of my machine.







Overall, my random sort algorithm is in $\boldsymbol{O}(\mathbf{n})$.

2. *Smallest Last Vertex: This ordering works by first initialising a vector/list of all the degrees contained by each vertex. From there, you then proceed to delete the vertex with the smallest degree. This process continues until all the vertices are deleted.*

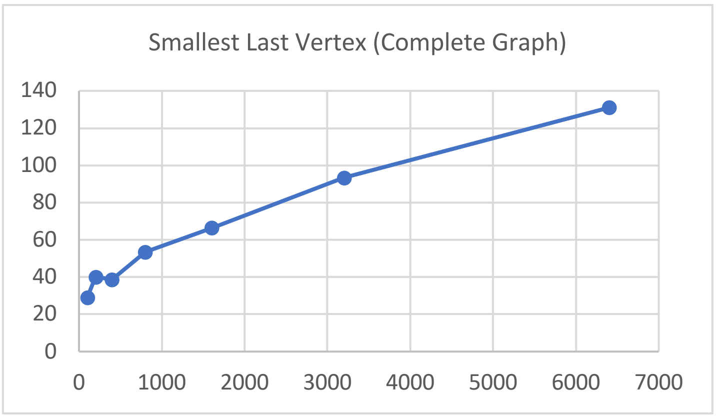
In the following implementation, we had a structure to follow in our project hand out:

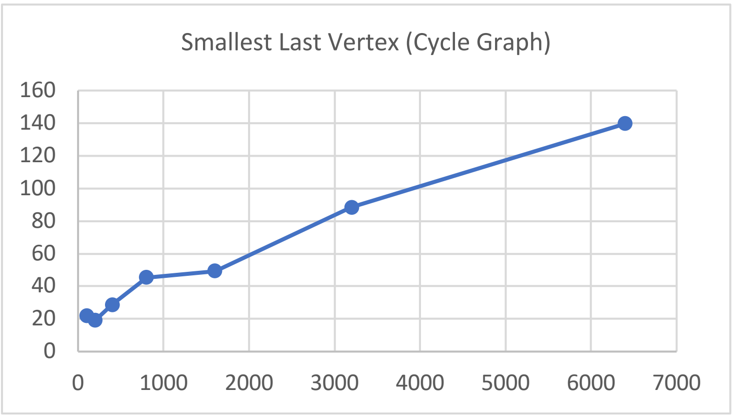
- Field 1: Vertex ID – When my graphs are generated, they have a vertex ID as well as a pointer to an edge list.
- Field 2: Pointer to an edge list – Similar to before, when my graphs are generated, there is a pointer to an edge list. In addition, we will later access this edge list to delete edges as needed.
- Field 3: Current Degree, -1 when deleted and colour value – For these fields, current degree is stored in a vector called "newDegreeSize" where for each vector, the degree value will get updated as edges are deleted. When all the edges for a vertex are successfully deleted, the "deleted" vector will be set to -1.
- Field 4: Pointer to a list of current degrees and a pointer to the deleted list – which is contained within this function. There is a pointer called "tempHolder" which points to a list of current degrees, which is then used to update the deleted vector.

```
void Graph::smallestLastVertix() {
//NOTE: SOURCE CODE HAS BEEN STRUCTURED AND ADAPTED FROM THE FOLLOWING
GITHUB PROJECT(S) AND SOURCES:
//https://github.com/MarkBrub/CS5350-
Graph_Coloring/blob/master/Coloring/Graph.cpp
//https://github.com/dhanwin247/Parallel-graph-coloring
//CHANGES HAVE BEEN MADE TO FIT MY UNDERSTANDING OF THE SMALLEST LAST
VERTEX ORDERING

    vector<vertex*> degrees(size);
    vector<int> newDegreeSize(size);
    vector<int> deleted(size);

    cout << "Original ordering: " << endl;
    for(int i = 0; i < vertices.size(); i++){
        cout << vertices[i].id << " " << vertices[i].degree << " " <<
vertices[i].colour << endl;
    }

    for (int i = 0; i < vertices.size(); i++){
        if(degrees[vertices[i].degree] != nullptr){
            degrees[vertices[i].degree]-> previous = &vertices[i];
            vertices[i].next = degrees[vertices[i].degree];
        }

        degrees[vertices[i].degree] = &vertices[i];
        newDegreeSize[i] = vertices[i].degree;
    }



    for(int i = 0; i < degrees.size(); i++){
        if(!degrees[i]){
            continue;
        }

        vertex* tempHolder = degrees[i];
        ordering.push_back(tempHolder);
        degrees[i] = tempHolder -> next;

        if(tempHolder-> next != nullptr){
            tempHolder->next->previous = nullptr;
        }

        tempHolder-> next = nullptr;
        deleted[tempHolder->id] = -1;

        Edge* currentEdge = edges[tempHolder->id];

        while (currentEdge != nullptr){
            int vert = currentEdge->dest;

            if(deleted[vert] == -1){
                currentEdge = currentEdge -> next;
                continue;
            }

            if (vertices[vert].next != nullptr){
                vertices[vert].next-> previous = vertices[vert].previous;
            }

            if(vertices[vert].previous != nullptr){
                vertices[vert].previous-> next = vertices[vert].next;
            }

            else {
                degrees[newDegreeSize[vert]] = vertices[vert].next;
            }

            int updateDegree = newDegreeSize[vert];
            newDegreeSize[vert] = updateDegree - 1;

            vertices[vert].previous = nullptr;
            vertices[vert].next = degrees[newDegreeSize[vert]];

            if(degrees[newDegreeSize[vert]] != nullptr){
                degrees[newDegreeSize[vert]] = &vertices[vert];
             }

            currentEdge = currentEdge->next;
        }

        if(i == 0) {
            i--;
            continue;
        }

        i -= 2;
    }

    std::reverse(ordering.begin(), ordering.end());

    cout << endl << "Smallest Last Vertex Ordering: " << endl;
    for (int i = 0; i < ordering.size(); i++){
        cout <<  ordering[i] ->id << " " << ordering[i]->degree << " " <<
ordering[i] ->colour << endl;
    }
}
```

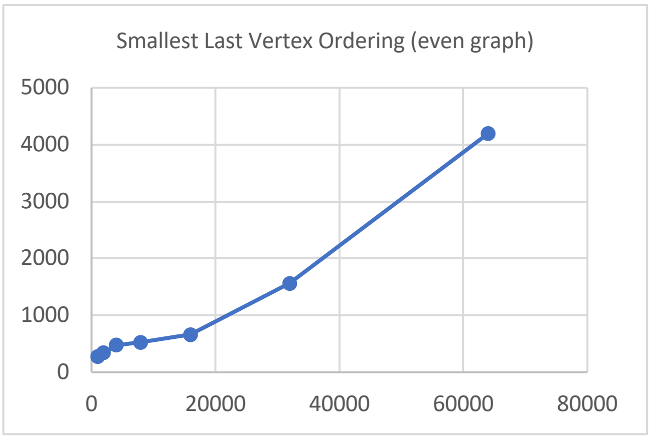
For this implementation, I was expecting it to run in **O(E + V)** because we need to both traverse the vertices and then the edge list, so that we can delete as needed.. After analysing the runtime, the timing for a complete and cycle graph we see that as the number of vertices double, the runtime also doubles which means V is running in linear time. For my distribution graphs, I expected this algorithm also run in linear time seeing as each graph will have 10,000 vertices, however the number of edges is doubling.

