Colin Weil

CS7350 Final Project

Computer Environment

Hardware Overview:

Model Name:	MacBook Pro
Model Identifier:	MacBookPro18,3
Chip:	Apple M1 Pro
Total Number of Cores:	8 (6 performance and 2 efficiency)
Memory:	16 GB
System Firmware Version:	7459.141.1
OS Loader Version:	7459.141.1

Application

CLion 2022.2.1

Build #CL-222.3739.54, built on August 16, 2022

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.23)
project(PA2)

set(CMAKE_CXX_STANDARD 14)

add_executable(PA2 main.cpp AdjacencyList.h AdjacencyList.cpp LinkedList.h CreateGraphs.cpp CreateGraphs.h
VertexNode.cpp VertexNode.h ColorSolver.cpp ColorSolver.h Vector.h)
```

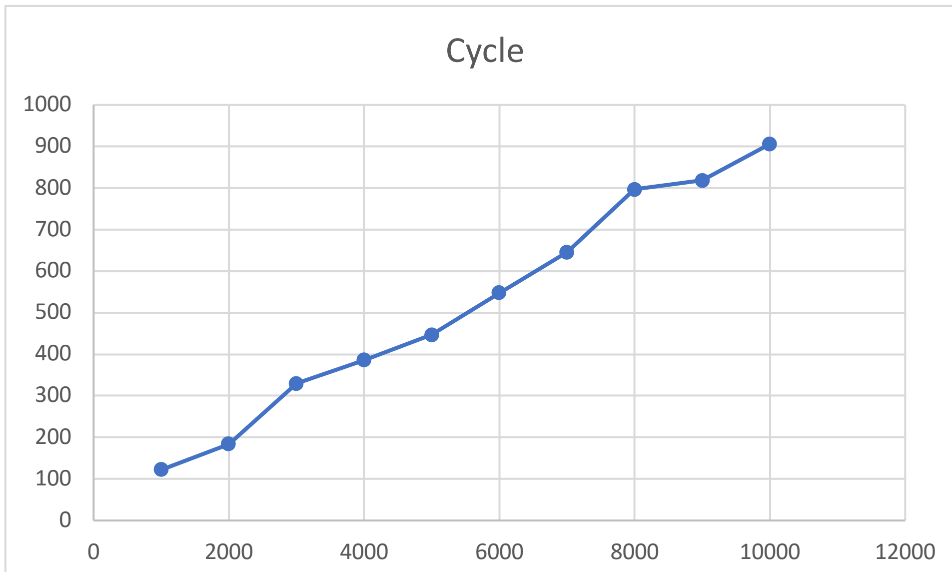
<u>Algorithms for Generating Conflict Graphs</u>

Part 1 – Non-Random Graphs Running Times:

1. Cycle

For creating the cycle graph, every vertex only has to connect to two other vertexes. I did this simply by connecting every vertex to the number before it and after it in a single for loop. I then had to connect the last vertex to the first vertex also but that is constant time. Because this process only uses one for loop to V, the asymptotic notation of creating a cycle is $\theta(V)$.

V	µs
1000	122
2000	184
3000	330
4000	386
5000	447
6000	548
7000	646
8000	797
9000	818
10000	906

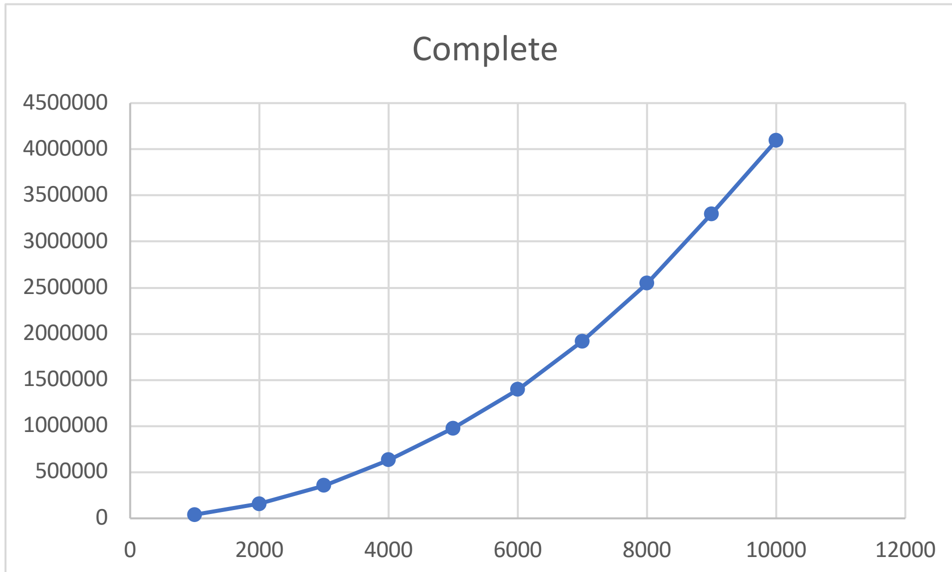


The table and graph support my analysis of $\theta(V)$for the cycle graph since as the size of n doubles from 4000 to 8000, the running time also essentially doubles from 386 to 797.

2. Complete

For creating the complete graph, every vertex connects to every other vertex. This is done in a double for-loop to V which is why creating a complete graph has the asymptotic notation of $\theta(V^2)$.

V	µs
1000	40025
2000	158838
3000	355702
4000	634072
5000	976471
6000	1395699
7000	1919460
8000	2548258
9000	3299583
10000	4092530



The table and graph support my analysis of $\theta(V^2)$ for the complete graph since as the size of n doubles from 1000 to 2000, the running time also essentially quadruples from 40,024 to 158,838 since $2^2 = 4x$.

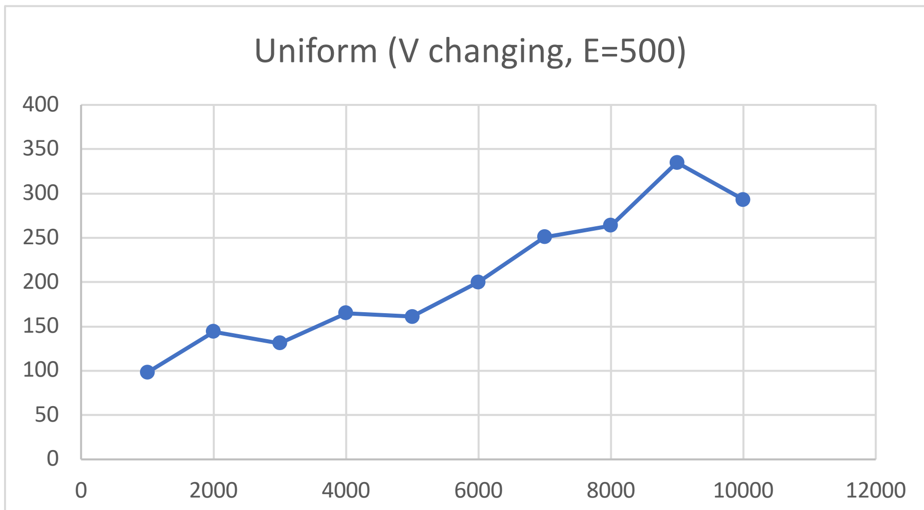
* For all random graphs after, there will be an element of randomness to the timings as a result of the way the random values are picked. They are implemented by picking a random value and checking if the edge was already added to the graph. If the edge was added already then it will keep picking new values. Because of this implementation, I saved on some memory however as the number of edges comes closer to the number in a complete graph, the program might become a lot slower which may also affect the timings.*

Part 2 – Random Graphs Running Times:

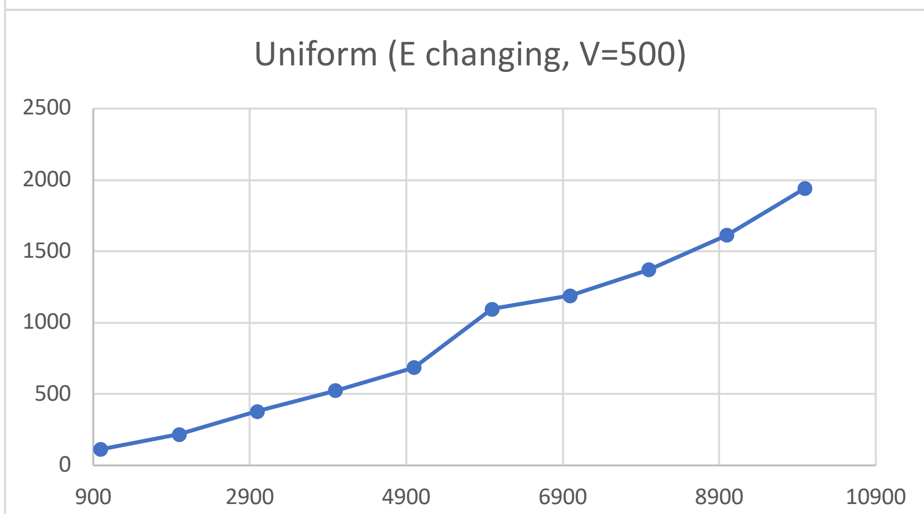
1. Random Uniform

For creating the random uniform graph, I created E number of edges by randomly choosing an x and y value that is in the range of the 1 to V. This is done in a single for-loop to E however after creating the graph, the return statements will also call the copy constructor of adjacency list which has to copy V number of array spaces which makes the asymptotic timing of creating a Random Uniform Graph $\Omega(E+V)$ and $O(\infty)$ because if it randomly never chooses the remaining possible edges, the program could go on forever.

V	μs
1000	98
2000	144
3000	131
4000	165
5000	161
6000	200
7000	251
8000	264
9000	335
10000	293



E	μs
1000	113
2000	218
3000	379
4000	524
5000	687
6000	1095
7000	1191
8000	1371
9000	1613
10000	1943

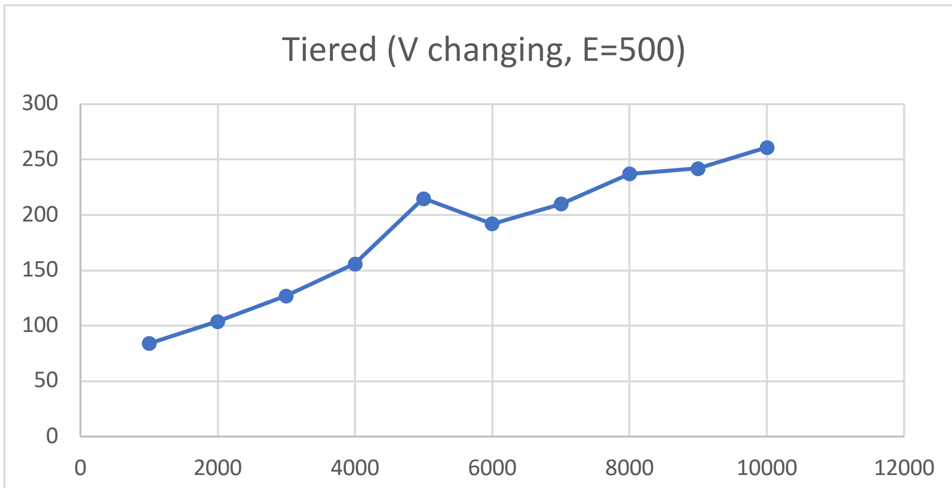


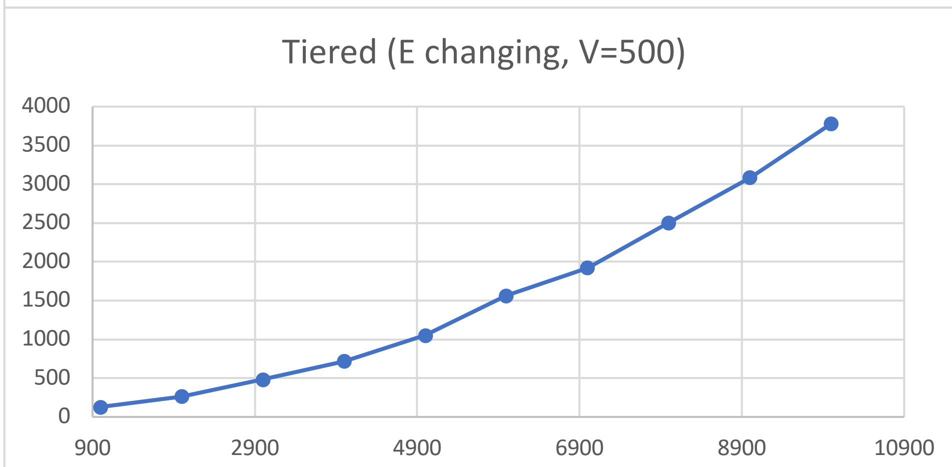
The tables and graph support my analysis of $\Omega(E+V)$ for creating a Random Uniform Graph since as the size of V and E increase, the timing increases linearly for both instances (with some bumps that are caused by the randomness and checking for not duplicating an edge).

2. Random Tiered

