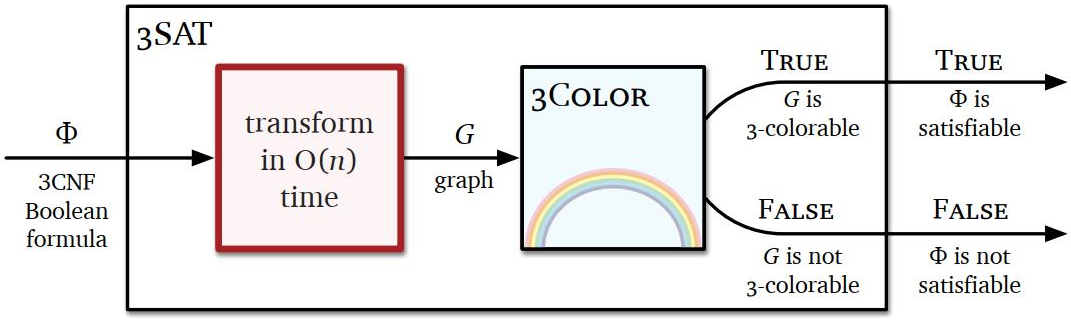
Project Report

Description:

Project 2 in CS: Reduction for NP-Complete problems.

1) The problem I chose is Graph Coloring, I will reduce 3-SAT to 3-Color by programming with C++ in order to prove Graph Coloring is a NP-Complete problem.
2) In this report, I will explain the algorithm of the Reduction first, and show the results of program, and then explain my programs.

Algorithms:



1) The main idea is to transform a random 3CNF Boolean formula to a suitable Graph, such that if the assignment of this boolean formula is True, the corresponding graph is also 3-Colorable.

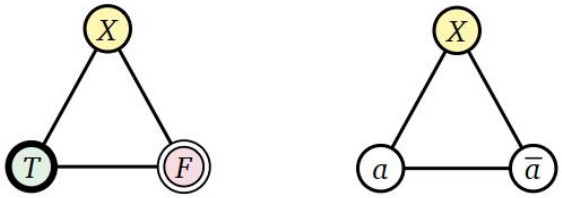


Figure 12.12. The truth gadget and a variable gadget for a.

2) In order to transform a CNF to a Graph, we need to consider 2 gadgets first - Truth gadget and Variable gadget.
3) For the Truth gadget, it is a triangle with 3 vertices T, F, and X, which stand for True, False, and Other respectively.
4) Then for the vertex X, every vertex of variable and its negation must be connected with it. Since the Graph we want to create is 3-Colorable, the vertex of variable can be colored either True or False, and its negation will have the opposite color.

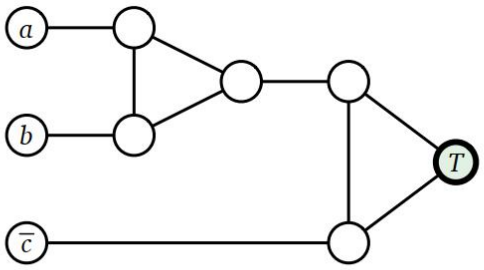


Figure 12.13. A clause gadget for $(a \vee b \vee \bar{c})$.

5) Finally, for each clause in the CNF, we will have more 5 nodes, I will denote them x0, x1, x2, x3, x4, from left to right, from up to down. The variables in the clause will connect to x0, x2, and x4 respectively, x3 and x4 will connect to the vertex T.

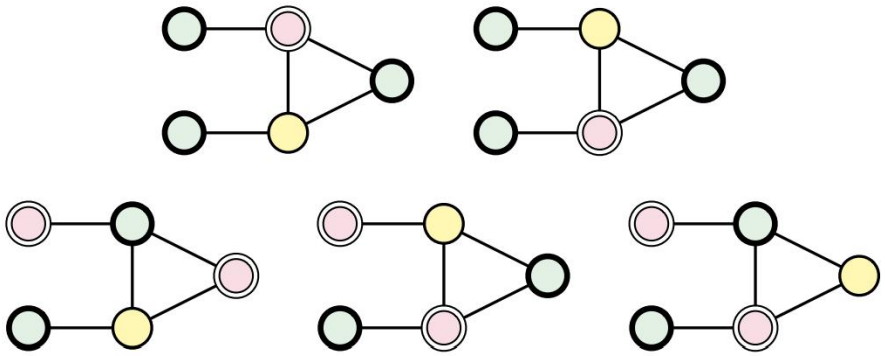


Figure 12.14. All valid 3-colorings of a "half-gadget", up to permutations of the colors

