Xuan (James) Zhai - CS5350 – Final Project Part B Report

#: Since this part requires the use of pointers, so I chose to use C++ for implementation, and generate this report in Words and Excel spreadsheet.
#: For checking the source code of this final project, visit: https://github.com/XuanZhai/Xuan-James-Zhai-CS5350/tree/main/FinalProject

1: Introduction

In this part B, I am going to implement multiple algorithms for coloring and identify their time complexities. The six algorithms (4 + 2 bonus) that I choose are Smallest Last Vertex Ordering (**SLVO**), Smallest Original Degree Last Vertex Ordering (**SOLVO**), Largest Original Degree Last Vertex Ordering (**LOLVO**), Uniform Random Ordering (**URO**), Breath-First-Search Ordering (**BFSO**), and Depth-First-Search Ordering (**DFSO**). To test the algorithms, I am going to ten groups of datasets with different sizes (100, 200, 300 ⋯ 1000). In each group, it will have a complete graph, a cycle graph, three random graphs with uniform randomness distribution, three random graphs with skewed randomness distribution, and three graphs with Gaussian randomness distribution. For all the random graphs, their number of edges will be ten times of their number of vertices. In the analysis, I will run each algorithm and each graph ten times and find the average time in microseconds, and I will put all the data I collected in tables for further analysis. Eventually, I will find the time complexity for each algorithm and identify its performance in different cases.

2: Code (UML Diagram)

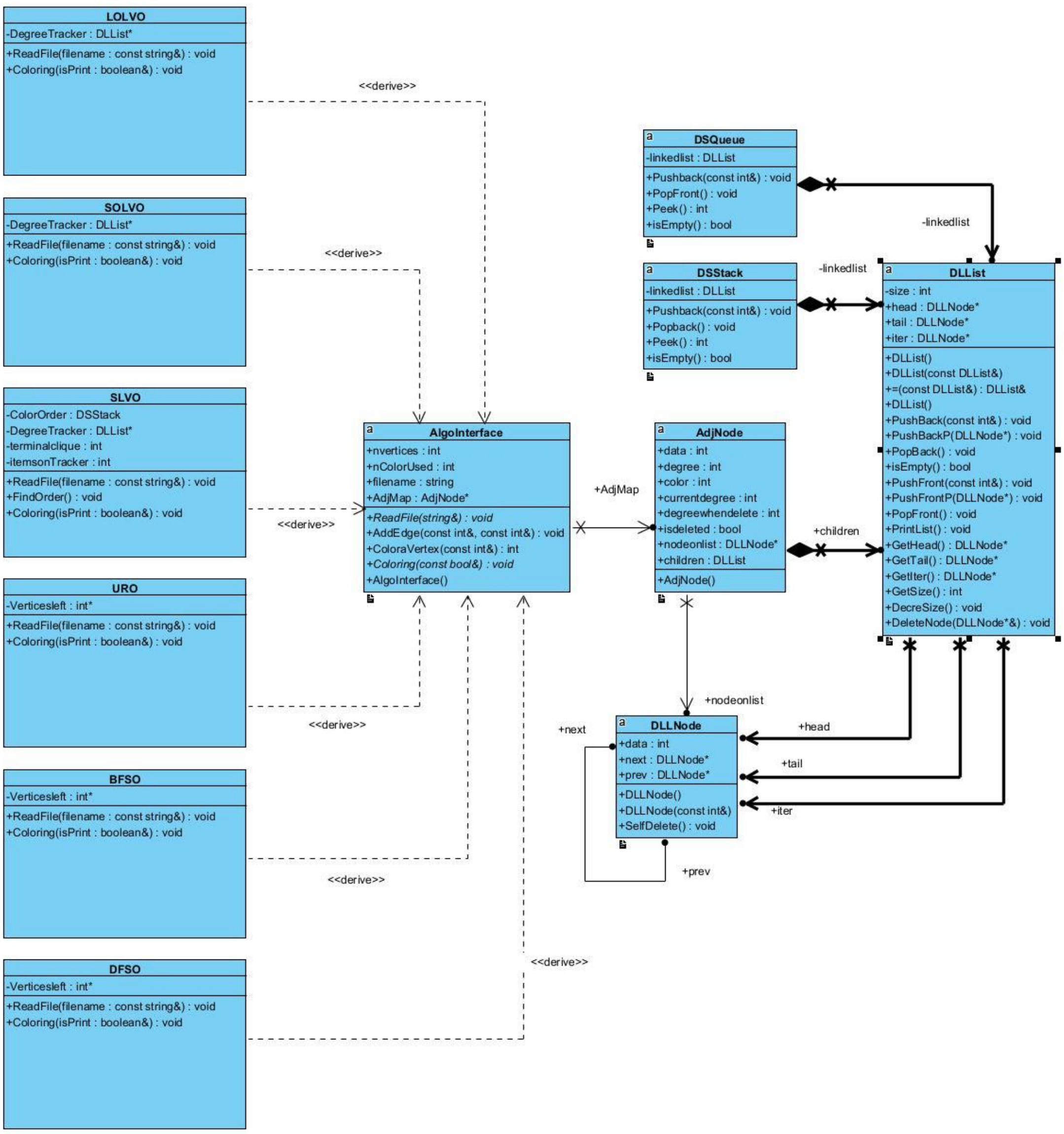

Figure 1: UML Diagram for the program.

3: Smallest Last Vertex Ordering (SLVO)

3.1: Code

For this program, I use three data structures: an adjacency linked list (an array of adj nodes), a degree table (an array of doubly linked lists), and a stack.

When it reads an input file, its time complexity is Θ(V+E). It will first read the number of vertices and allocate memories for the data structures. Then, it will read the pointers for each course and identify the degrees for each vertex. It will then loop different times and add edges to the adjacency linked list. After it read the whole file, it will loop through the adjacency linked list and add the nodes in the degree table and assign pointer to them. Therefore, the time complexity for reading a file is Θ(2V+E) or Θ(V+E).

When it does the coloring, it will first delete all the nodes and fill the stack. The algorithm will loop through all the nodes in the degree table; it will pick up the smallest and delete that vertex, moving all its neighbors with one layer up. When one vertex is deleted, the algorithm will also update the adjacent list and print the info. Finally, it will add that vertex to the stack. Finding the color is also Θ(V+E) because for each node on the degree table, deleting it and moving each of its neighbors one level up are all constant time, and moving all the neighbors is Θ(E), or maybe Θ(2E).

```
void SLVO::FindOrder() {

    int currindex = 0;

    while (itemsonTracker != 0){
        if(!DegreeTracker[currindex].isEmpty()){                // Check if the current level is empty
            if(itemsonTracker == currindex +1 && itemsonTracker > terminalclique){
                terminalclique = itemsonTracker;                // We got the terminal clique
            }


            int itempicked = DegreeTracker[currindex].GetHead()->data;      // Get the first number

            DegreeTracker[currindex].PopFront();                // Remove the first number
            itemsonTracker--;
            AdjMap[itempicked].isdeleted = true;                // Mark deleted
            AdjMap[itempicked].degreewhendelete = AdjMap[itempicked].currentdegree; // Get the degreewhendeleted

            AdjMap[itempicked].nodeonlist = nullptr;            // The node is no longer on the list

            AdjMap[itempicked].children.iter = AdjMap[itempicked].children.head;    // For each of his children
            ColorOrder.Pushback(itempicked);

            while (AdjMap[itempicked].children.iter != nullptr){
                int child = AdjMap[itempicked].children.iter->data;
                if(!AdjMap[child].isdeleted) {                  // Delete that child in the current level
                    DegreeTracker[AdjMap[child].currentdegree].DeleteNode( &: AdjMap[child].nodeonlist);
                    AdjMap[child].currentdegree--;      // Move to next level
                    DegreeTracker[AdjMap[child].currentdegree].PushBackP(AdjMap[child].nodeonlist);
                }
                AdjMap[itempicked].children.iter = AdjMap[itempicked].children.iter->next;
            }
            if(currindex != 0) {
                currindex--;            // Next time starts with the upper level.
            }
        }
        else{
            currindex++;            // If empty, go to the next level.
        }
    }
}
```

Figure 2: Source Code for Finding Orders.