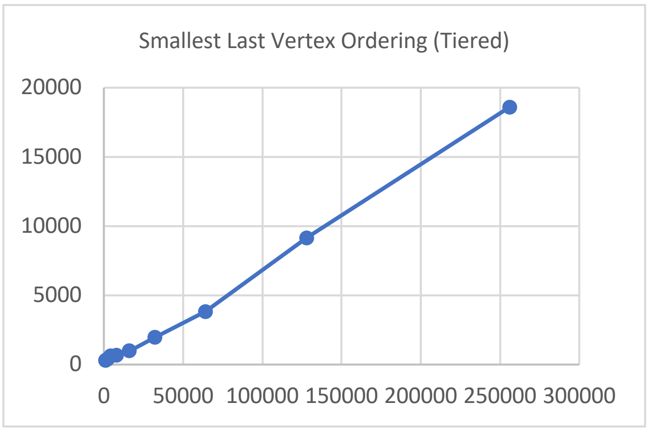
As we can see in the following graphs: the complete and cycle support the conclusion that this algorithm is running in **O(E + V).** This shows that the runtime of *adding more vertices is linear*.

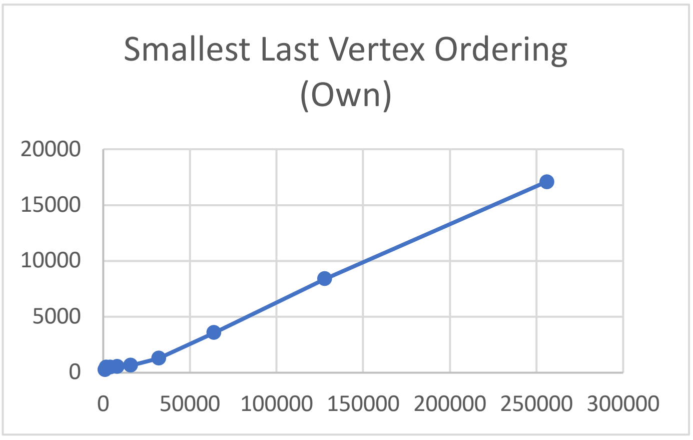




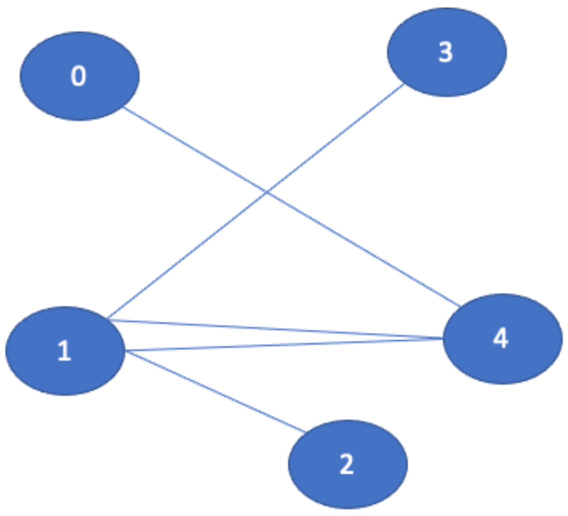
And when tracking the effect of increasing the number of edges on a graph, we can also see that the algorithm is in **O(E + V)** because as E doubles in size, the time will begin to approach a linear function.







To further support that this ordering works as expected, I tracked the deletion and colouring of a small graph. I created a 5 vertices graph with 5 edges which originally looks like the following:



From here we can see that:

Vertex 0 – >Degree 1

Vertex 1 -> Degree 4

Vertex 2 -> Degree 1

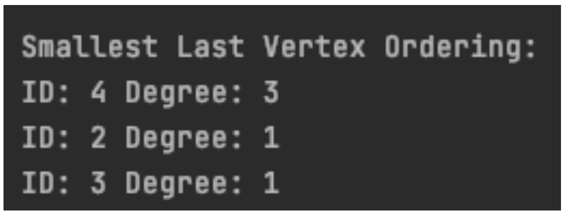
Vertex 3 -> Degree 1

Vertex 4 -> Degree 3

In this, the last nodes to get deleted will be: 4, 2, 3

There is a Terminal Clique of this graph is 3, which co-insides with the number of colours needed to colour this graph.

Program Output:



```
Smallest Last Vertex Ordering:
ID: 4 Degree: 3
ID: 2 Degree: 1
ID: 3 Degree: 1
```

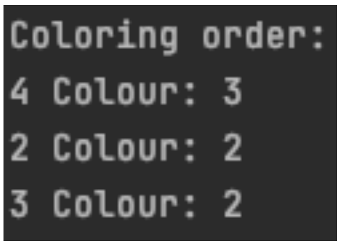
It will take 3 Colours to colour this graph:

Colour 1: 0, 1 OR 2,3

Colour 2: 2, 3 OR 0, 1

Colour 3: 4

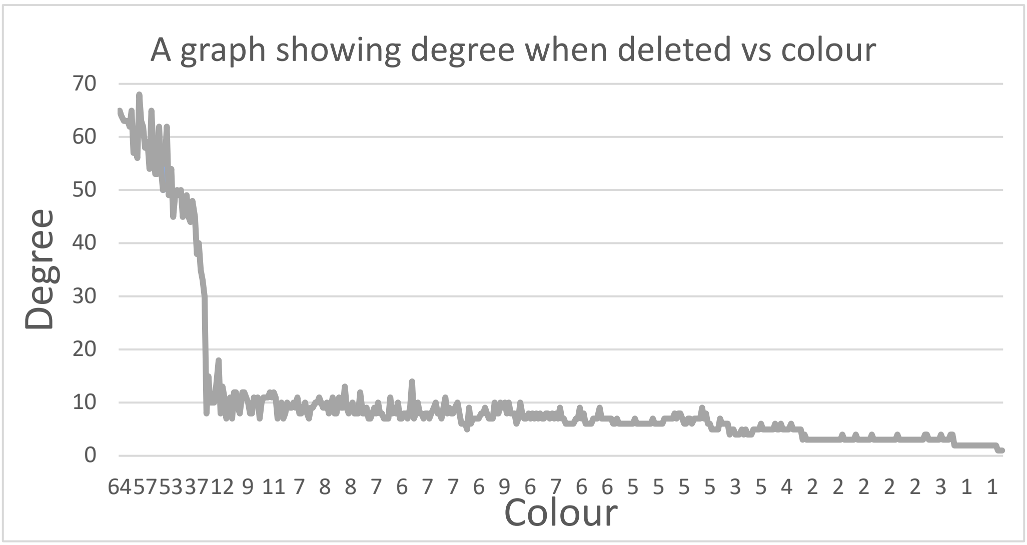
Program Output (Colouring Assignments):



```
Coloring order:
4 Colour: 3
2 Colour: 2
3 Colour: 2
```

There's outcome supports that vertices 2 & 3 will have the same colour and all remaining nodes will have a colour of 2. As well as the fact that the program is functioning as I expected it to.

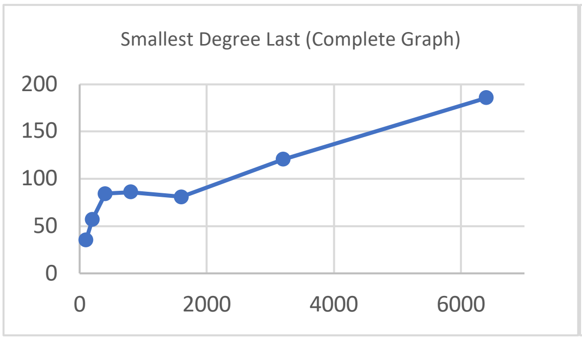
In addition to this, I also looked at the relationship between the degree and the colours. I was expecting it to follow a downwards trend, with the larger degree having a larger colour.

