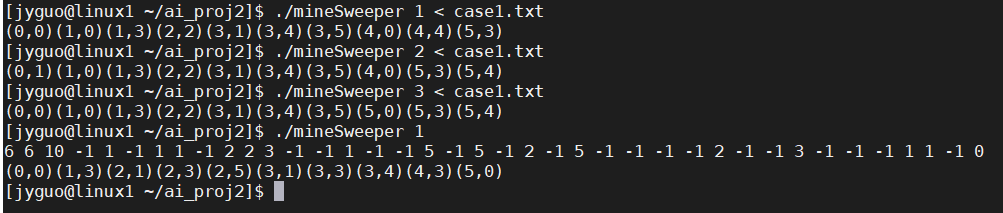
## Programming Assignment #2

0612129 郭家佑

## Demo:



This is a simple demo, and the parameter in the argv is stand for different kind of heuristics using.

1 -> MRV

2 -> Degree heuristic

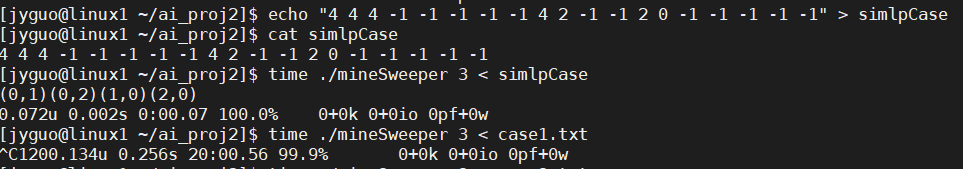
3 -> LCV

After inputting the constrain, the screen would print the position of the mines. However, if the solution is not existed, it would try almost all of possible and print nothing.

## Detail inside:

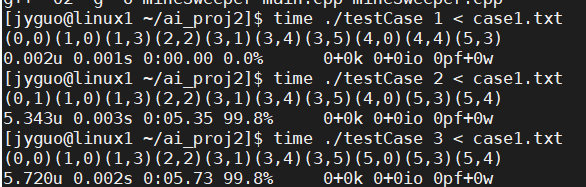
First, I would take a look of LCV, unlike MRV and degree heuristic, it is a heuristic to choose what value should assign to the variable first. So that it need to compute all of the case (in the default case, is 16\*2) initially and hard to choose a good value to avoid waste. In example 1 of the assignment given, if the (0, 0) and (0, 1) both are regarded as mines due to the domain of other variable is not changed, but the system would try a lot of situation and it would know that (0, 0) and (0, 1) couldn’t having same value at last.

In the picture, we can find out it can solve the small case. But it need to spend more than 20 min on the example 1.



However, the LCV is not need to be used barely. It can combine with MRV and LCV, that is, MRV mark those variable only have one value in its domain, and using LCV to choose what value should choose from remain variable witch have two value in its domain.

And this picture it case using LCV with MRV (with parameter 3), compare to other kind of heuristics.



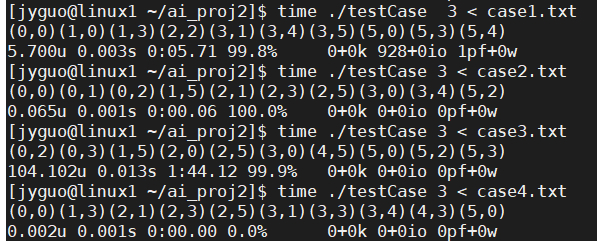
Sadly, LCV+MRV(in the following of report, I would use LCV to represent MRV+ LCV ) is also slower than barely using MRV and I think it is due to the LCV need large computation to choose a point is mine or not, but it isn’t such useful.

Anyway, I using “#ifndef” and “#ifdef” in “mineSweeper.cpp” to parted two kind of LCV and you can comment the line “#define LCV\_WITH\_MRV 1” in “minesweeper.hpp”.

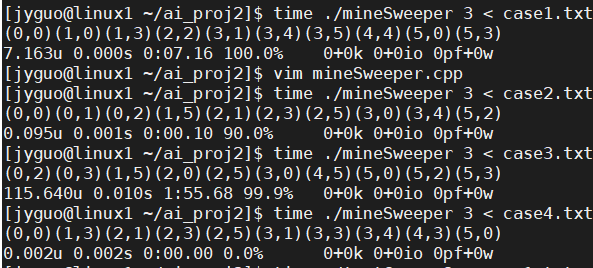
Next, in the program, I using the forward checking in usual. In “mineSweeper.cpp”, there are two function, “testMidCondition” and “testEndCondition”, and the “testMidCondition” used for checking the mines position is true or not for all of constrain, before the state that mines in the map are not all flagged.