For creating the random uniform graph, I created E number of edges by randomly choosing an x and y value that is in the range of the 1 to V with the first 10% of vertices having a 50% chance of being chosen and the last 90% of vertices having the other 50% chance of being chosen. This is done in a single for-loop to E however after creating the graph, the return statements will also call the copy constructor of adjacency list which has to copy V number of array spaces which makes the asymptotic timing of creating a Random Tiered Graph $\Omega(E + V)$ and $O(\infty)$ because if it randomly never chooses the remaining possible edges, the program could go on forever.

V	µs
1000	84
2000	104
3000	127
4000	156
5000	215
6000	192
7000	210
8000	237
9000	242
10000	261
E	µs
1000	129
2000	263
3000	482
4000	719
5000	1053
6000	1564
7000	1924
8000	2505
9000	3085
10000	3779



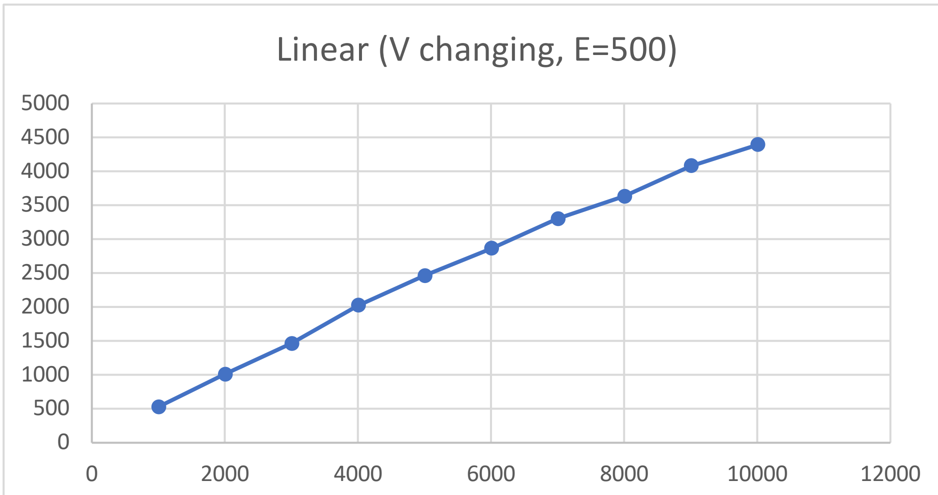


The tables and graph support my analysis of $\Omega(E + V)$ for creating a Random Uniform Graph since as the size of V and E increase, the timing increases linearly for the V changing graph. However, the graph table and graph support $O(\infty)$ because you can see as E increases in the E changing graph, the timing starts to not become linear. This is shown by the Tiered graph more than the Uniform graph because the Tiered will pick 50% of its vertices from 10% of the data making the program look for a smaller quantity of possible options from a large amount of options. This means that the program will randomly search for the possible options for a longer amount of time which is shown on the graph.

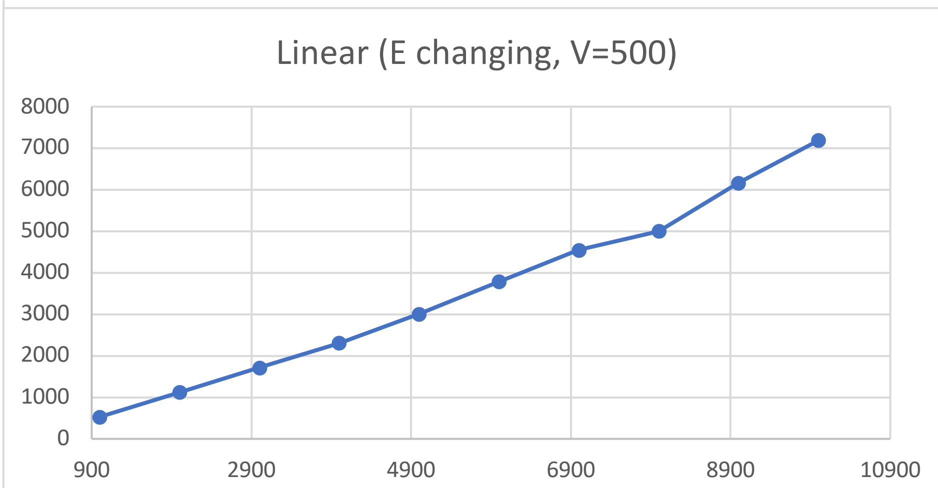
3. Linear Distribution (Own)

For creating the random linear graph, I created E number of edges by randomly choosing an x and y value that is in the range of the 1 to V with each vertex having a $\frac{Vertex\ \#}{\sum All\ Vertex\ \#'s}$ probability of being chosen. This is done in a single for-loop to E however after creating the graph, the return statements will also call the copy constructor of adjacency list which has to copy V number of array spaces which makes the asymptotic timing of creating a Random Tiered Graph $\Omega(E + V)$ and $O(\infty)$ because if it randomly never chooses the remaining possible edges, the program could go on forever.

V	µs
1000	529
2000	1007
3000	1462
4000	2025
5000	2463
6000	2861
7000	3303
8000	3629
9000	4079
10000	4390



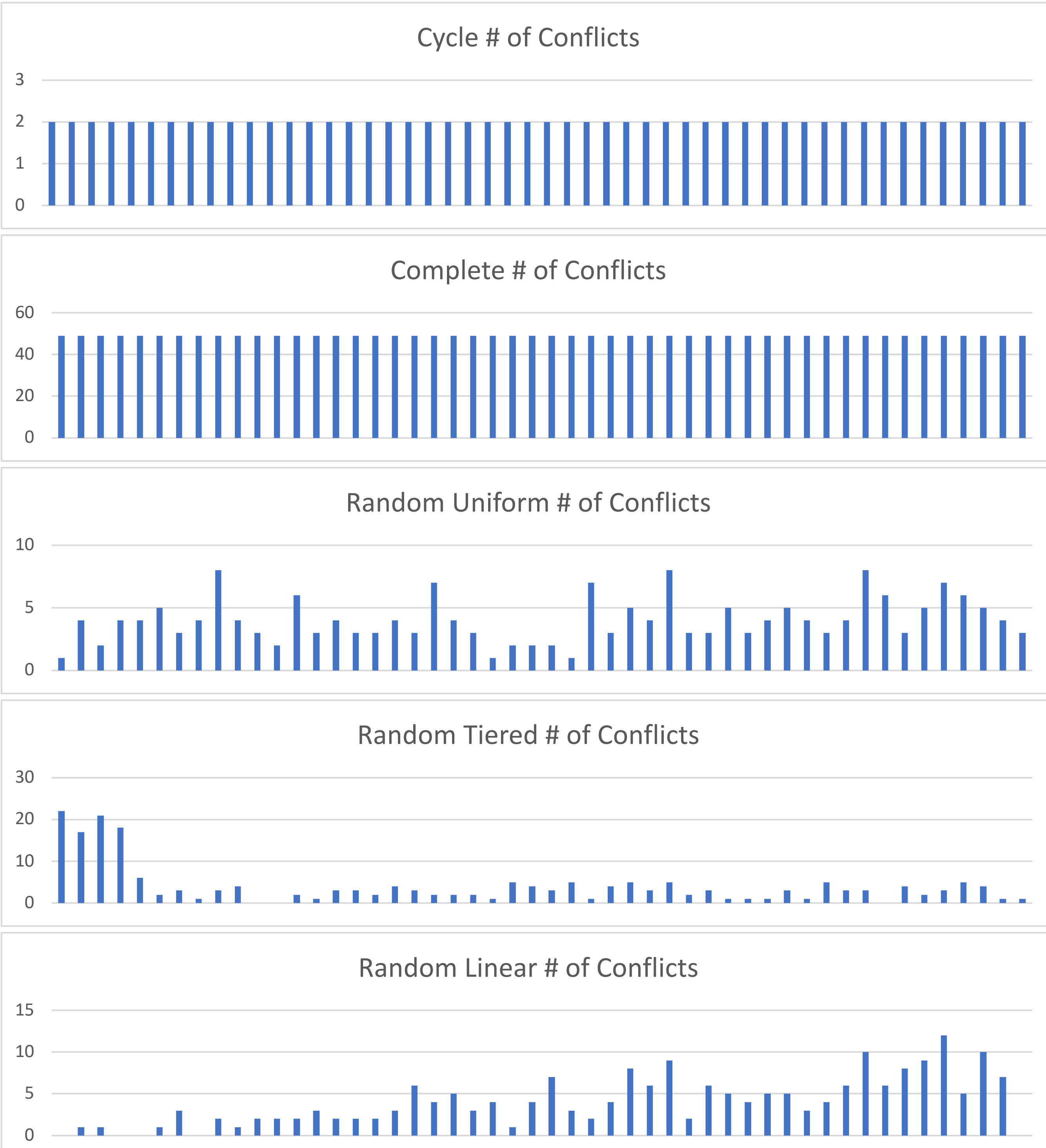
E	µs
1000	529
2000	1118
3000	1714
4000	2306
5000	3007
6000	3783
7000	4546
8000	5001
9000	6165
10000	7185



The tables and graph support my analysis of $\Omega(E + V)$ for creating a Random Uniform Graph since as the size of V and E increase, the timing increases linearly for both instances.

Part 3 – Conflicts for Each Graph:

Given the input for each graph is V = 50, E = 100 (when applicable), the conflicts are:



Vertex Ordering

Part 1 – Description of Ordering Implementations:

1. Smallest Last Vertex Ordering – This ordering starts by keeping track of the degree that every vertex has an array of lists. At each index, the list will contain pointers to all of the vertices that have that degree (ex. Index 2 will contain all vertices that have degree 2). While there are nodes left in this data structure, the function starts at the lowest degree and deletes the node. It then stores the node number in the last available index of an array that stores the ordering. After storing and deleting the vertex, the function will update all of its connections' degrees to show that it no longer connects to the deleted vertex and update it in the data structure accordingly. Then so it doesn't skip any vertices, the function will back up one index in the data structure and continue the process.

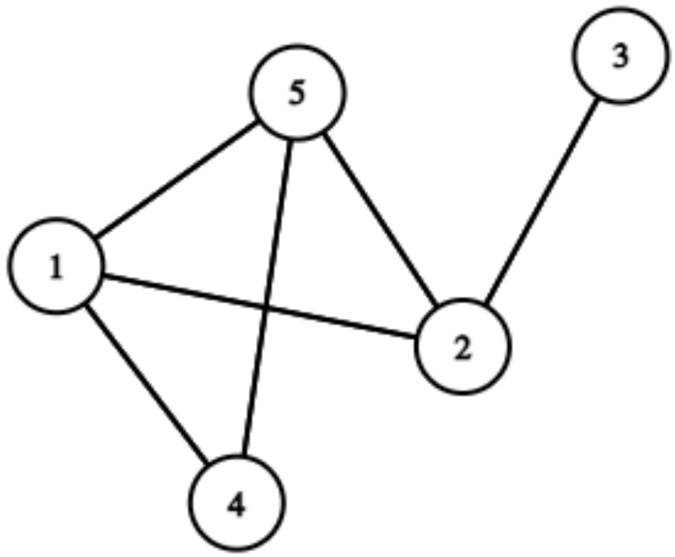
2. Smallest Original Degree Ordering – This is the same exact process as Smallest Last Vertex Ordering, however it will not update all of the vertices' connections after deleting it so that the vertices will not update in the data structure.

