Graph Coloring Project Report

Arush Mohan Gupta

Southern Methodist University

CS 7350 – 701C

Graph Coloring Project Report

The goal for this project is to analyze implementations of vertex ordering in a graph to solve the graph coloring problem. The project is divided into two parts: the first part looks at analyzing algorithms to generate graphs using adjacency lists, and the second part dives into comparing the runtimes and coloring efficacy of six different vertex ordering algorithms to color the graphs generated using part one. The implementation of all code being analyzed uses the basic data structures provided by the language to lower any latent overhead. All code blocks displayed in the document have been edited to show only the relevant functionality, the code is available in the repository listed in the appendix.

Computing Environment

For this project I’m working on a 2021 M1 Pro MacBook which has the following specifications:



In addition, for this project I’ve implemented my solution in Java (17.0.6 2023-01-17 LTS for 64-bit architecture) and have used Visual Studio Code (version 1.77.3) as my IDE. Furthermore, the M1 Pro Chip is known to be very good at multi-tasking so, I recorded ten timings for each algorithm and graphed the lowest runtimes to ensure that the best execution of the code was used and any time away from processing the code could be minimized.

Conflict Graph Generation

Graph generation was done by utilizing the concept of adjacency lists. A graph is initialized by a size, where an array of vertices is initialized with the given size and for every vertex a Boolean array of neighbors is initialized with the same size. When one of the five graph generation functions is called, edges between vertices are added by setting the neighbor to True. The following code block shows how this achieved:

public class Graph {

class Vertex {

*boolean*[] neighbors;

public *void* addNeighbor(*int* *neighborId*) {

*this*.neighbors[*neighborId*] = true;

*this*.degree++;

}

}

public *void* addEdge(*int* *src*, *int* *dest*) {

Vertex srcVertex = vertices[*src*];

Vertex destVertex = vertices[*dest*];

srcVertex.addNeighbor(*dest*);

destVertex.addNeighbor(*src*);

}

}

The addEdge() and addNeighbor() functions work in constant time, as the updates are made to elements in the existing array. First an edge is added from source to destination, then another is added from destination to source. The addition of the edges twice ensures that when either vertex is visited, we know that an edge exists between the two vertices since each contains a different neighbor list. Also using a Boolean array, makes checking the existence of an edge between two vertices constant time as shown in the code below:

public *boolean* hasEdge(*int* *src*, *int* *dest*) {

return vertices[*src*].neighbors[*dest*] && vertices[*dest*].neighbors[*src*];

}

For the following conflict graphs, a graph was initialized and then the graph generation functions were called to add vertices and edges to the graph. Each graph has 7 properties: graph size, max color, max degree , terminal clique size (only used for Smallest Last Vertex Ordering), and two graph-sized arrays of vertices and vertex ordering. Each vertex has five properties: id (starting at 0), degree, color (starting at 0), Boolean array of neighbors and two pointers to previous and next vertex. The Graph and Vertex classes were setup this way to enable data capture for all algorithms that will be implemented in the project. For runtime analysis five timing trials have been recorded for each graph generation algorithm, and the min. time column records the lowest of the five trials to be used for line graph to show the relation between time and vertices/edges (input).

1. ***Complete Graph****: A graph is complete if every vertex has an edge connecting to every other vertex. So, a graph with vertices will have edges. The following code displays the implementation:*

public *void* createCompleteGraph() {

for (*int* i = 0; i < size; i++) {

for (*int* j = i + 1; j < size; j++) {

addEdge(i, j);

}

}

}

From the code we can assume that the generation of a complete graph should be since for every vertex we are iterating over all other vertices to add an edge. To test this hypothesis, I have charted and graphed the runtime of the code, , by generating graphs containing 1,000 to 10,000 vertices that are increasing by 1,000 at each step. The table below shows the data captured:

*Table 1. Complete Graph*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Vertices | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Trial 5 | Min. Time (ms) |
| 1000 | 3 | 1 | 1 | 1 | 1 | 1 |
| 2000 | 5 | 5 | 5 | 6 | 5 | 5 |
| 3000 | 12 | 12 | 12 | 12 | 12 | 12 |
| 4000 | 24 | 23 | 24 | 23 | 24 | 23 |
| 5000 | 37 | 37 | 36 | 37 | 37 | 36 |
| 6000 | 53 | 52 | 53 | 52 | 53 | 52 |
| 7000 | 72 | 72 | 72 | 72 | 72 | 72 |
| 8000 | 110 | 111 | 110 | 110 | 111 | 110 |
| 9000 | 149 | 149 | 150 | 151 | 158 | 149 |
| 10000 | 230 | 226 | 224 | 222 | 229 | 222 |