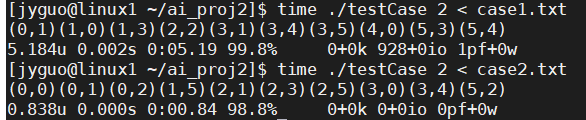
Otherwise, there are some checking in the program, like checking how many of mine in the table. And, in the MRV case, the function similar to testMidCondition is combined when choosing variable to assign. So that the MRV without forwarding checking is a bit hard to do. So that I checking the LCV without forwarding checking.

The first diagram is the normal case named as” testCase”.



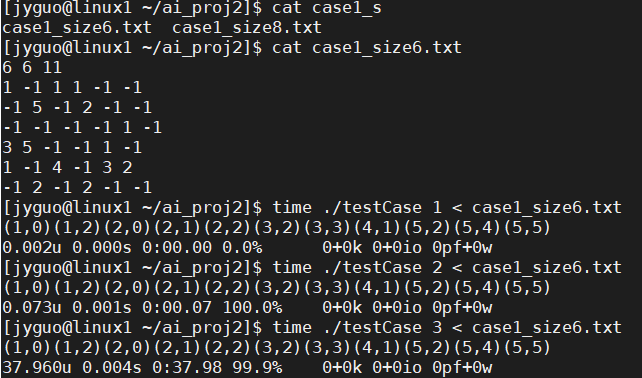
And the second one is commented the “testMidCondition” function.



Obviously, the case with the forward checking is more quick in most case.  
  
Similar test of degree heuristic. The different being more obviously.



Test in different size, first, I test 6\*6 matrix which made from “ramdon.c”. In fact, the position set as a mine with 25%, and it could be more or less.

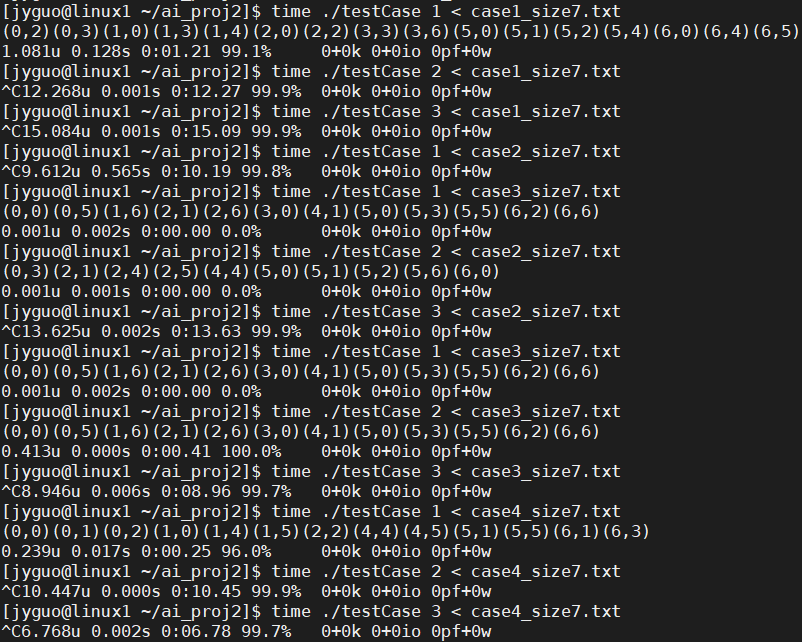


In severe case, the output is similar to example case given.

Then testing about 7\*7 size, sadly the LCV spending a long time to run. And the other case both can finish in few sec. I tested 10 random case, and the result showing that MRV finished 7 case in few sec, degree heuristic finish 5 case in few sec and LCV only finish 1 case in few second. About 8\*8 size, in 10 cases, the MRV only finish 2 case and the degree heuristic only finish 4 case.

About 9\*9 size, in 10 cases, all of function can’t finish any case in 30 sec.

The following picture is part of test.



## Algorithms and heuristic

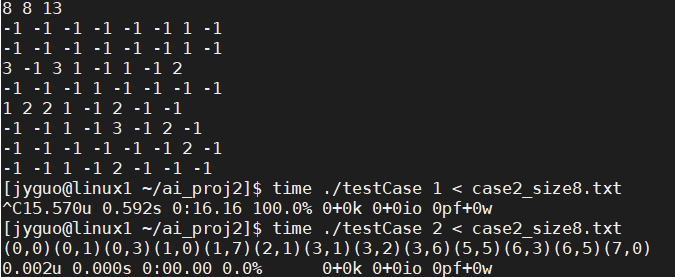
First, if I don’t use any heuristic to backtracking, the complexity is O(2^n) which n is variable number, in the assignment assigned as -1.