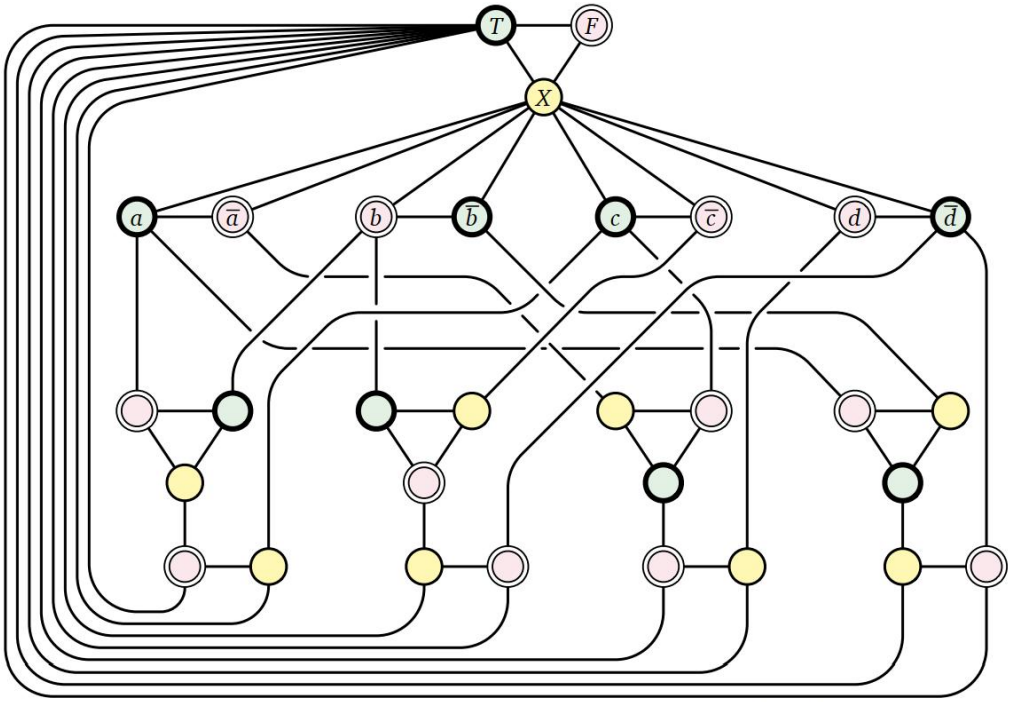
6) For coloring of each node, if the two vertices to the left have the same color, the rightmost vertex of the triangle must have the same color. On the other hand, if the two left vertices have different colors, the color of the right vertex can be chosen arbitrarily (T, F, X).
7) Then, the generated Gadget is 3-Colorable if there is at least one variable of the vertex is colored True. On the other hand, the Gadget is not 3-Colorable only if the input of the 3 variables are all colored False.
8) In conclusion, if we have a CF of m clauses, the corresponding Graph is 3-Colorable if every clause has at least one component is colored with True, which is the assignment of the CF is True.
9) Thus, the corresponding Graph is generated.

Some of my thinking (May not correct):

1) I think for the rightmost vertex in the second situation, it can not choose an arbitrary color except at least in the implementation of programming.
2) It could only choose the color except F (only T, X). Because when first 2 components have different color (T, F respectively) and this vertex get a F, it will has the same effect with "the first 2 components are F". Then if the third component are also F, it will be like 3 F, and the result will be "False" even we have a component of "T".

3) When I include F in the domain of this vertex, the form of (-1 v 1 v -1) are always recognized as False, but once I removed F from its domain, everything works.