3. Uniform Random Ordering – This ordering simply randomizes a list of numbers. I complete this in my vector class by creating a vector of numbers in order, then deleting them in a random order and adding them to a different vector in that order.

Part 2 – Walkthrough of SLVO on Small Graph:

Given a Graph with V = 5, E = 6 that looks like:

1 -> 4 -> 2 -> 5
2 -> 5 -> 3 -> 1
3 -> 2
4 -> 1 -> 5
5 -> 2 -> 1 -> 4



SLVO will solve by:

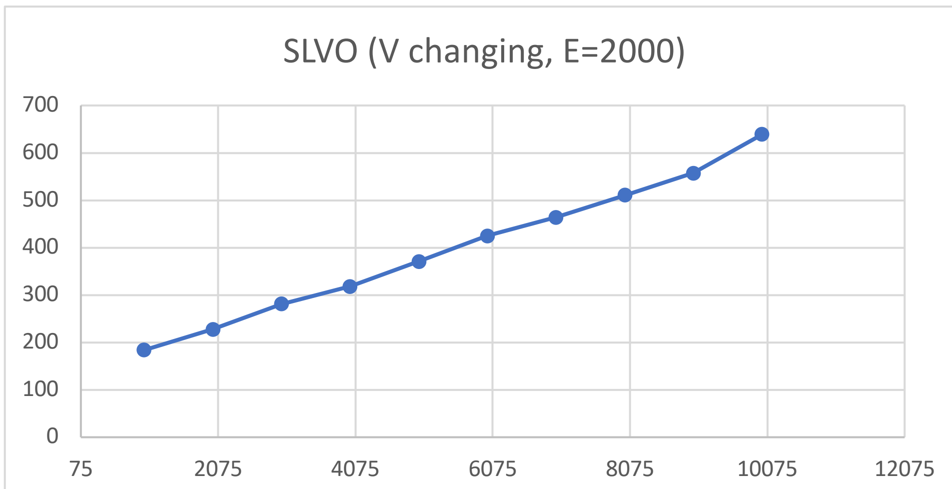
DELETED 3
DELETED 4
DELETED 2
DELETED 5
DELETED 1
SLVO Order: [1, 5, 2, 4, 3]

Part 3 – Vertex Ordering Running Times (Random Graph):

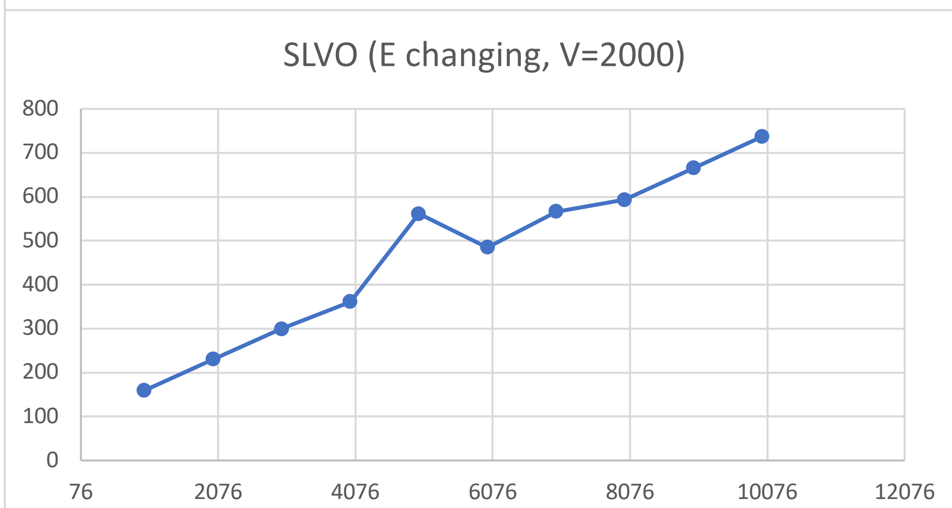
1. Smallest Last Vertex Ordering

The running of the SLVO algorithm is accomplishable in the asymptotic notation of $\theta(V + E)$. You can do this because almost all of the operations in the implemented function can be done in constant time. Adding from the doubly linked lists, deleting from the doubly linked list, and marking the vertex as deleted in the graph can all be done in constant time. The only operations that aren't will be looping through V number of vertexes for the ordering, and E number of edges to update the degree of the vertices. It will not be more than E because the function will not check the vertex in its connections if it is deleted.

V	µs
1000	184
2000	228
3000	281
4000	318
5000	371
6000	425
7000	464
8000	511
9000	557
10000	639



E	µs
1000	159
2000	230
3000	299
4000	361
5000	561
6000	485
7000	566
8000	593
9000	665
10000	737

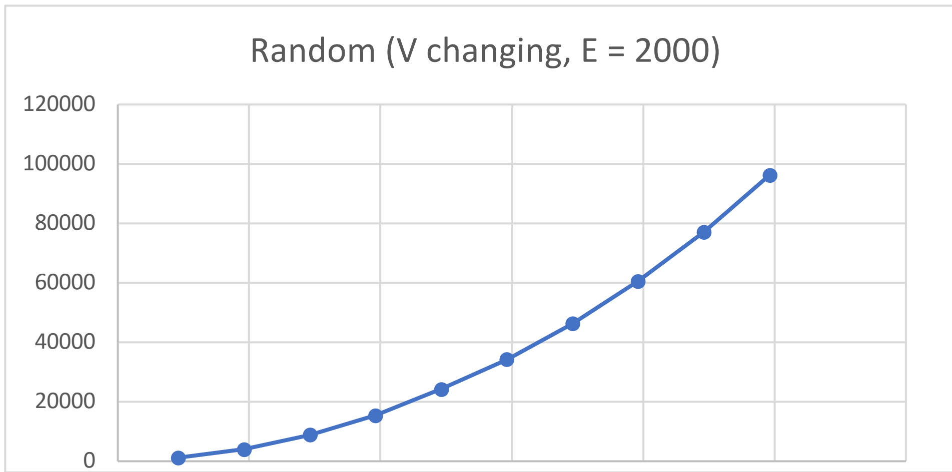


The tables and graph support my analysis of $\theta(V + E)$ for SLVO since as the size of V and E increase, the timing increases linearly for both instances.

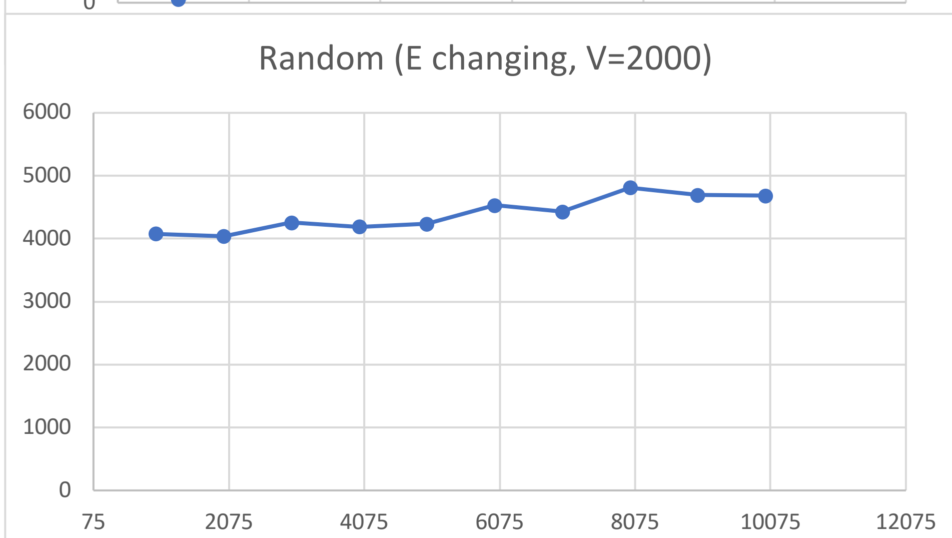
2. Uniform Random Ordering

The running of the Uniform Random Ordering is accomplishable in the asymptotic notation of $\theta(V^2)$. This is because the way that the creating random vector function is implemented. The function by looping V times through a preexisting vector that is in order and adding a randomly removed element from the vector to the ordering. Removing an element from the vector takes linear time because it has to create a new list and loop through the old one and add all the values besides the one removed. Since this linear remove is in a for loop, the notation is $\theta(V^2)$.

V	µs
1000	1206
2000	4061
3000	8877
4000	15470
5000	24200
6000	34294
7000	46315
8000	60588
9000	77176
10000	96286



E	µs
1000	4079
2000	4039
3000	4256
4000	4188
5000	4234
6000	4529
7000	4429
8000	4810
9000	4691
10000	4681



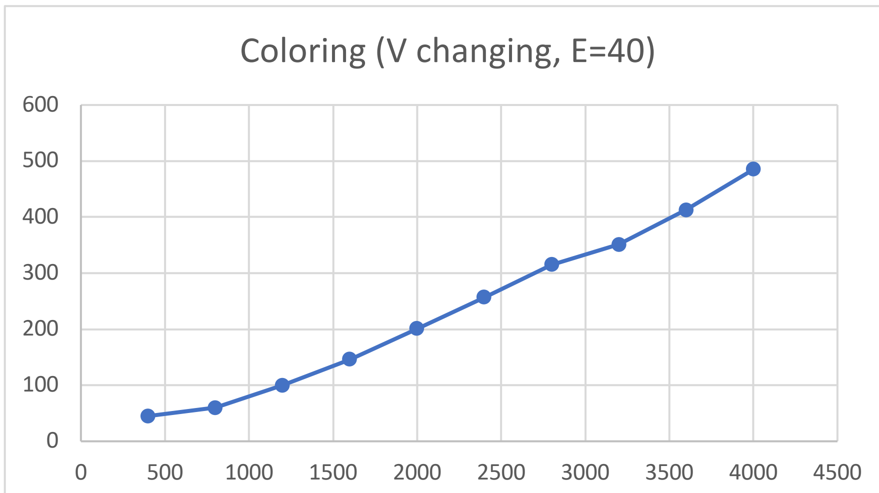
The tables and graph support my analysis of $\theta(V^2)$ for the Uniform Random Ordering since as the size of V doubles (1000 to 2000), the timing quadruples (1206 to 4061). Also the changing of E does not affect the runtime.

Coloring Algorithm

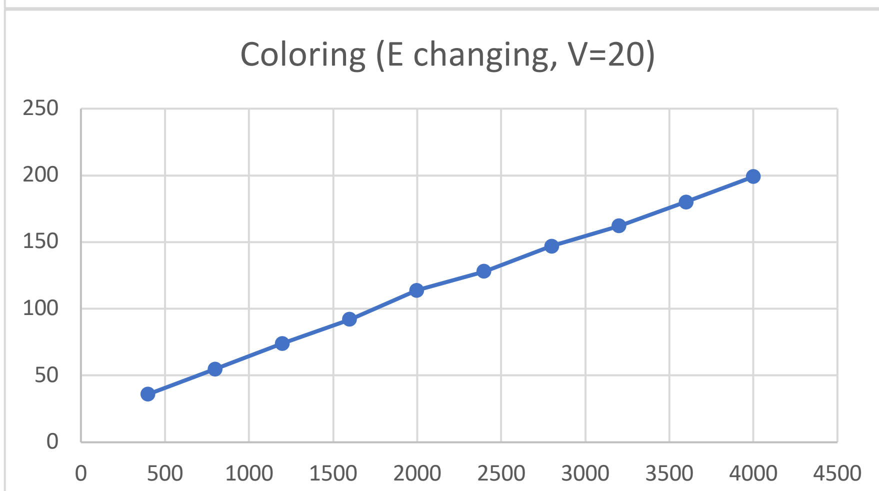
Given a random uniform graph and a random ordering, the coloring algorithm performs as follows:

The running of the Coloring is accomplishable in the asymptotic notation of $\theta(V + E)$. This is because the coloring has to loop through V number of vertexes to assign the coloring. In total, the coloring function will have to check E number of edges to make sure the colors don't conflict before assigning the color to the vertex.

V	µs
10	45
20	60
30	100
40	146
50	201
60	257
70	315
80	351
90	413
100	485



E	µs
20	36
40	55
60	74
80	92
100	114
120	128
140	147
160	162
180	180
200	199



The tables and graph support my analysis of $\theta(V + E)$ for the Coloring since as the size of V and E increase, the timing increases linearly for both instances.

<u>Vertex Ordering Capabilities</u>

Part 1 – Performance of Different Ordering on Various Graphs