With a filled stack, it will find the color for vertices based on the popped order. For a vertex it popped, it will create a color table; then it will loop through its neighbor and see which colors are used around. Besides, it will go from 0 to the number of original degrees and find the smallest color it can have. Finding a color for one vertex requires looping through all its neighbors two times, so finding all the colors will still be a constant time, or Θ(V+E). Note: This time complexity may be Θ(V+2E) since it will loop number of original degrees times instead of number of degrees when deleted times, and it may lead to one more duplicate. But since all my algorithms require that function, so I want to keep the time complexity for that part be the same.

```
int AlgoInterface::ColoraVertex(const int& index){
    bool* colormap = new bool[nvertices]{false};            // A list of color checker
    DLLNode* tempi = AdjMap[index].children.head;
    while (tempi != nullptr){
        if(AdjMap[tempi->data].color != -1){
            colormap[AdjMap[tempi->data].color] = true;      // Go through neighbours and check which colors are used.
        }
        tempi = tempi->next;
    }

    for(int i = 0; i < AdjMap[index].children.GetSize()+1; i++){
        if(!colormap[i]){                                // Starts from 0, check which color is not used.
            if(i > nColorUsed){
                nColorUsed = i;                          // Update the number of colors used if necessary
            }
            delete[] colormap;
            return i;
        }
    }
    printf( format: "No color found");
    delete[] colormap;
    return nColorUsed;
}
```

Figure 3: Source code for coloring a vertex

3.2: Sample Input & Output

```
6
7
9
12
16
21
24
2
3
2
3
5
1
3
0
4
1
2
4
0
5
3
2
5
1
4
3
```

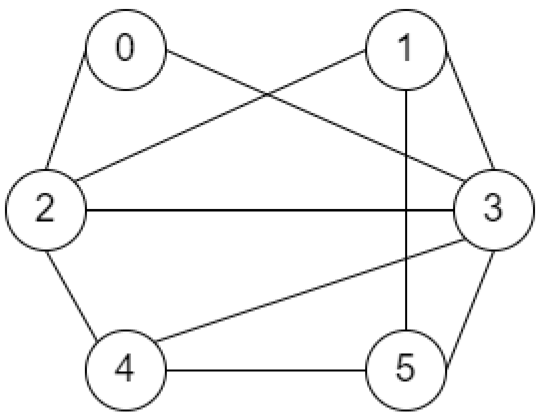

Figure 4: Sample Input (Will be used for all the algorithms)

```
Coloring: 3. The color is 0; Original Degree is: 5; Degree when deleted is: 0
Coloring: 4. The color is 1; Original Degree is: 3; Degree when deleted is: 1
Coloring: 5. The color is 2; Original Degree is: 3; Degree when deleted is: 2
Coloring: 2. The color is 2; Original Degree is: 4; Degree when deleted is: 2
Coloring: 1. The color is 1; Original Degree is: 3; Degree when deleted is: 3
Coloring: 0. The color is 1; Original Degree is: 2; Degree when deleted is: 2

=====================================
Total number of colors used: 3
The average Original degree: 3
The Maximum degree when deleted: 3
The size of terminal clique: 3
=====================================
```

Figure 5: Sample Output with SLVO

3.3 Analysis

Although SLVO requires an order before coloring, since both finding the order and finding the color are in linear time, the time complexity for that algorithm is still in linear time, or Θ(V+E). For the sample above, at step one the smallest vertex is 0. After removing 0, there will be four vertices has a degree of 3. Therefore, the program chose 1. After that, the smallest is 0 or 2, so the program chose 5 and 2. Finally, it will remove 3, and the order will be 0, 1, 2, 5, 4, 3; The terminal clique will be 3 which is a triangle formed by vertex 3, 4, 5. When we find the coloring, we just flipped the order, and do the coloring followed by that order.

4: Smallest Original Degree Last Vertex Ordering (SOLVO)

4.1: Code

This algorithm also has an adjacency linked list and a degree table, and the read file function will construct those two data structures with vertices and edges. However, this algorithm will not find a specific order. But instead, it will loop through all the nodes on the degree table, from the one with the greatest number of degrees to the one with the least number of degrees and do coloring. So, on the degree table the iterator will go from level maxdegree to 0.

```cpp
void SOLVO::Coloring(const bool& isPrint){
    int totaloriginaldegree = 0;                // From top to bottom instead of from bottom to top
    int stacksize = 0;
    int trackerlevel = nvertices-1;             // Here it starts at the maxdegree and goes to the smallest.
    ofstream outputfile;
    if(isPrint){
        outputfile.open("out_"+filename);
    }

    while (stacksize != nvertices){
        if(DegreeTracker[trackerlevel].isEmpty()){
            trackerlevel--;
        }
        else{
            DLLNode* temp = DegreeTracker[trackerlevel].GetHead();
            while (temp != nullptr){
                int selected = temp->data;
                int newcolor = ColoraVertex(selected);
                AdjMap[selected].color = newcolor;
                if(isPrint) {
                    totaloriginaldegree += AdjMap[selected].degree;
                    cout << "Coloring: " << selected << ". The color is " << newcolor << "; Original Degree is: " << AdjMap[selected].degree << "." << endl;
                    outputfile << temp << ", " << newcolor << endl;
                }
                stacksize++;
                temp = temp->next;
            }
            trackerlevel--;
        }
    }
    outputfile.close();
    cout << "\n====================================" << endl;
    cout << "Total number of colors used: " << nColorUsed+1 << endl;
    cout << "The average Original degree: " << totaloriginaldegree / stacksize << endl;
    cout << "====================================" << endl;
}
```

Figure 6: Source Code for Coloring (SOLVO)

4.2: Sample Output

```
Coloring: 3. The color is 0; Original Degree is: 5.
Coloring: 2. The color is 1; Original Degree is: 4.
Coloring: 1. The color is 2; Original Degree is: 3.
Coloring: 4. The color is 2; Original Degree is: 3.
Coloring: 5. The color is 1; Original Degree is: 3.
Coloring: 0. The color is 2; Original Degree is: 2.

====================================
Total number of colors used: 3
The average Original degree: 3
====================================
```

Figure 7: Sample Output (SOLVO)

4.3: Analysis

Although this algorithm does not find a specific order, it still needs to go over all the nodes on the degree table and coloring them. But it is still a constant time which is Θ(V+E). However, it is not showing on the sample output, but this algorithm does not guarantee to find the smallest number of colors it needs for coloring that graph.