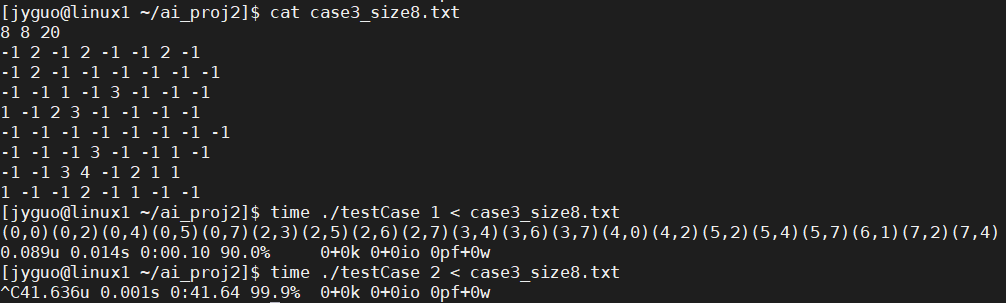
Although I am not sure the precise complexity with heuristic, different heuristic also show that they suit in different case.

MRV suits to solve the constrains in case which are dispersed, specially about the case whose variable’s domains are singletons.

And the degree heuristic suit to solve the concentrated constrain like the example 3 which constrain concentrated in center.

There are two 8\*8 case solved in the following picture. And they show their convenience on the situation.





## remaining questions and future investigation

Few day ago, I test some case with MRV and get the bad\_alloc() in few second. But until today, I don’t have the problem anymore and I am not sure why. And I also wonder the server I use to test those case is not such stable and the “time” command is not such reliable and produce enough information. But I think the virtue machine could be less stable and I don’t know what kind of data should be collected and what is useful tool in window system.

At lease, I know there are perf and gprof command in linux.

About my program, I using stack and also using recursion to finish this assignment in the SAME time. So that it meaning that I spend some space and time on not necessary part. But I don’t think I can change code in a short time and the function is complete. And it is part of future investigation about how to make the code better maybe shorter or faster.

## Appendix

Now I know that VS code can print the text with color.

Makefile:

all: main.cpp mineSweeper.cpp

    g++ -O2 -g -o mineSweeper main.cpp mineSweeper.cpp

minesweeper.cpp

#include "mineSweeper.hpp"

extern int board\_size\_x, board\_size\_y;

extern int variable\_num;

extern int mines\_num;

int runTime;

void func(vector<vector<int>>& map, int mode){

    //using stack to store different state with vector STL

    stack<vector<position\_state>> state;

    stack<vector<position\_state>>& state\_ref = state;

    //first state to put into stack

    vector<position\_state> first;

    vector<position\_state>& first\_ref = first;

    //making a lookup table to let me know how the 2D map's position state

    //store in the linear vector, if there are no correspond part, store

    //with -1.

    vector<vector<int>> lookup = map;

    vector<vector<int>> &lookup\_ref = lookup;

    int index = 0;

    for(int i=0; i < board\_size\_x; i++){

        for(int j=0; j< board\_size\_y; j++){

            if(lookup[i][j] == -1) lookup[i][j] = index++;

            else lookup[i][j] = -1;

        }

    }

    //complete the first state

    for(int i=0; i < board\_size\_x; i++){

        for(int j=0; j< board\_size\_y; j++){

            if(map[i][j] == -1) {

                position\_state tmp(i, j);

                first.push\_back(tmp);

            }

        }

    }

    state.push(first);

    //using different heuristic function with parameter

    switch(mode){

        case 1:

            MRV(map, state\_ref, lookup\_ref);

            break;

        case 2:

            DH(map, state\_ref, lookup\_ref);

            break;

        case 3:

            LCV(map, state\_ref, lookup\_ref);

            break;

        default:

            break;

    }

    //if the state return successfully, print all of position of mine.

    //In the loop, if there are no answer, it would print nothing.

    for(int i = 0; i<variable\_num; i++){

        if(state.top()[i].state == isMine){

            cout << "(" << state.top()[i].x << "," << state.top()[i].y << ")";

        }

    }

    cout << endl;

}

int MRV(vector<vector<int>>& map,

        stack<vector<position\_state>>& state,

        vector<vector<int>>& lookup){

    //copy the newest state

    vector<position\_state> priority;

    priority.assign(state.top().begin(), state.top().end());

    //To check what position only have one domain, that is,

    //it must be mine or nothing in the position.