Given the input for each graph is V = 10, E = 16 (when applicable), the number of colors used by each algorithm are:

1. Cycle:
   - Smallest Last Vertex Ordering – 2
   - Smallest Original Degree Ordering – 2
   - Random Uniform Ordering - 3
2. Complete:
   - Smallest Last Vertex Ordering – 10
   - Smallest Original Degree Ordering – 10
   - Random Uniform Ordering - 10
3. Random Uniform:
   - Smallest Last Vertex Ordering – 3
   - Smallest Original Degree Ordering – 4
   - Random Uniform Ordering - 3
4. Random Tiered:
   - Smallest Last Vertex Ordering – 3
   - Smallest Original Degree Ordering – 3
   - Random Uniform Ordering - 4
5. Random Linear:
   - Smallest Last Vertex Ordering – 4
   - Smallest Original Degree Ordering – 4
   - Random Uniform Ordering - 4

Part 2 – Description of Performance on Graphs and Analysis of Ordering Capabilities

Random Uniform Ordering:

Random Uniform Ordering was by far the worst ordering system. Not only was it quadratic in polynomial timing because of how it was implemented, but it also did not create the optimal solution. Although it finds the right solution for complete graphs because every vertex connects to all the other vertices therefore the coloring would be optimal in any order, I would still never use this ordering because of how slow it is compared to the other orderings.

Smallest Original Degree Ordering (SODO):

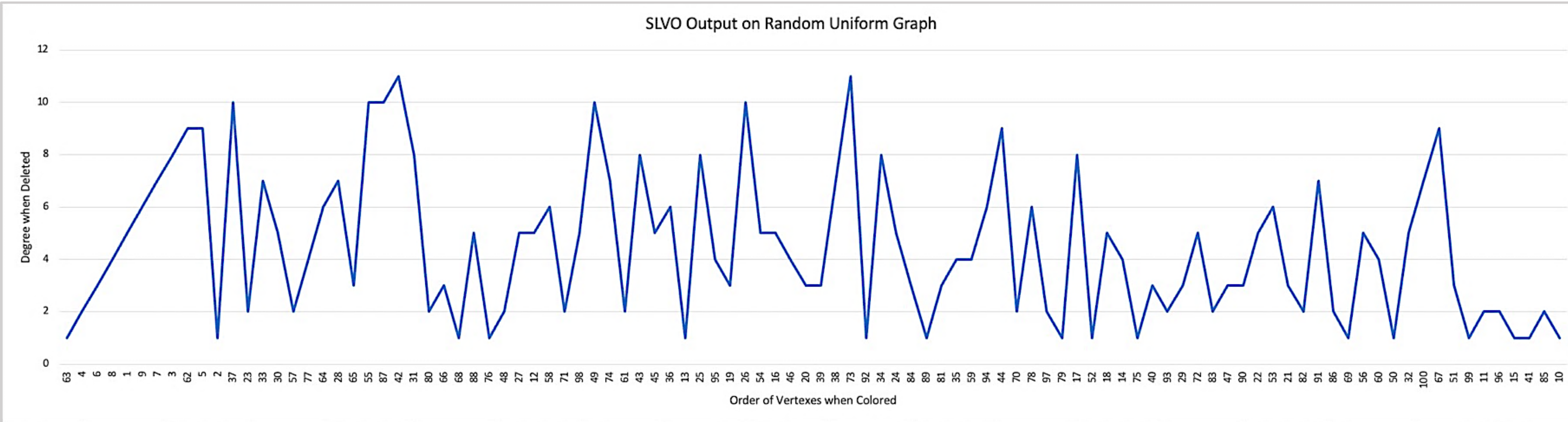
Smallest Original Degree Ordering is an improvement on Random Ordering, yet it still has some disadvantages. Much like Random Uniform Ordering, SODO will solve the complete graph correctly every time correctly, but an improvement is that SODO runs in a linear timing in relation to V. SODO also will create an ordering that is optimal for many graphs at a high rate, however it will not create the optimal solution every time, especially for the random graphs. For the way that the cycle is implemented in my program, SODO will solve it optimally however if a cycle input was not in order like the vertices were in mine, this could cause problems. For this reason, I would use SODO only on complete graphs because it is optimal and linear in relation to V for these graphs.

Smallest Last Vertex Ordering (SLVO):

Smallest Last Vertex Ordering creates an optimal ordering for all graphs that are created. It stays as linear timing because it scales with V and E individually linearly. This is all around a great algorithm because it is linear yet still creates the optimal solution for any graph possible. As a result, I could use this algorithm for any graph created and be confident that it would give me the optimal solution. The only circumstance I would not use SLVO was if I used SODO on a complete graph to save on a miniscule amount of time because SODO does increase timing as the number of E increases like SLVO does.

Part 3 – Output for SLVO

Given a Random Uniform Graph of V = 100 and E = 900:



<u>Output Besides Vertexes:</u>

Total Colors Used: 11
Average Original Degree (rounded down): 18
Maximum Degree when Deleted: 11
Size of Terminal Clique: 9

<u>Bounds of Colors</u>

The size of the terminal clique and the maximum degree when deleted of the graph bound the number of colors needed to completely color the graph.

The size of the terminal clique bounds the minimum of possible colors. This is because if there is a complete graph of size n in any part of the graph, there must be at least n number of colors to color the graph because every vertex is touching the other.

The maximum degree when deleted bounds maximum number of colors by maximum degree + 1 color. This is because if the highest degree of a vertex is of degree d, then that means that this vertex is touching d other vertices. The worst case possible would be that this vertex is part of a complete graph in which there would need d number of colors for each one of the vertices touched + 1 since it needs to account for its own color.

main.cpp