5: Largest Original Degree Last Vertex Ordering (LOLVO)

5.1: Code

This algorithm is similar to SOLVO, and the only difference is instead of coloring the vertices from the one with the greatest number of degrees to the one with the smallest, we do it vice versa.

```
void LOLVO::Coloring(const bool& isPrint){      // Same algorithm as what in SOLVO, only loop through the table
    int totaloriginaldegree = 0;
    int stacksize = 0;                  // It's the number of colored vertices
    int trackerlevel =0;                // It's the level in the degree table
    ofstream outputfile;
    if(isPrint){
        outputfile.open("out_"+filename);
    }

    while (stacksize != nvertices){         // Loop through each node on the degree table
        if(DegreeTracker[trackerlevel].isEmpty()){
            trackerlevel++;                 // If at that level there's no vertex, go to the next level
        }
        else{
            DLLNode* temp = DegreeTracker[trackerlevel].GetHead();      // Loop through that level
            while (temp != nullptr){
                int selected = temp->data;                  // Get a vertex
                int newcolor = ColoraVertex(selected);      // Find a color for it
                AdjMap[selected].color = newcolor;
                if(isPrint) {
                    totaloriginaldegree += AdjMap[selected].degree;
                    cout << "Coloring: " << selected << ". The color is " << newcolor << "; Original Degree is: " << AdjMap[selected].degree << "." << endl;
                    outputfile << temp << ", " << newcolor << endl;
                }
                stacksize++;
                temp = temp->next;
            }
            trackerlevel++;
        }
    }
    outputfile.close();
    cout << "\n====================================" << endl;
    cout << "Total number of colors used: " << nColorUsed+1 << endl;
    cout << "The average Original degree: " << totaloriginaldegree / stacksize << endl;
    cout << "====================================" << endl;
}
```

Figure 8: Source Code for Coloring (LOLVO)

5.2: Sample Output

```
Coloring: 0. The color is 0; Original Degree is: 2.
Coloring: 1. The color is 0; Original Degree is: 3.
Coloring: 4. The color is 0; Original Degree is: 3.
Coloring: 5. The color is 1; Original Degree is: 3.
Coloring: 2. The color is 1; Original Degree is: 4.
Coloring: 3. The color is 2; Original Degree is: 5.


====================================
Total number of colors used: 3
The average Original degree: 3
====================================
```

Figure 9: Sample Output (LOLVO)

5.3: Analysis

This algorithm is similar to SOLVO. So, it is in constant time. But since it starts with the one with the smallest number of original degrees, its performance in finding the minimum number of colors used may be worse in some cases compared to SOLVO. Here is an example. For an input like:

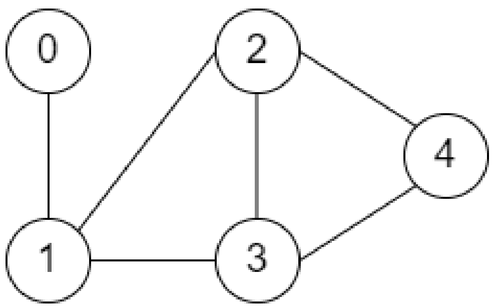

Figure 10: Input for Special Case (LOLVO & SOLVO)

The result from SOLVO is:

```
Coloring: 1. The color is 0; Original Degree is: 3.
Coloring: 2. The color is 1; Original Degree is: 3.
Coloring: 3. The color is 2; Original Degree is: 3.
Coloring: 4. The color is 0; Original Degree is: 2.
Coloring: 0. The color is 1; Original Degree is: 1.

=====================================
Total number of colors used: 3
The average Original degree: 2
=====================================
```

Figure 11: Output for Special Case (SOLVO)

But the result from LOLVO is what is below, which needs one more color.

```
Coloring: 0. The color is 0; Original Degree is: 1.
Coloring: 4. The color is 0; Original Degree is: 2.
Coloring: 1. The color is 1; Original Degree is: 3.
Coloring: 2. The color is 2; Original Degree is: 3.
Coloring: 3. The color is 3; Original Degree is: 3.

=====================================
Total number of colors used: 4
The average Original degree: 2
=====================================
```

Figure 12: Output for Special Case (LOLVO)

6: Uniform Random Ordering (URO)

6.1: Code

This algorithm needs two data structures which is the adjacency linked list and a list which tracking the unselected vertices. When it built the two lists after reading the file, while the list for unselected vertices is not empty, the algorithm will randomly pick (uniform distribution) a vertex and color it, and it will delete that node from the list which means that vertex is colored.

```cpp
void URO::Coloring(const bool& isPrint){
    int nverremain = nvertices;             // How many vertices are not colored
    int totaloriginaldegree = 0;
    int selectedsize = 0;
    ofstream outputfile;
    if(isPrint){
        outputfile.open("out_"+filename);
    }

    while (nverremain != 0){                // While not all been selected
        int selindex = rand()%nverremain;       // Randomly pick one that's unselected
        int selected = Verticesleft[selindex];
        Verticesleft[selindex] = Verticesleft[nverremain-1];  // Delete it in the array
        int newcolor = ColoraVertex(selected);          // Find a color for it
        AdjMap[selected].color = newcolor;

        totaloriginaldegree += AdjMap[selected].degree;
        selectedsize++;
        if(isPrint) {
            cout << "Coloring: " << selected << ". The color is " << newcolor << "; Original Degree is: " << AdjMap[selected].degree << "." << endl;
            outputfile << selected << ", " << newcolor << endl;
        }
        nverremain--;
    }
    outputfile.close();
    cout << "\n====================================" << endl;
    cout << "Total number of colors used: " << nColorUsed+1 << endl;
    cout << "The average Original degree: " << totaloriginaldegree / selectedsize << endl;
    cout << "====================================" << endl;
}
```

Figure 13: Source Code for Coloring (URO)

6.2: Sample Output

```
Coloring: 1. The color is 0; Original Degree is: 3.
Coloring: 5. The color is 1; Original Degree is: 3.
Coloring: 4. The color is 0; Original Degree is: 3.
Coloring: 3. The color is 2; Original Degree is: 5.
Coloring: 2. The color is 1; Original Degree is: 4.
Coloring: 0. The color is 0; Original Degree is: 2.

====================================
Total number of colors used: 3
The average Original degree: 3
====================================
```

Figure 14: Sample Output (URO)

6.3: Analysis

For each round, this algorithm will randomly pick a vertex, and delete it from the time; it is Θ (1) because randomly pick a vertex is constant, and when it wants to delete a vertex, it will first switch that node with the last one, and then pop the last one, so it is also a constant time complexity. Therefore, this algorithm has a time complexity which is also Θ (V+E). Nevertheless, since it just does the random picking, it does not guarantee to find the minimum number of colors.