    //The reflect flag to make sure the state would different

    //between the last state.

    int reflesh\_flag = 0;

    for(int i=0; i<board\_size\_x; i++){

        for(int j=0; j<board\_size\_y; j++){

            if(map[i][j] == -1) continue;

            int notSure\_num = 0;

            int isMine\_num = 0;

            for(int test=0; test<8; test++){

                int test\_x = i + surround\_x[test];

                int test\_y = j + surround\_y[test];

                if(test\_x >= 0 && test\_x < board\_size\_x &&

                        test\_y >= 0 && test\_y < board\_size\_y &&

                        map[test\_x][test\_y] == -1

                  ){

                    if(priority[lookup[test\_x][test\_y]].state == notSure) notSure\_num++;

                    else if(priority[lookup[test\_x][test\_y]].state == isMine) isMine\_num++;

                }

            }

            if(map[i][j] - isMine\_num == 0){

                for(int test=0; test<8; test++){

                    int test\_x = i + surround\_x[test];

                    int test\_y = j + surround\_y[test];

                    if(test\_x >= 0 && test\_x < board\_size\_x &&

                            test\_y >= 0 && test\_y < board\_size\_y &&

                            map[test\_x][test\_y] == -1

                      ){

                        if(priority[lookup[test\_x][test\_y]].state == notSure){

                            priority[lookup[test\_x][test\_y]].state = isntMine;

                            reflesh\_flag = 1;

                        }

                    }

                }

            }else if(map[i][j] - isMine\_num == notSure\_num){

                for(int test=0; test<8; test++){

                    int test\_x = i + surround\_x[test];

                    int test\_y = j + surround\_y[test];

                    if(test\_x >= 0 && test\_x < board\_size\_x &&

                            test\_y >= 0 && test\_y < board\_size\_y &&

                            map[test\_x][test\_y] == -1

                      ){

                        if(priority[lookup[test\_x][test\_y]].state == notSure) {

                            priority[lookup[test\_x][test\_y]].state = isMine;

                            reflesh\_flag = 1;

                        }

                    }

                }

            }else if(map[i][j] - isMine\_num < 0){

                state.pop();

                return 0;

            }

        }

    }

    //Forward checking

    int isMine\_num = 0;

    for(int i=0; i<variable\_num; i++){

        if(priority[i].state == isMine) isMine\_num++;

    }

    if(isMine\_num == mines\_num){

        if(!testEndCondition(map, priority, lookup)){

            state.pop();

            return 0;

        }

        state.push(priority);

        return 1;

    } else if(isMine\_num > mines\_num){

        state.pop();

        return 0;

    }

    //if the new state is same as the last one,

    //choose ramdon position as mine due to the

    //remind positiin's domain is all 1 and 0,

    //I using the "notSure" state to represent.

    if(reflesh\_flag){

        state.push(priority);

        if(MRV(map, state, lookup)) return 1;

    }else{

        for(int i=0; i<variable\_num; i++){

            if(priority[i].state == notSure){

                priority[i].state = isMine;

                state.push(priority);

                if(MRV(map, state, lookup)){

                    return 1;

                }else {

                    priority[i].state = notSure;

                }

            }

        }

    }

    return 0;

}

int DH(vector<vector<int>>& map,

        stack<vector<position\_state>>& state,

        vector<vector<int>>& lookup){

    //Full name is degree heuristic.

    //in the function, choosing the position has most constrains.

    //degree isnt represent the new state but different degreee,

    //that is, how much constrain effect the point.

    vector<position\_state> degree;

    degree.assign(state.top().begin(), state.top().end());

    vector<position\_state>& test\_state = degree;

    //forward checking first

    int isMine\_num = 0;

    for(int i=0; i<variable\_num; i++){

        if(degree[i].state == isMine) isMine\_num++;

    }

    if(isMine\_num > mines\_num){

        return 0;

    }else if(isMine\_num == mines\_num){

        if(testEndCondition(map, test\_state, lookup)) return 1;

        else return 0;

    }else {

        if(testMidCondition(map, test\_state, lookup) == 0) return 0;

    }

    //compute the constrain in different position

    //And if there are 0 constrain, regrad as a point

    //almost impossible to be a mine.