*Chart, line chart

Description automatically generated*

From Table 1, we can see that our hypothesis holds true, since we can see that when the number of vertices is doubled the amount of time taken to generate the graph increases by a factor of 4. For instance, the increase in vertices from 2000 to 4000 the time increases from 5 ms to 23 ms (increase factor of 4.6), similarly when the number of vertices increase from 3000 to 6000 vertices the time increases from 12 ms to 52 ms (increase factor of 4.3). The reason why it isn’t exactly 4 is because there are other constant operations occurring which can add up in real-world implementation.

1. ***Cycle Graph****: A graph is a cycle if you can start at a vertex and iterate through all vertices of the graph to end up at the starting vertex. So, a cycle graph with vertices will have edges. The following code shows the implementation:*

public *void* createCycleGraph() {

for (*int* i = 0; i < size; i++) {

addEdge(i, (i + 1) % size);

}

}

The is to ensure that every vertex is connected to the next and that the last vertex is connected to the first. From the code we can assume that the generation of a complete graph should be since for every vertex we are adding an edge to the next vertex. To test this hypothesis, I have charted and graphed the runtime of the code, by generating graphs containing 500 to 10,000 vertices that are increasing by 500 at each step. The table below shows the data captured:

*Table 2. Cycle Graph*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Vertices | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Trial 5 | Min. Time (ms) |
| 1000 | 80 | 100 | 89 | 72 | 92 | 72 |
| 2000 | 288 | 160 | 680 | 172 | 157 | 157 |
| 3000 | 178 | 199 | 181 | 286 | 184 | 178 |
| 4000 | 4575 | 3248 | 772 | 936 | 236 | 236 |
| 5000 | 387 | 289 | 320 | 5923 | 5546 | 289 |
| 6000 | 1800 | 2805 | 8811 | 364 | 415 | 364 |
| 7000 | 10391 | 10353 | 412 | 424 | 548 | 412 |
| 8000 | 480 | 513 | 533 | 511 | 521 | 480 |
| 9000 | 1414 | 715 | 548 | 530 | 10287 | 530 |
| 10000 | 588 | 1682 | 4575 | 737 | 659 | 588 |

Chart, line chart

Description automatically generated

From Table 2, we can see that our hypothesis holds true, since we can see that when the number of vertices is increased the amount of time taken to generate the graph increases by the same factor. For instance, the increase in vertices from 1500 to 3000 the time increases from 85 ms to 178 ms (increase factor of 2.09), similarly when the number of vertices increase from 5000 to 10000 vertices the time increases from 289 ms to 588 ms (increase factor of 2.03).

1. ***Graph with Uniformly Distributed Edges****: A graph is uniform if the edges are evenly distributed throughout the graph. So, every vertex is equally likely to be chosen for a conflict. The following code shows the implementation:*

public *void* createUniformDistributionGraph(*int* *E*) {

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

if (*E* > maxEdges) {

System.out.println("Too many edges");

return;

}

for (*int* i = 0; i < *E*; i++) {

*int* v1 = Utility.getRandomNumber(0, size - 1);

*int* v2 = Utility.getRandomNumber(0, size - 1);

if (v1 == v2 || v1 == size || v2 == size || hasEdge(v1, v2)) {

i--;

continue;

}

addEdge(v1, v2);

}

}

public static *int* getRandomNumber(*int* *min*, *int* *max*) {

*int* randomizer = *min* + (*int*)(StrictMath.random() \* ((*max* – *min*) + 1));

return randomizer;

}

The implementation first checks to make sure that number of edges requested aren’t more than the amount possible in a complete graph, and then iterates over the given number of edges to randomly select two vertices to add an edge if the two vertices are different and don’t already have an edge between them. The random vertex is obtained by getting a random number using Java’s built-in randomizer that runs in constant time and then scaled to the graph size. While we can expect the code to run in , the worst case for this code is because we are looking for new vertices to add an edge to if the same vertex or already connected vertices are chosen. To test this, I gathered data using by setting the graph size to 10,000 vertices and increased number of edges from 500 by doubling each time till 128,000. Table 3 and its corresponding graph show the data captured for this test:

*Table 3. Uniform Distribution (Edges vs Time)*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Edges | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Trial 5 | Min. Time () |
| 500 | 9 | 439 | 758 | 797 | 868 | 255 |
| 1000 | 560 | 511 | 526 | 521 | 450 | 450 |
| 2000 | 966 | 973 | 3433 | 1679 | 1079 | 966 |
| 4000 | 1404 | 1107 | 1907 | 1948 | 2180 | 1107 |
| 8000 | 1739 | 1852 | 1822 | 1936 | 1996 | 1739 |
| 16000 | 2625 | 2899 | 2956 | 3007 | 3092 | 2625 |
| 32000 | 4743 | 4784 | 4898 | 5027 | 5053 | 4743 |
| 64000 | 8938 | 9159 | 9355 | 9340 | 10910 | 8938 |
| 128000 | 19632 | 23642 | 25423 | 24880 | 20896 | 19632 |

Chart, line chart

Description automatically generated

From both the graph and table we can tell that the hypothesis was correct. From the data in Table 3, we can see that every time the number of edges is increased by a factor of two the amount of time taken to generate the graph also doubles. For instance, as number of edges is doubled from 64,000 to 128,000, the time taken increases from 8938 microseconds to 19632 microseconds (factor of 2.19). Then we can see from the spikes in data from Table 3 that sometimes the random generator gets unlucky with vertex selection and must repeatedly look for an appropriate pair. These instances show that in the worst-case scenario the graph generation could run indefinitely as it is trying to find distinct vertices that don’t already have an edge.

Chart, histogram

Description automatically generatedChart

Description automatically generatedFurthermore, an expected distribution of edges means that these vertices will have similar degree. The following graph visualizes this distribution both as a line graph, to illustrate the degree value for each vertex, and as a histogram, to show distribution of certain degree values.

1. ***Graph with Skewed Distribution****: A graph with a skewed distribution in this project means that the vertices with lower ID values have a higher chance to be picked for collision. The skew for this distribution is achieved by realizing that subtracting the square root of a number between 0 and 1 from 1 is a decreasing sequence which when scaled to the graph’s size gives us the desired skew.*

public *void* createSkewedDistributionGraph(*int* *E*) {

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

if (*E* > maxEdges) {

System.out.println("Too many edges");

return;

}

for (*int* i = 0; i < *E*; i++) {

*int* v1 = Utility.getSkewedNumber(size);

*int* v2 = Utility.getSkewedNumber(size);

if (v1 == v2 || hasEdge(v1, v2)) {

i--;

continue;

}

addEdge(v1, v2);

}

}

public static *int* getCustomNumber(*int* *size*) {

*int* num = (*int*)(*size* \* StrictMath.sqrt(StrictMath.random()));

return num;

}

Like the graph generated with a uniform distribution, this graph is generated by iterating times to find two random vertices to be connected. Thus, we can expect the average runtime for this implementation to be and the worst case to be since the random number generator can still provide us with the same or already connected vertices. To test this, I gathered data using the same approach used for Uniform Distribution Graphs. The following table and graph show the data captured for these tests:

*Table 4. Skewed Distribution (Edges vs Time)*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Edges | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Trial 5 | Min. Time () |
| 500 | 1592 | 311 | 227 | 140 | 148 | 140 |
| 1000 | 666 | 931 | 262 | 259 | 741 | 259 |
| 2000 | 496 | 578 | 1278 | 698 | 740 | 496 |
| 4000 | 1793 | 2415 | 1356 | 1133 | 1092 | 1092 |
| 8000 | 8496 | 8886 | 1668 | 1954 | 1859 | 1668 |
| 16000 | 3623 | 4026 | 3261 | 3441 | 3365 | 3261 |
| 32000 | 6898 | 7097 | 6790 | 6929 | 6497 | 6497 |
| 64000 | 14711 | 12955 | 12820 | 13388 | 17237 | 12820 |
| 128000 | 27268 | 29161 | 25946 | 29833 | 31309 | 25946 |

Chart, line chart

Description automatically generated

From both the graph and the table we can tell that the hypothesis was correct. From the data in Table 4, we can see that every time the number of edges is increased by a factor of two the amount of time taken to generate the graph also doubles. For instance, as number of edges is doubled from 8000 to 16000 the time taken increases from 1668 microseconds to 3261 microseconds (factor of 1.96), and when edges are doubled from 64000 to 128000, the time taken increases from 12820 microseconds to 25946 microseconds (factor of 2.02). Then we can see from the spikes in different trials from the table that sometimes the random generator gets unlucky with vertex selection and must repeatedly look for an appropriate pair. These instances show that in the worst-case scenario the graph generation could run indefinitely as it is trying to find distinct vertices that don’t already have an edge.