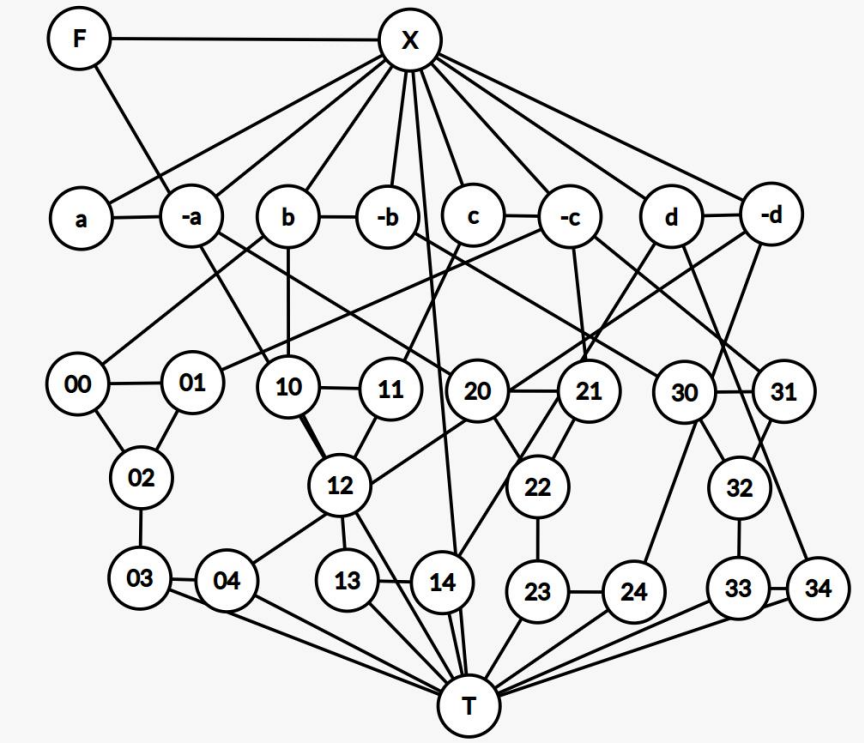
Example:



$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$

Figure 12.15. The 3-colorable graph derived from a satisfiable 3CNF formula.

My Results:



(b v -c v -d) ^ (b v c v d) ^ (-a v -c v -d) ^ (-b v -c v d)

```
xiaoqi@xiaoqi:~/Desktop/Pr$ g++ -o color color.cpp
xiaoqi@xiaoqi:~/Desktop/Pr$ ./color
Random Conjunctive Form:
|  0  1 -1 -1 |
|  0  1  1  1 |
| -1  0 -1 -1 |
|  0 -1 -1  1 |
(b v -c v -d) ^ (b v c v d) ^ (-a v -c v -d) ^ (-b v -c v d)
The assignment of the random 3CNF is: False!

Graph created with number of verteices = 31, and number of edges = 55
A situation adjacent vertices have the same color occurs!
Display the converted Graph from Conjunctive Form:
        Graph Vertices:
         X  T  F  a -a  b -b  c -c  d -d 00 01 02 03 04 10 11 12 13 14 20 21 22
        Vertices Color:
         X  T  F  T  F  T  F  T  F  T  F  T  F  X  F  X  F  X  T  F  X  T  X  F
        Edges:
         X: {(X,T) (X,F) (X,a) (X,-a) (X,b) (X,-b) (X,c) (X,-c) (X,d) (X,-d) }
         T: {(T,F) (T,03) (T,04) (T,13) (T,14) (T,23) (T,24) (T,33) (T,34) }
         F: {}
         a: {(a,-a) }
        -a: {(-a,20) }
         b: {(b,-b) (b,00) (b,10) }
        -b: {(-b,30) }
         c: {(c,-c) (c,11) }
        -c: {(-c,01) (-c,21) (-c,31) }
         d: {(d,-d) (d,14) (d,34) }
        -d: {(-d,04) (-d,24) }
        00: {(00,01) (00,02) }
        01: {(01,02) }
        02: {(02,03) }
        03: {(03,04) }
        04: {}
        10: {(10,11) (10,12) }
        11: {(11,12) }
        12: {(12,13) }
        13: {(13,14) }
        14: {}
        20: {(20,21) (20,22) }
        21: {(21,22) }
        22: {(22,23) }
        23: {(23,24) }
        24: {}
        30: {(30,31) (30,32) }
        31: {(31,32) }
        32: {(32,33) }
        33: {(33,34) }
        34: {}
The assignment of the converted Graph is:
Vertex 24 has the same color with Vertex T
False, it is not 3-Colorable!

Graph destroyed!
xiaoqi@xiaoqi:~/Desktop/Pr$
```

1) The Graph of my result is generated by https://csacademy.com/app/graph_editor/.

