Introduction to AI Programming Assignment #2 Report

Student ID: 0716085   Name: 賴品樺

Before Implement:

As this assignment demand, we can consider some different ways to solve this Minesweeper problem. Of course, brute-force can be considered. To optimal the performance, the heuristic function should be used. Thus, we have 3 new methods to solve this problem, MRV with forward checking, MRV and Degree Heuristic with forward checking, MRV and Degree Heuristic and LCV with forward checking. We can also ignore forward checking to generate another 3 methods. Before I implement, I noticed that in this problem, because the maximum of domain size of variable is 2, once I do consistency check, MRV will definitely be implemented. Hence, in optimal cases, I only compare the different between using Degree Heuristic or LCV.

Code interpretation:

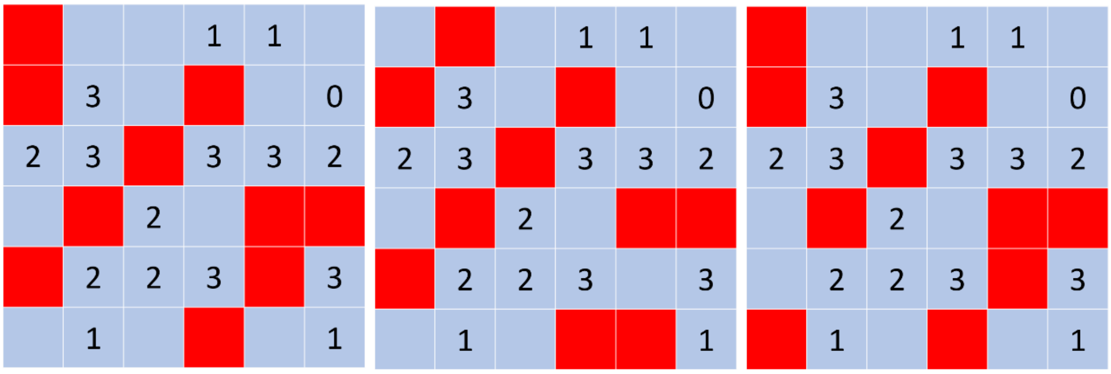
After doing arrangement, I want to compare 3 different methods. There are Brute-Force, MRV and Degree Heuristic, MRV and Degree Heuristic and LCV.
The way I record the node is using the whole Minesweeper board. In the board, it has ***n* * *m*** (board size) structure called ***Variable***.
In the ***Variable*** structure, there has 6 components. ***isVar*** is a boolean variable which records TRUE if this structure is a variable and FALSE if this structure is a hint. ***value*** is an integer which records hint if this structure is a hint and if this structure is a variable, ***value*** = -1 means that this variable is unassigned, ***value*** = 0 means that this variable is safe, ***value*** = 1 means that this variable is the mine. ***degree*** is an integer which records number of constrain the variable associates with. ***position*** is an integer which records the position this variable or hint located, and its' value is coordinate x times ***m*** plus coordinate y. ***mdl, mdu*** are integer that record minimum distance to lower bound of constrain and maximum distance to lower bound of constrain this variable corresponds to.
Constrain is recorded by structure ***Constrain***. In this structure, there has 2 components. ***value*** is an integer which records target sum. ***variable*** is an array of integer that record some position. The ***Constrain*** is satisfied

when ***value*** = SUM{***variable***[i].value} $\forall\ 0 \leqq i \leqq$ size of ***variable***.

No matter which method is, at every iteration, it will pop a node from stack and then do consistency check. If the node can pass the check, it will expand unassigned node. The different between each method is the order that pushing node in stack.
I use 2 value to compare the performance between different methods, ***Frontier_count*** and ***iteration***. ***Frontier_count*** is number of nodes which was expanded. ***iteration*** is how many times the program executes the while loop.
Generally, every iteration will expand number of unassigned variable times 2. However, I noticed that Brute-Force can't easily reach goal state by this way of node expanding. I simplify the way of expanding for Brute-Force so that for every iteration it will only choose one unassigned variable to do expanding.