Chart, line chart

Description automatically generatedFurthermore, an expected distribution of edges means that these vertices will have similar degree. The following graph visualizes this distribution both as a line graph, to illustrate the degree value for each vertex.

1. ***Graph with Custom Distribution****: A graph with a skewed distribution in this project means that the vertices with higher ID values have a higher chance to be picked for collision. This distribution is achieved by reversing the logic used for the skewed distribution.*

public *void* createCustomDistributionGraph(*int* *E*) {

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

if (*E* > maxEdges) {

System.out.println("Too many edges");

return;

}

for(*int* i = 0; i < *E*; i++) {

*int* v1 = Utility.getCustomNumber(size);

*int* v2 = Utility.getCustomNumber(size);

if (v1 == v2 || hasEdge(v1, v2)) {

i--;

continue;

}

addEdge(v1, v2);

}

}

public static *int* getCustomNumber(*int* *size*) {

*int* num = (*int*)(*size* \* StrictMath.sqrt(StrictMath.random()));

return num;

}

Like the graph generated with a uniform distribution, this graph is generated by iterating times to find two random vertices to be connected. Thus, we can expect the average runtime for this implementation to be and the worst case to be since the random number generator can still provide us with the same or already connected vertices. To test this, I gathered data using the same approach used for Uniform Distribution Graphs. The following table and graph show the data captured for these tests:

*Table 5. Custom Distribution (Edges vs Time)*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Edges | Trial 1 | Trial 2 | Trial 3 | Trial 4 | Trial 5 | Min. Time () |
| 500 | 288 | 509 | 628 | 726 | 780 | 288 |
| 1000 | 397 | 410 | 412 | 551 | 621 | 397 |
| 2000 | 559 | 628 | 633 | 635 | 642 | 559 |
| 4000 | 896 | 1053 | 1104 | 1108 | 1119 | 896 |
| 8000 | 1325 | 1537 | 1657 | 1668 | 1721 | 1325 |
| 16000 | 2333 | 2553 | 2566 | 2602 | 2635 | 2333 |
| 32000 | 4240 | 4320 | 4406 | 4417 | 4514 | 4240 |
| 64000 | 7355 | 7384 | 7435 | 7621 | 7688 | 7355 |
| 128000 | 15681 | 19294 | 20320 | 20765 | 21536 | 15681 |

Chart, line chart

Description automatically generated

From both the graph and the table we can tell that the hypothesis was correct. From the data in Table 5, we can see that every time the number of edges is increased by a factor of two the amount of time taken to generate the graph also doubles. For instance, as number of edges is doubled from 16000 to 32000 the time taken increases from 2333 microseconds to 4240 microseconds (factor of 1.82), and when edges are doubled from 64000 to 128000, the time taken increases from 7355 microseconds to 15681 microseconds (factor of 2.13). We can also see from the spikes in different trials from the table that sometimes the random generator gets unlucky with vertex selection and must repeatedly look for an appropriate pair. These instances show that in the worst-case scenario the graph generation could run indefinitely as it is trying to find distinct vertices that don’t already have an edge.

Furthermore, an expected distribution of edges means that these vertices will have similar degree. The following graph visualizes this distribution both as a line graph, to illustrate the degree value for each vertex.

Chart, line chart

Description automatically generated

Vertex Ordering

Having generated conflict graphs, now we look to tackle the graph coloring problem by doing so using six different vertex ordering algorithms to help color the graph in the most efficient way possible. Before getting into the coloring result and comparing these implementations, I will first provide a runtime analysis of the functions I have written. The algorithms developed for this project are: Smallest Last Vertex, Largest Last Vertex, Smallest Original Degree, Largest Original Degree, Random Ordering, and Depth-First Search.

1. ***Smallest Last Vertex Ordering***: This ordering uses the graph’s adjacency list and a degree list (made by grouping vertices by their degree as a doubly linked list). We iterate over the degree list and remove vertices with lowest degree from the graph and update the degree list after each deletion by moving the deleted vertex’s old neighbors to their appropriate new degree groupings. The degree of the last vertex removed now becomes the minimum by which further vertex groupings are updated. Once the ordering process is completed, the graph is empty and is colored in the order reverse to how they were removed.