    for(int i=0; i<board\_size\_x; i++){

        for(int j=0; j<board\_size\_y; j++){

            if(map[i][j] != -1){

                for(int test = 0; test < 8; test++){

                    int test\_x = i + surround\_x[test];

                    int test\_y = j + surround\_y[test];

                    if(test\_x >= 0 && test\_x < board\_size\_x &&

                            test\_y >= 0 && test\_y < board\_size\_y &&

                            map[test\_x][test\_y] == -1){

                        degree[lookup[test\_x][test\_y]].value += map[i][j];

                        if(map[i][j] == 0) degree[lookup[test\_x][test\_y]].value = INT\_MIN;

                    }

                }

            }

        }

    }

    //using heap to choose position effected by most constrain.

    make\_heap(degree.begin(), degree.end(), stateHeapMax());

    //Try the position with the heap one by one.

    vector<position\_state> new\_state = state.top();

    new\_state.assign(state.top().begin(), state.top().end());

    while(!degree.empty()){

        position\_state tmp = degree.front();

        if(tmp.state != isMine){

            mine\_state original\_state = new\_state[lookup[tmp.x][tmp.y]].state;

            new\_state[lookup[tmp.x][tmp.y]].state = isMine;

            state.push(new\_state);

            if(DH(map, state, lookup)){

                return 1;

            }else{

                new\_state[lookup[tmp.x][tmp.y]].state = original\_state;

                state.pop();

            }

        }

        pop\_heap(degree.begin(),degree.end(), stateHeapMax()); degree.pop\_back();

    }

    return 0;

}

int LCV(vector<vector<int>>& map,

        stack<vector<position\_state>>& state,

        vector<vector<int>>& lookup){

    vector<position\_state> corrent\_state;

    vector<position\_state>& test\_state = corrent\_state;

    corrent\_state.assign(state.top().begin(), state.top().end());

    //forward checking

    int isMine\_num = 0;

    for(int i=0; i<variable\_num; i++){

        if(corrent\_state[i].state == isMine) isMine\_num++;

    }

    if(isMine\_num > mines\_num){

        return 0;

    }else if(isMine\_num == mines\_num){

        if(testEndCondition(map, test\_state, lookup)) return 1;

        else return 0;

    }else {

        if(testMidCondition(map, test\_state, lookup) == 0) return 0;

    }

    //Compute the upper bound and low bound on constrain

    vector<vector<int>> upperbound(board\_size\_x, vector<int>(board\_size\_y));

    vector<vector<int>> lowerbound(board\_size\_x, vector<int>(board\_size\_y));

    for(int i=0; i<board\_size\_x; i++){

        for(int j=0; j<board\_size\_y; j++){

            if(map[i][j] == -1) continue;

            int notSure\_num = 0;

            int isMine\_num = 0;

            for(int test=0; test<8; test++){

                int test\_x = i + surround\_x[test];

                int test\_y = j + surround\_y[test];

                if(test\_x >= 0 && test\_x < board\_size\_x &&

                        test\_y >= 0 && test\_y < board\_size\_y &&

                        map[test\_x][test\_y] == -1

                  ){

                    if(corrent\_state[lookup[test\_x][test\_y]].state == notSure) notSure\_num++;

                    else if(corrent\_state[lookup[test\_x][test\_y]].state == isMine) isMine\_num++;

                }

            }

            upperbound[i][j] = isMine\_num + notSure\_num;

            lowerbound[i][j] = isMine\_num;

        }

    }

    vector<position\_state> priority;

//compute how much do each values effected other

//Parted in different case to test.

    for(int i=0; i<variable\_num; i++){

        if (corrent\_state[i].state != notSure) continue;

        int handle\_x = corrent\_state[i].x;

        int handle\_y = corrent\_state[i].y;

        int effect\_surround\_num\_upper = 0;

        int effect\_surround\_num\_lower = 0;

        mine\_state corrent\_point\_state = notSure;

        //a value in domain change at most effect 5\*5 size of point.

        vector<vector<int>> effect\_to\_isMine (5, vector<int>(5, 0));

        vector<vector<int>> effect\_to\_isntMine (5, vector<int>(5, 0));

//differe kind, comment the line if LCV\_WITH\_MRV in .hpp to use.