7: Breath-First-Search Ordering (BFSO)

7.1: Code

This algorithm will use a Breath-first-search algorithm, but instead of doing a searching, it will just loop through all the vertices. It will first have a "visited" array to avoid a cycle and a queue. This algorithm will start with the vertex "0" and each round it will push all the current node's neighbors to the queue. When the queue is empty, it does not mean it went through all the vertices because sometime a vertex may not have any adjacent vertices. For example, if vertex 0 does not have any edge connected, it will just color vertex 0. Therefore, it needs an iterator in the "visited" array to track the unvisited node. If the queue is empty, we will push the smallest unvisited node to the queue and update the iterator. The code will stop only if all the nodes are visited.

```cpp
void BFSO::Coloring(const bool& isPrint){
    bool *visited = new bool[nvertices]{false};
    DSQueue BFSQueue;
    int loopindex = 0;
    int nverremain = nvertices;
    int totaloriginaldegree = 0;
    int selectedsize = 0;
    ofstream outputfile;
    if(isPrint){
        outputfile.open( s: "out_"+filename);
    }
    BFSQueue.Pushback(0);       // Start the BFS algorithm
    visited[0] = true;

    while (nverremain != 0){
        if(BFSQueue.isEmpty()){
            loopindex++;
            if(!visited[loopindex]){
                BFSQueue.Pushback(loopindex);
            }
        }

        while(!BFSQueue.isEmpty()){             // BFS search that will loop through the graph.
            int selected = BFSQueue.Peek();
            BFSQueue.PopFront();
            int newcolor = ColoraVertex(selected);
            AdjMap[selected].color = newcolor;

            totaloriginaldegree += AdjMap[selected].degree;
            selectedsize++;
            if(isPrint) {
                cout << "Coloring: " << selected << ". The color is " << newcolor << "; Original Degree is: " << AdjMap[selected].degree << "." << endl;
                outputfile << selected << ", " << newcolor << endl;
            }
            nverremain--;

            DLLNode* temp = AdjMap[selected].children.head;
            while (temp != nullptr){
                if(!visited[temp->data]){
                    visited[temp->data] = true;
                    BFSQueue.Pushback(temp->data);
                }
                temp = temp->next;
            }

        }
    }
    outputfile.close();
    cout << "\n===================================" << endl;
    cout << "Total number of colors used: " << nColorUsed+1 << endl;
    cout << "The average Original degree: " << totaloriginaldegree / selectedsize << endl;
    cout << "===================================" << endl;
    delete[] visited;
}
```

Figure 15: Source Code for Coloring (BFSO)

7.2: Sample Output

```
Coloring: 0. The color is 0; Original Degree is: 2.
Coloring: 2. The color is 1; Original Degree is: 4.
Coloring: 3. The color is 2; Original Degree is: 5.
Coloring: 1. The color is 0; Original Degree is: 3.
Coloring: 4. The color is 0; Original Degree is: 3.
Coloring: 5. The color is 1; Original Degree is: 3.

====================================
Total number of colors used: 3
The average Original degree: 3
====================================
```

Figure 16: Sample Output (BFSO)

7.3: Analysis

Unlike URO or other similar algorithm, BFSO loop through the whole graph, a BFS algorithm in its worst case, is Θ(V+E). Therefore, BFSO algorithm may have a time complexity which is O(V+2E) or O(V+3E), but it is still a linear time. This algorithm also does not guarantee to find the minimum number of colors used since it starts at 0 and always go to a vertex's neighbors first. If the first several vertices it is coloring is the one with small original degree, it will be like SOLVO and cannot find the minimum number of colors.

8: Depth-First-Search Ordering (DFSO)

8.1: Code

The only difference between this algorithm and the BFSO is it uses a Depth-first-search algorithm. Thus, instead of using a queue, it will use a stack. Other than that, both algorithms are the same.

8.2: Sample Output

```
Coloring: 0. The color is 0; Original Degree is: 2.
Coloring: 3. The color is 1; Original Degree is: 5.
Coloring: 5. The color is 0; Original Degree is: 3.
Coloring: 4. The color is 2; Original Degree is: 3.
Coloring: 1. The color is 2; Original Degree is: 3.
Coloring: 2. The color is 3; Original Degree is: 4.

=====================================
Total number of colors used: 4
The average Original degree: 3
=====================================
```

Figure 17: Sample Output (DFSO)

8.3: Analysis

This algorithm also has a linear time complexity and does not guarantee to find the minimum number of colors. Its difference between BFSO is only shown in different graphs. For example, if it starts with vertices that has neighbors with large degrees, BFSO may be a better choice, while in other cases DFSO may be better.

9: Comparison and Check the Time Complexity

9.1: Code

For one dataset, the program will run all six algorithms 10 times and find the average time in microseconds. The function has one parameter which is the input file name, and it should be an absolute path. Since this program uses design patterns, it can just create an array of algorithms and run all of them in a for loop. The time measurement algorithm I used is the high_resolution_clock from the chrono library in C++.

```cpp
void RunTimeTest(const std::string& filename){
    AlgoInterface* Algos[6];
    Algos[0] = new SLVO;
    Algos[1] = new URO;
    Algos[2] = new SOLVO;
    Algos[3] = new LOLVO;
    Algos[4] = new BFSO;
    Algos[5] = new DFSO;

    long int times[6]{ 0 };

    for(int i = 0; i < 6; i++){
        long int time = 0;
        for(int j = 0; j < 10; j++) {
            Algos[i]->ReadFile(filename);
            auto start = high_resolution_clock::now();
            Algos[i]->Coloring(false);
            auto end = high_resolution_clock::now();
            time += duration_cast<microseconds>(end - start).count();
        }
        time /= 10;
        times[i] = time;
    }

    for(int i = 0; i < 6; i++){
        printf( "%d is: %ldms \n", i, times[i]);
    }

    for(auto & Algo : Algos){
        delete Algo;
    }
}
```

Figure 18: Source Code for Finding Time

9.2: Running Times (All running times will be in microseconds)

9.2.1: SLVO

When collected all the running time, we can find out for SLVO algorithm.

SLVO										
	100	200	300	400	500	600	700	800	900	1000
Complete	261	892	2044	3633	5635	8370	12080	15440	18547	24568
Cycle	27	69	108	184	232	343	403	594	621	749
Random1	86	194	322	438	525	740	903	1094	1283	1400
Random2	86	194	319	451	537	732	895	1076	1295	1509
Random3	90	193	326	451	534	743	899	1141	1237	1436
RandomS1	99	215	339	481	629	786	928	1094	1327	1458
RandomS2	102	214	335	462	614	769	953	1108	1325	1441
RandomS3	98	206	336	473	575	775	939	1253	1399	1441
RandomG1	90	192	317	439	587	739	900	1069	1304	1465
RandomG2	86	194	337	463	582	748	905	1061	1273	1417
RandomG3	86	192	320	456	587	751	904	1032	1283	1404

Figure 19: SLVO Running Time (Table)

When we put them into a graph, we will see:

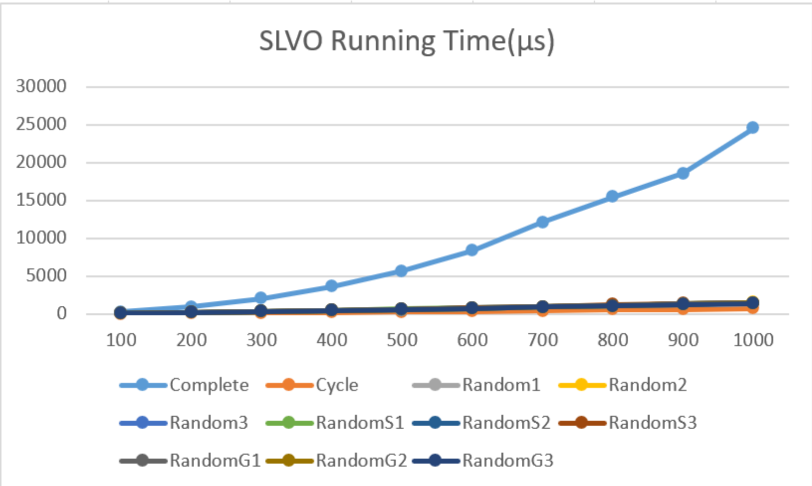

Figure 20: SLVO Running Time (Graph)

When it is a complete graph, E will be $\frac{n*(n-1)}{2}$. Therefore, Θ (V+E) will be close to Θ (V^2).