// Smallest Last Vertex Ordering

public *void* SLVO() {

Vertex[] degrees = **new** Vertex[size];

*int*[] newDegrees = **new** *int*[size];

*boolean*[] deleted = **new** *boolean*[size];

for (*int* i = 0; i < vertices.length; i++) {

Vertex vertex = vertices[i];

if (degrees[vertex.degree] != null) {

degrees[vertex.degree].prev = vertex;

vertex.next = degrees[vertex.degree];

}

degrees[vertex.degree] = vertex;

newDegrees[i] = vertex.degree;

}

for(*int* i = 0; i < degrees.length; i++) {

if(degrees[i] == null) {

continue;

}

Vertex currVertex = degrees[i];

ordering[orderingCap] = currVertex;

// Updates the degree of the vertex to reflect the degree on deletion

ordering[orderingCap].degree = newDegrees[currVertex.id];

orderingCap++;

degrees[i] = currVertex.next;

if (currVertex.next != null) {

currVertex.next.prev = null;

}

currVertex.next = null;

deleted[currVertex.id] = true;

*boolean*[] neighbors = currVertex.neighbors;

for (*int* neighborId = 0; neighborId < neighbors.length; neighborId++) {

// Check if vertex with neighborId is a neighbor

if (neighbors[neighborId]) {

if(deleted[neighborId] == true) {

continue;

}

Vertex neighbor = vertices[neighborId];

if(neighbor.next != null) {

neighbor.next.prev = neighbor.prev;

}

if (neighbor.prev != null) {

neighbor.prev.next = neighbor.next;

} else {

degrees[newDegrees[neighborId]] = neighbor.next;

}

newDegrees[neighborId] -= 1;

neighbor.prev = null;

neighbor.next = degrees[newDegrees[neighborId]];

if (degrees[newDegrees[neighborId]] != null) {

degrees[newDegrees[neighborId]].prev = neighbor;

}

degrees[newDegrees[neighborId]] = neighbor;

}

}

if (i == 0) {

i--;

continue;

}

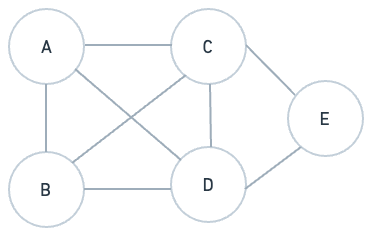
i -= 2;

}

reverseOrdering();

}

For this implementation of the Smallest Last Vertex Ordering, we are first iterating over the list of all vertices in the graph to initialize a degree list and then we are visiting each edge of a vertex once. This doesn’t become quadratic because the edges are being removed during the iteration which prevents repetition. Also, the vertices are moved up the degree list their degree-times which on average results in a runtime. Furthermore, if either the or terms are exponentially larger than the other then smaller term has very little impact on the runtime. This is because if there are a lot of vertices and very little edges in the graph, then the average degree of a vertex is lower which means on average a vertex will move very spots in the degree list. However, if the number of edges is very large then on average a vertex will move up several places up the degree list before being deleted. So, we can expect this implementation to have a linear runtime with as the ordered array is reversed which requires an iteration over vertices and updating of degree list for all neighbors of a vertex requires iterating over all edges. The following is an example of the implementation for this algorithm using a graph with 5 vertices to show its execution:



The following displays the structure of the above graph as used by the implementation:

Calendar

Description automatically generated

Here, the first line is the number of vertices in the graph, which is then followed by list of a vertex and its neighbors. The vertex and its neighbors are separated by colon, and the neighbors are space delimited. Since the graphs are built using numbers, we can see that: Vertex A is 0, Vertex B is 1, Vertex C is 2, Vertex D is 3, and Vertex E is 4. Also, given the SLVO algorithm we can expect that Vertex 4 will be removed first, followed by either Vertex E, Vertex C, Vertex D, and finally Vertex A. Furthermore, we can see that the graph can be colored using 4 colors, which is also the most optimal coloring for it since there exists a terminal clique of size 4 and the largest degree of a vertex is also 4.

The vertex ordering and coloring output from the implementation of the SLVO algorithm is as follows:

Text

Description automatically generatedText

Description automatically generated

We can see that Vertex 4 (D) was removed first, and the remaining vertices that are part of the largest subgraph were removed giving us a terminal clique of size 4, and maximum degree when removed of 3. These values provide us with an upper and lower bound on the coloring which are: lower bound is terminal clique size (4) and upper bound is one more than the largest degree when deleted (3 + 1 = 4). The results, in the screenshot with the colorings, shows that are max color is 4 and that our graph indeed could be colored using 4 colors. This supports my claim that the implementation is working as per the expected behavior described earlier.

