CS 700-34 Graph Coloring

Student: Shuhua Song

Due Date:	**Submission Data:**
Soft Copy: 05/8/2020	Soft Copy: 05/8/2020
Hard Copy: 05/8/2020	Hard Copy: 05/8/2020

Algorithm Steps:

I. Method1()

```
Step 0: Glist← an undirected graph is given
        ColorList← A list of colors is given
Step 1: newColor← get a new color from ColorListO(1)
Step 2: newNode←get an uncolored node from Glist
Step 3: OK← check all the newNode's neighbors to see if any of its neighbors had been
colored with the newColor, if there is, returns false, otherwise return true;
Step 4: if(OK)
         Color newNode with newColor
Step 5: repeat step 2 to step 4 until all un-colored nodes are checked
Step 6: repeat step 1 to step 5 until all nodes are colored
```

II . Method2()

```
Step 0:  Glist← an undirected graph is given
         ColorList← A list of colors is given
         UsedColorList←{} //empty
Step 1: newNode←get the next uncolor node from Glist
Step 2: newUsedColor←get the next used color from UsedColorList
Step 3: OK ← check all the newNode's neighbors to see if any of its neighbors had been
colored with the newColor, if there is, returns false, otherwise return true;
Step 4: if(OK== false) //try next used color
         Repeat step2 to step3 until OK == true or
         All used colors in UsedColorList have been tried
Step 5: (OK==true)
        Color newNode with new UsedColor
        Else
         newColor ← get a color from colorList
         color newNode with newColor
         Add the newColor into UsedColorList
Step 6: repeat 1 to step 5 until all nodes are colored
```

III. Data Structure

Class Node

Variable Member:

nodeId

color

Method: constructor(){}

Class GraphColoring

Variable Member:

numNodes // the total number in the graph

adjMatrix // a 2D array store the edge of 2 nodes, 1-connected, 0-nonConnected

usedColor // 1D array for storing the used color

colorChoice // 1D array for storing the choice of color

Method:

constructor(){}

loadMatrix() // load the original data from external file

method1(nodeColor, outFile)

method2(nodeColor, outFile)

bool CheckNeigbWithoutColor(nodeId, colorId, nodeColor)

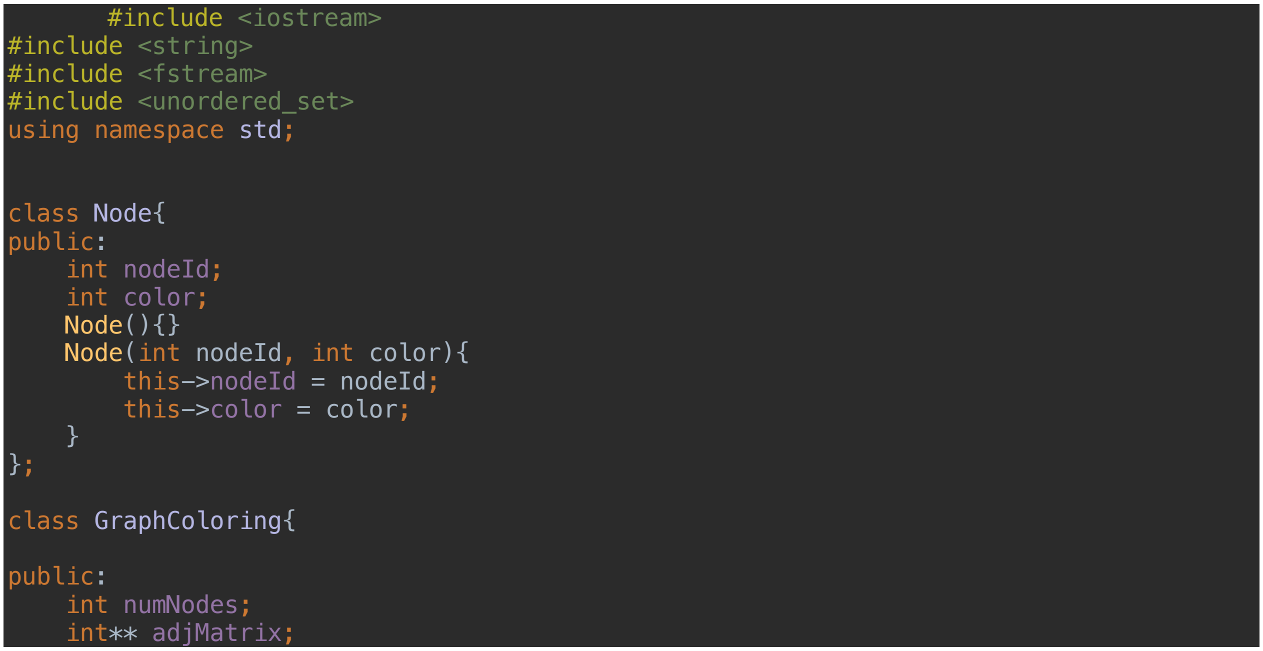
int getUncoloredNode(nodeColor) // find an uncolored node