```cpp
#include "CreateGraphs.h"
#include "ColorSolver.h"
using namespace std;

int main() {
  srand (time(NULL));
  CreateGraphs create;
  ColorSolver solver;

  // PART 1
  AdjacencyList cycle = create.CreateGraph(100, 500, "CYCLE", "NONE");
  AdjacencyList complete = create.CreateGraph(100, 500, "COMPLETE", "NONE");
  AdjacencyList uniform = create.CreateGraph(100, 500, "RANDOM", "UNIFORM");
  AdjacencyList tier = create.CreateGraph(100, 500, "RANDOM", "TIERED");
  AdjacencyList linear = create.CreateGraph(100, 500, "RANDOM", "LINEAR");

  cycle.OutputFile("../Graphs/cycle.csv");
  complete.OutputFile("../Graphs/complete.csv");
  uniform.OutputFile("../Graphs/uniform.csv");
  tier.OutputFile("../Graphs/tier.csv");
  linear.OutputFile("../Graphs/linear.csv");


  // PART 2

  // Complete output for SLVO
  string filenames[5] = {"complete","cycle","uniform","tier","linear"};
  AdjacencyList graph = create.createRandomTiered(100, 900);
  int *slvoOrder = solver.slvoSolver(graph);
  solver.colorGraph(slvoOrder, graph, "SLVO_" + filenames[2], true, true);

  // Get outputs for all combinations of ordering/ graph
  for(int i = 0; i < 5; i++) {
    AdjacencyList graph = create.readFromFile("../Graphs/" + filenames[i] + ".csv");
    cout << "------------------------------" << endl;
    cout << filenames[i] << endl;

    cout  << endl << "SLVO" << endl;
    int *slvoOrder = solver.slvoSolver(graph);
    solver.colorGraph(slvoOrder, graph, "SLVO_" + filenames[i], false, false);

    cout  << endl << "Smallest Original Degree" << endl;
    int *smallOrigOrder = solver.smallOriginalSolver(graph);
    solver.colorGraph(smallOrigOrder, graph, "SmallestOriginalDegree_" + filenames[i], false, false);

    cout  << endl << "Random Order" << endl;
    int *randomOrder = solver.uniformRandomSolver(graph);
    solver.colorGraph(randomOrder, graph, "RandomOrder_" + filenames[i], false, false);
  }
```

```
    // Functions to create data/ graph files
    create.CycleAndCompleteGraphTimings();
    create.RandomGraphCreationTimings();
    create.OutputGraphConflicts();

    solver.PrintExampleSLVO();
    solver.OrderSolverTimings();

    solver.ColorTimings();

    return 0;
}
```

AdjacencyList.h

```
#ifndef PA2_ADJACENCYLIST_H
#define PA2_ADJACENCYLIST_H
#include <algorithm>
#include <iostream>
#include "VertexNode.h"
using namespace std;

class AdjacencyList{
private:
    int edges;
    int V;
public:
    VertexNode* adjList;
    AdjacencyList();
    AdjacencyList(int V);
    AdjacencyList(const AdjacencyList &list);
    AdjacencyList &operator=(const AdjacencyList &l);
    void addEdge(int src, int dest);
    void addSingleEdge(int src, int dest);
    ~AdjacencyList();
    void display();
    void displayMinusOne();
    void OutputFile(string filename);
    int getNumEdges();
    int getNumVertices();
};

#endif
```

AdjacencyList.cpp

```
#include "AdjacencyList.h"
#include <vector>
#include <fstream>
using namespace std;

AdjacencyList::AdjacencyList(){
   V = 2000000;
   edges = 0;
   adjList = new VertexNode[V];
   for(int i = 0; i < V; i++) {
      VertexNode temp;
      adjList[i] = temp;
   }
}

AdjacencyList::AdjacencyList(int size){
   V = size;
   edges = 0;
   adjList = new VertexNode[V];
   for(int i = 0; i < V; i++) {
      VertexNode temp;
      adjList[i] = temp;
   }
}

AdjacencyList::AdjacencyList(const AdjacencyList &list){
   V = list.V;
   edges = 0;
   adjList = new VertexNode[V];
   for(int i = 0; i < V; i++) {
      adjList[i] = list.adjList[i];
   }
}
AdjacencyList& AdjacencyList::operator=(const AdjacencyList &list){
   V = list.V;
   edges = 0;
   adjList = new VertexNode[V];
   for(int i = 0; i < V; i++) {
      adjList[i] = list.adjList[i];
   }
   return *this;
}

void AdjacencyList::addEdge(int src, int dest){
   adjList[src].addNode(dest);
   adjList[dest].addNode(src);
   edges += 2;
}

void AdjacencyList::addSingleEdge(int src, int dest){
```

```
    adjList[src].addNode(dest);
    edges++;
}

AdjacencyList::~AdjacencyList(){
    delete[] adjList;
}

void AdjacencyList::display(){
    for(int i = 0; i < V; i++){
        cout << i+1;
        adjList[i].connections.printListPlusOne();
        cout << endl;
    }
}

void AdjacencyList::displayMinusOne(){
    for(int i = 0; i < V; i++){
        cout << i;
        adjList[i].connections.printList();
        cout << endl;
    }
}

void AdjacencyList::OutputFile(string filename){
    int outputSize = 1 + V + edges;
    int out[outputSize];
    out[0] = V;

    int index = 1 + V;
    for(int i = 0; i < V; i++) {
        out[i+1] = index;
        adjList[i].connections.ToOutputPlusOne(out, index);
    }

    ofstream outfile(filename);
    for(int i = 0; i < outputSize; i++)
        outfile << out[i] << endl;
    outfile.close();
}

int AdjacencyList::getNumEdges(){
    return edges;
}
int AdjacencyList::getNumVertices(){
    return V;
}
```

ColorSolver.h

```
#ifndef MAIN_CPP_COLORSOLVER_H
#define MAIN_CPP_COLORSOLVER_H
#include "AdjacencyList.h"
#include "Vector.h"
#include "CreateGraphs.h"

class ColorSolver {
public:
   ColorSolver();
   int colorGraph(int* order, AdjacencyList graph, string filename, bool output, bool slvo);
   int* slvoSolver(AdjacencyList& graph);           // Smallest last vertex
   int* smallOriginalSolver(AdjacencyList graph);   // small original degree
   int* uniformRandomSolver(AdjacencyList graph);   // uniform random

   void OrderSolverTimings();
   void ColorTimings();

   void PrintExampleSLVO();

//    void REVslvoSolver(AdjacencyList& graph);           // reverse slvo
//    void REVsmallOriginalSolver(AdjacencyList& graph);   // reverse small original degree
//    void inOrderSolver(AdjacencyList& graph);           // In number of node order
};


#endif
```

ColorSolver.cpp

```
#include "ColorSolver.h"

ColorSolver::ColorSolver(){}

int* ColorSolver::slvoSolver(AdjacencyList& graph){
   LinkedList<int>* vertexDegrees = new LinkedList<int>[graph.getNumVertices()-1];
   int graphSize = graph.getNumVertices();