//The most important of them is checking

//This is:

//map[test\_x][test\_y] - upperbound[test\_x][test\_y] == -1

//map[test\_x][test\_y] - lowerbound[test\_x][test\_y] == 1

//Two case

#ifndef LCV\_WITH\_MRV

        for(int test = 0; test < 8; test++){

            int test\_x = handle\_x + surround\_x[test];

            int test\_y = handle\_y + surround\_y[test];

            if(test\_x < 0 || test\_x >= board\_size\_x ||

                    test\_y < 0 || test\_y >= board\_size\_y ||

                    map[test\_x][test\_y] == -1) continue;

            if(map[test\_x][test\_y] > upperbound[test\_x][test\_y]){

                corrent\_point\_state = errorCondition;

            }else if(map[test\_x][test\_y] == upperbound[test\_x][test\_y]){

                if(corrent\_point\_state == notSure) corrent\_point\_state = isMine;

                else if(corrent\_point\_state == isntMine) corrent\_point\_state = errorCondition;

            }else if(map[test\_x][test\_y] - upperbound[test\_x][test\_y] == -1){

                for(int test\_block = 0; test\_block<8; test\_block++){

                    int test\_block\_x = test\_x + surround\_x[test\_block];

                    int test\_block\_y = test\_y + surround\_y[test\_block];

                    if(test\_block\_x >= 0 && test\_block\_x < board\_size\_x &&

                            test\_block\_x >= 0 && test\_block\_y < board\_size\_y &&

                            map[test\_block\_x][test\_block\_y] == -1 &&

                            corrent\_state[lookup[test\_block\_x][test\_block\_y]].state == notSure

                      ){

                        effect\_to\_isMine[2 + surround\_x[test] + surround\_x[test\_block]][2 + surround\_y[test] + surround\_y[test\_block]] = 1;

                    }

                }

            }

            if(map[test\_x][test\_y] < lowerbound[test\_x][test\_y]){

                corrent\_point\_state = errorCondition;

            }else if(map[test\_x][test\_y] == lowerbound[test\_x][test\_y]){

                if(corrent\_point\_state == notSure) corrent\_point\_state = isntMine;

                else if(corrent\_point\_state == isMine) corrent\_point\_state = errorCondition;

            }else if(map[test\_x][test\_y] - lowerbound[test\_x][test\_y] == 1){

                for(int test\_block = 0; test\_block<8; test\_block++){

                    int test\_block\_x = test\_x + surround\_x[test\_block];

                    int test\_block\_y = test\_y + surround\_y[test\_block];

                    if(test\_block\_x >= 0 && test\_block\_x < board\_size\_x &&

                            test\_block\_x >= 0 && test\_block\_y < board\_size\_y &&

                            map[test\_block\_x][test\_block\_y] == -1 &&

                            corrent\_state[lookup[test\_block\_x][test\_block\_y]].state == notSure

                      ){

                        effect\_to\_isntMine[2 + surround\_x[test] + surround\_x[test\_block]][2 + surround\_y[test] + surround\_y[test\_block]] = 1;

                    }

                }

            }

            if(corrent\_point\_state == errorCondition) break;

        }

#endif

#ifdef LCV\_WITH\_MRV

        for(int test = 0; test < 8; test++){

            int test\_x = handle\_x + surround\_x[test];

            int test\_y = handle\_y + surround\_y[test];

            if(test\_x < 0 || test\_x >= board\_size\_x ||

                test\_y < 0 || test\_y >= board\_size\_y ||

                map[test\_x][test\_y] == -1) continue;

            if(map[test\_x][test\_y] >= upperbound[test\_x][test\_y]){

                if(corrent\_point\_state == notSure) corrent\_point\_state = isntMine;

                else if(corrent\_point\_state == isMine) corrent\_point\_state = errorCondition;

                if(map[test\_x][test\_y] == upperbound[test\_x][test\_y]){

                    for(int test\_block = 0; test\_block<8; test\_block++){

                        int test\_block\_x = test\_x + surround\_x[test\_block];

                        int test\_block\_y = test\_y + surround\_y[test\_block];

                        if(test\_block\_x >= 0 && test\_block\_x < board\_size\_x &&

                                test\_block\_x >= 0 && test\_block\_y < board\_size\_y &&

                                map[test\_block\_x][test\_block\_y] == -1 &&

                                corrent\_state[lookup[test\_block\_x][test\_block\_y]].state == notSure

                          ){

                            corrent\_state[lookup[test\_block\_x][test\_block\_y]].state = isMine;

                        }

                    }

                }

            }else if(map[test\_x][test\_y] - upperbound[test\_x][test\_y] == -1){

                for(int test\_block = 0; test\_block<8; test\_block++){

                    int test\_block\_x = test\_x + surround\_x[test\_block];

                    int test\_block\_y = test\_y + surround\_y[test\_block];

                    if(test\_block\_x >= 0 && test\_block\_x < board\_size\_x &&

                            test\_block\_x >= 0 && test\_block\_y < board\_size\_y &&

                            map[test\_block\_x][test\_block\_y] == -1 &&

                            corrent\_state[lookup[test\_block\_x][test\_block\_y]].state == notSure

                      ){

                        effect\_to\_isMine[2 + surround\_x[test] + surround\_x[test\_block]][2 + surround\_y[test] + surround\_y[test\_block]];