To understand the performance of this implementation, I have graphed the time taken for ordering against the number of vertices/edges to show the runtime as follows:

1. Chart, line chart

   Description automatically generatedChart, line chart

   Description automatically generated*Complete Graph*

For a complete graph, as more vertices are added to the graph, the number of edges increases quadratically (by factor). We can glean from the two graphs above that when the number of edges far exceeds the number of vertices, the runtime can be expected to increase linearly with the edges.

1. *Cycle Graph*

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generatedFor a complete graph, the number of vertices and edges are the same. So, we can see that as the number of vertices doubles the number of edges also doubles and time taken to order the vertices increases by a factor of 4. Thus, providing support that when the vertices and edges increase by the same factor the runtime increases linearly.

1. Chart, line chart

   Description automatically generatedChart, line chart

   Description automatically generated*Uniform Distribution*

For further analysis I also graphed the implementation’s performance for a uniformly distributed graph where I first increased the number of vertices while keeping the density of the edges in the graph at 50% and then initialized a graph with 10,000 vertices and increased edges by doubling from 500 to nearly 33 million. In both scenarios it is evident that as the number of edges and vertices increase size the time taken to order the vertices also increases linearly in the order . From the second analysis, using edge increments, I saw that the time taken to order the vertices remained almost constant from 500 to 256,000 edges. From this one of the conclusions, we can draw is that for sparse graphs the ordering of vertices is determined more by the number of vertices in the graph and that for very dense graphs do the additional edges play a role in increasing the runtime.

In addition, I have also gathered the data on the relationship between vertex’s degree when deleted and the order in which a vertex is colored. I expect that as the order ID increases the degree of the vertex also increases and that there might be a drop off in the degree of the vertex towards the end since the vertices removed at the very beginning tend to have degrees on delete lower than the maximum degree after deletion. The following graph shows this relation:

Chart, line chart

Description automatically generated

1. ***Largest Last Vertex Ordering***: This ordering colors the vertices in the order reverse to the one generated by Smallest Last Vertex Ordering algorithm. This is because the terminal clique will get colored at the end here.

// Largest Last Vertex Ordering

public *void* LLVO() {

SLVO();

reverseOrdering();

}

Since the implementation for this ordering uses Smallest Last Vertex Ordering (SLVO) to iterate through the graph and then reverse the ordering, I expect it to have the same runtime as SLVO. To test my hypothesis, I graphed my results for this ordering as follows:

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

From the data we can see that for a complete graph as the number of vertices is doubled the time to order the vertices increases by more than the double due to the quadrupling of the edges. The implementation performs with an runtime because the removal of edges during ordering ensures that we go over all edges exactly once and that all vertices are visited to ensure deletion from graph to complete the ordering.

1. ***Smallest Original Degree Last Ordering***: This ordering creates a degree list like the one implemented for Smallest Last Vertex Ordering and then iterates over the degree list in ascending order to add vertices to an ordering list. For this ordering we should expect the implementation to run in .

// Smallest Original Degree Last Ordering

public *void* SODO() {

Vertex[] degrees = **new** Vertex[size];

for (*int* i = 0; i < vertices.length; i++) {

Vertex vertex = vertices[i];

if (degrees[vertex.degree] != null) {

degrees[vertex.degree].prev = vertex;

vertex.next = degrees[vertex.degree];

}

degrees[vertex.degree] = vertex;

}

for(*int* i = 0; i < degrees.length; i++) {

Vertex curr = degrees[i];

while (curr != null) {

ordering[orderingCap] = curr;

orderingCap++;

curr = curr.next;

}

}

}

Chart, line chart, scatter chart

Description automatically generatedWe can see that the first loop creates the degree list and adds vertices with the same degree to a linked list. Then the second loop iterates over each linked list attached to the degree list and adds the vertices to an ordering list. To understand the performance of this implementation I have graphed its runtime as follows:

The above graph has been generated by creating uniform distribution graphs with 10000 vertices and edges ranging from 500 to 33 million. From the graph above we can see that as the number of edges is doubled the runtime stays consistent. This is because to order the vertices in the graph the implementation iterates over all vertices and then iterates over the linked list of vertices with the same degree. Since every vertex of a complete graph has the same degree, the iteration over degree list is just another iteration over the vertices.

1. ***Largest Original Degree Last Ordering***: This ordering colors the vertices in the order reverse to the one generated by Smallest Original Degree Last Ordering (SODO) algorithm. The vertex with the largest degree will get colored first.

// Largest Original Degree Last Ordering

public *void* LODO() {

SODO();

reverseOrdering();

}

Since the LODO implementation uses the SODO algorithm to order the vertices and then an additional linear operation of reverse the resulting ordering list, we can expect it to perform with the same as Smallest Original Degree Last Ordering.