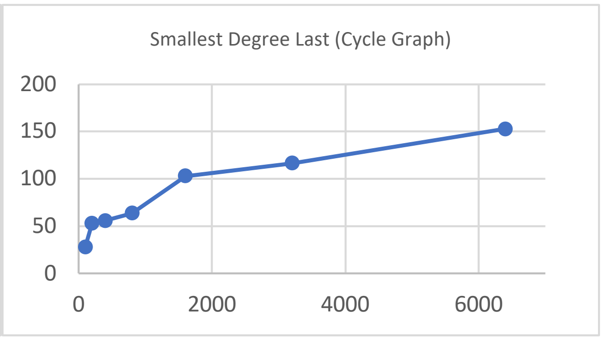


3. *Smallest Original Degree Last: In this ordering, we order our graph by degree with the smallest original degree last.*

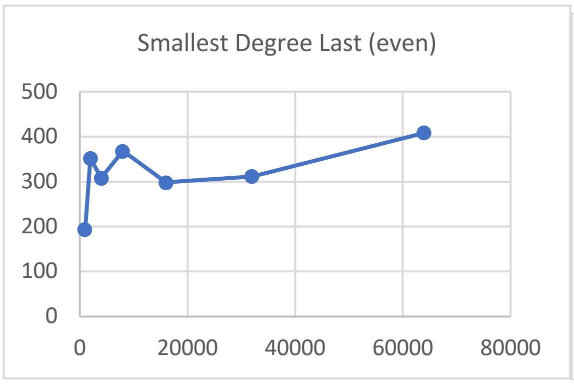
```
void Graph::smallestDegreeFirst() {
    vector<vertex*> degrees(size);

    for (int i = 0; i < vertices.size(); i++){
        if(degrees[vertices[i].degree] != nullptr){
            degrees[vertices[i].degree]-> previous = &vertices[i];
            vertices[i].next = degrees[vertices[i].degree];
        }
        degrees[vertices[i].degree] = &vertices[i];
    }

    for (int i = 0; i < degrees.size(); i++) {
        vertex *current = degrees[i];
        while (current != nullptr) {
            ordering.push_back(current);
            current = current->next;
        }
    }

}
```

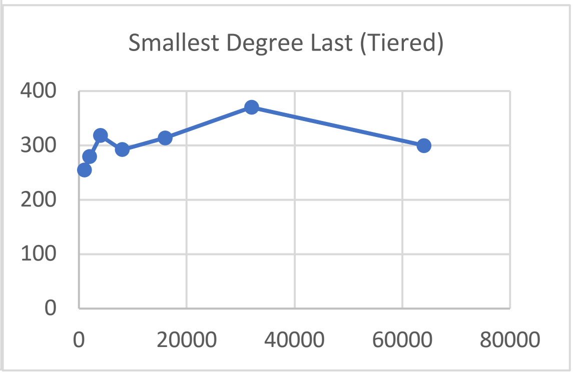
For this ordering, we are told that it should run **Theta (E + V).** Looking at the effect of doubling the number of vertices, we can see that the timing gets roughly **(sqrt(V))** bigger:

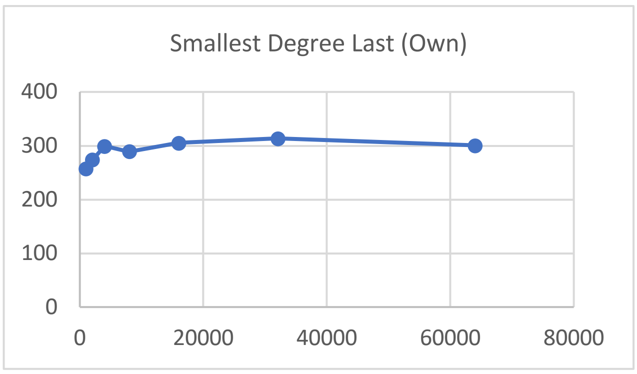




With a complete and a cycle graph we can see that as the number of vertices double, the time is getting roughly 1.3 times larger – this relationship can be seen extremely clearly in the largest degree first ordering. In addition to this, looking at what happened when the number of edges double, the time stays almost constant - which is expected because we are ordering based purely on the degree of a vertices.





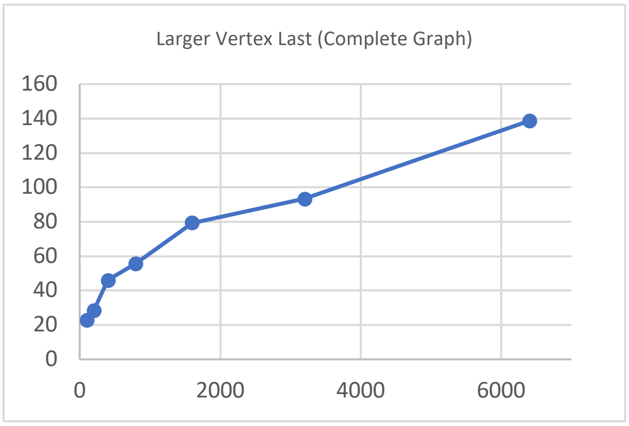


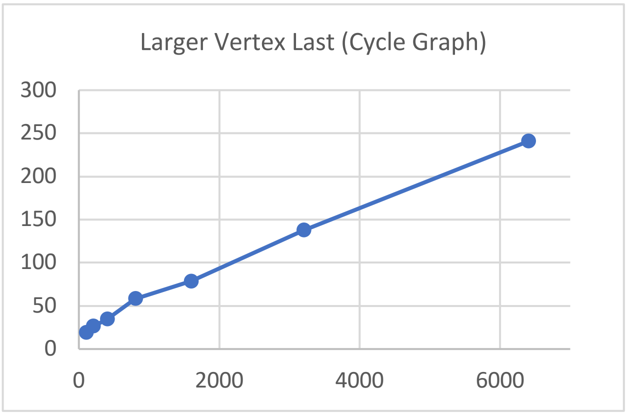
As an effect the overall runtime of this algorithm is **O(E + V).**

4. *Largest Vertex Last: This ordering is in reverse of a smallest last vertex ordering. As an effect, this ordering method will return the reverse of a smallest last vertex ordering.*

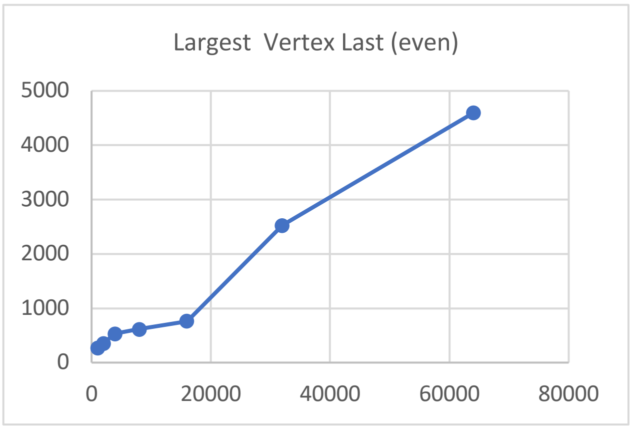
```
void Graph::largestVertexLast()  {
    smallestLastVertix();
    std::reverse(ordering.begin(), ordering.end());
}
```

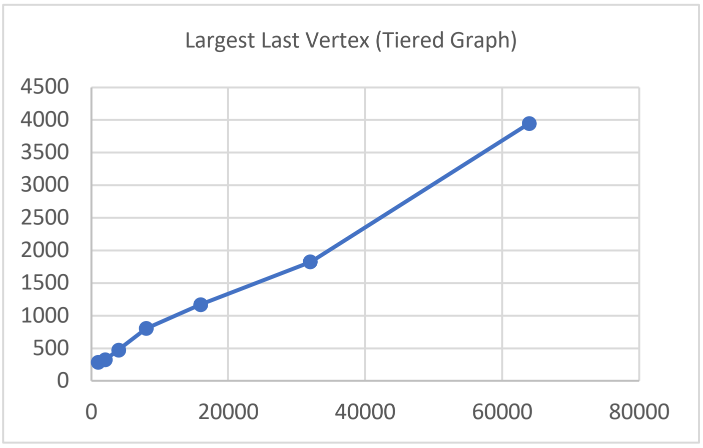
Due to the nature of how I programmed this, I expected it run in the same space as the smallest last vertex ordering. Once again, I looked at the runtimes it took to sort a graph with differing numbers of vertices. In my case, that was my cycle and complete graphs. We can see that as the number of vertices doubled, the runtime also doubled. Putting V into a linear time.