From the table and graph above we can see that, if we double the size of dataset, the running time becomes 4 times than before, so it proves our assumption that this SLVO algorithm is Θ (V+E).

9.2.2: URO

Here is the table and graph for URO algorithm:

URO										
	100	200	300	400	500	600	700	800	900	1000
Complete	120	376	768	1366	2024	3021	4153	5813	6438	8318
Cycle	15	43	80	144	199	256	367	464	567	682
Random1	35	93	157	237	291	410	541	693	880	888
Random2	37	86	165	236	285	419	528	642	791	894
Random3	38	85	163	229	286	420	528	655	775	892
RandomS1	34	88	168	236	332	419	526	678	767	847
RandomS2	36	91	163	237	322	411	509	679	793	910
RandomS3	34	95	158	237	278	411	520	635	802	977
RandomG1	37	92	163	246	312	412	540	627	782	873
RandomG2	34	92	153	252	321	412	538	637	772	872
RandomG3	34	89	157	223	325	408	523	621	789	952

Figure 21: URO Running Time (Table)

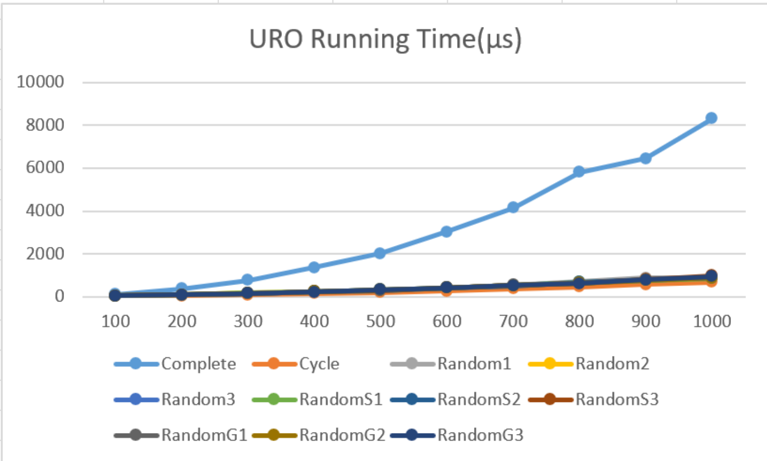

Figure 22: URO Running Time (Graph)

Again, when we look at the running time for the complete graph, we will find out it is still Θ (V^2). When we just look at the time for a cycle graph, we will see it is a linear time. Therefore, it is Θ (V+E).

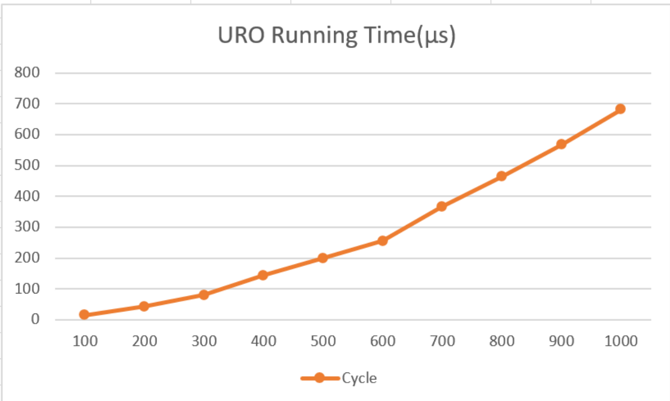

Figure 23: URO Running Time (Cycle's graph)

9.2.3: SOLVO

When we check the result from SOLVO algorithm, and later the LOLVO algorithm, we can find out although it's also Θ (V+E), its running time is little bit faster than the URO algorithm. That may cause by the random number generation and selection process, while for SOLVO and LOLVO it just loops through the degree table.

SOLVO										
	100	200	300	400	500	600	700	800	900	1000
Complete	90	293	654	1119	1665	2606	3401	5333	5693	6882
Cycle	14	37	76	114	177	244	324	408	526	634
Random1	37	92	150	233	279	425	530	653	814	861
Random2	33	85	164	238	279	432	538	646	814	864
Random3	34	89	151	232	280	417	529	665	729	868
RandomS1	35	95	156	233	328	428	549	683	801	856
RandomS2	35	89	154	240	335	419	534	638	793	860
RandomS3	38	84	155	240	273	420	538	648	920	882
RandomG1	32	92	155	241	316	416	535	634	803	853
RandomG2	31	85	152	257	321	416	546	620	783	861
RandomG3	34	92	152	243	321	419	522	617	794	894

Figure 24: SOLVO Running Time (Table)

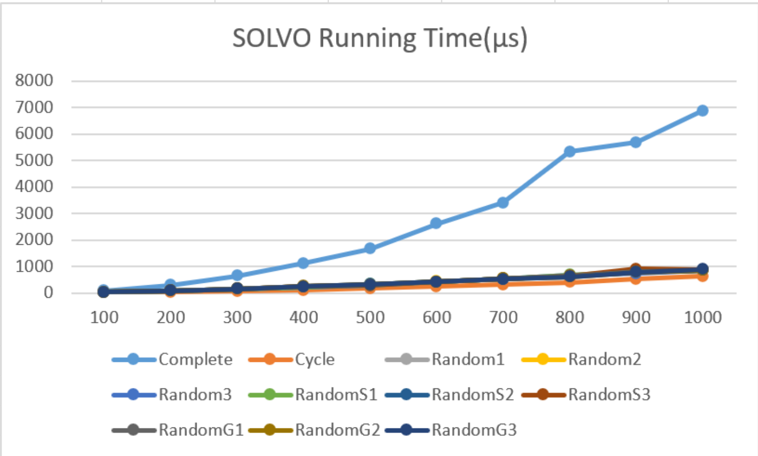

Figure 25: SOLVO Running Time (Graph)

9.2.4: LOLVO

Here is the result from LOLVO algorithm. Since it loops through the degree table in a reverse order, the running time should be close to the one from SOLVO algorithm, and the time complexity should be the same. One interesting thing is from 800 vertices to 900 vertices, the running time for complete graph decreased.