                    }

                }

            }

            if(map[test\_x][test\_y] <= lowerbound[test\_x][test\_y]){

                if(corrent\_point\_state == notSure) corrent\_point\_state = isMine;

                else if(corrent\_point\_state == isntMine) corrent\_point\_state = errorCondition;

                if(map[test\_x][test\_y] == lowerbound[test\_x][test\_y]){

                    for(int test\_block = 0; test\_block<8; test\_block++){

                        int test\_block\_x = test\_x + surround\_x[test\_block];

                        int test\_block\_y = test\_y + surround\_y[test\_block];

                        if(test\_block\_x >= 0 && test\_block\_x < board\_size\_x &&

                                test\_block\_x >= 0 && test\_block\_y < board\_size\_y &&

                                map[test\_block\_x][test\_block\_y] == -1 &&

                                corrent\_state[lookup[test\_block\_x][test\_block\_y]].state == notSure

                          ){

                            corrent\_state[lookup[test\_block\_x][test\_block\_y]].state = isntMine;

                        }

                    }

                }

            }else if(map[test\_x][test\_y] - lowerbound[test\_x][test\_y] == 1){

                for(int test\_block = 0; test\_block<8; test\_block++){

                    int test\_block\_x = test\_x + surround\_x[test\_block];

                    int test\_block\_y = test\_y + surround\_y[test\_block];

                    if(test\_block\_x >= 0 && test\_block\_x < board\_size\_x &&

                            test\_block\_x >= 0 && test\_block\_y < board\_size\_y &&

                            map[test\_block\_x][test\_block\_y] == -1 &&

                            corrent\_state[lookup[test\_block\_x][test\_block\_y]].state == notSure

                      ){

                        effect\_to\_isntMine[2 + surround\_x[test] + surround\_x[test\_block]][2 + surround\_y[test] + surround\_y[test\_block]];

                    }

                }

            }

            if(corrent\_point\_state == errorCondition) break;

        }

#endif

        for(int m=0; m<5; m++) for(int n=0; n<5; n++) if(effect\_to\_isntMine[m][n]) effect\_surround\_num\_lower++;

        for(int m=0; m<5; m++) for(int n=0; n<5; n++) if(effect\_to\_isMine[m][n]) effect\_surround\_num\_upper++;

        //compute what value need to try first by a heap

        if(corrent\_point\_state == errorCondition) continue;

        else{

            if(corrent\_point\_state == notSure || corrent\_point\_state == isMine){

                position\_state tmp(handle\_x, handle\_y);

                tmp.value = effect\_surround\_num\_lower;

                tmp.state = isMine;

                priority.push\_back(tmp);

            }

            if(corrent\_point\_state == notSure || corrent\_point\_state == isntMine){

                position\_state tmp(handle\_x, handle\_y);

                tmp.value = effect\_surround\_num\_upper;

                tmp.state = isntMine;

                priority.push\_back(tmp);

            }

        }

    }

    //make the heap to make more ramdon

    random\_shuffle ( priority.begin(), priority.end());

    //making heap

    make\_heap(priority.begin(), priority.end(), stateHeapMin());

    //try

    while(!priority.empty()){

        runTime--;

        vector<position\_state> new\_state = corrent\_state;

        if (new\_state[lookup[priority.front().x][priority.front().y]].state == notSure)

            new\_state[lookup[priority.front().x][priority.front().y]] = priority.front();

        pop\_heap(priority.begin(),priority.end(), stateHeapMin()); priority.pop\_back();

        state.push(new\_state);

        if(LCV(map, state, lookup)) return 1;

        else state.pop();

    }

    return 0;

}

//Two funtion to check

//This one to check reaching end case or not

int testEndCondition(vector<vector<int>>& map,

        vector<position\_state>& test\_array,

        vector<vector<int>>& lookup

        ){

    for(int i=0; i<board\_size\_x; i++){

        for(int j=0; j<board\_size\_y; j++){

            if(map[i][j] == -1) continue;

            int isMine\_num = 0;

            for(int test=0; test<8; test++){

                int test\_x = i + surround\_x[test];

                int test\_y = j + surround\_y[test];

                if(test\_x >= 0 && test\_x < board\_size\_x &&

                        test\_y >= 0 && test\_y < board\_size\_y &&

                        map[test\_x][test\_y] == -1

                  ){

                    if(test\_array[lookup[test\_x][test\_y]].state == isMine) isMine\_num++;