bool allNodeColored(nodeColor) // check if all node have been colored

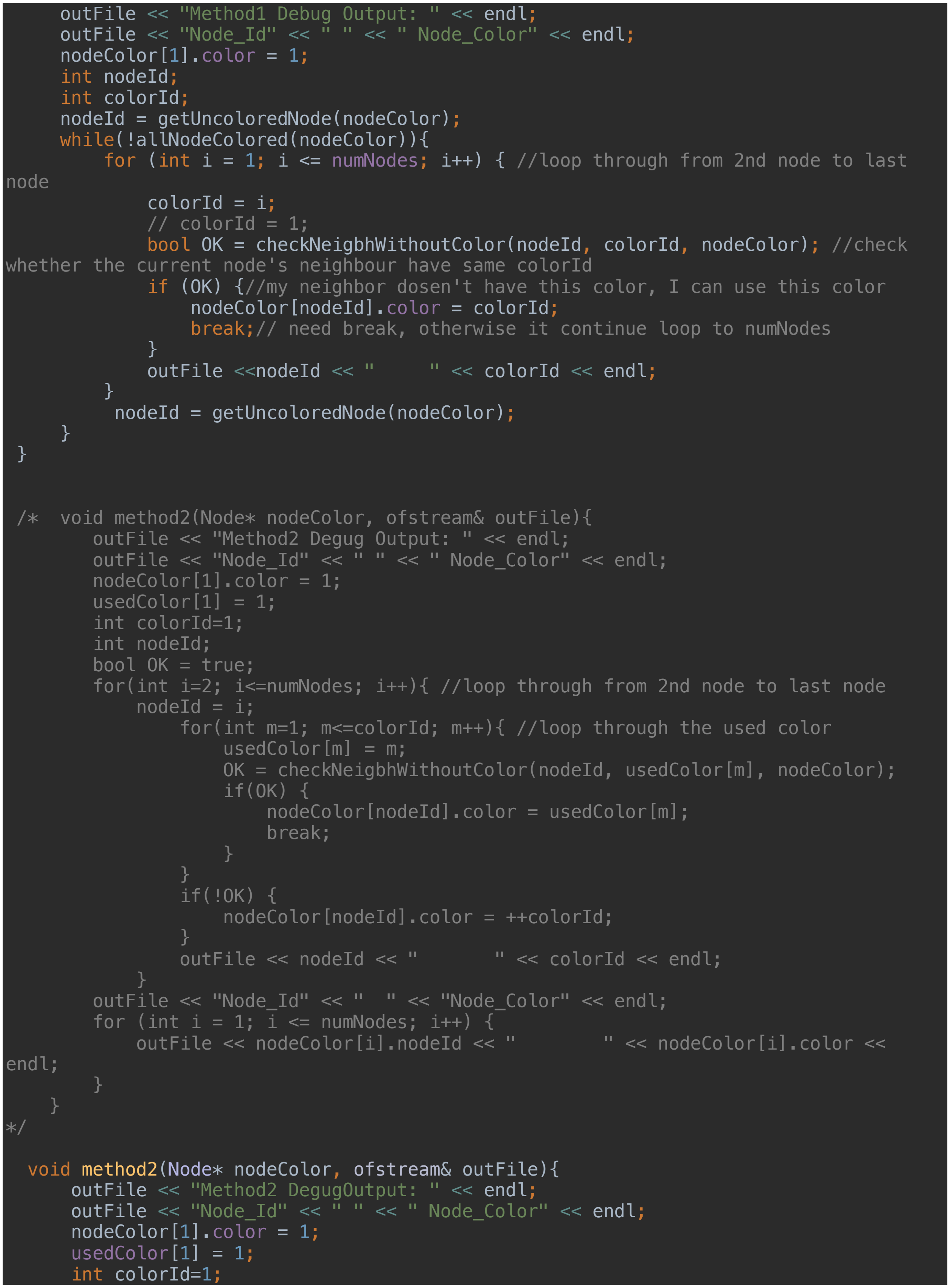
printMatrix(nodeColor, outFile) // print out the data Structure