LOLVO										
	100	200	300	400	500	600	700	800	900	1000
Complete	83	285	647	1189	1659	2592	3332	5498	5430	7041
Cycle	13	37	68	118	180	245	312	406	516	617
Random1	36	90	160	234	301	420	543	655	837	886
Random2	29	88	166	240	304	431	538	670	789	865
Random3	39	91	155	234	271	439	539	659	871	879
RandomS1	37	99	167	256	339	438	546	643	807	918
RandomS2	36	93	165	251	336	430	553	691	805	876
RandomS3	37	88	165	242	281	445	552	699	865	863
RandomG1	31	91	162	240	321	416	556	637	787	839
RandomG2	33	86	155	242	336	421	537	621	785	883
RandomG3	34	90	163	232	326	424	532	675	793	842

Figure 26: LOLVO Running Time (Table)

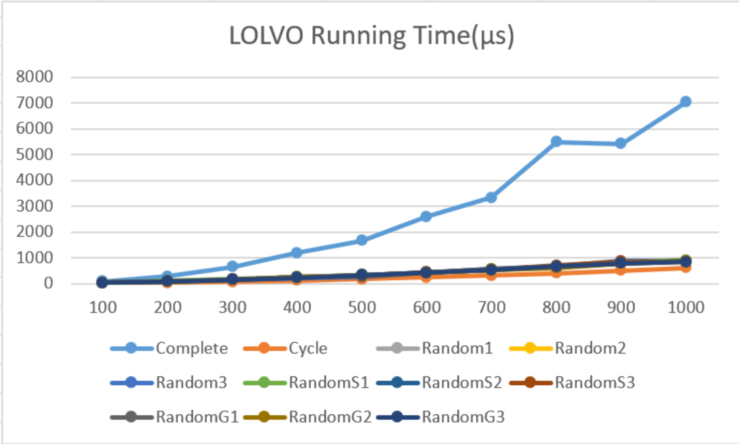

Figure 27: LOLVO Running Time (Graph)

9.2.5: BFSO

Here are the results from BFSO algorithm. We can see that its running time is between SVLO and the three algorithms above, but it is still a linear time. This may be O (V+3E), but still also Θ (V+E). However, for random graphs, sometime BFSO (and DFSO) has really a bad running time. Check RandomS3 when the number of vertices is 800 below, although it also happened in other algorithm's result, I think that is because it is a special skewed random graph which smaller vertices are all connected, and it takes time to loop through the array to find visited vertices.

BFSO										
	100	200	300	400	500	600	700	800	900	1000
Complete	133	436	952	1663	2577	3658	5015	7757	8168	10937
Cycle	21	51	89	138	199	274	367	473	568	683
Random1	54	120	202	290	377	525	678	813	1014	1065
Random2	56	117	193	309	279	535	682	811	948	1185
Random3	49	119	202	302	354	533	683	815	931	1079
RandomS1	49	124	205	305	407	553	674	792	934	1027
RandomS2	50	125	204	307	411	523	685	799	960	1061
RandomS3	50	113	203	287	370	527	689	1213	1029	1046
RandomG1	51	123	201	300	389	505	642	768	919	1038
RandomG2	47	118	196	303	391	500	644	754	904	1006
RandomG3	56	120	219	303	411	533	699	808	919	1039

Figure 28: BFSO Running Time (Table)

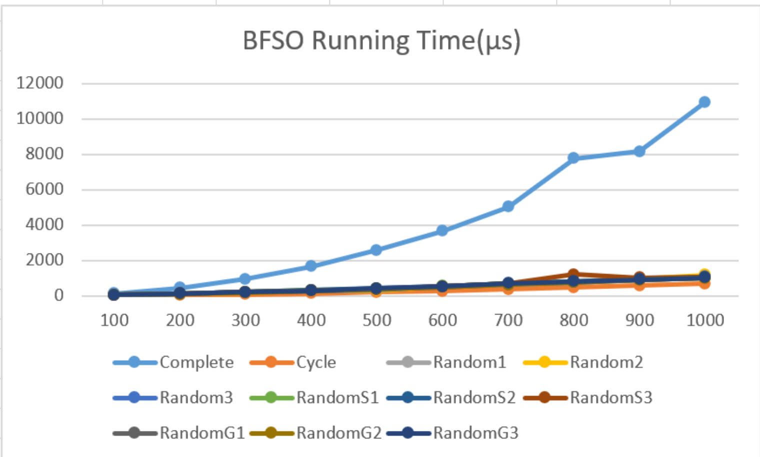

Figure 29: BFSO Running Time (Graph)

9.2.6: DFSO

Here are the results from the DFSO algorithm. Since in this algorithm we changed the BFSO algorithm from a queue to a stack. The overall running time should be close, and the time complexity should also be a linear Θ(V+E).

DFSO										
	100	200	300	400	500	600	700	800	900	1000
Complete	164	496	1072	1850	2973	4356	5659	7608	8831	11266
Cycle	20	48	84	135	197	266	352	447	554	688
Random1	47	117	198	291	353	520	668	860	983	1055
Random2	49	115	190	281	354	526	663	897	917	1095
Random3	48	118	207	304	340	513	651	778	1006	1063
RandomS1	50	119	208	295	403	515	1421	762	893	973
RandomS2	54	115	198	288	339	500	657	763	1043	1043
RandomS3	51	116	189	289	343	504	643	743	963	1058
RandomG1	48	117	202	294	389	494	629	731	910	1050
RandomG2	47	119	195	288	394	503	631	740	901	1017
RandomG3	54	115	232	304	392	532	622	786	897	1013

Figure 30: DFSO Running Time (Table)

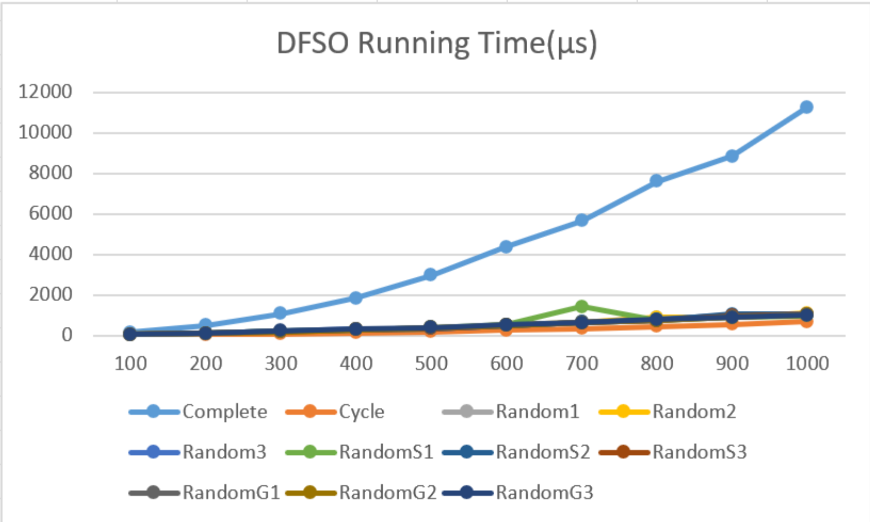

Figure 31: DFSO Running Time (Graph)

10: Algorithm's Comparison

10.1: Complete Graph

For a Complete Graph, when we put all the algorithms' running time together, we'll find out that SLVO is the slowest, since it needs to find an order before doing the coloring. After it, BFSO and DFSO are relatively slow because they need to get an order based on a Breath-first search or a Depth-first search. SOLVO and LOLVO are the fastest because it knew the order (on the degree table) when it read in the file. URO is at the middle since it needs to do a random selection to get an order, but still, it is fast and close to SOLVO and LOLVO.