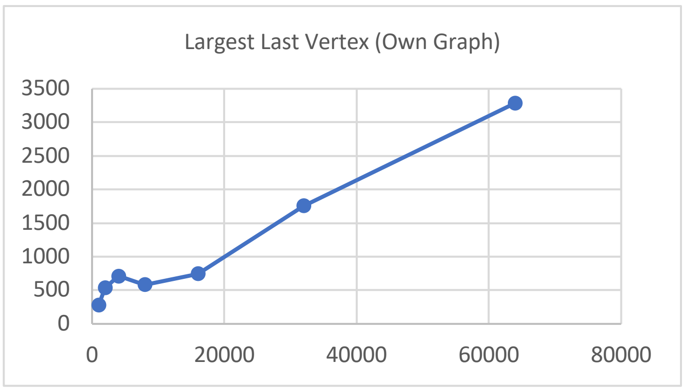




Like the smallest last vertex graph, we can also see that as the number of edges double on a graph, the time to order the graph also doubles.





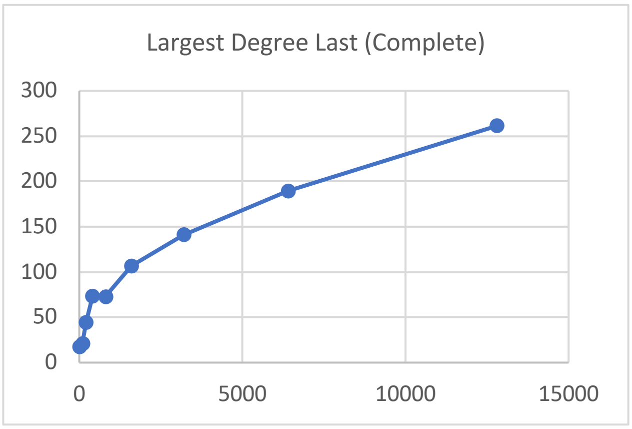


I think this algorithm is still in **O (E + V).**

5. *Largest Degree First: This ordering is the reverse of the Smallest Degree First ordering. As a result, to implement this ordering I called the smallestDegreeFirst function and reversed the ordering vector so that the largest degree would be coloured first.*

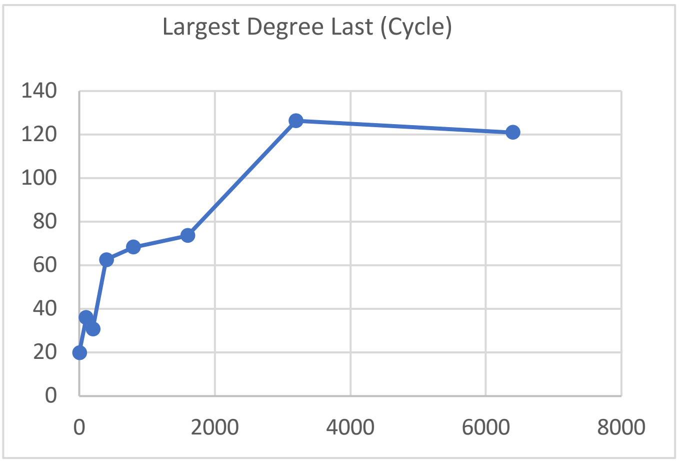
```
void Graph::largestDegreeFirst() {
    smallestDegreeFirst();
    std::reverse(ordering.begin(), ordering.end());
}
```

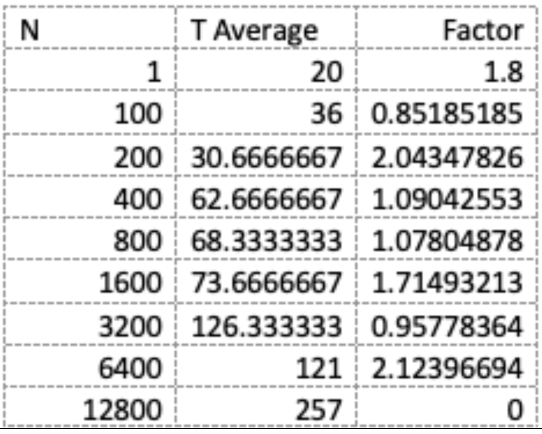
For this ordering, it was very clear that as the number of vertices increase is running in a **O(sqrt(n))** because as n doubles, time is roughly increasing by a factor of 1.3. The graph for a complete graph shows this relationship extremely clearly for this part of the algorithm.



n	time	Factor
1	17.33333	1.192307697
100	20.66667	2.145161285
200	44.33333	1.646616543
400	73	0.99543379
800	72.66667	1.467889912
1600	106.6667	1.321874996
3200	141	1.345153667
6400	189.6667	1.377855883
12800	261.3333	0

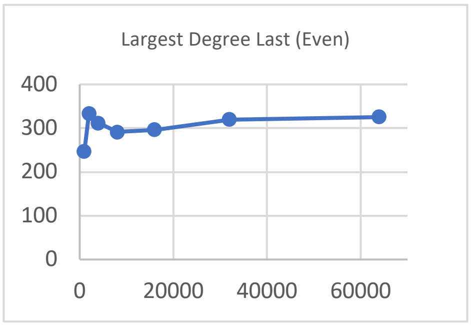
A similar relationship can be seen in the runtime table for the cycle graphs.

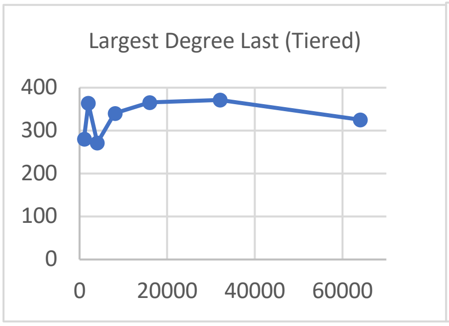


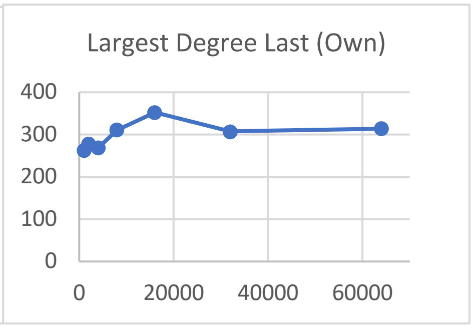


N	T Average	Factor
1	20	1.8
100	36	0.85185185
200	30.6666667	2.04347826
400	62.6666667	1.09042553
800	68.3333333	1.07804878
1600	73.6666667	1.71493213
3200	126.333333	0.95778364
6400	121	2.12396694
12800	257	0

Once again, the number of edges will roughly have a constant relation as they double.





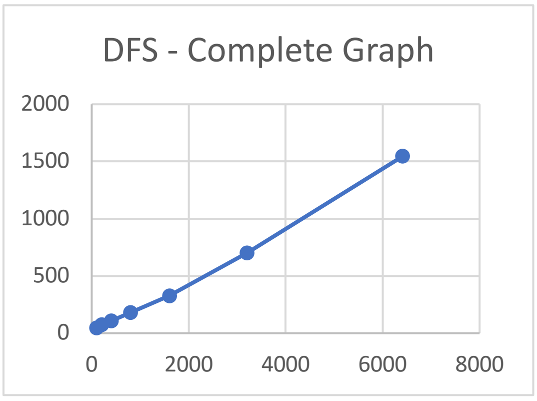


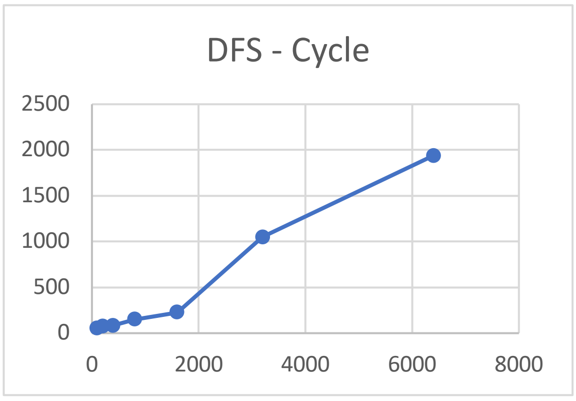
Overall, this ordering is still in **O(E + V)**, as expected**.**

6. *My own graph ordering: For this ordering, I implemented depth-first search. This consists of two functions, the ordering function itself and the helper function*
   - The myOwnOrdering() function initialises a vector of vertices that still need to be checked if they have been found. I also initialise a vector of Booleans. From there I call the helper function.
   - The helper function will complete a depth first traversal of the graph and will mark if the vertices have been visited during the traversal.