1. ***Random Ordering***: This ordering colors the vertices in random order. As the vertices are to be picked randomly, the number of edges has no bearing on the runtime of this implementation. To perform the random picking of vertices, I first added all vertices to the ordering list and then shuffled the list using a Fisher-Yates shuffle. The shuffle is performed in O(n) since it is swapping elements in an array by picking random index, where swapping occurs for each vertex and the random index generator works in constant time.

// Random Vertex Ordering

public *void* RNVO() {

for(Vertex v: vertices) {

ordering[orderingCap] = v;

orderingCap++;

}

shuffleOrdering();

}

// Implementing the Fisher-Yates Shuffle

// Source: https://stackoverflow.com/questions/1519736/random-shuffling-of-an-array

public *void* shuffleOrdering() {

*int* n = ordering.length;

for(*int* i = n - 1; i > 0; i--) {

*int* j = Utility.getShuffledIndex(i);

swapOrderingElem(i, j);

}

}

1. ***Depth First Search Based Ordering***: The final ordering I have implemented for this project utilizes the depth first search algorithm to order the vertices in the graph. To perform this ordering, we iterate through all vertices and check if any of their neighbors have been visited yet, the first neighbor that has not yet been visited is then visited and the process is repeated until all vertices have been visited and added to the ordering list.

// Depth-First Search Ordering

public *void* DFSO() {

*boolean*[] visited = **new** *boolean*[size];

for (Vertex vertex : vertices) {

if (!visited[vertex.id]) {

dfs(vertex, visited);

}

}

shuffleOrdering();

}

public *void* dfs(Vertex *vertex*, *boolean*[] *visited*) {

*visited*[*vertex*.id] = true;

*boolean*[] neighbors = *vertex*.neighbors;

for (*int* neighborId = 0; neighborId < neighbors.length; neighborId++) {

if (neighbors[neighborId]) {

Vertex neighbor = vertices[neighborId];

if (!*visited*[neighborId]) {

dfs(neighbor, *visited*);

}

}

}

ordering[orderingCap] = *vertex*;

orderingCap++;

}

For the runtime of this ordering, I expect it to have a linear relation with vertices and a constant relation with edges. This is because as more vertices are added to the graph that means iterating over more vertices to mark every vertex as visited. Whereas if more edges are added to the graph nothing changes because if a vertex has been visited it won’t get visited again through another route.

Coloring

In this section of the paper, I will go over the coloring algorithm I have implemented for the project and discuss how the discussed graphs were colored using the orderings discussed in the previous section. The coloring algorithm is made up of two functions. The first function iterates over all vertices, colors them individually using the second function and keeps track of the max color used. The second function iterates over the list of a vertex’s neighbors and assigns color as necessary based on the existing coloring.

// Color the graph based on the ordering

public *void* colorGraph() {

for (Vertex vertex : ordering) {

vertex.color = colorVertex(vertex);

if (vertex.color > maxColor) {

maxColor = vertex.color;

}

}

}

// Color individual vertices

// Reduced vertex coloring to First Missing Positive Integer

// Link to the problem: https://leetcode.com/problems/first-missing-positive/

public *int* colorVertex(Vertex *vertex*) {

*int*[] colors = **new** *int*[size];

// Initialize list with current neighbor color

*boolean*[] neighbors = *vertex*.neighbors;

for (*int* i = 0; i < neighbors.length; i++) {

if (neighbors[i]) {

colors[i] = vertices[i].color;

}

}

for (*int* i = 0; i < colors.length; i++) {

*int* val = colors[i];

while(0 < val && val <= colors.length && colors[val - 1] != val) {

*int* temp = colors[i];

colors[i] = colors[val - 1];

colors[val - 1] = temp;

val = colors[i];

}

}

// Find the first gap in the sequence of colors

for (*int* i = 0; i < colors.length; i++) {

if (colors[i] != i + 1) {

return i + 1;

}

}

// No gap was found, so get next color

return colors.length + 1;

}

The colorVertex() function solves the coloring problem by instantiating an array of colors that is initialized with the existing colors of the vertex’s neighbors. Then this colors array is sorted using a cycle sort so that adjacent values in the array have the least possible gap between them. Finally, this colors array is iterated over once more to see if the colors array is now a contiguous sequence. If a gap is found between two colors, then this gap value is returned or the next available color is returned. This coloring algorithm works with runtime because for every vertex all neighbors are visited to update a vertex’s color.