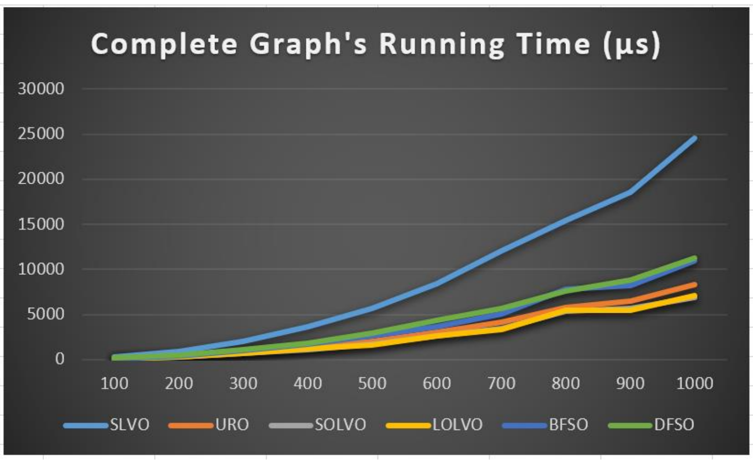

Figure 32: Running Time for a Complete Graph

10.2: Cycle Graph

When it is a cycle graph, the comparison result is showing below. Although the SLVO algorithm is still the slowest, but it is close to the rest. That is because for a cycle, since all vertices' degrees are one. Finding an order will just modify the first layer on the degree table, so it may be faster. For the rest of the algorithms, we can see the order is similar to the one in Complete Graph's comparison.

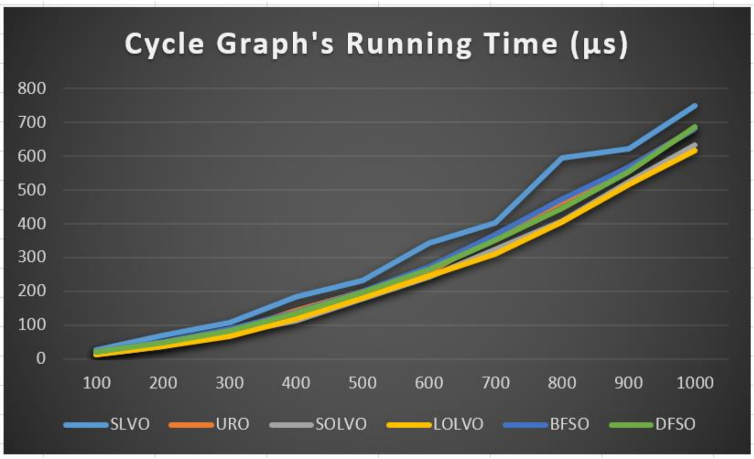

Figure 33: Running Time for a Cycle Graph

10.3: Random Graph

For the random graph, since they will also follow the order we found in complete graph and cycle graph, this part we will instead find the performance difference for each algorithm in different types of randomness distribution.

10.3.1: SLVO with Three Types of Distribution

From the result below we can see that the running time for different types of random graph are close; the Random Graph with Skewed Distribution may have a little bit bigger running time. It makes sense since for SLVO algorithm, both finding an order and coloring them do not have a connection with the vertex. The Random Graph with Skewed Distribution may take a longer running time because the average number of degrees for lower vertices is slightly bigger than the one for bigger vertices. (Check PartA 7.2) But since that difference is smaller, its effect on the running time is also small.

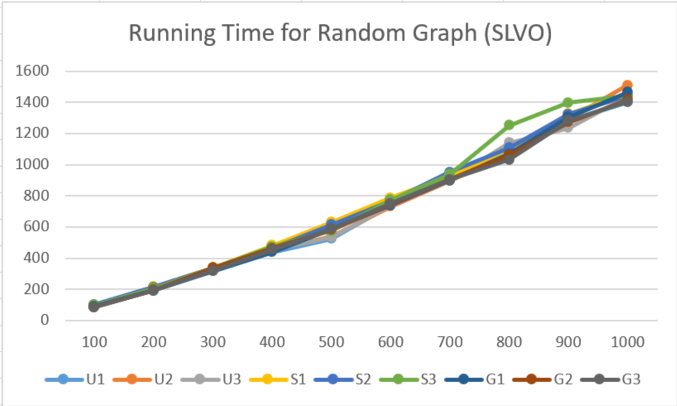

Figure 34: Running Time for Random Graph (SLVO)

10.3.2: URO with Three Types of Distribution

In the case of URO, the difference in running time is so small that which could be ignored. That's because the URO algorithm does not care about vertex; all it does it randomly pick a vertex and color it.

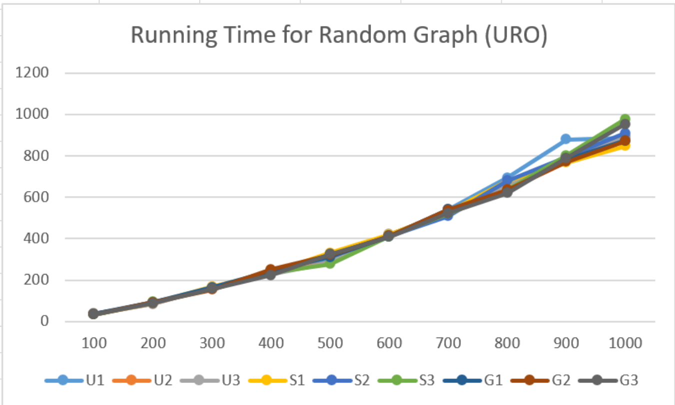

Figure 35: Running Time for Random Graph (SLVO)

10.3.3: SOLVO&LOLVO with Three Types of Distribution

Both results from SOLVO and LOLVO are like SLVO, because they all use a Degree Table. For Random Graph with Skewed Distribution, Smaller Vertex may have a higher number of degrees, and finding color for that vertex may takes more time, so that is why the Random Graph with Skewed distribution may take a bit more time than the other two.

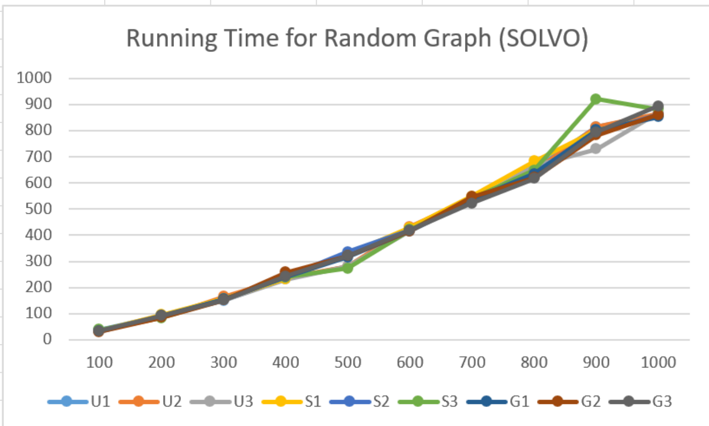

Figure 36: Running Time for Random Graph (SOLVO)

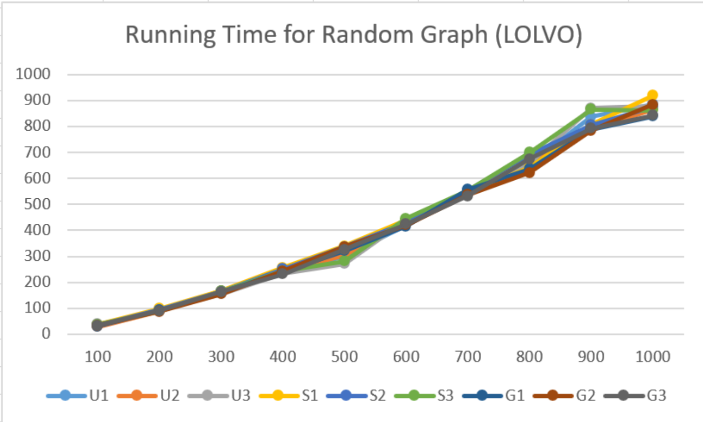

Figure 37: Running Time for Random Graph (LOLVO)

10.3.4: BFSO&DFSO with Three Types of Distribution

The result from BFSO and DFSO algorithms are different. That is because for both algorithms, they will start at 0, and when every time the queue/stack, it will push the smallest uncolored vertex to the container. As a result, if the smaller vertex has more degrees, both visiting and coloring will take more time. That result is reflected in the graphs below. The running time for those two algorithms is not consistent even the number of vertices and edges are the same. Sometimes it takes a huge more time if the small vertices has more degrees (like 800-S3 in BFSO and 700-S1 in DFSO), while sometimes takes smaller time if vice versa (like 500-U2 in BFSO).