2) My program firstly generate a random 3CNF with an assignment. Then, after generation, it shows a corresponding Graph to the CNF with all necessary information: number of vertices, number of edges, name and color of each vertex, edges of each vertex.
3) And finally, it shows a assignment of the Graph by my algorithm such that whether this Graph is 3-Colorable or not, and it will also explain to you the explicit errors with the vertices.

Main Function:

```
int main() {
        //n = number of variables.
        //m = number of clauses.
        //k = length of clauses.
        int n = 4;
        int m = 4;
        int k = 3;
        srand(time(NULL));
        int** kins = randomKSAT(n, m, k);
        int** cf = ksat2CF(n, m, k, kins);

        string alphabet[10] = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j"};
        string** cf_alphabet = convert_cf(cf, n, m, k, alphabet);
        display_cf(cf, n, m, k, cf_alphabet);
        cout << "The assignment of the random 3CNF is: ";
        if (cfAnswer(cf, n, m)) {
                cout << "True!" << endl << endl;
        }
        else {
                cout << "False!" << endl << endl;
        }

        Graph* graph = new Graph(n, m);
```

```
        transformation(n, m, k, cf_alphabet, graph, alphabet);
        graph->display_graph();
        cout << "The assignment of the converted Graph is: " << endl;
        if (checkColor(graph, m)) {
                cout << "True, it is 3-Colorable!" << endl << endl;
        }
        else {
                cout << "False, it is not 3-Colorable!" << endl << endl;
        }

        //Free memories.
        for (int i = 0; i < m; i++) {
                delete[] cf_alphabet[i];
        }
        delete[] cf_alphabet;
        delete graph;
        return 0;
}
```

It did what I explained explicitly above.

Random k-SAT:

```
//Generate a random instance of k-SAT over n variables with m clauses.
int** randomKSAT(int n, int m, int k) {
        //Neccessary memory allocation for 2D array using new.
        int** cf = new int*[m];
        for (int i = 0; i < m; i++) {
                cf[i] = new int[k];
        }
        for (int i = 0; i < m; i++) {
                int* uv = new int[k] (); //Initialized to 0.
                int nuv = 0;
                while (nuv < k) {
                        int v = rand() % n + 1;
                        bool check = false;
                        for (int i = 0; i < k; i++) {
                                if (uv[i] == v) {
                                        check = true;
                                        break;
                                }
                        }
                        if (check == false) {
                                int ran = rand() % 2;
                                cf[i][nuv] = (2 * ran - 1) * v;
                                uv[nuv] = v;
                                nuv++;
                        }
                }
                delete[] uv;
        }
        return cf;
```

Assignment of the k-SAT:

```
//A clause is true if one of its components is true.
bool clauseAnswer(int* clasue, int n) {
        bool tmp = false;
        for (int i = 0; i < n; i++) {
                if (clasue[i] == 1) {
                        tmp = true;
                }
        }
        return tmp;
}

//A conjunctive form is a list of clauses,
//it is true if every clause is true.
bool cfAnswer(int** cf, int n, int m) {
        bool tmp = true;
        for (int i = 0; i < m; i++) {
                if (!clauseAnswer(cf[i], n)) {
                        tmp = false;
                }
        }
        return tmp;
}
```

Struct and Class that represents Vertex and Graph:

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

//A struct that represents a vertex in graph.
typedef struct vertex {
        string name;
        string color;
        int edge_number;
        int* edges;
} Vertex;