Vertex Ordering Capabilities

Now that we have looked at all our graph-generation, vertex ordering and coloring algorithms, we can compare how the different graphs get colored using different vertex orderings.

1. Chart, line chart

   Description automatically generated*Complete Graphs*: For complete graphs we can expect that the colorings should the same as the number of vertices since all of them are connected.

As we can see in the graph above that the number of colors used increases linearly with the number of vertices in a graph supporting the prediction.

1. *Cycle Graphs*: For cycle graphs we can expect that the colorings should optimally be 2 since we can color every other vertex the using same color.

Chart

Description automatically generated

All orderings except for Random and Depth First Search were able to color the graphs using 2 colors. These two orderings didn’t perform too poorly though, they were able to color the graphs using 3 colors consistently.

1. *Distribution-based Graphs*: For graphs with a uniform/skewed/custom distribution of edges the smallest last vertex, and largest original degree last orderings performed the best consistently. Whereas largest last vertex ordering performed the worst to color the graphs. The depth first search ordering performed as well as the random ordering.

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Conclusion

From the data collected and visualized in the previous section, we can see that the smallest last vertex ordering performs well in both the runtime and coloring of any graph. Alongside, largest original degree last can also perform coloring of a graph with the same efficiency as the smallest last vertex ordering. Furthermore, using random ordering to color a graph is as good and maybe better than using a depth first search to order the vertices. Smallest original degree last and largest last vertex ordering are the worst at coloring a graph.

For Complete Graphs, we can state that the number of colors to color it can be baselined using the number of vertices in the graph. All our orderings for the complete graphs provided us with the same coloring result.

For Cycle Graphs, we can state that the least efficient way to color it would require 3 colors, whereas it can be colored using 2 based on the vertex ordering used. The number of vertices and edges do not impact how a cycle graph gets colored. This is because every vertex is connected only to two other vertices so the coloring can be at max 3 if vertices aren’t selected optimally.

For graphs with different distributions, we can state that if the edges of a graph are spread throughout a graph, as shown by uniform distribution, and not concentrated to only a particular set of vertices, as shown by skewed and custom distribution, the graph will consistently require the least number of colors to be colored for the same number of edges in the graph. This observation can be extrapolated to the real-world for scheduling purposes. For instance, when building a schedule for courses to be taken by students at a university for a semester, the need for extra sections for a course can be realized upon seeing how the graph representing this relation gets colored. If it has an efficient coloring, then the graph is a lot more uniform than skewed. If the graph is skewed, then courses with highest conflicts can be provided extra sections to ease the graph and make it easier for students to enroll in them. This used to be quite a common occurrence during my undergraduate years, where students were on waitlists that were 300 students strong because all other degree requirement courses conflicted with the courses offered by the Computer Science Department.

Lastly, we can also see the advantage of using smallest last vertex ordering over all others. It executed consistently with a runtime of O(V+E) which when scaled to larger graphs with higher edge densities provides a better computation time than the other ordering algorithms. Furthermore, smallest last vertex ordering consistently provided the most efficient coloring for every graph. Smallest Original Degree Last and Largest Last Vertex orderings can be dismissed for usage since they were not efficient at coloring even if their algorithms did provide a linear runtime for all graphs. And we can also conclude that in trying to come up with a new ordering algorithm it is best to compare its efficiency and performance to random ordering since the new ordering algorithm will more often than not be equally as good if not poor than random ordering.

References

1. Fisher Yates Shuffle: <https://stackoverflow.com/questions/1519736/random-shuffling-of-an-array>
2. Coloring a vertex by finding minimum unused color: <https://leetcode.com/problems/first-missing-positive/>
3. Random Number: <https://stackoverflow.com/questions/363681/how-do-i-generate-random-integers-within-a-specific-range-in-java>

Appendix

1. Code Repository: <https://github.com/arushgupta/graph-coloring>
2. Graph Generation Data: <https://docs.google.com/spreadsheets/d/1XN3wdySzxcAWExzUJm4b2trUAHis_7zvWlgxb0oneOE/edit?usp=sharing>
3. Ordering Data: <https://docs.google.com/spreadsheets/d/1hKBAWfh_qMFkyb6VpkmPg-T63_3U-KGLTs4QG_M7xOo/edit?usp=sharing>
4. Coloring Data: <https://docs.google.com/spreadsheets/d/1uppggoWPdM-EIwMgjvgXinuScTc-l7u3u1th_BaW8es/edit?usp=sharing>