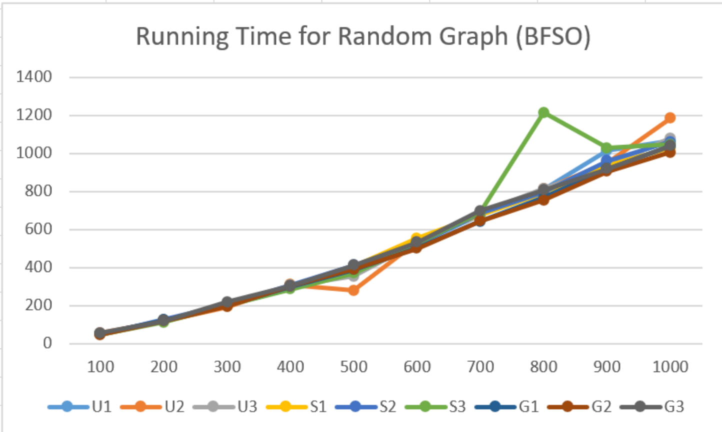

Figure 38: Running Time for Random Graph (BFSO)

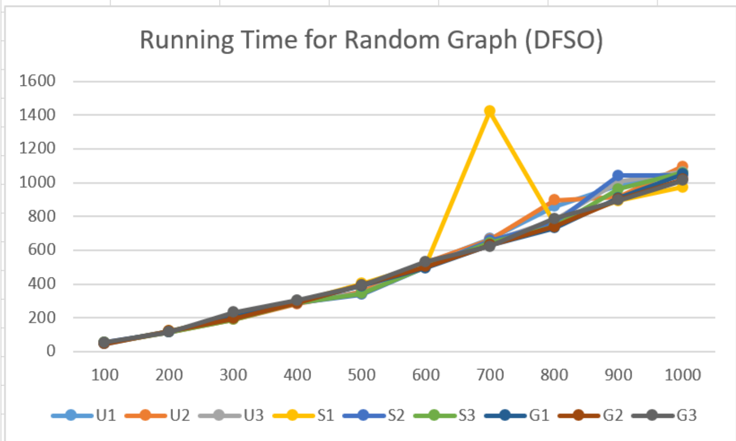

Figure 39: Running Time for Random Graph (BFSO)

11: Number of colors analysis

In this section, I modify the source code for each algorithm so that it will print the number of colors it used out. After that, I will collect all the data into a table and identify their accuracy. Here I just use 100 vertices.

nvertices	100					
	SLVO	URO	SOLVO	LOLVO	BFSO	DFSO
Complete	100	100	100	100	100	100
Cycle	2	2	2	2	2	2
Random1	10	11	10	12	11	11
Random2	11	11	11	13	11	11
Random3	10	11	10	12	10	11
RandomS1	12	16	12	19	13	15
RandomS2	15	17	15	20	16	19
RandomS3	12	15	13	20	14	17
RandomG1	10	10	10	12	10	11
RandomG2	10	11	10	11	11	10
RandomG3	9	11	11	12	11	11

Figure 40: Number of colors used for a graph with 100 vertices

	SLVO	URO	SOLVO	LOLVO	BFSO	DFSO	
Complete	100	Y	Y	Y	Y	Y	5(5)
Cycle	2	Y	Y	Y	Y	Y	5(5)
Random1	10	N	Y	N	N	N	1(5)
Random2	11	Y	Y	N	Y	Y	4(5)
Random3	10	N	Y	N	Y	Y	3(5)
RandomS1	12	N	Y	N	N	N	1(5)
RandomS2	15	N	Y	N	N	N	1(5)
RandomS3	12	N	N	N	N	N	0(5)
RandomG1	10	Y	Y	N	Y	N	3(5)
RandomG2	10	N	Y	N	N	Y	2(5)
RandomG3	9	N	N	N	N	N	0(5)
		4(11)	9(11)	2(11)	5(11)	5(11)	

Figure 41: Accuracy of algorithm to a graph with 100 vertices

When I modify the table to the Figure 41, it is clearer to find out that other than SLVO who guarantee to find the minimum number of colors used. SOLVO is the best. It reaches an accuracy which is around 81%. URO, BFSO, and DFSO are close in accuracy, which are all around 36%-63%. LOLVO is the worst one with an accuracy only 18%. It makes sense based and proved my analysis at section 5.3. Among different types of graphs, all algorithms find the minimum number of colors used for Complete and Cycle graph. Random Graph with skewed distribution has the worst result overall, which due to the number of degrees for small vertices.

12: Conclusion

From the analysis all above, we know that all the algorithms selected are linear Θ(V+E). **Smallest Last Vertex Ordering (SLVO)** has the largest running time; its time complexity is about Θ(2V+3E). That is because this algorithm needs to find the order first before doing the coloring. Finding the order is Θ(V+E) and finding the coloring is Θ(V+2E). Nevertheless, this algorithm will guarantee to find the minimum number of colors. **Smallest Original Degree Last Vertex Ordering (SOLVO)** is also a great algorithm; it has a good accuracy compared to the rest. Moreover, SOLVO and **Largest Original Degree Last Vertex Ordering (LOLVO)** are the two fastest algorithms between the six, which is Θ(V+2E) or Θ(V+E). However, SOLVO and LOLVO do not guarantee to find the minimum number of colors, and LOLVO has a terrible accuracy in finding that. Uniform Random Ordering (URO) is a coloring algorithm which will randomly choose a vertex and color it. Its running time is close to SOLVO and LOLVO, it may be slightly larger because it takes time to generate a random number and identify the unselected vertices, but that difference is tiny. **Breath-First-Search Ordering (BFSO)** and **Depth-First-Search Ordering (DFSO)** are algorithms that will use a search algorithm for looping through the graph. They also have a linear time complexity Θ(V+E), but they are slower than SOLVO, LOLVO, and URO because both BFS and DFS take O(V+E) time to visit all the nodes, while the other three are O(V). However, for random graphs, especially the ones that generated with skewed distribution, BFSO and DFSO's running time are not consistent compared to the others, sometimes their running time maybe higher/lower than expected, depending on the numbers of degrees for small vertices.

Between the three types of random generator, the one with Skewed distribution often takes more running time for coloring, and that may be because average number of degrees, especially for small vertices, are larger than the other two. But overall, their differences are small.