Colin Weil
CS5350 Programming Assignment 2

Problem

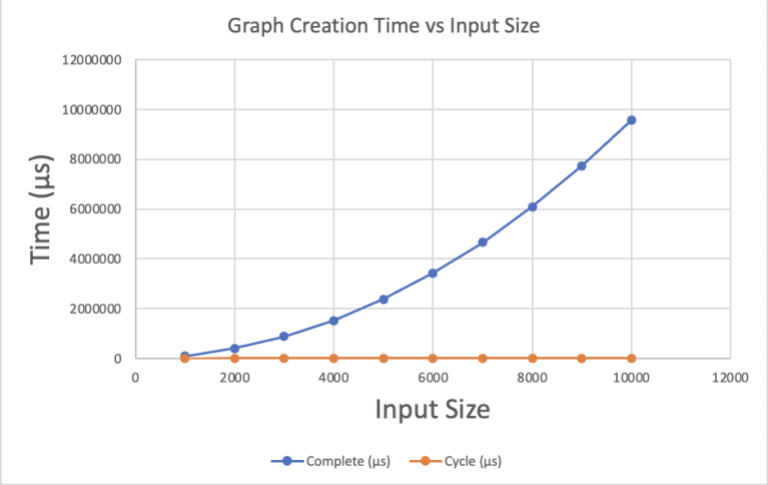
Create a program that accepts a number of vertices "V", create both an undirect complete graph and cycle graph with "V" vertices using an adjacency list data structure of your creation and then output the graph in the format below. Do not include the outputting of the graph in your timing analysis.

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 $\Theta(n^2)$.

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 $\Theta(n)$.

Here is the running time chart and graph:

n	Complete (µs)	Cycle (µs)
1000	96554	291
2000	406820	647
3000	872828	888
4000	1525858	1167
5000	2382520	1456
6000	3424530	1740
7000	4662863	2017
8000	6098540	2299
9000	7733144	2610
10000	9562269	2904



The table and graph support my analysis of $\Theta(n^2)$ for the complete graph since as the size of n doubles (from 1000 to 2000) the running time also essential quadruples since $2^2 = x4$ (from 96,554 to 406,820).

The table and graph support my analysis of $\Theta(n)$ for the cycle graph since as the size of n doubles (from 4000 to 8000) the running time also essential doubles (from 1167 to 2299).

Here is the code in C++: The highlighted code is the tested code. The rest of the code is scaffolding code that can be common for all problems.

The output for both of the graphs are below all of the code

Main.cpp

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

AdjacencyList 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 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;
}
```

```
int main() {
    ofstream output("../timings.csv");
    output << "n" << "," << "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;
        cout << i << "," << completeRunTime.count() << "," <<
cycleRunTime.count() << endl;
    }
    output.close();

    AdjacencyList com = createCycle(5);
    AdjacencyList cy = createComplete(5);

    com.OutputFile("../complete.txt");
    cy.OutputFile("../cycle.txt");

    return 0;
}
```

AdjacencyList.cpp

```
#ifndef PA2_ADJACENCYLIST_H
#define PA2_ADJACENCYLIST_H
#include <algorithm>
#include <iostream>
#include <list>
using namespace std;

class AdjacencyList{
private:
    int V;
    list<int>* adjLists;
public:
    AdjacencyList(int V);
    void addEdge(int src, int dest);
    void addSingleEdge(int src, int dest);
    ~AdjacencyList();
    void display();
    void OutputFile(string filename);
};

#endif //PA2_ADJACENCYLIST_H
```

AdjacencyList.cpp

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

AdjacencyList::AdjacencyList(int size){
    V = size;
    adjLists = new list<int>[V];
    for(int i = 0; i < V; i++) {
        list<int> temp;
        adjLists[i] = temp;
    }
}

void AdjacencyList::addEdge(int src, int dest){
    adjLists[src].push_back(dest);
    adjLists[dest].push_back(src);
}

void AdjacencyList::addSingleEdge(int src, int dest){
    adjLists[src].push_back(dest);
}
```

```
AdjacencyList::~AdjacencyList(){
    delete[] adjLists;
}

void AdjacencyList::display(){
    for(int i = 0; i < V; i++){
        cout << i+1;
        for (auto const &i: adjLists[i]) {
            cout << "->" << i+1 ;
        }
        cout << endl;
    }
}

void AdjacencyList::OutputFile(string filename){
    int outputSize = 1 + V;
    for(int i = 0; i < V; i++)
        outputSize += adjLists[i].size();

    int out[outputSize];
    out[0] = V;

    int index = 1 + V;
    for(int i = 0; i < V; i++) {
        int edges = adjLists[i].size();
        out[i+1] = index;
        for (auto const &i: adjLists[i]) {
            out[index] = i+1;
            index++;
        }
    }

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

Complete Output with V = 5

5
6
10
14
18
22
2
3
4
5
1
3
4
5
1
2
4
5
1
2
3
5
1
2
3
4

Cycle Output with V = 5

5
6
8
10
12
14
2
5
1
3
2
4
3
5
4
1