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

#include "randomgraph.h"
#include "graph.h"
#include "tstack.h"


/**
 *
 * classes to check for k-colorability
 * class TwoColorable uses depth-first search (using stack)
 * to check for 2-colorability
 *
 * class BacktrackColorable uses plain backtracking to
 * check for k-colorability
 *
 * Both classes use integers 1-k for k-coloring,
 * with 0 meaning "not currently colored"
 *
 * Owen Astrachan
 * April 23, 2001
 */

class TwoColorable
{
  public:
    TwoColorable();
    bool isColorable(Graph * g);

  private:

    bool mark(int start);
    int  opposite(int color)          const;

    Graph * myGraph;
    tvector<int> myColors;
};

int TwoColorable::opposite(int color) const
// pre: color = 1 or 2
// post: returns the opposite color (1->2 and 2->1)
{
    if (color == 1) return 2;
    else            return 1;
}

TwoColorable::TwoColorable()
    : myGraph(0),
      myColors(0)
{

}

bool TwoColorable::isColorable(Graph * g)
// post: return true if g is two-colorable, else return false
{
    myGraph = g;
    myColors = tvector<int>(g->vertexSize(),0);
    int k;
    bool retVal = true;
    for(k=0; k < g->vertexSize(); k++) {
        if (myColors[k] == 0) {           // not visited yet?
            if (! mark(k)) {              // mark it and descendants
                retVal = false;
                break;
            }
        }
```



```
    }
    return retVal;
}

bool TwoColorable::mark(int start)
// pre: 0 <= start < myGraph.vertexSize(), vertex start not colored
// post: returns true if component reachable from start is 2-colorable
//       and leaves coloring in myColors
//       return false if myGraph *not* 2-colorable
{
    tstack<string> stack;
    string current;
    tvector<string> adj;
    int k;

    stack.push(myGraph->getName(start));
    myColors[start] = 1;
    while (stack.size() > 0) {
        stack.pop(current);
        int color = myColors[myGraph->getNum(current)];

        myGraph->getAdjacent(current, adj);
        for(k=0; k < adj.size(); k++) {
            int vIndex = myGraph->getNum(adj[k]);
            if (myColors[vIndex] == 0) {
                myColors[vIndex] = opposite(color);
                stack.push(adj[k]);
            }
            else if (myColors[vIndex] != opposite(color)) {
                return false;
            }
        }
    }
    return true;
}


class BacktrackColorable
{
  public:
    BacktrackColorable();
    bool isColorable(Graph * g, int count);

    bool checkColors(Graph * g) const;

  private:

    bool mark(int start, int badColor, tvector<int> & markedVerts);
    void unmark(tvector<int>& markedVerts);
    void mergeToFrom(tvector<int>& to, tvector<int>& from);

    Graph *      myGraph;
    tvector<int> myColors;
    int          myCount;
    int          myHits;
};

BacktrackColorable::BacktrackColorable()
    : myGraph(0),
      myColors(0)
{

}

bool BacktrackColorable::isColorable(Graph * g, int count)
// pre:  2 <= count
// post: returns true if g is count-colorable, else falexs
{
    myGraph = g;
```

```
    myCount = count;
    myColors = tvector<int>(g->vertexSize(),0);
    int k;
    bool retVal = true;
    tvector<int> markedVerts;
    myHits = 0;                  // counts # vertices visited

    for(k=0; k < g->vertexSize(); k++) {
        if (myColors[k] == 0) {                  // not seen yet?
            if (! mark(k,0, markedVerts)) {      // color and check
                retVal = false;
                break;
            }
        }
    }
    cout << "# vertices visited = " << myHits << endl;

    return retVal;
}


void BacktrackColorable::mergeToFrom(tvector<int>& to, tvector<int>& from)
// post: elements of from added to end of to, from is empty
{
    for(int k=0; k < from.size(); k++) {
        to.push_back(from[k]);
    }
    from.clear();
}

bool BacktrackColorable::checkColors(Graph * g) const
// post: return true if coloring is valid, else return false
{
    tvector<string> vertices, adj;
    int k,j;
    g->getVertices(vertices);

    for(j=0; j < vertices.size(); j++) {
        g->getAdjacent(vertices[j], adj);
        for(k=0; k < adj.size(); k++) {
            if (myColors[j] == myColors[g->getNum(adj[k])]) {
                return false;
            }
        }
    }
    return true;
}

void BacktrackColorable::unmark(tvector<int>& markedVerts)
// post: all vertices in markedVerts are cleared/marked as unvisited
{
    int k;
    for(k=0; k < markedVerts.size(); k++) {
        myColors[markedVerts[k]] = 0;
    }
    markedVerts.clear();
}

bool BacktrackColorable::mark(int vIndex, int badColor,
                              tvector<int>& markedVerts)
// pre: 0 <= vIndex < mygraph.vertexSize(),
//      vIndex not yet marked as colored ,
//      badColor is the *only* color vIndex cannot be colored
// post: returns true iff coloring succeeds
//       if true, elements visited during marking of vIndex are pushed onto
//       markedVerts (vIndex is *not* pushed)
//
{
    tvector<string> adj;
```