                }

            }

            if(map[i][j] !=  isMine\_num) return 0;

        }

    }

    return 1;

}

//This one use for forward checking

int testMidCondition(vector<vector<int>>& map,

        vector<position\_state>& test\_state,

        vector<vector<int>>& lookup

        ){

    for(int i=0; i<board\_size\_x; i++){

        for(int j=0; j<board\_size\_y; j++){

            if(map[i][j] == -1) continue;

            int isMine\_num = 0;

            for(int test=0; test<8; test++){

                int test\_x = i + surround\_x[test];

                int test\_y = j + surround\_y[test];

                if(test\_x >= 0 && test\_x < board\_size\_x &&

                        test\_y >= 0 && test\_y < board\_size\_y &&

                        map[test\_x][test\_y] == -1

                  ){

                    if(test\_state[lookup[test\_x][test\_y]].state == isMine) isMine\_num++;

                }

            }

            if(map[i][j] - isMine\_num < 0){

                return 0;

            }

        }

    }

    return 1;

}

mineSweeper.hpp

#include <iostream>

#include <vector>

#include <stack>

#include <array>

#include <algorithm>

#include <list>

#include <bits/stdc++.h>

#define LCV\_WITH\_MRV 1

using namespace std;

const int surround\_x[8] = {1, 1, 0, -1, -1, -1, 0, 1};

const int surround\_y[8] = {0, 1, 1, 1, 0, -1, -1, -1};

enum mine\_state{

    notSure, isMine, isntMine, errorCondition

};

class position\_state{

    public:

        int x, y;

        int value;

        mine\_state state;

        position\_state(int init\_x, int init\_y){

            state = notSure;

            x = init\_x;

            y = init\_y;

            value = 0;

        }

};

struct stateHeapMax{

    bool operator()(const position\_state& x, const position\_state& y){

        return x.value < y.value;

    }

};

struct stateHeapMin{

    bool operator()(const position\_state& x, const position\_state& y){

        return x.value >= y.value;

    }

};

void func(vector<vector<int>>&, int);

int MRV(vector<vector<int>>&,

        stack<vector<position\_state>>&,

        vector<vector<int>>&

        );

int DH(vector<vector<int>>&,

        stack<vector<position\_state>>&,

        vector<vector<int>>&

        );

int LCV(vector<vector<int>>&,

        stack<vector<position\_state>>&,

        vector<vector<int>>&

        );

int testEndCondition(vector<vector<int>>&,

        vector<position\_state>&,

        vector<vector<int>>&

        );

int testMidCondition(vector<vector<int>>&,

        vector<position\_state>&,

        vector<vector<int>>&

        );

random.c (use for make matrix)

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <time.h>

const int surround\_x[8] = {1, 1, 0, -1, -1, -1, 0, 1};

const int surround\_y[8] = {0, 1, 1, 1, 0, -1, -1, -1};

int main(int argc, char\*\* argv){

    srand( time(NULL) );

    int size\_x = atoi(argv[1]);

    int size\_y = (argc == 3) ? atoi(argv[2]) : size\_x;

    int map[size\_x][size\_y];

    int mine\_num = 0;

    for(int i=0; i<size\_x; i++){

        for(int j=0; j<size\_y; j++){

            int x = rand() % 4;

            if(!x){

                map[i][j] = -1;

                mine\_num++;

            }

            else map[i][j] = 0;

        }

    }

    for(int i=0; i<size\_x; i++){

        for(int j=0; j<size\_y; j++){

            int x = rand() % 2;

            if(!map[i][j] && x){

                for(int test = 0 ; test<8; test++){

                    if(i+surround\_x[test] >= 0 && i+surround\_x[test] < size\_x

                    && j+surround\_y[test] >= 0 && j+surround\_y[test] < size\_y

                    && map[i+surround\_x[test]][j+surround\_y[test]] == -1

                    ) map[i][j]++;

                }

            }

        }

    }

    for(int i=0; i<size\_x; i++){

        for(int j=0; j<size\_y; j++){

            if(!map[i][j]) map[i][j] = -1;

        }

    }

    printf("%d %d %d\n", size\_x, size\_y, mine\_num);

    for(int i=0; i<size\_x; i++){

        for(int j=0; j<size\_y; j++){

            printf("%d ", map[i][j]);

        }

        printf("\n");

    }

    return 0;

}