//A class that represents an undirected graph.
class Graph {
public:
        int vertex_number;      //Number of vertices.
        int edge_number;        //Number/Maximum of edges.
        Vertex** vertices;      //Each vertex has its individua

        void setName(int v, string name);
        void setColor(int v, string color);
        void addEdge(int v, int w);
        int findVertex(string name);
        string findColor(string name);
        void display_graph();
```

```
        //Constructor of a k-Colorable Graph.
        //n = number of variables, m = number of clauses.
        //vertex_number = 2*n + 5*m + 3.
        //edge_number = 3*n + 10*m + 3.
        Graph(int n, int m) {
                vertex_number = 2*n + 5*m + 3;
                edge_number = 3*n + 10*m + 3;
                vertices = new Vertex*[vertex_number];
                for (int i = 0; i < vertex_number; i++) {
                        vertices[i] = new Vertex;
                        vertices[i]->edge_number = 0;
                        vertices[i]->edges = new int[vertex_number]();
                }
                cout << "Graph created with number of verteices = "<< vertex_number
                        << ", and number of edges = " << edge_number << endl;
        }

        //Destructor of the class, free all memories that have been allocated.
        ~Graph() {
                for (int i = 0; i < vertex_number; i++) {
                        delete[] vertices[i]->edges;
                        delete vertices[i];
                }
                cout << "Graph destroyed!" << endl;
        }
```

Transformation of CF to Graph:

```
//Transformation of a 3CNF to a Graph.
void transformation(int n, int m, int k, string** cf_alphabet, Graph* graph, s
        //Vertices 0 - 2 are T, F, X. T for true, F for false, X for Other
        graph->setName(0, "X");
        graph->setName(1, "T");
        graph->setName(2, "F");
        graph->setColor(0, "X");
        graph->setColor(1, "T");
        graph->setColor(2, "F");
        graph->addEdge(0, 1);
        graph->addEdge(0, 2);
        graph->addEdge(1, 2);

        //Vertices 3 - 2n + 2 for each variables and its negation based on cf.
        for (int i = 2; i < n + 2; i++) {
                graph->setName(2*i - 1, alphabet[i - 2]);
                graph->setName(2*i, "-" + alphabet[i - 2]);
                graph->setColor(2*i - 1, "T");
                graph->setColor(2*i, "F");
                graph->addEdge(2*i - 1, 0);
                graph->addEdge(2*i, 0);
                graph->addEdge(2*i - 1, 2*i);
        }
```

```
        //m clauses, each clauses need 5 nodes: x1 - x5.
        //Vertices 2n + 3 - 2n + 3 + 5m.
        for (int i = 0; i < m; i++) {
                for (int j = 0; j < 5; j++) {
                        graph->setName(2*n + 3 + 5*i + j, to_string(i) + to_string(j));
                }
        }

        //A clause a v b v -c:
        //first 2 elements in the clause have the same color,
        //means x2 have the same color,
        //otherwise, x2 choose an arbitrary color.
        for (int i = 0; i < m; i++) {
                //For nodes x0, x1, x2.
                if (cf_alphabet[i][0].length() == cf_alphabet[i][1].length()) {
                        graph->setColor(graph->findVertex(to_string(i) + to_string(2)), g
                }
                else {
                        graph->setColor(graph->findVertex(to_string(i) + to_string(2)), r
                }
                graph->setColor(graph->findVertex(to_string(i) + to_string(0)), diffColor
                graph->setColor(graph->findVertex(to_string(i) + to_string(1)), diffColor
                graph->addEdge(graph->findVertex(cf_alphabet[i][0]), graph->findVertex(to
                graph->addEdge(graph->findVertex(cf_alphabet[i][1]), graph->findVertex(to
                graph->addEdge(graph->findVertex(to_string(i) + to_string(0)), graph->fin
                graph->addEdge(graph->findVertex(to_string(i) + to_string(0)), graph->fin
                graph->addEdge(graph->findVertex(to_string(i) + to_string(1)), graph->fin
```

Space are not enough here, you can see it in color.cpp.