```
    tvector<int> localMarked;
    int k,color;
    string current = myGraph->getName(vIndex);
    myGraph->getAdjacent(current, adj);
    myHits++;                                    // visited one more vertex

    for(color=1; color <= myCount; color++) {   // check all colors

        if (color == badColor) {   // don't try this one, no good for vIndex
            continue;
        }

        myColors[vIndex] = color;  // mark and then check adjacent vertices
        bool okColor = true;

        for(k=0; k < adj.size(); k++) {
            int aIndex = myGraph->getNum(adj[k]);
            if (myColors[aIndex] == 0) {            // not yet colored

                localMarked.push_back(aIndex);
                if (! mark(aIndex, color, localMarked)) {
                    okColor = false;
                    break;
                }
            }
            else if (myColors[aIndex] == color) {  // marked and conflicts
                okColor = false;
                break;
            }
        }
        if (okColor) {                          // coloring ok so add marked
            mergeToFrom(markedVerts, localMarked);  // vertices to master list
            return true;
        }
        unmark(localMarked);                    // not ok, clear locals
    }
    myColors[vIndex] = 0;
    return false;
}

void doTest(Graph * g)
// post: tests for 2-colorability and 2-5 colorabilty are done for g
{
    TwoColorable color;

    if (color.isColorable(g)) {
        cout << "\tis 2-colorable" << endl;
    }
    else {
        cout << "\tNOT 2-colorable" << endl;
    }

    BacktrackColorable bcolor;

    for(int k=2; k <= 5; k++) {
        if (bcolor.isColorable(g,k)) {
            cout << "\tis " << k << "-colorable " << endl;
            if (! bcolor.checkColors(g)){
                cout << "\t\tbut check failed " << endl;
            }
        }
        else {
            cout << "\tis NOT " << k << "-colorable" << endl;
        }
    }

}
```

```
int main(int argc, char * argv[])
{
    double edgeProb = 0.05;
    int    vertexCount = 25;

    int k=1;
    while (k < argc) {
        string arg = argv[k];
        if (arg == "-v") {
            vertexCount = atoi(argv[k+1]);
            k+=2;
        }
        else if (arg == "-p") {
            edgeProb = atof(argv[k+1]);
            k += 2;
        }
        else {
            cout << "unknown option " << arg << endl;
            k += 1;
        }
    }

    Graph * g = 0;

    g = RandomGraph::makeOddEven(100);
    cout << "odd/even graph" << endl;
    doTest(g);

    cout << "\n---------\nrandom graph" << endl;
    g = RandomGraph::makeGraph(vertexCount, edgeProb);
    doTest(g);

    return 0;
}
```

```
#ifndef _RANDOM_GRAPH_H
#define _RANDOM_GRAPH_H

#include <iostream>
#include <string>
using namespace std;

#include "graph.h"
#include "strutils.h"  // for tostring
#include "randgen.h"


class RandomGraph
{
  public:

    static Graph * makeGraph(int vertexCount, double edgeProb);
    static Graph * makeOddEven(int vertexCount);
};



inline Graph * RandomGraph::makeGraph(int vertexCount, double edgeProb)
{
    Graph * g = new Graph();
    int j,k;
    RandGen rgen;

    for(k=0; k < vertexCount; k++) {
        g->addVertex(tostring(k));
    }
    int count = 0;
    for(j=0; j < vertexCount; j++) {
        for(k=0; k < vertexCount; k++) {
            if (j != k && rgen.RandReal() < edgeProb) {
                g->addEdge(g->getName(j), g->getName(k));
                count++;
            }
        }
    }
    cout << "added " << count << " edges" << endl;

    return g;
}

inline Graph * RandomGraph::makeOddEven(int vertexCount)
{
    Graph * g = new Graph();
    int j,k;

    for(k=0; k < vertexCount; k++) {
        g->addVertex(tostring(k));
    }
    int count = 0;
    for(j=0; j < vertexCount; j++) {
        for(k=0; k < vertexCount; k++) {
            if ( (j % 2) != (k % 2) ) {
                g->addEdge(g->getName(j), g->getName(k));
                count++;
            }
        }
    }
    cout << "added " << count << " edges" << endl;

    return g;
}


#endif
```