   // Create Blank Lists at every possible number of connections
   for(int i = 0; i < graphSize; i++)
      vertexDegrees[i] = LinkedList<int>();

   // Add node number to list where it has number of connections in array
   for(int i = 0; i < graphSize; i++)
      graph.adjList[i].orderPointer = vertexDegrees[graph.adjList[i].connections.getSize()].push_back(i);

   int* ordering = new int[graphSize];
   int orderingCount = 0;
   int smallIndex = 0;
```

```
    while(orderingCount < graphSize){
        if(vertexDegrees[smallIndex].getSize() != 0) {
            // Pulling Smallest Vertex off Tree and marking deleted
            int deleted = vertexDegrees[smallIndex].removeHead();
            ordering[graphSize-orderingCount-1] = deleted;
            orderingCount++;
            graph.adjList[deleted].deleted = true;

            // Loop through all of its connections
            ListNode<int> *cur = graph.adjList[deleted].connections.getHead();
            while (cur != nullptr) {
                int vertexNum = cur->data;
                // If the connection is not deleted:
                // delete from current list, subtract connections, add onto new list, and update pointer
                if (!graph.adjList[vertexNum].deleted) {
                    ListNode<int> *toDelete = graph.adjList[vertexNum].orderPointer;
                    vertexDegrees[graph.adjList[vertexNum].curConnections].deleteNode(toDelete);
                    graph.adjList[vertexNum].curConnections--;
                    graph.adjList[vertexNum].orderPointer =
                            vertexDegrees[graph.adjList[vertexNum].curConnections].push_back(vertexNum);
                }
                cur = cur->next;
            }
            // Go back one index in array
            if(smallIndex>0) smallIndex-=2;
            else smallIndex--;
        }
        smallIndex++;
    }
    return ordering;
}

int* ColorSolver::smallOriginalSolver(AdjacencyList graph) {
    LinkedList<int>* vertexDegrees = new LinkedList<int>[graph.getNumVertices()-1];
    int graphSize = graph.getNumVertices();

    // Create Blank Lists at every possible number of connections
    for(int i = 0; i < graphSize; i++)
        vertexDegrees[i] = LinkedList<int>();

    // Add node number to list where it has number of connections in array
    for(int i = 0; i < graphSize; i++)
        graph.adjList[i].orderPointer = vertexDegrees[graph.adjList[i].connections.getSize()].push_back(i);

    int* ordering = new int[graphSize];
    int orderingCount = 0;
    int smallIndex = 0;

    while(orderingCount < graphSize){
        if(vertexDegrees[smallIndex].getSize() != 0) {
            // Pulling Smallest Vertex off Tree and marking deleted
            int deleted = vertexDegrees[smallIndex].removeHead();
```

```
        ordering[graphSize-orderingCount-1] = deleted;
        orderingCount++;
        graph.adjList[deleted].deleted = true;

        // Go back one index in array
        if(smallIndex>0) smallIndex-=2;
        else smallIndex--;
      }
      smallIndex++;
    }
    return ordering;
}

int* ColorSolver::uniformRandomSolver(AdjacencyList graph){
    int graphSize = graph.getNumVertices();
    int* ordering = new int[graphSize];

    Vector<int> temp;
    temp.createRandom(graphSize);
    for(int i = 0; i < graphSize; i++)
      ordering[i] = temp[i];
    return ordering;
}

int getSmallestColorInList(AdjacencyList graph, VertexNode& node, int numColors){
    for(int i = 1; i <= numColors; i++){
      bool found = false;
      ListNode<int> *cur =node.connections.getHead();
      while (cur != nullptr) {
        if(graph.adjList[cur->data].color == i){
          found = true;
          break;
        }
        cur = cur->next;
      }
      if(!found) return i;
    }
    return numColors + 1;
}

int ColorSLVO(int* order, AdjacencyList& graph, string filename){
    ofstream out("../Color_Outputs/"+filename+".csv");
    int numColors = 0;
    cout << filename << endl;
    cout << "VERTEX NUM" << "," << "ORIGINAL DEGREE" << "," << "DEGREE WHEN DELETED" << endl;
    int degreeCount = 0;
    int maxDeleted = 0;
    int cliqueSize = 0;

    for(int i = 0; i < graph.getNumVertices(); i++){
      VertexNode& curVertex = graph.adjList[order[i]];
      curVertex.color = getSmallestColorInList(graph, curVertex, numColors);
      if(curVertex.color > numColors) {
```

```
        numColors++;
        if(i+1 == numColors)
            cliqueSize++;
      }
      if(curVertex.curConnections > maxDeleted)
        maxDeleted = curVertex.curConnections;

      out << order[i]+1 << "," << curVertex.color << endl;
      cout << order[i]+1 << "," << curVertex.size << "," << curVertex.curConnections << endl;
      degreeCount += curVertex.size;
    }
    cout << "Total Colors Used: " << numColors << endl;
    cout << "Average Original Degree (rounded down): " << degreeCount/graph.getNumVertices() << endl;
    cout << "Maximum Degree when Deleted: " << maxDeleted << endl;
    cout << "Size of Terminal Clique: " << cliqueSize << endl;

    out.close();
    return numColors;
}

int ColorNonSLVO(int* order, AdjacencyList& graph, string filename){
    ofstream out("../Color_Outputs/"+filename+".csv");
    int numColors = 0;
    cout << filename << endl;
    cout << "VERTEX NUM" << "," << "ORIGINAL DEGREE" << endl;
    int degreeCount = 0;

    for(int i = 0; i < graph.getNumVertices(); i++){
      VertexNode& curVertex = graph.adjList[order[i]];
      curVertex.color = getSmallestColorInList(graph, curVertex, numColors);
      if(curVertex.color > numColors) numColors++;

      out << order[i]+1 << "," << curVertex.color << endl;
      cout << order[i]+1 << "," << curVertex.size << endl;
      degreeCount += curVertex.size;
    }
    cout << "Total Colors Used: " << numColors << endl;
    cout << "Average Original Degree (rounded down): " << degreeCount/graph.getNumVertices() << endl;

    out.close();
    return numColors;
}

int ColorNoOutput(int* order, AdjacencyList& graph){
    int numColors = 0;
    int degreeCount = 0;

    for(int i = 0; i < graph.getNumVertices(); i++){
      VertexNode& curVertex = graph.adjList[order[i]];
      curVertex.color = getSmallestColorInList(graph, curVertex, numColors);
      if(curVertex.color > numColors) numColors++;

      degreeCount += curVertex.size;
```

```
    }
    cout << "Total Colors Used: " << numColors << endl;
    cout << "Average Original Degree (rounded down): " << degreeCount/graph.getNumVertices() << endl;
    return numColors;
}

int ColorSolver::colorGraph(int* order, AdjacencyList graph, string filename, bool output, bool slvo){
    if(output) {
        if (slvo)
            return ColorSLVO(order, graph, filename);
        else
            return ColorNonSLVO(order, graph, filename);
    }
    else return ColorNoOutput(order, graph);
}

void ColorSolver::OrderSolverTimings(){
    ofstream output("../Timings/OrderSolverTimings.csv");
    output << "V" << "," << "E" << "," << "SLVO" << "," << "Smallest Original" << "," << "Uniform Random" << endl;

    CreateGraphs create;
    int E = 2000;
    for(int i = 1000; i <= 10000; i+=1000)
    {
        AdjacencyList graph = create.createRandomUniform(i, E);

        auto beginTime = chrono::high_resolution_clock::now();
        slvoSolver(graph);
        auto endTime = chrono::high_resolution_clock::now();
        auto slvoTime =
                chrono::duration_cast<chrono::microseconds>(endTime - beginTime);

        beginTime = chrono::high_resolution_clock::now();
        smallOriginalSolver(graph);
        endTime = chrono::high_resolution_clock::now();
        auto smallOrginalTime =
                chrono::duration_cast<chrono::microseconds>(endTime - beginTime);

        beginTime = chrono::high_resolution_clock::now();
        uniformRandomSolver(graph);
        endTime = chrono::high_resolution_clock::now();
        auto uniformRandomTime =
                chrono::duration_cast<chrono::microseconds>(endTime - beginTime);


        output << i << "," << E << "," << slvoTime.count() << "," << smallOrginalTime.count() << ","
            << uniformRandomTime.count() << endl;
    }
    output << endl;
    int V = 2000;
    for(int i = 1000; i <= 10000; i+=1000)
    {
        AdjacencyList graph = create.createRandomUniform(V, i);
```

```
        auto beginTime = chrono::high_resolution_clock::now();
        slvoSolver(graph);
        auto endTime = chrono::high_resolution_clock::now();
        auto slvoTime =
            chrono::duration_cast<chrono::microseconds>(endTime - beginTime);

        beginTime = chrono::high_resolution_clock::now();
        smallOriginalSolver(graph);
        endTime = chrono::high_resolution_clock::now();
        auto smallOrginalTime =
            chrono::duration_cast<chrono::microseconds>(endTime - beginTime);

        beginTime = chrono::high_resolution_clock::now();
        uniformRandomSolver(graph);
        endTime = chrono::high_resolution_clock::now();
        auto uniformRandomTime =
            chrono::duration_cast<chrono::microseconds>(endTime - beginTime);


        output << V << "," << i << "," << slvoTime.count() << "," << smallOrginalTime.count() << ","
            << uniformRandomTime.count() << endl;
    }
    output.close();
}

void ColorSolver::ColorTimings(){
    ofstream output("../Timings/ColorSolverTimings.csv");
    output << "V" << "," << "µs" << endl;

    CreateGraphs create;
    for(int i = 10; i <= 100; i+=10)
    {
        AdjacencyList graph = create.createComplete(i);
        int *randomOrder = uniformRandomSolver(graph);

        auto beginTime = chrono::high_resolution_clock::now();
        colorGraph(randomOrder, graph, "none", false, false);
        auto endTime = chrono::high_resolution_clock::now();
        auto time =
            chrono::duration_cast<chrono::microseconds>(endTime - beginTime);

        output << i << "," << time.count() << endl;
    }
    output << endl;
    output.close();
}

void ColorSolver::PrintExampleSLVO(){
    CreateGraphs create;
    AdjacencyList graph = create.createRandomUniform(5,6);
    cout << "EXAMPLE" << endl;
    graph.display();
```