```
                //For nodes x3, x4.
                //Try to let x4 be the same color with x2, in order to let x3 choose
                if (graph->findColor(to_string(i) + to_string(2)) != graph->findColo
                graph->setColor(graph->findVertex(to_string(i) + to_string(4)), grap
                }
                else {
                        graph->setColor(graph->findVertex(to_string(i) + to_string(4
                }
                graph->setColor(graph->findVertex(to_string(i) + to_string(3)), diff
                graph->addEdge(graph->findVertex(cf_alphabet[i][2]), graph->findVert
                graph->addEdge(graph->findVertex(to_string(i) + to_string(3)), graph
                graph->addEdge(graph->findVertex(to_string(i) + to_string(3)), graph
                graph->addEdge(graph->findVertex(to_string(i) + to_string(3)), 1);
                graph->addEdge(graph->findVertex(to_string(i) + to_string(4)), 1);
        }
}
```

Assignment of Graph:

```
bool checkColor(Graph* graph, int m) {
        bool tmp = true;
        for (int i = 0; i < m; i++) {
                if (graph->findColor(to_string(i) + to_string(3)) == graph->findColor(to_strin
                        tmp = false;
                        cout << "Vertex " << i << 3 << " has the same color with Vertex " << i
                }
                if (graph->findColor(to_string(i) + to_string(3)) == "T") {
                        tmp = false;
                        cout << "Vertex " << i << 3 << " has the same color with Vertex T" <<
                }
                if (graph->findColor(to_string(i) + to_string(4)) == "T") {
                        tmp = false;
                        cout << "Vertex " << i << 4 << " has the same color with Vertex T" <<
                }
        }
        return tmp;
}
```

There will only be errors (Situations that adjacent vertices have the same color) with x3 and x4 with the vertex T because the algorithm is just designed like this (Every time I set color of a vertex, I will set the color which are different with the color of its adjacent vertices. So we only need to check this part.

Compile (run.sh):

Since it's just a simple program written by C++, you can simply compile it using:
"g++ -o color color.cpp", "./color".
I also write a script program "run.sh" which will do exactly the commands above.

Other Codes:

```
void Graph::setName(int v, string name) {
        vertices[v]->name = name;
}

void Graph::setColor(int v, string color) {
        vertices[v]->color = color;
}

void Graph::addEdge(int v, int w) {
        vertices[v]->edges[w] = 1;
        vertices[w]->edges[v] = 1;
        vertices[v]->edge_number++;
        vertices[w]->edge_number++;
}

int Graph::findVertex(string name) {
        for (int i = 0; i < vertex_number; i++) {
                if (vertices[i]->name == name) {
                        return i;
                }
        }
        cout << "Find vertex failed! Program aborted!" << endl;
        exit(-1);
}
```

```
string Graph::findColor(string name) {
        for (int i = 0; i < vertex_number; i++) {
                if (vertices[i]->name == name) {
                        return vertices[i]->color;
                }
        }
        cout << "Find color failed! Program aborted!" << endl;
        exit(-1);
}

void Graph::display_graph() {
        cout << "Display the converted Graph from Conjunctive Form: " << endl;
        cout << "        Graph Vertices: " << endl << "        ";
        for (int i = 0; i < vertex_number; i++) {
                if (vertices[i]->name.length() == 1) {
                        cout << " ";
                }
                cout << vertices[i]->name << " ";
        }
        cout << endl;

        cout << "        Vertices Color: " << endl << "        ";
        for (int i = 0; i < vertex_number; i++) {
                cout << " " <<  vertices[i]->color << " ";
        }
        cout << endl;

        cout << "        Edges: " << endl;
        for (int i = 0; i < vertex_number; i++) {
```

```
                if(vertices[i]->name.length() == 1) {
                        cout << " ";
                }
                cout << "        " << vertices[i]->name << ": {";
                for (int j = i; j < vertex_number; j++) {
                        if (vertices[i]->edges[j] == 1) {
                                cout << "(" << vertices[i]->name << "," << vertices[j
                        }
                }
                cout << "}" << endl;
        }
}
```