Testing result:

test case 1:
6 6 10 -1 -1 -1 1 1 -1 -1 3 -1 -1 -1 0 2 3 -1 3 3 2 -1 -1 2 -1 -1 -1 -1 2 2 3 -1 3 -1 1 -1 -1 -1 1



BruteForce, Optimal_MRVDegree, Optimal_MRVDegreeLCV
BruteForce: Frontier_count = 133, Iteration = 123
Optimal_MRVDegree: Frontier_count = 151, Iteration = 63
Optimal_MRVDegreeLCV: Frontier_count = 97, Iteration = 7

test case 2:
6 6 10 -1 -1 -1 1 1 1 3 4 -1 2 -1 -1 2 -1 -1 -1 -1 -1 -1 -1 2 2 -1 2 1 2 -1 -1 1 -1 -1 1 -1 1 0 -1
BruteForce: Frontier_count = 41, Iteration = 31
Optimal_MRVDegree: Frontier_count = 153, Iteration = 80
Optimal_MRVDegreeLCV: Frontier_count = 83, Iteration = 6

test case 3:

6 6 10 -1 -1 -1 -1 -1 -1 -1 2 2 2 3 -1 -1 2 0 0 2 -1 -1 2 0 0 2 -1 -1 3 2 2 2 -1 -1 -1 -1 -1 -1 -1

BruteForce: Frontier_count = 159, Iteration = 149

Optimal_MRVDegree: Frontier_count = 243, Iteration = 24

Optimal_MRVDegreeLCV: Frontier_count = 225, Iteration = 17

test case 4:

6 6 10 -1 1 -1 1 1 -1 2 2 3 -1 -1 1 -1 -1 5 -1 5 -1 2 -1 5 -1 -1 -1 -1 2 -1 -1 3 -1 -1 -1 1 1 -1 0

BruteForce: Frontier_count = 55, Iteration = 45

Optimal_MRVDegree: Frontier_count = 41, Iteration = 7

Optimal_MRVDegreeLCV: Frontier_count = 41, Iteration = 4

```
PS D:\NCTU\108_2\Intro. to Artificial Intelligence\Prog2> .\a.exe
6 6 10  -1 1 -1 1 1 -1  2 2 3 -1 -1 1  -1 -1 5 -1 5 -1  2 -1 5 -1 -1 -1  -1 2 -1 -1 3 -1  -1 -1 1 1 -1 0
Case BruteForce:
Solution:
(1)[1](0)[1][1](0)
[2][2][3](1)(0)[1]
(0)(1)[5](1)[5](1)
[2](1)[5](1)(1)(0)
(0)[2](0)(1)[3](0)
(1)(0)[1][1](0)[0]

Frontier_count =  55
Iteration =  45

Case Optimal(MRV+Degree):
Solution:
(1)[1](0)[1][1](0)
[2][2][3](1)(0)[1]
(0)(1)[5](1)[5](1)
[2](1)[5](1)(0)(1)
(0)[2](0)(1)[3](0)
(1)(0)[1][1](0)[0]

Frontier_count =  41
Iteration =  7

Case Optimal(MRV+Degree+LCV):
Solution:
(1)[1](0)[1][1](0)
[2][2][3](1)(0)[1]
(0)(1)[5](1)[5](1)
[2](1)[5](1)(0)(1)
(0)[2](0)(1)[3](0)
(1)(0)[1][1](0)[0]

Frontier_count =  41
Iteration =  4
```

test case 5 (no solution case):

6 6 10 -1 -1 -1 1 1 -1 -1 3 -1 -1 -1 0 2 3 -1 3 2 2 -1 -1 2 -1 -1 -1 -1 2 2 3 -1 3 -1 1 -1 -1 -1 1