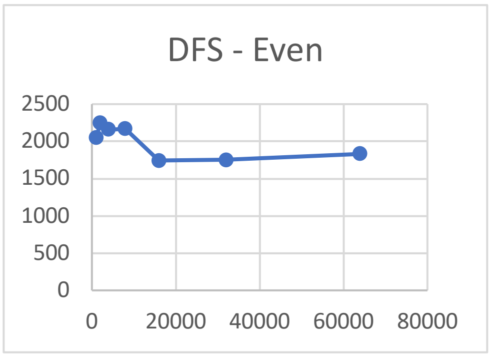
```
   void Graph::myOwnOrderingHelper(vertex *v, vector<bool> found) {
    //THIS CODE IS MADE UP OF THE FOLLOWING SOURCES:
    //
https://cppsecrets.com/users/4530104114105116104105107109971081051075654641
03109971051084699111109/C00-Program-implementing-Topological-Sort.php
    //https://www.geeksforgeeks.org/cpp-program-for-topological-sorting/
    if(found[v->id]){
        return;
    }

    found[v->id] = true;

    ordering.push_back(v);

    while(v-> next != nullptr){
        myOwnOrderingHelper(v->next, found);
    }
}

void Graph::myOwnOrdering() {
    //THIS CODE IS MADE UP OF THE FOLLOWING SOURCES:
    //
https://cppsecrets.com/users/4530104114105116104105107109971081051075654641
03109971051084699111109/C00-Program-implementing-Topological-Sort.php
    //https://www.geeksforgeeks.org/cpp-program-for-topological-sorting/

    std::vector<vertex*> toBeAdded;
    std::vector<bool> found(size,false);

    for(int i = 0; i < vertices.size(); i++){
        toBeAdded.push_back(&vertices[i]);
    }

    for(int i = 0; i < size; i++){
        myOwnOrderingHelper(toBeAdded[i], found);
    }

}
```

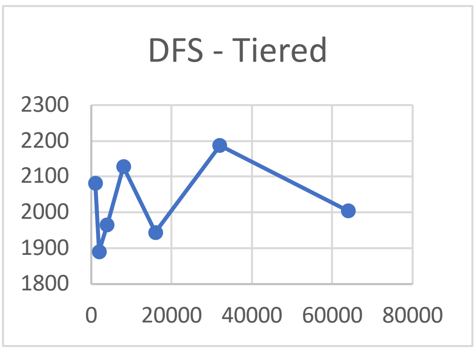
Looking at the runtime of this algorithm, I was expecting it run in **O(V + E)** and on average, as the number of vertices doubled, the runtime of the algorithm roughly doubled. As an affect V is a linear function. We can see this relationship with our use of a cycle and complete graph where we doubled the input size of the vertices for our timing.

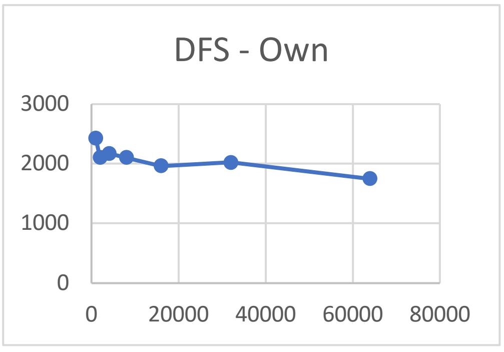




For the edges, I expected them to run in a near to constant time, because in this algorithm we are simply checking what vertex is next and if it has been visited before. This is supported because as E doubled, the time stayed roughly the same or within the same range and we can see this in our distribution graphs.



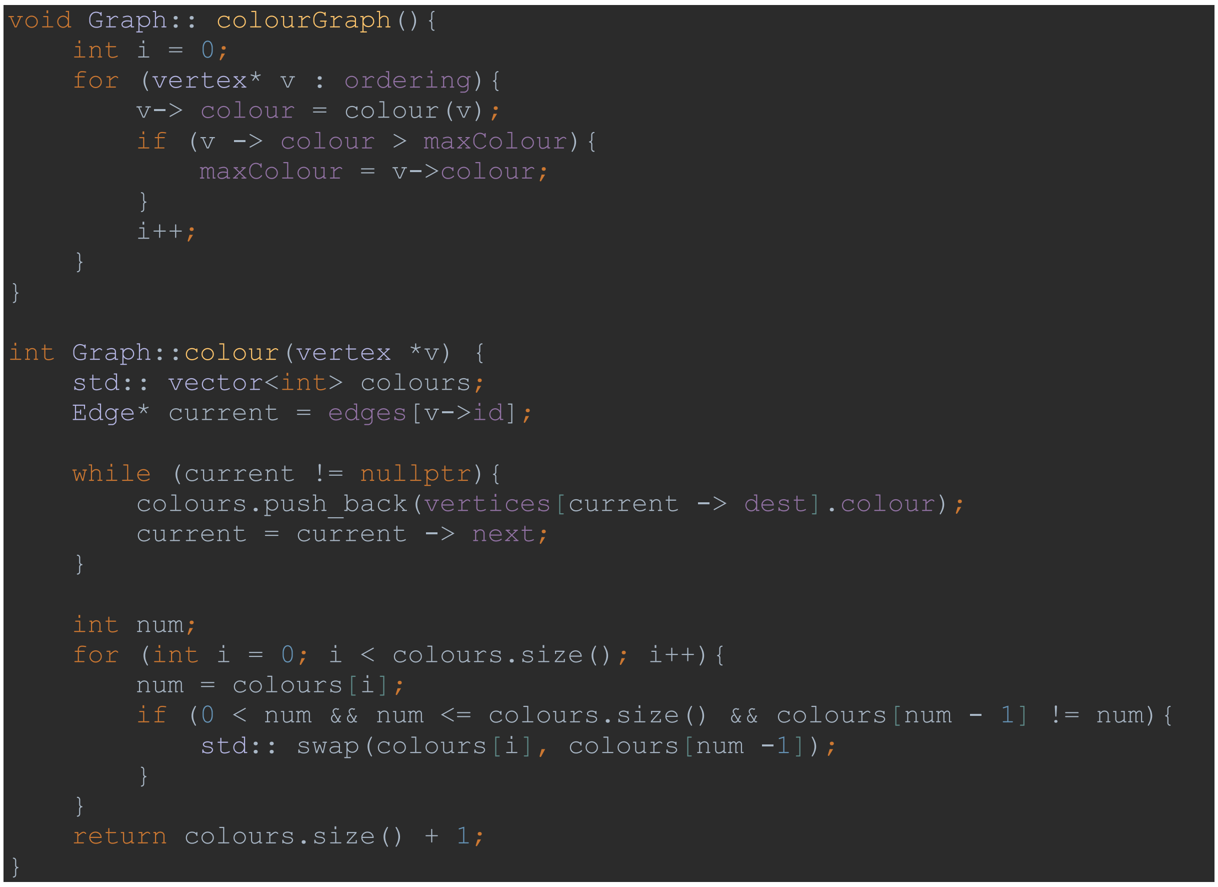




Colouring Algorithm:

In this section, I will be discussing my colouring algorithm and it how it performed with different types of graphs and orderings. My algorithm consists of two functions:

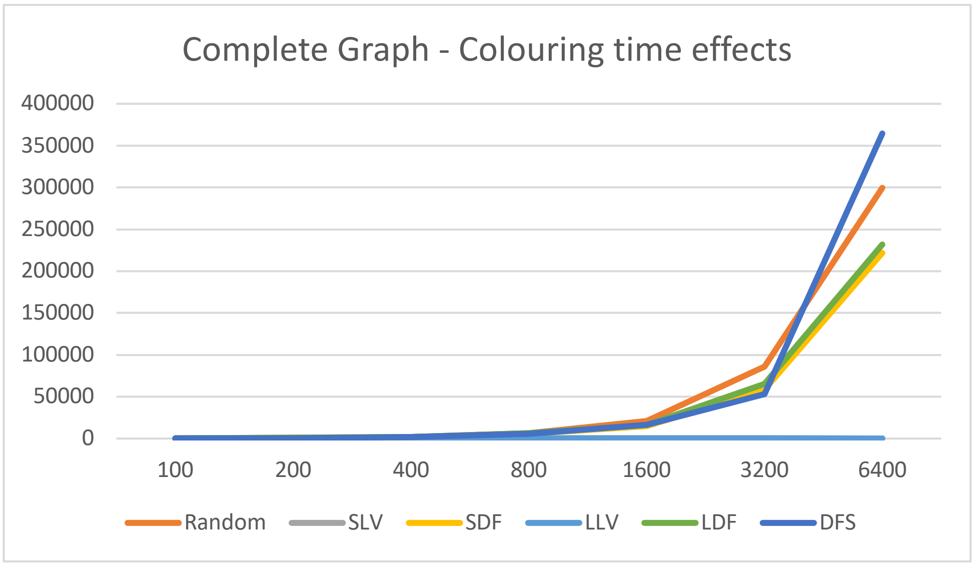
- A colourGraph() function is what calls the colouring function. It is going to start at the first vertex of the ordering and assign colours. At which point, it will then compare the maxColour we've had so far and update it accordingly.
- A colour () function is what is actively responsibly for the graph colouring. This will traverse all the edges of a vertex and assign colours as necessary.



```
void Graph:: colourGraph(){
    int i = 0;
    for (vertex* v : ordering){
        v-> colour = colour(v);
        if (v -> colour > maxColour){
            maxColour = v->colour;
        }
        i++;
    }
}

int Graph::colour(vertex *v) {
    std:: vector<int> colours;
    Edge* current = edges[v->id];

    while (current != nullptr){
        colours.push_back(vertices[current -> dest].colour);
        current = current -> next;
    }

    int num;
    for (int i = 0; i < colours.size(); i++){
        num = colours[i];
        if (0 < num && num <= colours.size() && colours[num - 1] != num){
            std:: swap(colours[i], colours[num -1]);
        }
    }
    return colours.size() + 1;
}
```

The colouring algorithm, similar others in this project can be affected both by the number of vertices in a graph as well as the number of edges. Overall it will run in **O(V + E)** time.

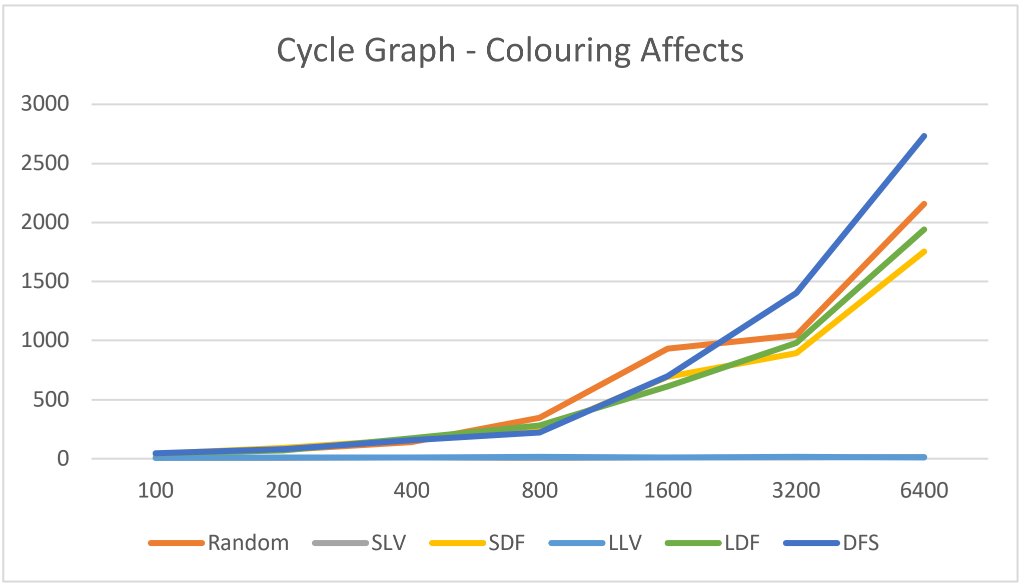
It runs at its worse when using a complete graph. This is because not only does it have to traverse every vertex, but it also then must traverse every edge.



On average, as the number of vertices double the time taken to colour the graph quadruples. As an effect, this algorithm is running between **O(sqrt n) to** O(**n^2). We** can see this in the average runtimes of this graph, where the numbers are showing how many times bigger the time is getting as V doubles:

Complete Graph Runtime Increase

Random	SLV	SDF	LLV	LDF	DFS
3.26542158	1.28357744	3.29552905	1.3362542	3.34782425	3.59130549

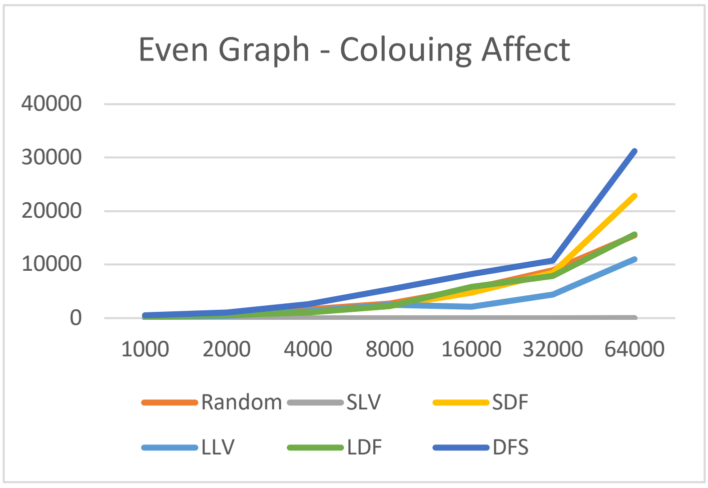


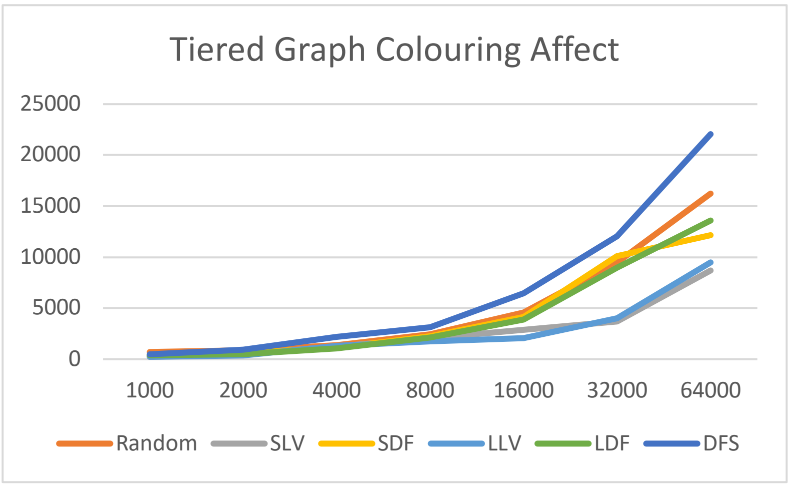
A cycle graph on the other hand, will only need 3 colours regardless of size, and is a good example of the effects of increasing the number of vertices has the colouring algorithm because at most each vertex will have two edges. In an effect, we can see that the vertex portion of this algorithm has the potential to run between **O(sqrt n) to O(n)** time because, as V doubles time taken to colour is get between 1.2 to 2 times bigger.