```
  LinkedList<int>* vertexDegrees = new LinkedList<int>[graph.getNumVertices()-1];
  int graphSize = graph.getNumVertices();

  // Create Blank Lists at every possible number of connections
  for(int i = 0; i < graphSize; i++)
    vertexDegrees[i] = LinkedList<int>();

  // Add node number to list where it has number of connections in array
  for(int i = 0; i < graphSize; i++)
    graph.adjList[i].orderPointer = vertexDegrees[graph.adjList[i].connections.getSize()].push_back(i);

  int* ordering = new int[graphSize];
  int orderingCount = 0;
  int smallIndex = 0;

  while(orderingCount < graphSize){
    if(vertexDegrees[smallIndex].getSize() != 0) {
      // Pulling Smallest Vertex off Tree and marking deleted
      int deleted = vertexDegrees[smallIndex].removeHead();
      ordering[graphSize-orderingCount-1] = deleted;
      orderingCount++;
      graph.adjList[deleted].deleted = true;
      cout << "DELETED " << deleted+1 << endl;

      // Loop through all of its connections
      ListNode<int> *cur = graph.adjList[deleted].connections.getHead();
      while (cur != nullptr) {
        int vertexNum = cur->data;
        // If the connection is not deleted:
        // delete from current list, subtract connections, add onto new list, and update pointer
        if (!graph.adjList[vertexNum].deleted) {
          ListNode<int> *toDelete = graph.adjList[vertexNum].orderPointer;
          vertexDegrees[graph.adjList[vertexNum].curConnections].deleteNode(toDelete);
          graph.adjList[vertexNum].curConnections--;
          graph.adjList[vertexNum].orderPointer =
              vertexDegrees[graph.adjList[vertexNum].curConnections].push_back(vertexNum);
        }
        cur = cur->next;
      }
      // Go back one index in array
      if(smallIndex>0) smallIndex-=2;
      else smallIndex--;
    }
    smallIndex++;
  }
  cout << "SLVO Order: [" << ordering[0]+1;
  for(int i = 1; i < graphSize; i++)
    cout << ", " << ordering[i]+1;
  cout << "]" << endl;
}
```

CreateGraphs.h

```
#ifndef MAIN_CPP_CREATEGRAPHS_H
#define MAIN_CPP_CREATEGRAPHS_H
#include "AdjacencyList.h"
#include <iostream>

class CreateGraphs {
public:
   AdjacencyList CreateGraph(int V, int E, string G, string DIST);

   AdjacencyList createComplete(int V);
   AdjacencyList createCycle(int V);

   AdjacencyList createRandomUniform(int V, int E);
   AdjacencyList createRandomTiered(int V, int E);

   AdjacencyList createRandomYours(int V, int E);

   AdjacencyList readFromFile(string filepath);

   int calculateNumberOfLinearDist(int);

   void CycleAndCompleteGraphTimings();
   void RandomGraphCreationTimings();

   void OutputGraphConflicts();
};


#endif
```

CreateGraphs.cpp

```
//
// Created by Colin Weil on 11/28/22.
//

#include "CreateGraphs.h"
#include <chrono>
#include <fstream>
#include <random>
using namespace std;

AdjacencyList CreateGraphs::CreateGraph(int V, int E, string G, string DIST){
    if(G == "COMPLETE"){
        return createComplete(V);
    }
    else if(G == "CYCLE"){
        return createCycle(V);
    }
    else{
```

```
        if(DIST == "UNIFORM"){
            return createRandomUniform(V,E);
        }
        else if(DIST == "TIERED"){
            return createRandomTiered(V,E);
        }
        else{
            return createRandomYours(V,E);
        }
    }
}

AdjacencyList CreateGraphs::createRandomUniform(int V, int E){
    AdjacencyList graph(V);
    for(int i = 0; i < E; i++){
        int x = rand() % V;
        int y = rand() % V;
        while(graph.adjList[x].connections.getSize() == V-1){
            x = rand() % V;
        }
        while(x == y || graph.adjList[x].connections.existsInList(y)){
            y = rand() % V;
        }
        graph.addEdge(x,y);
    }
    return graph;
}

int getTiered(int V){
    int x = rand() % (2 * (V/10 * 9));
    if(x < V/10 * 9) x = x % V/10;
    else x = (x - V/10 * 8);
    return x;
}
AdjacencyList CreateGraphs::createRandomTiered(int V, int E){
    AdjacencyList graph(V);
    for(int i = 0; i < E; i++){
        int x = getTiered(V);
        int y = getTiered(V);
        while(graph.adjList[x].connections.getSize() == V-1){
            x = getTiered(V);
        }
        while(x == y || graph.adjList[x].connections.existsInList(y)){
            y = getTiered(V);
        }
        graph.addEdge(x,y);
    }
    return graph;
}

int CreateGraphs::calculateNumberOfLinearDist(int v){
    int V = v-1;
    int count = ((V * (V+1))/2);
    int x = rand() % count;
    for(int j = V; j > 0; j--){
        count = count - j;
        if(count < x) return j-1;
```

```
    }
    return V;
}

AdjacencyList CreateGraphs::createRandomYours(int V, int E){
    AdjacencyList graph(V);
    for(int i = 0; i < E; i++){
        int x = calculateNumberOfLinearDist(V);
        int y = calculateNumberOfLinearDist(V);
        while(graph.adjList[x].connections.getSize() == V-1){
            x = calculateNumberOfLinearDist(V);
        }
        while(x == y || graph.adjList[x].connections.existsInList(y)){
            y = calculateNumberOfLinearDist(V);
        }
        graph.addEdge(x,y);
    }
    return graph;
}

AdjacencyList CreateGraphs::createComplete(int V){
    AdjacencyList graph(V);
    for(int i = 0; i < V; i++){
        for(int j = 0; j < V; j++){
            if(i != j)
                graph.addSingleEdge(i, j);
        }
    }
    return graph;
}

AdjacencyList CreateGraphs::createCycle(int V){
    AdjacencyList graph(V);
    for(int i = 1; i < V; i++){
        graph.addEdge(i, i-1);
    }
    graph.addEdge(0, V-1);
    return graph;
}

void CreateGraphs::CycleAndCompleteGraphTimings(){
    ofstream output("../Timings/CycleAndCompleteGraphTimings.csv");
    output << "V" << "," << "Complete" << "," << "Cycle" << endl;

    for(int i = 1000; i <= 10000; i+=1000)
    {
        auto beginTime = chrono::high_resolution_clock::now();
        createComplete(i);
        auto endTime = chrono::high_resolution_clock::now();
        auto completeRunTime =
                chrono::duration_cast<chrono::microseconds>(endTime -
beginTime);

        beginTime = chrono::high_resolution_clock::now();
        createCycle(i);
        endTime = chrono::high_resolution_clock::now();
        auto cycleRunTime =
```

```
                chrono::duration_cast<chrono::microseconds>(endTime -
beginTime);

        output << i << "," << completeRunTime.count() << "," <<
cycleRunTime.count() << endl;
    }
    output.close();
}

void CreateGraphs::RandomGraphCreationTimings(){
    ofstream output("../Timings/RandomGraphCreationTimings.csv");
    output << "V" << "," << "E" << "," << "Uniform" << "," << "Tiered" << ","
<< "Linear" << endl;

    int E = 500;
    for(int i = 1000; i <= 10000; i+=1000)
    {
        auto beginTime = chrono::high_resolution_clock::now();
        createRandomUniform(i, E);
        auto endTime = chrono::high_resolution_clock::now();
        auto uniformRunTime =
                chrono::duration_cast<chrono::microseconds>(endTime -
beginTime);

        beginTime = chrono::high_resolution_clock::now();
        createRandomTiered(i, E);
        endTime = chrono::high_resolution_clock::now();
        auto tieredRunTime =
                chrono::duration_cast<chrono::microseconds>(endTime -
beginTime);

        beginTime = chrono::high_resolution_clock::now();
        createRandomYours(i, E);
        endTime = chrono::high_resolution_clock::now();
        auto linearRunTime =
                chrono::duration_cast<chrono::microseconds>(endTime -
beginTime);

        output << i << "," << E << "," << uniformRunTime.count() << ","
            << tieredRunTime.count() << "," << linearRunTime.count() << endl;
    }

    int V = 500;
    for(int i = 1000; i <= 10000; i+=1000)
    {
        auto beginTime = chrono::high_resolution_clock::now();
        createRandomUniform(V, i);
        auto endTime = chrono::high_resolution_clock::now();
        auto uniformRunTime =
                chrono::duration_cast<chrono::microseconds>(endTime -
beginTime);

        beginTime = chrono::high_resolution_clock::now();
        createRandomTiered(V, i);
        endTime = chrono::high_resolution_clock::now();
        auto tieredRunTime =
                chrono::duration_cast<chrono::microseconds>(endTime -
```

```
beginTime);

        beginTime = chrono::high_resolution_clock::now();
        createRandomYours(V, i);
        endTime = chrono::high_resolution_clock::now();
        auto linearRunTime =
                chrono::duration_cast<chrono::microseconds>(endTime -
beginTime);

        output << V << "," << i << "," << uniformRunTime.count() << ","
               << tieredRunTime.count() << "," << linearRunTime.count() <<
endl;
    }
    output.close();
}

AdjacencyList CreateGraphs::readFromFile(string filepath){
    ifstream file(filepath);

    int size;
    file >> size;
    int lineNumbers[size];
    for(int i = 0; i < size; i++){
        file >> lineNumbers[i];
    }