BruteForce: Frontier_count = 201, Iteration = 201

Optimal_MRVDegree: Frontier_count = 33, Iteration = 33

Optimal_MRVDegreeLCV: Frontier_count = 33, Iteration = 33

```
PS D:\NCTU\108_2\Intro. to Artificial Intelligence\Prog2> .\a.exe
6 6 10  -1 -1 -1 1 1 -1 -1 3 -1 -1 -1 0 2 3 -1 3 2 2 -1 -1 2 -1 -1 -1 -1 2 2 3 -1 3 -1 1 -1 -1 -1 1
Case BruteForce:
No solution

Frontier_count =  201
Iteration =  201

Case Optimal(MRV+Degree):
No solution

Frontier_count =  33
Iteration =  33

Case Optimal(MRV+Degree+LCV):
No solution

Frontier_count =  33
Iteration =  33

PS D:\NCTU\108_2\Intro. to Artificial Intelligence\Prog2>
```

Observation of test:

We can say that ***Frontier_count*** corresponds to space complexity, ***iteration*** corresponds to time complexity. Although I have simplified the node expanding of Brute-Force, which makes its' ***Frontier_count*** seems to be small. When we do an observation on ***iteration***, Brute-Force performs the worst through 3 methods, which means it tries lots of useless cases. Optimal_MRVDegree and Optimal_MRVDegreeLCV have improve a lot on ***iteration***. In fact, the ***Frontier_count*** can also reduce in these 2 of methods because many nodes we expand are useless, they just stay at the bottom of stack for ensuring every possible node can be used if needed.

Conclusion of test:

With forward check and heuristic function can really improve the performance. In Brute-Force, I even need to simplify the expanding method and it also need to try lots of iteration to get answer. And for Optimal_MRVDegree and Optimal_MRVDegreeLCV, the affection of heuristic function can be composited. It means that if some heuristic functions can really improve the performance, once we do the connection between each of them, not only can we improve the performance but we can obtain a great methods than separate them.

Appendix:

main.cpp:

```
#include <iostream>
#include <queue>
#include <vector>
```

```
#include <climits> // INT_MAX

using namespace std;
class Variable{
public:
	bool isVar;
	int value,degree,position,mdl,mdu;
	Variable(int val = -1){
		value = val;
		position = -1;
		mdl = mdu = INT_MAX; // MIN_DISTANCE_TO_LOWER,
MIN_DISTANCE_TO_UPPER
		degree = 1; // number of Constrain the Variable associates,
min_degree = 1, num of bomb
		isVar = false; // isVAR-> value = -1: unassigned variable,0:
clear,1: bomb
	}
	friend bool operator>(const Variable & v1,const Variable & v2){
		return v1.degree > v2.degree; // v2 FO, Degree heuristic
	}
};
class Constrain{
public:
	int value;
	vector<int> variable;
	Constrain(int val = -1):value(val){}
};
void print_solution(vector<Variable> & solution);
void BruteForce(vector<Variable> & board,vector<Constrain> & cons);
void Optimal_MRVDegree(vector<Variable> & board,vector<Constrain>
& cons);
void Optimal_MRVDegreeLCV(vector<Variable> &
board,vector<Constrain> & cons);
int n,m,NumofMines;
int main(int argc, char const *argv[]){
	scanf("%d%d%d",&n,&m,&NumofMines);
	vector<Variable> board(n * m);
	vector<Constrain> cons;
```

```
	cons.push_back(Constrain(NumofMines)); // Num of Mines
constrain
	for(int pos = 0;pos < n * m;pos++){
		scanf("%d",&board[pos].value);
		board[pos].position = pos;
		if(board[pos].value == -1){ // var record
			cons[0].variable.push_back(pos);
			board[pos].isVar = true;
		}
		else
			cons.push_back(Constrain(pos));
	}
	int dx[] = {-1,-1,-1,0,0,1,1,1};
	int dy[] = {-1,0,1,-1,1,-1,0,1};
	for(int i = 1;i < cons.size();i++){
		int x = cons[i].value / m,y = cons[i].value % m;
		for(int di = 0;di < 8;di++){
			int nx = x + dx[di],ny = y + dy[di];
			if(0 <= nx && nx < n && 0 <= ny && ny < m &&
board[nx * m + ny].value == -1){
				cons[i].variable.push_back(nx * m + ny);
				board[nx * m + ny].degree++;
			}
		}
		cons[i].value = board[x * m + y].value; // replace hint pos into
value
	}
	printf("Case BruteForce:\n");
	BruteForce(board,cons);
	printf("Case Optimal(MRV+Degree):\n");
	Optimal_MRVDegree(board,cons);
	printf("Case Optimal(MRV+Degree+LCV):\n");
	Optimal_MRVDegreeLCV(board,cons);
	return 0;
}
void BruteForce(vector<Variable> & board,vector<Constrain> & cons){
	int Frontier_count = 1,iteration = 0;
	bool SolutionExist = false;
```

```
vector<vector<Variable> > stack;
stack.push_back(board);
while(!stack.empty()){
	iteration++;
	vector<Variable> node = stack.back(); // board in current node
	stack.pop_back();
	bool goal = true,drop_node = false;
	for(int i = 0;i < cons.size();i++){
		int lower = 0,upper = cons[i].variable.size();
		for(int j = 0;j < cons[i].variable.size();j++){
			if(node[cons[i].variable[j]].value == 0)
				upper--;
			else if(node[cons[i].variable[j]].value == 1)
				lower++;
		}
		if(lower == cons[i].value && upper == cons[i].value)
			continue; // check next constrain
		else if(lower > cons[i].value || upper < cons[i].value){
			drop_node = true; // A constrain can't be satisfied
			break;
		}
		else
			goal = false; // NOT goal node
	}
	if(drop_node)
		continue; //
	if(goal){
		SolutionExist = true;
		printf("Solution:\n");
		print_solution(node);
		printf("\nFrontier_count =	%d\n",Frontier_count);
		printf("Iteration =	%d\n\n",iteration);
		break;
	}

	// cons[0].variable has all variable
	for(int i = 0;i < cons[0].variable.size();i++)
		if(node[cons[0].variable[i]].value == -1){
```

```
                    node[cons[0].variable[i]].value = 0;
                    stack.push_back(node);
                    node[cons[0].variable[i]].value = 1;
                    stack.push_back(node);
                    Frontier_count += 2;
                    break;
                }
        }
        if(!SolutionExist){
            printf("No solution\n");
            printf("\nFrontier_count =    %d\n",Frontier_count);
            printf("Iteration =    %d\n\n",iteration);
        }
}
void Optimal_MRVDegree(vector<Variable> & board,vector<Constrain>
& cons){
        int Frontier_count = 1,iteration = 0;
        bool SolutionExist = false;
        vector<vector<Variable> > stack;
        stack.push_back(board);
        while(!stack.empty()){
            iteration++;
            vector<Variable> node = stack.back(); // board in current node
            stack.pop_back();
            bool goal = true,drop_node = false;
            for(int i = 0;i < cons.size();i++){
                // check all constrain upper and lower bound and do goal
test(all constrains are satified)
                // MRV is also implemented in this for loop,
                // domain_size = 1 can be assigned at current state directly
                int lower = 0,upper = cons[i].variable.size();
                for(int j = 0;j < cons[i].variable.size();j++){
                    if(node[cons[i].variable[j]].value == 0)
                        upper--;
                    else if(node[cons[i].variable[j]].value == 1)
                        lower++;
                }
                if(lower == cons[i].value && upper == cons[i].value)
```

```