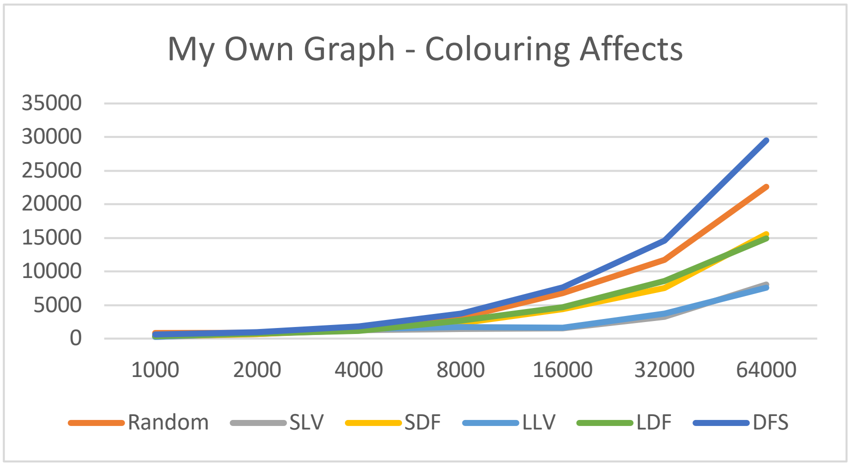
Cycle Graph Runtime Increase

Random	SLV	SDF	LLV	LDF	DFS
2.00839873	1.19180765	1.94182641	1.13097928	1.91401969	2.03512666

When looking at how the number of edges impacted the run time, I once again pre-generated a graph of 10,000 vertices and varied the number of edges. In general, when colouring a distributed graph the colouring algorithm took roughly **O(cube root of E)** time. As the number of edges doubled, the time got roughly 1.8 times bigger.







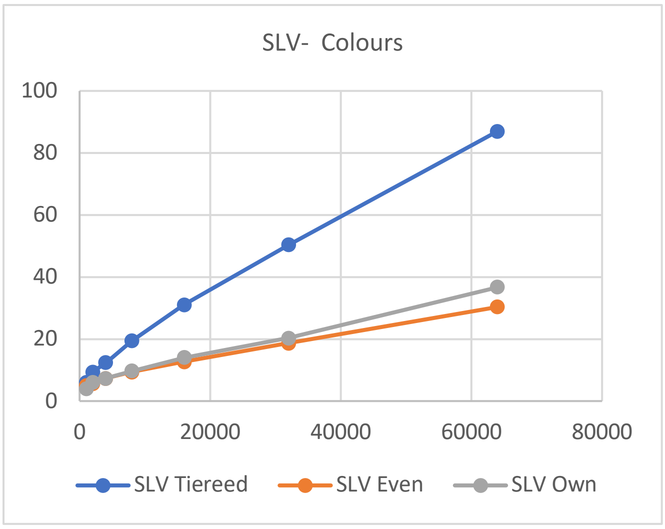
In addition to the graph, this is supported by the average increase in runtime.

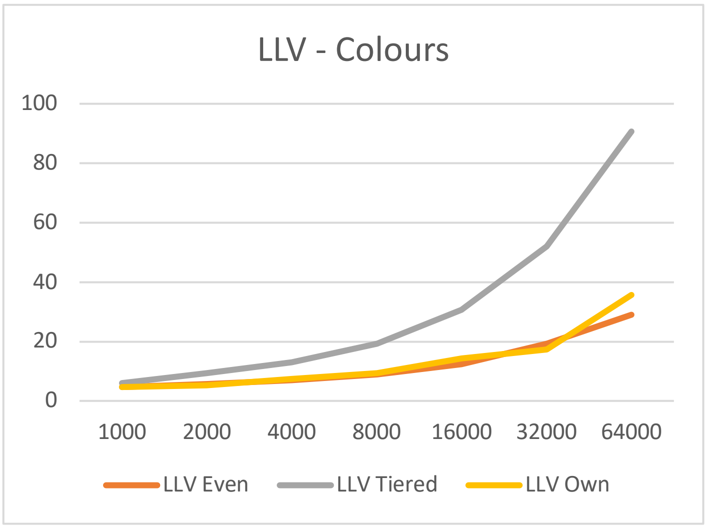
Distribution Graph Runtime Increase

	Random	SLV	SDF	LLV	LDF	DFS
Even	1.77345247	1.19180765	2.01772231	2.11413204	1.99263406	2.04915996
Tiered	1.70841707	1.89800458	1.84591842	1.98765647	1.84879557	1.91770143
Own	1.77759433	1.88139229	1.90105471	1.93312977	1.87321385	1.91489282

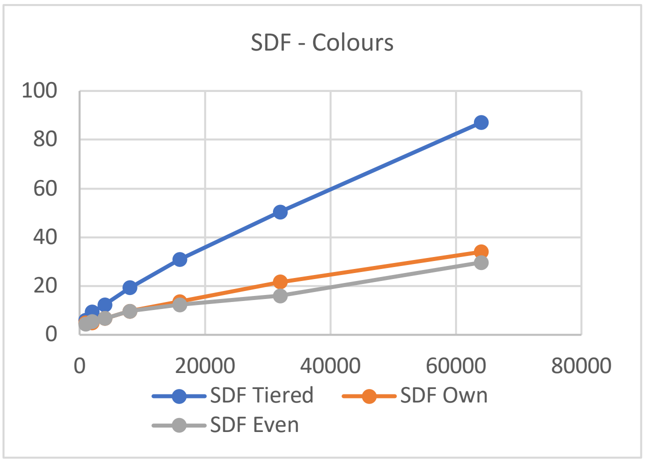
The colouring algorithm itself was running at roughly the same rate, some orderings did perform better than others. In general, the Smallest Last Vertex was running the fastest and the random ordering was running slower.

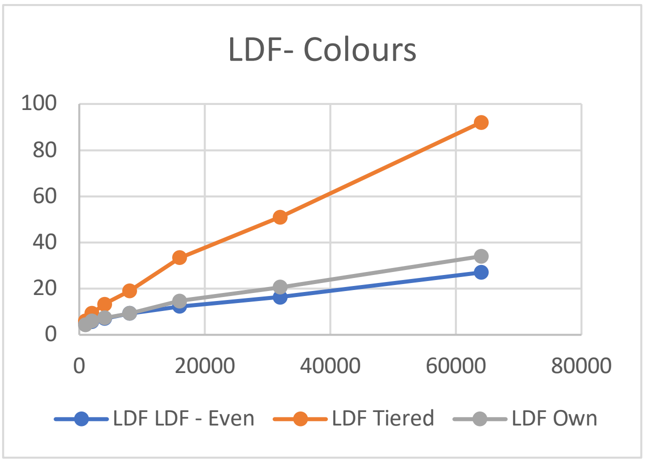
Looking deeper at the number of colours, regardless of what ordering method used a complete graph always had the same number of colours as a vertex. In addition, a cycle graph always had 3 colours regardless of size and ordering. However, and graph type did affect the total number of colours.