printColorAssignment(outFile, nodeColor) //print out the result

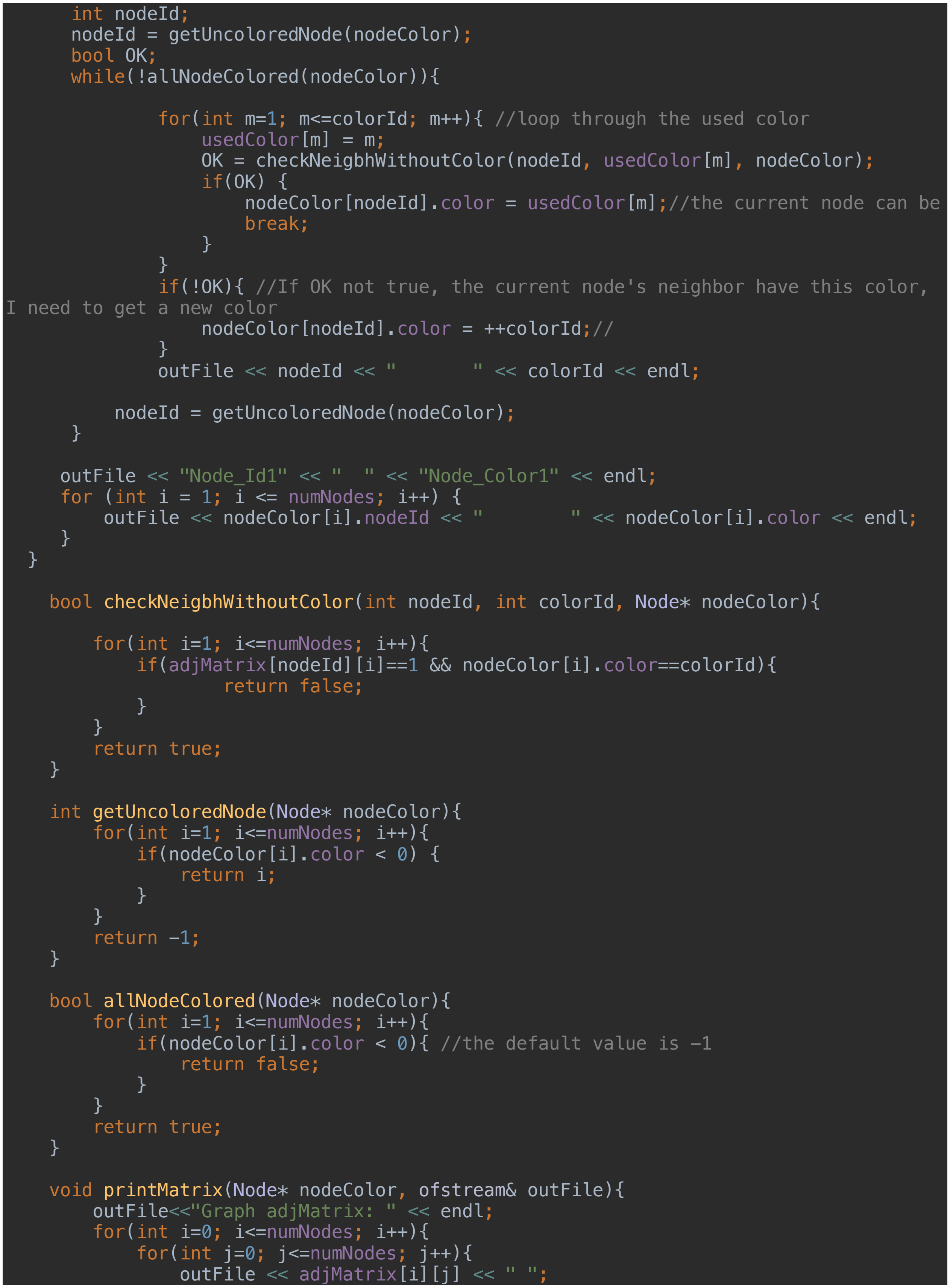
source code



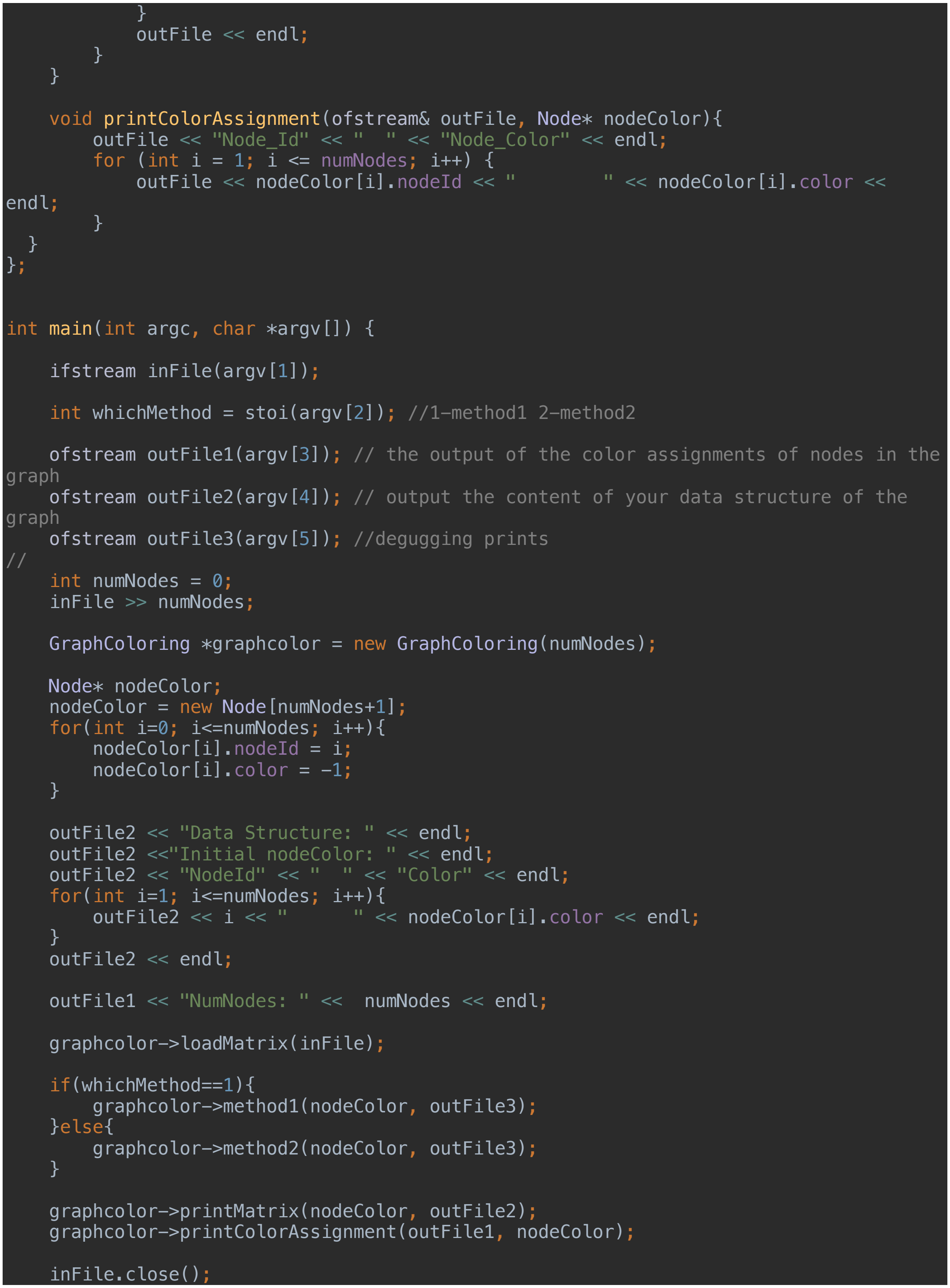
```
       #include <iostream>
#include <string>
#include <fstream>
#include <unordered_set>
using namespace std;


class Node{
public:
    int nodeId;
    int color;
    Node(){}
    Node(int nodeId, int color){
        this->nodeId = nodeId;
        this->color = color;
    }
};

class GraphColoring{

public:
    int numNodes;
    int** adjMatrix;
    int* colorChoice; // A list of colors is given
    int* usedColor;


    GraphColoring(int numNodes){
        this->numNodes = numNodes;

        adjMatrix = new int*[numNodes+1];
        for(int i=0; i<numNodes+1; i++){
            adjMatrix[i] = new int[numNodes+1]{0};
        }


        colorChoice = new int[numNodes+1];
        for(int i=0; i<=numNodes; i++){
            colorChoice[i] = i;
        }
        usedColor = new int[numNodes+1]{-1};
    }

    void loadMatrix(ifstream& inFile){
        int x = 0;
        int y = 0;
        while(!inFile.eof()){
            inFile >> x;
            inFile >> y;
            adjMatrix[x][y] = 1;
            adjMatrix[y][x] = 1;
        }
    }

 /*   void method1(Node* nodeColor) {
        cout << "Method1: " << endl;
        nodeColor[1].color = 1;
        int nodeId;
        int colorId;

       for(int k=2; k<=numNodes; k++){ //loop through from 2nd node to last node
            for (int i = 1; i <= numNodes; i++) { //loop through from 1st color to last color
                colorId = i;
                // colorId = 1;
                bool OK = true;
                for (int j = 1; j <= numNodes; j++) { //loop through from 1st uncolor node to last uncolor node
                    if (adjMatrix[k][j] == 1 && nodeColor[j].color == colorId) {
                        OK = false;
                        break;
                    }
                }
                if (OK) {
                    nodeColor[k].color = colorId;
                    break;
                }
            }
          // nodeId = getUncoloredNode(nodeColor);
        }
    }
    */

 void method1(Node* nodeColor, ofstream& outFile) {
```



```
     outFile << "Method1 Debug Output: " << endl;
     outFile << "Node_Id" << " " << " Node_Color" << endl;