    AdjacencyList graph(size);
    for(int i = 0; i < size-1; i++){
        for(int j = lineNumbers[i]; j < lineNumbers[i+1]; j++){
            int dest;
            file >> dest;
            graph.addSingleEdge(i, dest-1);
        }
    }
    int dest;
    while(file >> dest)
        graph.addSingleEdge(size-1, dest-1);

    return graph;
}

void outputGraphConflict(AdjacencyList graph, string filename){
    ofstream output("../GraphConflicts/" + filename);
    output << "Vertex" << "," << "# of Conflicts" << endl;
    for(int i = 0; i < graph.getNumVertices(); i++){
        output << i+1 << "," << graph.adjList[i].size << endl;
    }
    output.close();
}

void CreateGraphs::OutputGraphConflicts(){
    int V = 50; int E = 100;
    AdjacencyList cycle = CreateGraph(V, E, "CYCLE", "NONE");
    AdjacencyList complete = CreateGraph(V, E, "COMPLETE", "NONE");
    AdjacencyList uniform = CreateGraph(V, E, "RANDOM", "UNIFORM");
    AdjacencyList tier = CreateGraph(V, E, "RANDOM", "TIERED");
    AdjacencyList linear = CreateGraph(V, E, "RANDOM", "LINEAR");
```

```
    outputGraphConflict(cycle, "cycle.csv");
    outputGraphConflict(complete, "complete.csv");
    outputGraphConflict(uniform, "uniform.csv");
    outputGraphConflict(tier, "tier.csv");
    outputGraphConflict(linear, "linear.csv");
}
```

LinkedList.h

```
#include <iostream>
#include <fstream>
using namespace std;

template<class T>
class ListNode{
public:
  T data;
  ListNode* next;
  ListNode* previous;

  ListNode(T d, ListNode<T>* n = nullptr, ListNode<T>* p =
  nullptr){
    data = d;
    next = n;
    previous = p;
  }

  bool operator==(const ListNode& n) const { return data ==
                                  n.data; }
  T& getData(){ return data; }
  ListNode* getNext(){ return next; }
  ListNode* getPrevious(){ return previous; }
};

template<class T>
class LinkedList {
private:
  ListNode<T> *head;
  ListNode<T> *tail;
  int size;
public:
  LinkedList();
  LinkedList(T data);
  LinkedList(const LinkedList &list);
  LinkedList &operator=(const LinkedList &l);

  void ToOutputPlusOne(int* out, int &index);
  void printListPlusOne();
  void printList();
  ~LinkedList();
  int getSize();
  ListNode<T>* push_back(T data);
  bool existsInList(T data);
```

```
    ListNode<T>* getTail();
    ListNode<T>* getHead();
    T removeHead();
    ListNode<T>* deleteNode(ListNode<T>* point);
};

template<class T>
ListNode<T>* LinkedList<T>::getTail(){ return tail; }

template<class T>
ListNode<T>* LinkedList<T>::getHead(){ return head; }

template<class T>
T LinkedList<T>::removeHead(){
    if(head == nullptr)
        return -1;
    T data = head->data;

    ListNode<T>* temp = head;
    head = temp->next;
    if(head != nullptr) head->previous = nullptr;

    size--;
    delete temp;
    return data;
}

template<class T>
bool LinkedList<T>::existsInList(T data){
    ListNode<T>* cur = head;
    while(cur != nullptr){
        if(cur->data == data)
            return true;
        cur = cur->next;
    }
    return false;
}

template<class T>
ListNode<T>* LinkedList<T>::push_back(T data) {
    // No data
    if(head == nullptr) {
        head = new ListNode<T>(data);
        tail = head;
    }
    else{
        ListNode<T> *temp = new ListNode<T>(data, nullptr, tail);
        tail->next = temp;
        tail = temp;
    }
    size++;
    return tail;
}
```

```
template<class T>
LinkedList<T>::LinkedList() {
   head = nullptr;
   tail = nullptr;
   size = 0;
}

template<class T>
LinkedList<T>::LinkedList(T data) {
   head = new ListNode<T>(data);
   size = 1;
}

template<class T>
LinkedList<T>::LinkedList(const LinkedList &list){
   head = nullptr;
   ListNode<T> *cur = list.head;
   while (cur != nullptr) {
      push_back(cur->data);
      cur = cur->next;
   }
   size = list.size;
}

template<class T>
LinkedList<T>& LinkedList<T>::operator=(const LinkedList<T>& list){
   head = nullptr;
   ListNode<T> *cur = list.head;
   while (cur != nullptr) {
      push_back(cur->data);
      cur = cur->next;
   }
   size = list.size;
   return *this;
}

template<class T>
void LinkedList<T>::ToOutputPlusOne(int* out, int &index) {
   ListNode<T>* cur = head;
   while(cur != nullptr){
      out[index] = cur->data+1;
      index++;
      cur = cur->next;
   }
}

template<class T>
void LinkedList<T>::printListPlusOne() {
   if(head == nullptr){
      return;
   }
   ListNode<T>* cur = head;
```

```
    while(cur != nullptr){
        cout << " -> " << cur->data + 1;
        cur = cur->next;
    }
}

template<class T>
void LinkedList<T>::printList() {
    if(head == nullptr){
        return;
    }
    ListNode<T>* cur = head;
    while(cur != nullptr){
        cout << " -> " << cur->data;
        cur = cur->next;
    }
}

template<class T>
LinkedList<T>::~LinkedList() {
    ListNode<T>* temp;
    while (head != nullptr) {
        temp = head;
        head = head->next;
        delete temp;
    }
}

template<class T>
int LinkedList<T>::getSize(){
    return size;
}

template<class T>
ListNode<T>* LinkedList<T>::deleteNode(ListNode<T>* cur){
    // Not head
    if(cur->previous != nullptr){
        // not head, not tail
        if(cur->next != nullptr) {
            cur->previous->next = cur->next;
            cur->next->previous = cur->previous;
            delete cur;
        }
        // not head, but tail
        else{
            tail = cur->previous;
            tail->next = nullptr;
            delete cur;
        }
    }
    // data is at head
    else{
        // head, not tail
```

```
      if(cur->next != nullptr) {
         head = cur->next;
         head->previous = nullptr;
         delete cur;
      }
      // head and tail
      else{
         head = nullptr;
         delete cur;
      }
   }
   size--;
   return head;
}
```

Vector.h

```
#ifndef MAIN_CPP_VECTOR_H
#define MAIN_CPP_VECTOR_H
#include <ostream>
#include <iostream>
//#include <random>
using namespace std;

template<class T>
class Vector{
private:
   T* data;
   int size;
   int cap;
public:
   void resize();
   void push_back(const T& x);
   int getSize();
   Vector();
   ~Vector();
   Vector (const Vector& vector);
   Vector& operator=(const Vector& vector);
   T& operator[](int index);
   void remove(int index);
   void createRandom(int size);
   void print();
};

template<class T>
void Vector<T>::createRandom(int s){
   Vector<int> temp;
   for(int i = 0; i < s; i++)
      temp.push_back(i);

   for(int i = 0; i < s; i++){
```

```
        int randIndex = rand() % temp.size;
        push_back(temp[randIndex]);
        temp.remove(randIndex);
    }
}

template<class T>
void Vector<T>::print(){
    for(int i = 0; i < size; i ++)
        cout << data[i] << " ";
    cout << endl;
}

// Vector Constructor
template<class T>
Vector<T>::Vector() {
    data = new T[10];
    size = 0;
    cap = 10;
}

// Vector Destructor
template<class T>
Vector<T>::~Vector() {
    if(data != nullptr)
        delete[] data;
}

// Returns size of vector
template<class T>
int Vector<T>::getSize(){
    return size;
}

// Vector copy constructor
template<class T>
Vector<T>::Vector(const Vector<T>& vector){
    size = vector.size;
    cap = vector.cap;
    data = new T[cap];
    for(int i = 0; i < size; i++){
        data[i] = vector.data[i];
    }
}

// Resizes vector to twice the capacity as before
template<class T>
void Vector<T>::resize() {
    cap = cap*2;
    T* temp = new T[cap];
    for(int i = 0; i < size; i++){
        temp[i] = data[i];
    }
```

```
    delete[] data;
    data = temp;
}

// Adds a value to the end of the vector
template<class T>
void Vector<T>::push_back(const T &x) {
    if(cap == size){
        resize();
    }
    data[size++] = x;
}

// Overided equal operator
template<class T>
Vector<T>& Vector<T>::operator=(const Vector<T> &vector) {
    if(data!=nullptr){
        delete[] data;
    }
    size = vector.size;
    cap = vector.cap;
    data = new T[cap];
    for(int i = 0; i < size; i++){
        data[i] = vector.data[i];
    }
    return *this;
}

// Returns element in [index] of vector
template<class T>
T& Vector<T>::operator[](int index){
    if(index>=size || index<0){
        throw:: std::out_of_range("not in index of vector");
    }
    return data[index];
}

template<typename T>
void Vector<T>::remove(int index) {
    T *tempData = new T[size-1];
    for(int i = 0; i < size; i++) {
        if(i < index)
            tempData[i] = data[i];
        else if(i > index)
            tempData[i-1] = data[i];
    }
    delete[] data;
    data = tempData;
    size--;
}


#endif
```

VertexNode.h

```cpp
#ifndef MAIN_CPP_VERTEXNODE_H
#define MAIN_CPP_VERTEXNODE_H
#include "LinkedList.h"


class VertexNode {
public:
    VertexNode();
    void addNode(int dest);
    LinkedList<int> connections;
    ListNode<int>* orderPointer;
    bool deleted;
    int size;
    int curConnections;
    int color;
};


#endif
```

VertexNode.cpp

```cpp
#include "VertexNode.h"

VertexNode::VertexNode(){
    deleted = false;
    size = 0;
    orderPointer = nullptr;
    curConnections = 0;
    color = -1;
}
void VertexNode::addNode(int dest){
    connections.push_back(dest);
    size++;
    curConnections++;
}
```