```
//Convert a random instance of KSAT in the conjunctive form
//representation given in the previous function randomKSAT().
int** ksat2CF(int n, int m, int k, int** kcf) {
        //m is the length of the row of kcf.
        //k is the length of the line of kcf.
        //n is the length of the line we want to convert kcf to.
        int** cf = new int*[m];
        for (int i = 0; i < m; i++) {
                cf[i] = new int[n] ();
        }
        for (int i = 0; i < m; i++) {
                for (int j = 0; j < k; j++) {
                        int v = abs(kcf[i][j]);
                        int sg = kcf[i][j]/v;
                        cf[i][v - 1] = sg;
                }
        }
        return cf;
}

//Display the random CF we created.
void display_cf(int** cf, int n, int m, int k, string** cf_alphabet) {
        cout << "Random Conjunctive Form: " << endl;
        for (int i = 0; i < m; i++) {
                cout << "| ";
                for (int j = 0; j < n; j++) {
                        printf("%2d ", cf[i][j]);
                }
                cout << "|" << endl;
        }

        for (int i = 0; i < m; i++) {
                cout << "(";
                for (int j = 0; j < k; j++) {
                        cout << cf_alphabet[i][j];
                        if (j != k - 1) {
                                cout << " v ";
                        }
                }
                cout << ")";
                if (i != m - 1) {
                        cout << " ^ ";
                }
        }
        cout << endl;
}
```

```
string** convert_cf(int** cf, int n, int m, int k, string* alphabet) {
        string** tmp = new string*[m];
        for (int i = 0; i < m; i++) {
                tmp[i] = new string[k];
        }
        int cursor = 0;
        for (int i = 0; i < m; i++) {
                for (int j = 0; j < n; j++) {
                        if (cf[i][j] == 0) {
                                continue;
                        }
                        else if (cf[i][j] == 1) {
                                tmp[i][cursor++] = alphabet[j];
                        }
                        else if (cf[i][j] == -1) {
                                tmp[i][cursor++] = "-" + alphabet[j];
                        }
                }
                cursor = 0;
        }
        return tmp;
}
```

```
//Return a random color except "F" for nodes x2 when first 2
//components have different color, if it returns a "F" for x2,
//it will has the same effect with "the first 2 components are "F",
//then if the third component are also "F", it will be like 3 "F",
//and the result will be "False" even we have a component of "T".
string randomColor() {
        int tmp = rand() % 2;
        if (tmp == 0) {
                return "X";
        }
        else {
                return "T";
        }
}

string diffColor1(string color) {
        int tmp = rand() % 2;
        if (color == "X") {
                if (tmp == 0) {
                        return "T";
                }
                else {
                        return "F";
                }
        }
        else if (color == "T") {
                if (tmp == 0) {
                        return "X";
```

```
                if (tmp == 0) {
                        return "X";
                }
                else {
                        return "F";
                }
        }
        else {
                if (tmp == 0) {
                        return "X";
                }
                else {
                        return "T";
                }
        }
}

string diffColor2(string color1, string color2) {
        if (color1 == "X" && color2 == "T") {
                return "F";
        }
        else if (color1 == "T" && color2 == "X") {
                return "F";
        }
        else if (color1 == "X" && color2 == "F") {
                return "T";
        }
        else if (color1 == "F" && color2 == "X") {
                return "T";
```

```
        else if (color1 == "T" && color2 == "F") {
                return "X";
        }
        else if (color1 == "F" && color2 == "T") {
                return "X";
        }
        else if (color1 == "T" && color2 == "T") {
                return diffColor1("T");
        }
        else if (color1 == "F" && color2 == "F") {
                return diffColor1("F");
        }
        else if (color1 == "X" && color2 == "X") {
                return diffColor1("X");
        }
        else {
                cout << "diffColor2() failed!" << endl;
                return "T";
        }
}
```

```
string diffColor3(string color1, string color2, string color3) {
        if (color1 == color2) {
                return diffColor2(color1, color3);
        }
        else if (color1 == color3) {
                return diffColor2(color1, color2);
        }
        else if (color2 == color3) {
                return diffColor2(color1, color2);
        }
        else {
                cout << "A situation adjacent vertices have the same color occurs!"
                return "T";
        }
}
```

That’s the end of this report, thank you very much for your attention!
Xiaoqi LIU 999009335