     nodeColor[1].color = 1;
     int nodeId;
     int colorId;
     nodeId = getUncoloredNode(nodeColor);
     while(!allNodeColored(nodeColor)){
         for (int i = 1; i <= numNodes; i++) { //loop through from 2nd node to last 
node
             colorId = i;
             // colorId = 1;
             bool OK = checkNeigbhWithoutColor(nodeId, colorId, nodeColor); //check 
whether the current node's neighbour have same colorId
             if (OK) {//my neighbor dosen't have this color, I can use this color
                 nodeColor[nodeId].color = colorId;
                 break;// need break, otherwise it continue loop to numNodes
             }
             outFile <<nodeId << "     " << colorId << endl;
         }
          nodeId = getUncoloredNode(nodeColor);
     }
 }


 /*  void method2(Node* nodeColor, ofstream& outFile){
        outFile << "Method2 Degug Output: " << endl;
        outFile << "Node_Id" << " " << " Node_Color" << endl;
        nodeColor[1].color = 1;
        usedColor[1] = 1;
        int colorId=1;
        int nodeId;
        bool OK = true;
        for(int i=2; i<=numNodes; i++){ //loop through from 2nd node to last node
            nodeId = i;
                for(int m=1; m<=colorId; m++){ //loop through the used color
                    usedColor[m] = m;
                    OK = checkNeigbhWithoutColor(nodeId, usedColor[m], nodeColor);
                    if(OK) {
                        nodeColor[nodeId].color = usedColor[m];
                        break;
                    }
                }
                if(!OK) {
                    nodeColor[nodeId].color = ++colorId;
                }
                outFile << nodeId << "       " << colorId << endl;
            }
        outFile << "Node_Id" << "  " << "Node_Color" << endl;
        for (int i = 1; i <= numNodes; i++) {
            outFile << nodeColor[i].nodeId << "        " << nodeColor[i].color << 
endl;
        }
    }
*/

  void method2(Node* nodeColor, ofstream& outFile){
      outFile << "Method2 DegugOutput: " << endl;
      outFile << "Node_Id" << " " << " Node_Color" << endl;
      nodeColor[1].color = 1;
      usedColor[1] = 1;
      int colorId=1;
```



```
      int nodeId;
      nodeId = getUncoloredNode(nodeColor);
      bool OK;
      while(!allNodeColored(nodeColor)){