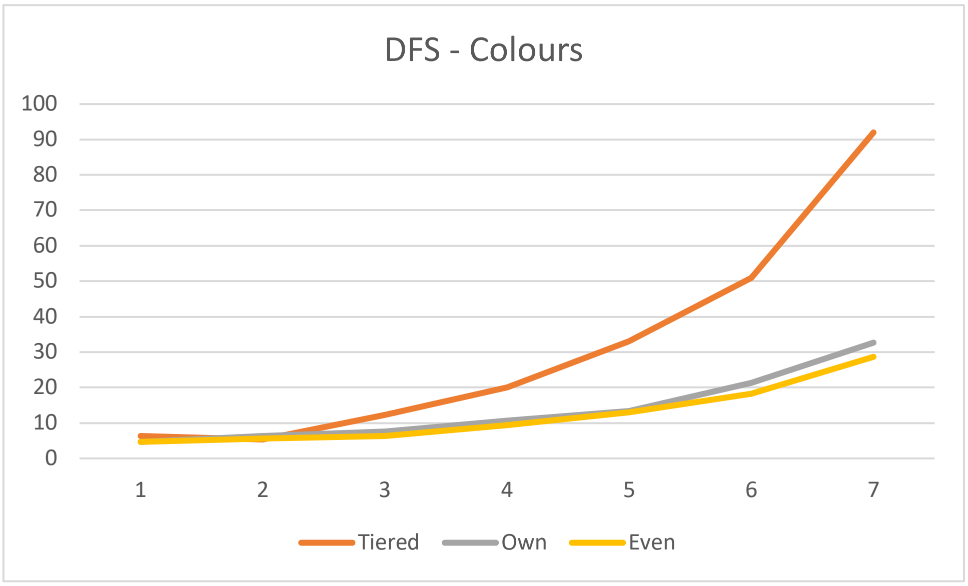


For example, for a smallest last vertex graph the tiered ordering on average needed the most colours. We saw a similar relationship in the Largest Last Vertex too. Bur the tiered graph consistently required the most colours.





In a smallest degree first ordering and in the largest degree first, the tired graph still preformed needed the most colours.



Lastly, with a depth first search ordering, the even graph and my distributed graph are following a similar trend with regards to how many colours they need. However, the tiered graph still needed significantly more colours.

Vertex Ordering Capability's:

For this analysis there are three main concepts that were followed:

1. If an edge is present between two vertices, then a student is taking both of those courses. This also means they cannot be scheduled at the same time.
2. The colour of each graph represents a time slot where it can be scheduled.
3. The number of colours represents the number of timeslots that are needed in a day to schedule of classes.

Overall, in terms of colouring a graph:

- A Complete Graph: will always need the *same number of colours as vertices*. This is because all vertex's will be connected to each other. This makes it a **good baseline** to understand if your algorithm works effectively and can help you test the robustness of your orderings.
- A Cycle graph: Will always *produce three colours*, regardless of the number of vertices. This is because each vertex will only have two edges and will connect to its neighbour. This is good for testing how the number of vertices truly impacts an algorithm as the number of edges stays constant.
- An Even Distribution: This on average produced some of the best colouring results with consistently needing the *least number of colours* overall as the number of edges increased. This is because edges are evenly connected across the graph, and not focused more onto one set of vertices. Overall – providing the least number of timeslots meaning it would be a more realistic way to schedule because you can have more happening at one time.
- A Tiered Distribution: On average, this produced one of the worst colouring results. Overall – *it required more colours as the number of edges increased* in comparison to the other distribution graphs. This is because most of the conflicts where focused on

the first 10% of the graph, meaning it was highly connected in that area. However, a tired distribution might be more realistic in terms of student schedules: for example, if you had a popular major, a student needed to take multiple/different classes to graduate, and it could cause it to be skewed more towards a specific set of classes.

- My Own Distribution: My distributions tried to mimic more of an average students schedule where roughly 2/3 of the classes taken in a semester are the same for about 75% of the student body with the remainder being more major specific. As a result, this distribution produced similar colouring results to an even distribution, in some orderings it did need more colours.

Looking at the ordering algorithms:

- A random ordering: A random ordering is extremely consistent in terms or runtime. This is because it is purely shuffling the graph before colouring it. As a result, it will run in linear time. It is not particular the most efficient algorithm to run in terms of timing, because as the number of students increase the amount of time it takes to order the graph doubles. However, the interconnectedness of the graph and edges does not impact the ordering. But it does impact the colouring algorithm and can cause it to have a higher runtime.
- Smallest Last Vertex/Largest Last Vertex: A smallest last vertex ordering/largest large vertex ordering is running in **theta (V + E) time.** Where the impact of doubling the number of vertices is linear, and the impact of doubling the number of edges between the vertices is also linear. These orderings produced some of the best results with regards to colouring the graph, especially with the tiered graph. For this type of graph, the smallest last vertex produced the best outcome for a tiered graph in comparison to the other orderings. As a result, this ordering would be better as graphs conflicts get

less predictable. When the graphs are closer to being evenly distributed, the number of colours needed because of different graphs is roughly them same.

- Smallest Degree First/Largest Degree First: A smallest degree first/largest degree first algorithm was also found to be running in **theta(V+E)** time. As the number of vertices doubled, on average the runtime of ordering this graph was getting **sqrt(V)** times bigger, and as the number of edges doubled the time stayed constant with regards to ordering the graph. The main purpose of this algorithm was to store the graph by its degree, but it did not delete the edges. The ordering on an evenly distributed graph and my own tiered graph produced a similar number of colours to that of a smallest last vertex ordering. However, with the tiered graph it on average required more colours than the smallest last vertex ordering.
- Depth First Search: This algorithm ran very similarly to the random ordering in terms of run time, but also ran in **theta (V + E)** because it had to check all the vertices and its respective edges. As a result, the run time of checking all the vertices was linear and the runtime of checking the if the edge to another vertex has been visited was constant. It performed very similar to the smallest last vertex graph in that regard. Although for the evenly distributed graph and my own tiered graph it preformed similarly to the other orderings, it needed the most colours on the tiered graph.

Overall, I think the best scheduling algorithm was the **smallest last vertex** ordering. This is because it provided the best colouring outcomes overall. It out preformed the other orderings when it came to colouring the graphs as it produced the least number of overall time slots needed to schedule students for their classes.

Sources:

I tried my best to provide specific links to sources I used within my code. Below is a list of all resources I accessed and used during this project.

https://www.codesdope.com/blog/article/c-linked-lists-in-c-singly-linked-list/

https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/

https://www.softwaretestinghelp.com/linked-list/

https://cplusplus.com/articles/E3wTURfi/

https://cplusplus.com/reference/random/uniform_int_distribution/

https://www.interviewbit.com/blog/graph-coloring-problem/

https://cplusplus.com/forum/general/82433/

https://github.com/mabseher/htd/blob/master/src/htd/RandomOrderingAlgorithm.cpp

https://leetcode.com/problems/first-missing-positive/solutions/17071/my-short-c-solution-o-1-space-and-o-n-time/

https://github.com/MarkBrub/CS5350-Graph_Coloring/blob/master/Coloring/Graph.cpp

https://faculty.cs.niu.edu/~freedman/340/340notes/340graph.htm

https://www.youtube.com/watch?v=Cl3A_9hokjU

https://gist.github.com/MagallanesFito/b9b1215512f3378d674ff43f5542014a

https://github.com/dhanwin247/Parallel-graph-coloring

https://dl.acm.org/doi/pdf/10.1145/2402.322385

https://stackoverflow.com/questions/58388493/is-there-a-difference-between-dfs-and-topological-sort-can-topological-ordering

https://www.geeksforgeeks.org/cpp-program-for-topological-sorting/

https://cppsecrets.com/users/4530104114105116104105107109971081051075654641031099710510846991111109/C00-Program-implementing-Topological-Sort.php

Source Code/ Run Timetable:

Link to GitHub Repo:

https://github.com/nsood1/AlgorthimEngineeringFinalProject/blob/main/graphcolouring/Graph.h

Link to Timing Sheet 1:

https://docs.google.com/spreadsheets/d/1vBt3yEkUkam2I8G000CTq3T3g-1zvTPD/edit?usp=share_link&ouid=109861524476041243052&rtpof=true&sd=true

Link to Timing Sheet 2: https://docs.google.com/spreadsheets/d/1-wQSobtd7qw_oyyE7QAgbka6nvJZDhD6/edit?usp=share_link&ouid=109861524476041243052&rtpof=true&sd=true