                continue; // check next constrain
            else if(lower > cons[i].value || upper < cons[i].value){
                drop_node = true; // A constrain can't be satisfied
                break;
            }
            else if(lower == cons[i].value || upper == cons[i].value){
                for(int j = 0;j < cons[i].variable.size();j++){
                    if(node[cons[i].variable[j]].value == -1) // var's
domain has become singletons
                        node[cons[i].variable[j]].value = lower ==
cons[i].value?0:1;
                }
            }
            else
                goal = false; // NOT goal node
        }
        if(drop_node)
            continue; //
        if(goal){
            SolutionExist = true;
            printf("Solution:\n");
            print_solution(node);
            printf("\nFrontier_count =    %d\n",Frontier_count);
            printf("Iteration =    %d\n\n",iteration);
            break;
        }

        // cons[0].variable has all variable
        priority_queue<Variable,vector<Variable>,greater<Variable> >
q;
        for(int i = 0;i < cons[0].variable.size();i++)
            if(node[cons[0].variable[i]].value == -1) // unassigned
node
                q.push(node[cons[0].variable[i]]);
        while(!q.empty()){
            int pos = q.top().position;
            q.pop();
            node[pos].value = 0;
```

```
                stack.push_back(node);
                node[pos].value = 1;
                stack.push_back(node);
                node[pos].value = -1;
                Frontier_count += 2;
            }

        }
        if(!SolutionExist){
            printf("No solution\n");
            printf("\nFrontier_count =    %d\n",Frontier_count);
            printf("Iteration =    %d\n\n",iteration);
        }
}
void Optimal_MRVDegreeLCV(vector<Variable> &
board,vector<Constrain> & cons){
        bool SolutionExist = false;
        int Frontier_count = 1,iteration = 0;
        vector<vector<Variable> > stack;
        stack.push_back(board);
        while(!stack.empty()){
            iteration++;
            vector<Variable> node = stack.back(); // board in current node
            stack.pop_back();
            bool goal = true,drop_node = false;
            for(int i = 0;i < cons.size();i++){
                // check all constrain upper and lower bound and do goal
test(all constrains are satified)
                // MRV is also implemented in this for loop,
                // domain_size = 1 can be assigned at current state directly
                int lower = 0,upper = cons[i].variable.size();
                for(int j = 0;j < cons[i].variable.size();j++){
                    if(node[cons[i].variable[j]].value == 0)
                        upper--;
                    else if(node[cons[i].variable[j]].value == 1)
                        lower++;
                }
                if(lower == cons[i].value && upper == cons[i].value)
```

```
					continue; // check next constrain
				else if(lower > cons[i].value || upper < cons[i].value){
					drop_node = true; // A constrain can't be satisfied
					break;
				}
				else if(lower == cons[i].value || upper == cons[i].value){
					for(int j = 0;j < cons[i].variable.size();j++){
						if(node[cons[i].variable[j]].value == -1) // var's
domain has become singletons
							node[cons[i].variable[j]].value = lower ==
cons[i].value?0:1;
					}
				}
				else{
					for(int j = 0;j < cons[i].variable.size();j++){
						if(node[cons[i].variable[j]].mdu > upper -
cons[i].value)
							node[cons[i].variable[j]].mdu = upper -
cons[i].value;
						if(node[cons[i].variable[j]].mdl > cons[i].value -
lower)
							node[cons[i].variable[j]].mdl =
cons[i].value - lower;
					}
					goal = false; // NOT goal node
				}
			}
			if(drop_node)
				continue; //
			if(goal){
				SolutionExist = true;
				printf("Solution:\n");
				print_solution(node);
				printf("\nFrontier_count =	%d\n",Frontier_count);
				printf("Iteration =	%d\n\n",iteration);
				break;
			}
			// cons[0].variable has all variable
```

```
		priority_queue<Variable,vector<Variable>,greater<Variable> >
q;
		for(int i = 0;i < cons[0].variable.size();i++)
			if(node[cons[0].variable[i]].value == -1)
				q.push(node[cons[0].variable[i]]);
		while(!q.empty()){
			int pos = q.top().position;
			q.pop();
			// LCV, consider that 0 and 1 which can be assigned first
			if(node[pos].mdu < node[pos].mdl){
				node[pos].value = 0;
				stack.push_back(node);
				node[pos].value = 1;
				stack.push_back(node);
			}
			else{
				node[pos].value = 1;
				stack.push_back(node);
				node[pos].value = 0;
				stack.push_back(node);
			}
			node[pos].value = -1;
			Frontier_count += 2;
		}
	}
	if(!SolutionExist){
		printf("No solution\n");
		printf("\nFrontier_count =	%d\n",Frontier_count);
		printf("Iteration =	%d\n\n",iteration);
	}
}
void print_solution(vector<Variable> & solution){
	for(int i = 0;i < n;i++){
		for(int j = 0;j < m;j++)
			printf(solution[i * m + j].isVar?"(%d)":"[%d]",solution[i *
m + j].value);
		printf("\n");
	}
```

}