              for(int m=1; m<=colorId; m++){ //loop through the used color
                  usedColor[m] = m;
                  OK = checkNeigbhWithoutColor(nodeId, usedColor[m], nodeColor);
                  if(OK) {
                      nodeColor[nodeId].color = usedColor[m];//the current node can be
                      break;
                  }
              }
              if(!OK){ //If OK not true, the current node's neighbor have this color,
I need to get a new color
                  nodeColor[nodeId].color = ++colorId;//
              }
              outFile << nodeId << "       " << colorId << endl;

          nodeId = getUncoloredNode(nodeColor);
      }

     outFile << "Node_Id1" << "  " << "Node_Color1" << endl;
     for (int i = 1; i <= numNodes; i++) {
         outFile << nodeColor[i].nodeId << "        " << nodeColor[i].color << endl;
     }
  }

    bool checkNeigbhWithoutColor(int nodeId, int colorId, Node* nodeColor){

        for(int i=1; i<=numNodes; i++){
            if(adjMatrix[nodeId][i]==1 && nodeColor[i].color==colorId){
                    return false;
            }
        }
        return true;
    }

    int getUncoloredNode(Node* nodeColor){
        for(int i=1; i<=numNodes; i++){
            if(nodeColor[i].color < 0) {
                return i;
            }
        }
        return -1;
    }

    bool allNodeColored(Node* nodeColor){
        for(int i=1; i<=numNodes; i++){
            if(nodeColor[i].color < 0){ //the default value is -1
                return false;
            }
        }
        return true;
    }

    void printMatrix(Node* nodeColor, ofstream& outFile){
        outFile<<"Graph adjMatrix: " << endl;
        for(int i=0; i<=numNodes; i++){
            for(int j=0; j<=numNodes; j++){
                outFile << adjMatrix[i][j] << " ";
```



```
            }
            outFile << endl;
        }
    }

    void printColorAssignment(ofstream& outFile, Node* nodeColor){
        outFile << "Node_Id" << "  " << "Node_Color" << endl;
        for (int i = 1; i <= numNodes; i++) {
            outFile << nodeColor[i].nodeId << "        " << nodeColor[i].color <<
endl;
        }
  }
};


int main(int argc, char *argv[]) {

    ifstream inFile(argv[1]);

    int whichMethod = stoi(argv[2]); //1-method1 2-method2

    ofstream outFile1(argv[3]); // the output of the color assignments of nodes in the
graph
    ofstream outFile2(argv[4]); // output the content of your data structure of the
graph
    ofstream outFile3(argv[5]); //degugging prints
//
    int numNodes = 0;
    inFile >> numNodes;

    GraphColoring *graphcolor = new GraphColoring(numNodes);

    Node* nodeColor;
    nodeColor = new Node[numNodes+1];
    for(int i=0; i<=numNodes; i++){
        nodeColor[i].nodeId = i;
        nodeColor[i].color = -1;
    }

    outFile2 << "Data Structure: " << endl;
    outFile2 <<"Initial nodeColor: " << endl;
    outFile2 << "NodeId" << "  " << "Color" << endl;
    for(int i=1; i<=numNodes; i++){
        outFile2 << i << "      " << nodeColor[i].color << endl;
    }
    outFile2 << endl;

    outFile1 << "NumNodes: " <<  numNodes << endl;

    graphcolor->loadMatrix(inFile);

    if(whichMethod==1){
        graphcolor->method1(nodeColor, outFile3);
    }else{
        graphcolor->method2(nodeColor, outFile3);
    }

    graphcolor->printMatrix(nodeColor, outFile2);
    graphcolor->printColorAssignment(outFile1, nodeColor);

    inFile.close();
    outFile1.close();
    outFile2.close();
    outFile3.close();
    return 0;
}
```

method-1 on data1

```
NumNodes: 8
Node_Id  Node_Color
1        1
2        2
3        3
4        4
5        5
6        1
7        1
8        2
```

method-2 on data1

```
NumNodes: 8
Node_Id  Node_Color
1        1
2        2
3        3
4        4
5        5
6        1
7        1
8        2
```

method-1 on data2

```
NumNodes: 10
Node_Id  Node_Color
1        1
2        1
3        2
4        2
5        2
6        3
7        3
8        3
9        4
10        4
```

method-2 on data2

```
 NumNodes: 10
Node_Id  Node_Color
1       1
2       1
3       2
4       2
5       2
6       3
7       3
8       3
9       4
10       4
```

method-1 on data3

```
 NumNodes: 19
Node_Id  Node_Color
1       1
2       1
3       1
4       2
5       2
6       3
7       1
8       1
9       3
10       3
11       3
12       2
13       3
14       1
15       1
16       4
17       2
18       3
19       1
```

method-2 on data3

```
 NumNodes: 19
Node_Id  Node_Color
1       1
2       1
3       1
4       2
5       2
```

6	3
7	1
8	1
9	3
10	3
11	3
12	2
13	3
14	1
15	1
16	4
17	2
18	3
19	1